Table of Contents

PART 1 - Getting started

Rx is a .NET library for processing event streams. Why might you want that?

Why Rx?

Users want timely information. If you're expecting a parcel, live reports of the delivery van's progress give you more freedom than a suspect 2 hour delivery window. Financial applications depend on continuous streams of up-to-date data. We expect our phones and computers to provide us with all sorts of important notifications. And some applications simply can't work without live information—online collaboration tools and multiplayer games absolutely depend on the rapid distribution and delivery of data.

In short, our systems need to react when interesting things happen.

Live information streams are a basic, ubiquitous element of computer systems. Despite this, they are often a second class citizen in programming languages. Most languages support sequences of data through something like an array, which presumes that the data is sitting in memory ready for our code to read at its leisure. If your application deals with events, arrays might work for historical data, but they aren't a good way to represent events that occur while the application is running. And although streamed data is a pretty venerable concept in computing, it tends to be clunky, with the abstractions often surfaced through APIs that are poorly integrated with our programming language's type system.

This is bad. Live data is critical to a wide range of applications. It should be as easy to work with as lists, dictionaries, and other collections.

The Reactive Extensions for .NET (Rx.NET or Rx for short) elevate live data sources to first class citizens. Rx does not require any special programming language support. It exploits .NET's type system to represent streams of data in a way that .NET languages such as C#, F#, and VB.NET can all work with as naturally as they use collection types.

(A brief grammatical aside: although the phrase "Reactive Extensions" is plural, when we reduce it to just Rx.NET or Rx, we treat it as a singular noun. This is inconsistent, but saying "Rx are…" sounds plain weird.)

For example, C# has integrated query features that we might use to find all of the entries in a list that meet some criteria. If we have some `List<Trade> trades` variable, we might write this:

```
var bigTrades =
    from trade in trades
    where trade.Volume > 1_000_000;
```

With Rx, we could use this exact same code with live data. Instead of being a `List<Trade>`, the `trades` variable could be an `IObservable<Trade>`. (`IObservable<T>` is the fundamental abstraction in Rx. It is essentially a live version of `IEnumerable<T>`.) In this case, `bigTrades` would also be an `IObservable<Trade>`, a live data source able to notify us of all trades whose `Volume` exceeds one million. Crucially, it will report each such trade immediately—this is what we mean by a 'live' data source.

Rx is a powerfully productive development tool. It enables developers to work with live event streams using language features familiar to all .NET developers. It enables an elegant, declarative approach that often allows us to express complex behaviour more elegantly and with less code than would be possible without Rx.

Rx builds on LINQ (Language Integrated Query). This enables us to use the query syntax shown above (or you can use the explicit function call approach that some .NET developers prefer). LINQ is widely used in .NET both for data access (e.g., in Entity Framework Core), but also for working with in-memory collections (with LINQ to Objects), meaning that experienced .NET developers will tend to feel at home with Rx. Crucially, LINQ is a highly composable design: you can connect operators together in any combination you like, expressing potentially complex processing in a straightforward way.

When is Rx appropriate?

Rx is designed for processing sequences of events, meaning that it suits some scenarios better than others. The next sections describe some of these scenarios, and also cases in which it is a less obvious match but still worth considering. Finally, we describe some cases in which it is possible to use Rx but where alternatives are likely to be better.

Good Fit with Rx

Rx is well suited to representing events that originate from outside of your code, and which your application needs to respond to, such as:

- Integration events like a broadcast from a message bus or a push event from WebSockets API or other low latency middleware like Azure Event Grid, Azure Event Hubs and Azure Service Bus
- Reports from monitoring devices such as a flow sensor in a water utility's infrastructure, or the monitoring and diagnostic features in networking equipment
- Location data from mobile systems such as AIS messages from ships, or automotive telemetry
- Operating system events such as filesystem activity, or WMI events
- Road traffic information, such as notifications of accidents or changes in average speed
- Integration with a Complex Event Processing (CEP) engine

- UI events such as mouse movement or button clicks

Rx is also good way to model domain events—these may occur as a result of some of the events just described, but after processing them to produce events that more directly represent application concepts. These might include:

- Property or state changes on domain objects such as "Order Status Updated", or "Registration Accepted"
- Changes to collections of domain objects, such as "New Registration Created"

Events might also represent insights derived from incoming events and historical data such as:

- A broadband customer might have become an unwitting participant in a DDoS attack
- CNC Milling Machine MFZH12's number 4 axis bearing is exhibiting signs of wear at a significantly higher rate than the nominal profile
- If the user wants to arrive on time at their meeting half way across town, the current traffic conditions suggest they should leave in the next 10 minutes

These three sets of examples show how applications might progressively increase the value of the information as they process events. We start with raw events, which we then enhance to produce domain-specific events, and we then perform analysis to produce notifications that the application's users will really care about. Each stage of processing increases the value of the messages that emerge. Each stage will typically also reduce the volume of messages—if we presented the raw events in the first category directly to users, they might be overwhelmed by the volume of messages, making it impossible to spot the important events. But if we only present them with notifications when our processing has detected something important, this will enable them to work more efficiently and accurately.

The `System.Reactive` library provides tools for building exactly this kind of value-adding process, in which we tame high-volume raw event sources to produce high-value live insights. It provides a suite of operators that enable our code to express this kind of processing declaratively, as you'll see in subsequent chapters.

Rx is also well suited for introducing and managing concurrency for the purpose of *offloading*. That is, performing a given set of work concurrently, so that the thread that detected an event doesn't also have to be the thread that handles that event. A very popular use of this is maintaining a responsive UI. (UI event handling has become *such* a popular use of Rx—both in .NET. but also in RxJS, which originated as an offshoot of Rx.NET—that it would be easy to think that this is what it's for. But its success there should not blind us to its wider applicability.)

You should consider using Rx if you have an existing `IEnumerable<T>` that is attempting to model live events. While `IEnumerable<T>` *can* model data in motion (by using lazy evaluation like `yield`

`return`), there's a problem. If the code consuming the collection has reached the point where it wants the next item (e.g., because a `foreach` loop has just completed an iteration) but no item is yet available, the `IEnumerable<T>` implementation would have no choice but to block the calling thread in its `MoveNext` until such time as data is available, which can cause scalability problems in some applications. Even in cases where thread blocking is acceptable (or if you use the newer `IAsyncEnumerable<T>`, which can take advantage of C#'s `await foreach` feature to avoid blocking a thread in these cases) `IEnumerable<T>` and `IAsyncEnumerable<T>` are misleading types for representing live information sources. These interfaces represent a 'pull' programming model: code asks for the next item in the sequence. Rx is a more natural choice for modelling information sources that naturally produce information on their own schedule.

Possible Fit with Rx

Rx can be used to represent asynchronous operations. .NET's `Task` or `Task<T>` effectively represent a single event, and `IObservable<T>` can be thought if as a generalization of this to a sequence of events. (The relationship between, say, `Task<int>` and `IObservable<int>` is similar to the relationship between `int` and `IEnumerable<int>`.)

This means that there are some scenarios that can be dealt with either using tasks and the `async` keyword or through Rx. If at any point in your processing you need to deal with multiple values as well as single ones, Rx can do both; tasks don't handle multiple items so well. You can have a `Task<IEnumerable<int>>`, which enables you to `await` for a collection, and that's fine if all the items in the collection can be collected in a single step. The limitation with this is that once the task has produced its `IEnumerable<int>` result, your `await` has completed, and you're back to non-asynchronous iteration over that `IEnumerable<int>`. If the data can't be fetched in a single step—perhaps the `IEnumerable<int>` represents data from an API in which results are fetched in batches of 100 items at a time—its `MoveNext` will have to block your thread every time it needs to wait.

For the first 5 years of its existence, Rx was arguably the best way to represent collections that wouldn't necessarily have all the items available immediately. However, the introduction of `IAsyncEnumerable<T>` in .NET Core 3.0 and C# 8 provided a way to handle sequences while remaining in the world of `async`/`await` (and the `Microsoft.Bcl.AsyncInterfaces` NuGet package makes this available on .NET Framework and .NET Standard 2.0). So the choice to use Rx to now tends to boil down to whether a 'pull' model (exemplified by `foreach` or `await foreach`) or a 'push' model (in which code supplies callbacks to be invoked by the event source when items become available) is a better fit for the concepts being modelled.

Earlier, I mentioned *offloading*: using Rx to push work onto other threads. Although this technique can enable Rx to introduce and manage concurrency for the purposes of *scaling* or performing *parallel*

computations, other dedicated frameworks like TPL (Task Parallel Library) Dataflow or PLINQ are more appropriate for performing parallel compute intensive work. However, TPL Dataflow offers some integration with Rx through its `AsObserver` and `AsObservable` extension methods. So it is common to use Rx to integrate TPL Dataflow with the rest of your application.

Poor Fit with Rx

Rx's `IObservable<T>` is not a replacement for `IEnumerable<T>` or `IAsyncEnumerable<T>`. It would be a mistake to take something that is naturally pull based and force it to be push based.

Also, there are some situations in which the simplicity of Rx's programming model can work against you. For example, although some message queuing technologies such as MSMQ are by definition sequential, and thus might look like a good fit for Rx, these are often chosen for their transaction handling support. Rx does not have any direct way to surface transaction semantics, so in scenarios that require this you might be better off just working directly with the relevant technology's API.

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

Rx in action

You can get up and running with a simple Rx example very quickly. If you have the .NET SDK installed. Run the following at a command line:

```
mkdir TryRx
cd TryRx
dotnet new console
dotnet add package System.Reactive
```

Alternatively, if you have Visual Studio installed, create a new .NET Console project, and then use the NuGet package manager to add a reference to `System.Reactive`.

This code creates an observable source (`ticks`) that produces an event once every second. The code also passes a handler to that source that writes a message to the console for each event:

```
using System.Reactive.Linq;

IObservable<long> ticks = Observable.Timer(
    dueTime: TimeSpan.Zero,
    period: TimeSpan.FromSeconds(1));

ticks.Subscribe(
```

```
    tick => Console.WriteLine($"Tick {tick}"));

Console.ReadLine();
```

If this doesn't seem very exciting, it's because it's about as basic an example as it's possible to create, and at its heart, Rx has a very simple programming model. The power comes from composition—we can use the building blocks in the `System.Reactive` library to describe the processing that will takes us from raw, low-level events to high-value insights. But to do that, we must first understand Rx's key types, `IObservable<T>` and `IObserver<T>`.

Key types

Rx is a powerful framework that can greatly simplify code that responds to events. But to write good Reactive code you have to understand the basic concepts. The fundamental building block of Rx is an interface called `IObservable<T>`. Understanding this, and its counterpart `IObserver<T>`, is the key to success with Rx.

The preceding chapter showed this LINQ query expression as the first example:

```
var bigTrades =
    from trade in trades
    where trade.Volume > 1_000_000;
```

Most .NET developers will be familiar with LINQ in at least one of its many popular forms such as LINQ to Objects, or Entity Framework Core queries. Most LINQ implementations allow you to query *data at rest*—LINQ to Objects works on arrays or other collections; LINQ queries in Entity Framework Core run against data in a database. But Rx is different: it offers the ability to define queries over live event streams—what you might call *data in motion*.

If you don't like the query expression syntax, you can write exactly equivalent code by invoking LINQ operators directly:

```
var bigTrades = trades.Where(trade => trade.Volume > 1_000_000);
```

Whichever style we use, this is the LINQ way of saying that we want `bigTrades` to have just those items in `trades` where the `Volume` property is greater than one million.

We can't tell exactly what these examples do because we can't see the type of the `trades` or `bigTrades` variables. The meaning of this code is going to vary greatly depending on these types. If we were using LINQ to objects, these would both likely be `IEnumerable<Trade>`. That would mean that these variables both referred to objects representing collections whose contents we could

enumerate with a `foreach` loop. This would represent *data at rest*, data that our code could inspect directly.

But let's make it clear what the code means by being explicit about the type:

```
IObservable<Trade> bigTrades = trades.Where(trade => trade.Volume >
↪  1_000_000);
```

This removes all ambiguity. It is now clear that we're not dealing with data at rest. We're working with an `IObservable<Trade>`. But what exactly is that?

IObservable

The `IObservable<T>` interface represents Rx's fundamental abstraction: a sequence of values of some type T. In a very abstract sense, this means it represents the same thing as `IEnumerable<T>`. The difference is in how code consumes those values. Whereas `IEnumerable<T>` enables code to retrieve values (typically with a `foreach` loop), an `IObservable<T>` provides values when they become available. This distinction is sometimes characterised as *push* vs *pull*. We can *pull* values out of an `IEnumerable<T>` by executing a `foreach` loop, but and `IObservable<T>` will *push* values into our code.

How can an `IObservable<T>` push its values into our code? If we want these values, our code must *subscribe* to the `IObservable<T>`, which means providing it with some methods it can invoke.

In fact, subscription is the only operation an `IObservable<T>` directly supports. Here's the entire definition of the interface:

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

You can see the source for `IObservable<T>` on GitHub. Notice that it is part of the .NET runtime libraries, and not the `System.Reactive` NuGet package. `IObservable<T>` represents such a fundamentally important abstraction that it is baked into .NET. (So you might be wondering what the `System.Reactive` NuGet package is for. The .NET runtime libraries define only the `IObservable<T>` and `IObserver<T>` interfaces, and not the LINQ implementation. The Rx NuGet package gives us LINQ support, and also deals with threading.)

This interface's only method makes it clear what we can do with an `IObservable<T>`: if we want to receive the events it has to offer, we can *subscribe* to it. (We can also unsubscribe: the `Subscribe` method returns an `IDisposable`, and if we call `Dispose` on that it cancels our subscription.) The

`Subscribe` method requires us to pass in an implementation of `IObserver<T>`, which we will get to shortly.

Observant readers will have noticed that an example in the preceding chapter looks like it shouldn't work. That code created an `IObservable<long>` that produced events once per second, and then it subscribed to it with this code:

```
ticks.Subscribe(
    tick => Console.WriteLine($"Tick {tick}"));
```

That's passing a delegate, and not the `IObserver<T>` that `IObservable<T>.Subscribe` requires. We'll get to `IObserver<T>` shortly, but all that's happening here is that this example is using an extension method from the `System.Reactive` NuGet package:

```
// From the System.Reactive library's ObservableExtensions class
public static IDisposable Subscribe<T>(this IObservable<T> source,
↪   Action<T> onNext)
```

This is a helper method that wraps a delegate in an implementation of `IObserver<T>` and then passes that to `IObservable<T>.Subscribe`. The effect is that we can write just a simple method (instead of a complete implementation of `IObserver<T>`) and the observable source will invoke our callback each time it wants to supply a value.

Hot and Cold Sources

Since an `IObservable<T>` cannot supply us with values until we subscribe, the time at which we subscribe can be important. Imagine an `IObservable<Trade>` describing trades occurring in some market. If the information it supplies is live, it's not going to tell you about any trades that occurred before you subscribed. In Rx, sources of this kind are described as being *hot*.

Not all sources are *hot*. There's nothing stopping an `IObservable<T>` always supplying the exact same sequence of events to any subscriber no matter when the call to `Subscribe` occurs. (Imagine an `IObservable<Trade>` which, instead of reporting live information, generates notifications based on recorded historical trade data.) Sources where it doesn't matter at all when you subscribe are known as *cold* source.

Here are some sources that might be represented as hot observables:

- Measurements from a sensor
- Price ticks from a trading exchange
- An event source that distributes events immediatly such as Azure Event Grid
- mouse movements
- timer events

- broadcasts like ESB channels or UDP network packets.

And some examples of cold observables:

- An observable representing the contents of a collection (such as is returned by the `ToObservable` extension method for `IEnumerable<T>`)
- An observable producing a fixed range of values, such as `Observable.Range` produces
- An observable that generates events based on an algorithm, such as `Observable.Generate` produces
- An observable representing a factory for an asynchronous operation, as `FromAsync` returns
- An observable of the kind typically created using `Observable.Create`
- An observable representing items from a streaming event provides such as Azure Event Hub or Kafka (or any other streaming-style source which holds onto events from the past to be able to deliver events from a particular moment in the stream; so *not* an events source in the Azure Event Grid style)

Not all sources are strictly completely *hot* or *cold*. For example, you could imagine a slight variation on a live `IObservable<Trade>` where the source always reports the most recent trade to new subscribers. Subscribers can count on immediately receiving something, and will then be kept up to date as new information arrives. The fact that new subscribers will always receive (potentially quite old) information is a *cold*-like characteristic, but it's only that first event that is *cold*—it's still likely that a brand new subscriber will have missed lots of information that would have been available to earlier subscribers, making this source more *hot* than *cold*.

There's an interesting special case in which a source of events has been designed to enable applications to receive every single event in order, exactly once. Event streaming systems such as Kafka or Azure Event Hub have this characteristic—they retain events for a while, to ensure that consumers don't miss out even if they fall behind from time to time. The standard input (*stdin*) for a process also has this characteristic: if you run a command line tool and start typing input before it is ready to process it, the operating system will hold that input in a buffer, to ensure that nothing is lost. Windows does something similar for desktop applications—each application thread gets a message queue so that if you click or type when it's not able to respond, the input will eventually be processed. We might think of these sources as *cold*-then-*host*. They're like *cold* sources in that we won't miss anything just because it took us some time to start receiving events, but once we start retrieving the data, then we can't generally rewind back to the start. So once we're up and running they are more like *hot* events.

This kind of *cold*-then-*hot* source can present a problem if we want to attach multiple subscribers. If the source starts providing events as soon as subscription occurs, then that's fine for the very first subscriber: it will receive any events that were backed up waiting for us to start. But if we wanted to attach multiple subscribers, we've got a problem: that first subscriber might receive all the notifications

that were sitting waiting in some buffer before we manage to attach the second subscriber. The second subscriber will miss out.

In these cases, we really want some way to rig up all our subscribers before kicking things off. We want subscription to be separate from the act of starting. By default, subscribing to a source implies that we want it to start, but rx defines a specialised interface that can give us more control: `IConnectableObservable<T>`. This derives from `IObservable<T>`, and adds just a single method, `Connect`:

```
public interface IConnectableObservable<out T> : IObservable<T>
{
    IDisposable Connect();
}
```

This is useful in these scenarios where there will be some process that fetches or generates events and we need to make sure we're prepared before that starts. Because an `IConnectableObservable<T>` won't start until you call `Connect`, it provides you with a way to attach however many subscribers you need before events begin to flow.

The 'temperature' of a source is not necessarily evident from its type. Even when the underlying source is an `IConnectableObservable<T>`, that can often be hidden behind layers of code. So whether a source is hot, cold, or something in between, most of the time we just see an `IObservable<T>`. Since `IObservable<T>` defines just one method, `Subscribe`, you might be wondering how we can do anything interesting with it. The power comes from the LINQ operators that the `System.Reactive` NuGet library supplies.

LINQ Operators and Composition

So far I've shown only a very simple LINQ example, using the `Where` operator to filter events down to ones that meet certain criteria. To give you a flavour of how we can build more advanced functionality through composition, I'm going to introduce an example scenario.

Suppose you want to write a program that watches some folder on a filesystem, and performs automatic processing any time something in that folder changes. For example, web developers often want to trigger automatic rebuilds of their client side code when they save changes in the editor so they can quickly see the effect of their changes. Often in these cases, filesystem changes often come in bursts—text editors might perform a few distinct operations when saving a file. (Some save modifications to a new file, then perform a couple of renames once this is complete, because this avoids data loss if a power failure or system crash happens to occur at the moment you save the file.) So you typically won't want to take action as soon as you detect file activity. It would be better to give it a moment to see if any more activity occurs, and take action only after everything has settled down.

So we should not react directly to filesystem activity. We want take action at those moments when everything goes quiet after a flurry of activity. The following code defines an Rx operator that detects and reports such things. If you're new to Rx (which seems likely if you're reading this) it probably won't be instantly obvious how this works, so I'll walk through it.

```
static class RxExt
{
    public static IObservable<IList<T>> Quiescent<T>(
        this IObservable<T> src,
        TimeSpan minimumInactivityPeriod,
        IScheduler scheduler)
    {
        IObservable<int> onoffs =
            from _ in src
            from delta in Observable.Return(1,
↪   scheduler).Concat(Observable.Return(-1,
↪   scheduler).Delay(minimumInactivityPeriod, scheduler))
            select delta;
        IObservable<int> outstanding = onoffs.Scan(0, (total, delta) =>
↪   total + delta);
        IObservable<int> zeroCrossings = outstanding.Where(total => total
↪   == 0);
        return src.Buffer(zeroCrossings);
    }
}
```

The first thing to say about this is that we are effectively defining a custom LINQ-style operator: this is an extension method which, like all of the LINQ operators Rx supplies, takes an `IObservable<T>` as its implicit argument, and produces another observable source as its result. The return type is slightly different: it's `IObservable<IList<T>>`. That's because once we return to a state of inactivity, we will want to process everything that just happened, so this operator will produce a list containing every value that the source reported in its most recent flurry of activity.

When we want to show how an Rx operator behaves, we typically draw a 'marble' diagram. This is a diagram showing one or more `IObservable<T>` event sources, with each one being illustrated by a horizontal line. Each event that a source produces is illustrated by a circle (or 'marble') on that line, with the horizontal position representing timing. Typically, the line has a vertical bar on its left indicating the instant at which the source comes into existence, unless it happens to produce events from the very instant it is created, in which case it will start with a marble. If the line has an arrowhead on the right, that indicates that the observable's lifetime extends beyond the diagram. Here's a diagram showing how the `Quiescent` operator above response to a particular input:

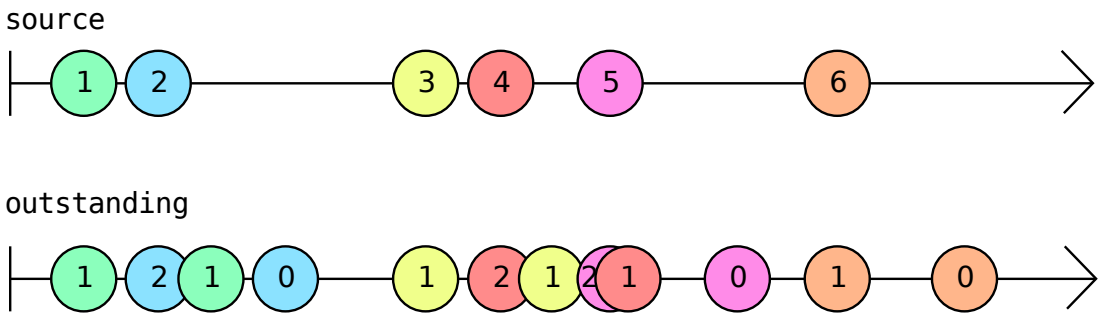
An Rx marble diagram illustrating two observables.

This shows that the source produced a couple of events (the values `1` and `2`, in this example), and then stopped for a bit. A little while after it stopped, the observable returned by the `Quiescent` operator produced a single event with a list containing both of those events (`[1,2]`). Then the source started up again, producing the values `3`, `4`, and `5` in fairly quick succession, and then going quiet for a bit. Again, once the quiet spell had gone on for long enough, the source returned by `Quiescent` produced a single event containing all of the source events from this second burst of activity (`[3,4,5]`). And then the final bit of source activity shown in this diagram consists of a single event, `6`, followed by more inactivity, and again, once the inactivity has gone on for long enough the `Quiescent` source produces a single event to report this. And since that last 'burst' of activity from the source contained only a single event, the list reported by this final output from the `Quiescent` observable is a list with a single value: `[6]`.

So how does the code shown achieve this? The first thing to notice about the `Quiescent` method is that it's just using other Rx LINQ operators (the `Return`, `Scan`, `Where`, and `Buffer` operators are explicitly visible, and the query expression will be using the `SelectMany` operator, because that's what C# query expressions do when they contain two `from` clauses in a row) in a combination that produces the final `IObservable<IList<T>>` output.

This is Rx's *compositional* approach, and it is how we normally use Rx. We use a mixture of operators, combined (*composed*) in a way that produces the effect we want.

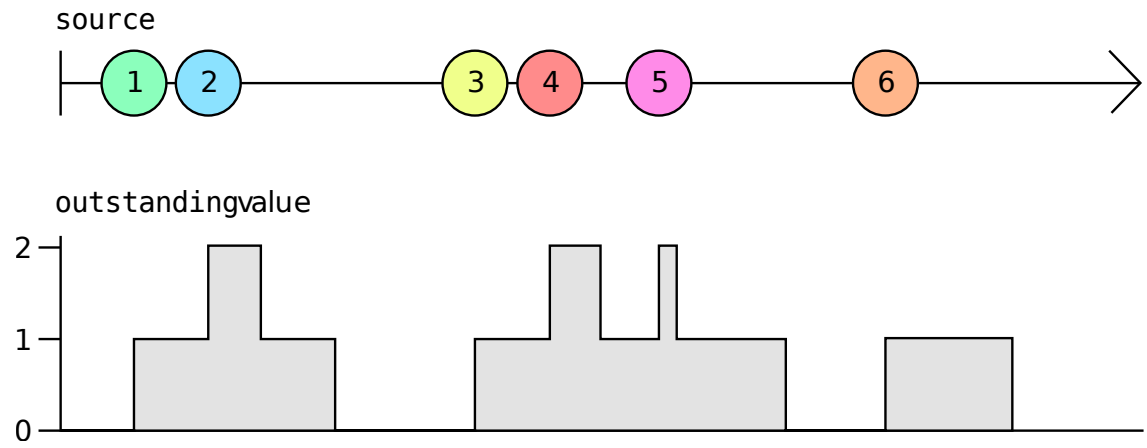
But how is this producing the effect we want? There are a few ways we could get the behaviour that we're looking for from a `Quiescent` operator, but the basic idea of this particular implementation is that it keeps count of how many events have happened recently, and then produces a result every time that number drops back to zero. The `oustanding` variable refers to the `IObservable<int>` that tracks the number of recent events, and this marble diagram shows what it produces in response to the same `source` events as were shown on the preceding diagram:



I've colour coded the events this time so that I can show the relationship between `source` events and corresponding events produced by `outstanding`. Each time `source` produces an event, `oustanding` produces an event at the same time, in which the value is one higher than the preceding

value produced by `outstanding`. But each such `source` event also causes `outstanding` to produce a second event two seconds later. (It's two seconds because in these examples, I've presumed that the first argument to `Quiescent` is `TimeSpan.FromSeconds(2)`, as shown on the first marble diagram.) That second event always produces a value that is one lower than whatever the preceding value was.

This means that each event to emerge from `outstanding` tells us how many events `source` produced within the last two seconds. This diagram shows that same information in a slightly different form—it shows the most recent value produced by `outstanding` as a graph. You can see the value goes up by one each time `source` produces a new value. And two seconds after each value produced by `source`, it drops back down by one.



In simple cases like the final event 6, in which it's the only event that happens at around that time, the `outstanding` value goes up by one when the event happens, and drops down again two seconds later. Over on the left of the picture it's a little more complex: we get two events in fairly quick succession, so the `outstanding` value goes up to one and then up to two, before falling back down to one and then down to zero again. The middle section looks a little more messy—the count goes up by one when the `source` produces event 3, and then up to two when event 4 comes in. It then drops down to one again once two seconds have passed since the 3 event, but then another event, 5, comes in taking the total back up to two. Shortly after that it drops back to one again because it has now been two seconds since the 4 event happened. And then a bit later, two seconds after the 5 event it drops back to zero again.

That middle section is the messiest, but it's also most representative of the kind of activity this operator is designed to deal with. Remember, the whole point here is that we're expecting to see flurries of activity, and if those represents filesystem activity, they will tend to be slightly chaotic in nature, because storage devices don't always have entirely predictable performance characteristics (especially if it's a magnetic storage device with moving parts, or remote storage in which variable networking

delays might come into play).

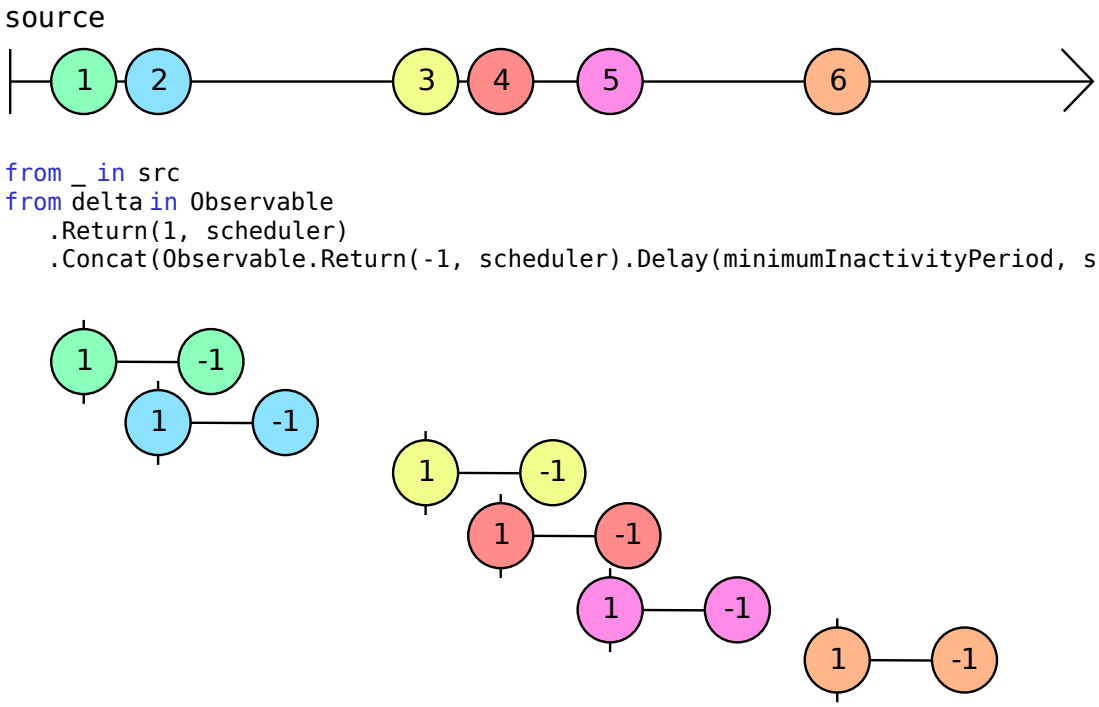
With this measure of recent activity in hand, we can spot the end of bursts of activity by watching for when `oustanding` drops back to zero, which is what the observable referred to by `zeroCrossing` in the code above does. (That's just using the `Where` operator to filter out everything except the events where `outstanding`'s current value returns to zero.)

But how does `outstanding` itself work? The basic approach here is that every time `source` produces a value, we actually create a brand new `IObservable<int>`, which produces exactly two values. It immediately produces the value 1, and then after the specified timespan (2 seconds in these examples) it produces the value -1. That's what's going in in this clause of the query expression:

```
from delta in Observable.Return(1, scheduler).Concat(Observable.Return(-1,
↪  scheduler).Delay(minimumInactivityPeriod, scheduler))
```

I said Rx is all about composition, and that's certainly the case here. We are using the very simple `Return` operator to create an `IObservable<int>` that immediately produces just a single value and then terminates. This code calls that twice, once to produce the value `1` and again to produce the value `-1`. It uses the `Delay` operator so that instead of getting that `-1` value immediately, we get an observable that waits for the specified time period (2 seconds in these examples, but whatever `minimumInactivityPeriod` is in general) before producing the value. And then we use `Concat` to stitch those two together into a single `IObservable<int>` that produces the value `1`, and then two seconds later produces the value `-1`.

Although this produces a brand new `IObservable<int>` for each `source` event, the `from` clause shown above is part of a query expression of the form `from ... from .. select`, which the C# compiler turns into a call to `SelectMany`, which has the effect of flattening those all back into a single observable, which is what the `onoffs` variable refers to. This marble diagram illustrates that:



This also shows the `outstanding` observable again, but we can now see where that comes from: it is just the running total of the values emitted by the `onoffs` observable. This running total observable is created with this code:

```
IObservable<int> outstanding = onoffs.Scan(0, (total, delta) => total +
↪  delta);
```

Rx's `Scan` operator works much like the standard LINQ `Aggregate` operator, in that it cumulatively applies an operation (adding, in this case) to every single item in a sequence. The different is that whereas `Aggregate` produces just the final result once it reaches the end of the sequence, `Scan` shows all of its working, producing the accumulated value so far after each input. So this means that

`outstanding` will produce an event every time `onoffs` produces one, and that event's value will be the running total—the sum total of every value from `onoffs` so far.

So that's how `outstanding` comes to tell us how many events `source` produced within the last two seconds (or whatever `minimumActivityPeriod` has been specified).

The final piece of the puzzle is how we go from the `zeroCrossings` (which produces an event every time the source has gone quiescent) to the output `IObservable<IList<T>>`, which provides all of the events that happened in the most recent burst of activity. Here we're just using Rx's `Buffer` operator, which is designed for exactly this scenario: it slices its input into chunks, producing an event for each chunk, the value of which is an `IList<T>` containing the items for the chunk. `Buffer` can slice things up a few ways, but in this case we're using the form that starts a new slice each time some `IObservable<T>` produces an item. Specifically, we're telling `Buffer` to slice up the `source` by creating a new chunk every time `zeroCrossings` produces a new event.

(One last detail, just in case you saw it and were wondering, is that this method requires an `IScheduler`. This is an Rx abstraction for dealing with timing and concurrency. We need it because we need to be able to generate events after a one second delay, and that sort of time-driven activity requires a scheduler.)

We'll get into all of these operators, and the workings of schedulers, in more detail in later chapters. For now, the key point is that we typically use Rx by creating a combination of LINQ operators that process and combine `IObservable<T>` sources to define the logic that we require.

Notice that nothing in that example actually called the one and only method that `IObservable<T>` defines (`Subscribe`). There will always be something somewhere that ultimately consumes the events, but most of the work of using Rx tends to entail declaratively defining the `IObservable<T>`s we need.

Now that you've seen an example of what Rx programming looks like, we can address some obvious questions about why Rx exists at all.

What was wrong with .NET Events?

.NET has had built-in support for events from the very first version that shipped over two decades ago—events are part of .NET's type system. The C# language has intrinsic support for this in the form of the `event` keyword, along with specialized syntax for subscribing to events. So why, when Rx turned up some 10 years later, did it feel the need to invent its own representation for streams of events? What was wrong with the `event` keyword?

The basic problem with .NET events is that they get special handling from the .NET type system. Ironically, this makes them less flexible than if there had been no built-in support for the idea of events.

Without .NET events, we would have needed some sort of object-based representation of events, at which point you can do all the same things with events that you can do with any other objects: you could store them in fields, pass them as arguments to methods, define methods on them and so on.

To be fair to .NET version 1, it wasn't really possible to define a good object-based representation of events without generics, and .NET didn't get those until version 2 (three and a half years after .NET 1.0 shipped). Different event sources need to be able to report different data, and .NET events provided a way to parameterize events by type. But once generics came along, it became possible to define types such as `IObservable<T>`, and the main advantage that events offered went away. (The other benefit was some language support for implementing and subscribing to events, but in principle that's something that could have been done for Rx if Microsoft had chosen to. It's not a feature that required events to be fundamentally different from other features of the type system.)

Consider the example we've just worked through. It was possible to define our own custom LINQ operator, `Quiescent`, because `IObservable<T>` is just an interface like any other, meaning that we're free to write extension methods for it. You can't write an extension method for an event.

Also, we are able to wrap or adapt `IObservable<T>` sources. `Quiescent` took an `IObservable<T>` as an input, and combined various Rx operators to produce another observable as an output. Its input was a source of events that could be subscribed to, and its output was also a source of events that could be subscribed to. You can't do this with .NET events—you can't write a method that accepts an event as an argument, or that returns an event.

These limitations are sometimes described by saying that .NET events are not *first class citizens*—there are things you can do with values or references in .NET that you can't do with events.

If we represent an event source as a plain old interface, then it *is* a first class citizen: it can use all of the functionality we expect with other objects and values precisely because it's not something special.

What about Streams?

I've described `IObservable<T>` as representing a *stream* of events. This raises an obvious question: .NET already has `System.IO.Stream`, so why not just use that?

The short answer is that streams are weird because they represent an ancient concept in computing dating back long before the first ever Windows operating system shipped, and as such they have quite a lot of historical baggage. This means that even a scenario as simple as "I have some data, and want to make that available immediately to all interested parties" is surprisingly complex to implement though the `Stream` type.

Moreover, `Stream` doesn't provide any way to indicate what type of data will emerge—it only knows about bytes. Since .NET's type system supports generics, it is natural to want the types that represent

event streams to indicate the event type through a type parameter.

So even if you did use `Stream` as part of your implementation, you'd want to introduce some sort of wrapper abstraction. If `IObservable<T>` didn't exist, you'd need to invent it.

It's certainly possible to use IO streams in Rx, but they are not the right primary abstraction.

(If you are unconvinced, see Appendix A: What's Wrong with Classic IO Streams for a far more detailed explanation of exactly why `Stream` is not well suited to this task.)

Now that we've seen why `IObservable<T>` needs to exist, we need to look at its counterpart, `IObserver<T>`.

IObserver

Earlier, I showed the definition of `IObservable<T>`. As you saw, it has just one method, `Subscribe`. And this method takes just one argument, of type `IObserver<T>`. So if you want to observe the events that an `IObservable<T>` has to offer, you must supply it with an `IObserver<T>`. In the examples so far, we've just supplied a simple callback, and Rx has wrapped that in an implementation of `IObserver<T>` for us, but to use Rx effectively you need to understand `IObserver<T>`. It is not a complex interface:

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

As with `IObservable<T>`, you can find the source for `IObserver<T>` in the .NET runtime GitHub repository, because both of these interfaces are built into the runtime libraries.

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 {value}");
    }

    public void OnError(Exception error)
    {
        Console.WriteLine($"Sequence faulted with {error}");
```

```
    }

    public void OnCompleted()
    {
        Console.WriteLine("Sequence terminated");
    }
}
```

In the preceding chapter, I used a `Subscribe` extension method that accepted a delegate which it invoked each time the source produced an item. This method is defined by Rx's `ObservableExtensions` class, which also defines various other extension methods for `IObservable<T>`. It includes overloads of `Subscribe` that enable me to write code that has the same effect as the preceding example, without needing to provide my own implementation of `IObserver<T>`:

```
source.Subscribe(
    value => Console.WriteLine($"Received value {value}"),
    error => Console.WriteLine($"Sequence faulted with {error}"),
    () => Console.WriteLine("Sequence terminated")
);
```

The overloads of `Subscribe` where we don't pass all three methods (e.g., my earlier example just supplied a single callback corresponding to `OnNext`) are equivalent to writing an `IObserver<T>` implementation where one or more of the methods simply has an empty body. Whether we find it more convenient to write our own type that implements `IObserver<T>`, or just supply callbacks for some or all of its `OnNext`, `OnError` and `OnCompleted` method, the basic behaviour is the same: an `IObservable<T>` source reports each event with a call to `OnNext`, and tells us that the events have come to an end either by calling `OnError` or `OnCompleted`.

If you're wondering whether the relationship between `IObservable<T>` and `IObserver<T>` is similar to the relationship between `IEnumerable<T>` and `IEnumerator<T>`, then you're onto something. Both `IEnumerator<T>` and `IObservable<T>` represent *potential* sequences. With both of these interfaces, they will only supply data if we ask them for it. To get values out of an `IEnumerable<T>`, an `IEnumerator<T>` needs to come into existence, and similarly, to get values out of an `IObservable<T>` requires an `IObserver<T>`.

The difference reflects the fundamental *pull vs push* difference between `IEnumerable<T>` and `IObservable<T>`. Whereas with `IEnumerable<T>` we ask the source to create an `IEnumerator<T>` for us which we can then use to retrieve items (which is what a C# `foreach` loop does), with `IObservable<T>`, the source does not *implement* `IObserver<T>`: it expects *us* to supply an `IObserver<T>` and it will then push its values into that observer.

So why does `IObserver<T>` have these three methods? Remember when I said that in an abstract

sense, `IObserver<T>` represents the same thing as `IEnumerable<T>`? I meant it. It might be an abstract sense, but it is precise: `IObservable<T>` and `IObserver<T>` were designed to preserve the exact meaning of `IEnumerable<T>` and `IEnumerator<T>`, changing only the detailed mechanism of consumption.

To see what that means, think about what happens when you iterate over an `IEnumerable<T>` (with, say, a `foreach` loop). With each iteration (and more precisely, on each call to the enumerator's `MoveNext` method) there are three things that could happen:

- `MoveNext` could return `true` to indicate that a value is available in the enumerator's `Current` property
- `MoveNext` could throw an exception
- `MoveNext` could return `false` to indicate that you've reached the end of the collection

These three outcomes correspond precisely to the three method defined by `IObserver<T>`. We could describe these in slightly more abstract terms:

- Here's another item
- It has all gone wrong
- There are no more items

That describes the three things that either can happen next when consuming either an `IEnumerable<T>` or an `IObservable<T>`. The only difference is the means by which consumers discover this. With an `IEnumerable<T>` source, each call to `MoveNext` will tell us which of these three applies. And with an `IObservable<T>` source, it will tell you one of these three things with a call to the corresponding member of your `IObserver<T>` implementation.

The Fundamental Rules of Rx Sequences

Notice that two of the three outcomes in the list above are terminal. If you're iterating through an `IEnumerable<T>` with a `foreach` loop, and it throws an exception, the `foreach` loop will terminate. The C# compiler understands that if `MoveNext` throws, the `IEnumerator<T>` is now done, so it disposes it and then allows the exception to propagate. Likewise, if you get to the end of a sequence, then you're done, and the compiler understands that too: the code it generates for a `foreach` loop detects when `MoveNext` returns false and when that happens it disposes the enumerator and then moves onto the code after the loop.

These rules might seem so obvious that we might never even think about them when iterating over `IEnumerable<T>` sequences. What might be less immediately obvious is that exactly the same rules apply for an `IObservable<T>` sequence. If an observable source either tells an observer that the

sequence has finished, or reports an error, then in either case, that is the last thing the source is allowed to do to the observer.

That means these examples would be breaking the rules:

```
public static void WrongOnError(IObserver<int> obs)
{
    obs.OnNext(1);
    obs.OnError(new ArgumentException("This isn't an argument!"));
    obs.OnNext(2);  // Against the rules! We already reported failure, so
↪   iteration must stop
}

public static void WrongOnCompleted(IObserver<int> obs)
{
    obs.OnNext(1);
    obs.OnCompleted();
    obs.OnNext(2);  // Against the rules! We already said we were done, so
↪   iteration must stop
}

public static void WrongOnErrorAndOnCompleted(IObserver<int> obs)
{
    obs.OnNext(1);
    obs.OnError(new ArgumentException("A connected series of statements was
↪   not supplied"));

    // This next call is against the rule because we reported an error, and
    ↪  you're not
    // allowed to make any further calls after you did that.
    obs.OnCompleted();
}

public static void WrongOnCompletedAndOnError(IObserver<int> obs)
{
    obs.OnNext(1);
    obs.OnCompleted();

    // This next call is against the rule because already said we were
    ↪  done.
    // When you terminate a sequence you have to pick between OnCompleted
    ↪  or OnError
    obs.OnError(new ArgumentException("Definite proposition not
↪   established"));
```

}

These correspond in a pretty straightforward way to things we already know about `IEnumerable<T>`:

- `WrongOnError`: if an enumerator throws from `MoveNext`, it's done and you mustn't call `MoveNext` again, so you won't be getting any more items out of it
- `WrongOnCompleted`: If an enumerator returns `false` from `MoveNext`, it's done and you mustn't call `MoveNext` again, so you won't be getting any more items out of it
- `WrongOnErrorAndOnCompleted` If an enumerator throws from `MoveNext`, that means its done, it's done and you mustn't call `MoveNext` again, meaning it won't have any opportunity to tell that it's done by returning `false` from `MoveNext`
- `WrongOnCompletedAndOnError` If an enumerator returns `false` from `MoveNext`, it's done and you mustn't call `MoveNext` again, meaning it won't have any opportunity to also throw an exception

Because `IObservable<T>` is push-based, the onus for obeying all of these rules fall on the observable source. With `IEnumerable<T>`, which is pull-based, it's up to the code using the `IEnumerator<T>` (e.g. a `foreach` loop) to obey these rules. But they are essentially the same rules.

There's an additional rule for `IObserver<T>`: if you call `OnNext` you must wait for it to return before making any more method calls into the same `IObserver<T>`. That means this code breaks the rules:

```
public static void EverythingEverywhereAllAtOnce(IEnumerable<int> obs)
{
    Random r = new();
    for (int i = 0; i < 10000; ++i)
    {
        int v = r.Next();
        Task.Run(() => obs.OnNext(v));
    }}
```

This calls `obs.OnNext` 10,000 times, but it executes these calls as individual tasks to be run on the thread pool. The thread pool is designed to be able to execute work in parallel, and that's a a problem here because nothing here ensures that one call to `OnNext` completes before the next begins. We've broken the rule that says we must wait for each call to `OnNext` to return before calling either `OnNext`, `OnError`, or `OnComplete` on the same observer. (Note: this assumes that the caller won't subscribe the same observer to multiple different sources. If you do that, you can't assume that all calls to its `OnNext` will obey the rules, because the different sources won't have any way of knowing they're talking to the same observer.)

This rule in which we must wait for `OnNext` to return is tricky and subtle. It's perhaps less obvious than the others, because there's no equivalent rule for `IEnumerable<T>`—the opportunity to break this rule only arises when the source pushes data into the application. You might look at the example above and think "well who would do that?" However, multithreading is just an easy way to show that it is technically possible to break the rule. The harder cases are where single-threaded re-entrancy occurs. Take this code:

```
public class GoUntilStopped
{
    private readonly IObserver<int> observer;
    private bool running;

    public GoUntilStopped(IObserver<int> observer)
    {
        this.observer = observer;
    }

    public void Go()
    {
        this.running = true;
        for (int i = 0; this.running; ++i)
        {
            this.observer.OnNext(i);
        }
    }

    public void Stop()
    {
        this.running = false;
        this.observer.OnCompleted();
    }
}
```

This class takes an `IObserver<int>` as a constructor argument. When you call its Go method, it repeatedly calls the observer's `OnNext` until something calls its `Stop` method.

Can you see the bug?

We can take a look at what happens by supplying an `IObserver<int>` implementation:

```
public class MyObserver : IObserver<int>
{
    private GoUntilStopped? runner;
```

```
    public void Run()
    {
        this.runner = new(this);
        Console.WriteLine("Starting...");
        this.runner.Go();
        Console.WriteLine("Finished");
    }

    public void OnCompleted()
    {
        Console.WriteLine("OnCompleted");
    }

    public void OnError(Exception error) { }

    public void OnNext(int value)
    {
        Console.WriteLine($"OnNext {value}");
        if (value > 3)
        {
            Console.WriteLine($"OnNext calling Stop");
            this.runner?.Stop();
        }
        Console.WriteLine($"OnNext returning");
    }
}
```

Notice that the `OnNext` method looks at its input, and if it's greater than 3, it tells the `GoUntilStopped` object to stop.

Let's look at the output:

```
Starting...
OnNext 0
OnNext returning
OnNext 1
OnNext returning
OnNext 2
OnNext returning
OnNext 3
OnNext returning
OnNext 4
```

```
OnNext calling Stop
OnCompleted
OnNext returning
Finished
```

The problem is right near the end. Specifically, these two lines:

```
OnCompleted
OnNext returning
```

This tells us that the call to our observer's `OnCompleted` happened before a call in progress to `OnNext` returned. It didn't take multiple threads to make this occur. It happened because the code in `OnNext` decides whether it wants to keep receiving events, and when it wants to stop, it immediately calls the `GoUntilStopped` object's `Stop` method. There's nothing wrong with that—observers are allowed to make outbound calls to other objects inside `OnNext`, and it's actually quite common for an observer to inspect an incoming event and decide that it wants to stop.

The problem is in the `GoUntilStopped.Stop` method. This calls `OnCompleted` but it makes no attempt to determine whether a call to `OnNext` is in progress.

This can be a surprisingly tricky problem to solve. Suppose `GoUntilStopped` *did* detect that there was a call in progress to `OnNext`. What then? In the multithreaded case, we could have solved this by using `lock` or some other synchronization primitive to ensure that calls into the observer happened one at at time, but that won't work here: the call to `Stop` has happened on *the same thread* that called `OnNext`. The call stack will look something like this at the moment where `Stop` has been called and it wants to call `OnCompleted`:

```
`GoUntilStopped.Go`
  `MyObserver.OnNext`
    `GoUntilStopped.Stop`
```

Our `GoUntilStopped.Stop` method needs to wait for `OnNext` to return before calling `OnCompleted`. But notice that the `OnNext` method can't return until our `Stop` method returns. We've managed to create a deadlock with single-threaded code!

In this case it's not all that hard to fix: we could modify `Stop` so it just sets the `running` field to `false`, and then move the call to `OnComplete` into the `Go` method, after the `for` loop. But more generally this can be a hard problem to fix, and it's one of the reasons for using the `System.Reactive` library instead of just attempting to implement `IObservable<T>` and `IObserver<T>` directly. Rx has general purpose mechanisms for solving exactly this kind of problem. (We'll see these when we look at

Scheduling.) Moreover, all of the implementations Rx provides take advantage of these mechanisms for you. If you're using Rx by composing its built-in operators in a declarative way, you never have to think about these rules, because all of Rx's operators obey the rules.

So as long as you're using Rx to build the observable sources you need, you get to depend on these rules in your callbacks that receive the events, but it's mostly Rx's problem to keep to the rules. So the main effect of these rules is that it makes life simpler for code that consumes events.

These rules are sometimes expressed as a *grammar*. For example, consider this regular expression:

```
(OnNext)*(OnError|OnComplete)
```

This formally captures the basic idea: there can be any number of calls to `OnNext` (maybe even zero calls), that occur in sequence, followed by either an `OnError` or an `OnComplete`, but not both, and there must be nothing after either of these.

One last point: sequences may be infinite. This is true for `IEnumerable<T>`—it's perfectly possible for an enumerator to return `true` every time `MoveNext` is returned, in which case a `foreach` loop iterating over it will never reach the end. It might choose to stop (with a `break` or `return`), or some exception that did not originate from the enumerator might cause the loop to terminate, but it's absolutely acceptable for an `IEnumerable<T>` to produce items for as long as you keep asking for them. The same is true of a `IObservable<T>`—if you subscribe to an observable source, and by the time your program exits you've not received a call to either `OnComplete` or `OnError`, that's not a bug.

So you might argue that this is a slightly better way to describe the rules formally:

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

More subtly, observable sources are allowed to do nothing at all. In fact there's a built-in implementation to save developers from the effort of writing a source that does nothing: if you call `Observable.Never<int>()` it will return an `IObservable<int>`, and if you subscribe to that, it will never call any methods on your observer. This might not look immediately useful—it is logically equivalent to an `IEnumerable<T>` in which the enumerator's `MoveNext` method never returns. That might not be usefully distinguishable from crashing. It's slightly different with Rx, because when we model this "no items emerge ever" behaviour, we don't need to block a thread forever to do it. We can just decide never to call any methods on the observer. This may seem daft, but as you've seen with the `Quiescent` example, sometimes we create observable sources not because we want the actual items that emerge from it, but because we're interested in the instants when interesting things happen. It can sometimes be useful to be able to model "nothing interesting ever happens" cases.

We're not quite done with the Rx's rules, but the last one applies only when we choose to unsubscribe from a source before it comes to a natural end.

Subscription Lifetime

There's one more aspect of the relationship between observers and observables to understand: the lifetime of a subscription.

You already know from the rules of `IObserver<T>` that a call to either `OnComplete` or `OnError` denotes the end of a sequence. We passed an `IObserver<T>` to `IObservable<T>.Subscribe`, and now the subscription is over. But what if we want to stop the subscription earlier?

I mentioned earlier that the `Subscribe` method returns an `IDisposable`, which enables us to cancel our subscription. Perhaps we only subscribed to a source because our application opened some window showing the status of some process, and we wanted to update the window to reflect that's process's progress. If the user closes that window, we no longer have any use for the notifications. And although we could just ignore all further notifications, that could be a problem if the thing we're monitoring never reaches a natural end—our observer would continue to receive notifications for the lifetime of the application. This is a waste of CPU power (with corresponding implications for battery life and environmental impact) and it can also prevent the garbage collector from reclaiming memory that should have become free.

So we are free to indicate that we no longer wish to receive notifications by calling `Dispose` on the object returned by `Subscribe`. There are, however, a few non-obvious details.

Disposal of Subscriptions is Optional

You are not required to call `Dispose` on the object returned by `Subscribe`. Obviously if you want to remain subscribed to events for the lifetime of your process, this makes sense: you never stop using the object, so of course you don't dispose it. But what might be less obvious is that if you subscribe to an `IObservable<T>` that does come to an end, it automatically tidies up after itself.

`IObservable<T>` implementations are not allowed to assume that you will definitely call `Dispose`, so they are required to perform any necessary cleanup if they stop by calling the observer's `OnCompleted` or `OnError`. This is unusual—in most cases where a .NET API returns a brand new object created on your behalf that implements `IDisposable`, it's an error not to dispose it. But `IDisposable` objects representing Rx subscriptions are an exception to this rule. You only need to dispose them if you want them to stop earlier than they otherwise would.

Cancelling Subscriptions may be Slow or Even Ineffectual

`Dispose` won't necessarily take effect instantly. Obviously it will take some non-zero amount of time in between your code calling into `Dispose`, and the `Dispose` implementation reaching the point where it actually does something. Less obviously, some observable sources may need to do non-trivial work to shut things down.

A source might create a thread to be able to monitor for and report whatever events it represents. (That would happen with the filesystem source shown above when running on Linux on .NET 7, because the `FileSystemWatcher` class itself creates its own thread on Linux.) It might take a while for the thread to detect that it is supposed to shut down.

It is fairly common practice for an `IObservable<T>` to represent some underlying work. For example, Rx can take any factory method that returns a `Task<T>` and wrap it as an `IObservable<T>`. It will invoke the factory once for each call to `Subscribe`, so if there are multiple subscribers to a single `IObservable<T>` of this kind, each one effectively gets its own `Task<T>`. This wrapper is able to supply the factory with a `CancellationToken`, and if an observer unsubscribes by calling `Dispose` before the task naturally runs to completion, it will put that `CancellationToken` into a cancelled state. This might have the effect of bringing the task to a halt, but that will work only if the task happens to be monitoring the `CancellationToken`. Even if it is, it might take some time to bring things to a complete halt. Crucially, the `Dispose` call doesn't wait for that to happen—it will attempt to initiate cancellation but it may return before cancellation is complete.

The Rules of Rx Sequences when Unsubscribing

The fundamental rules of Rx sequences described earlier only considered sources that decided when (or whether) to come to a halt. What if a subscriber unsubscribes early? There is only one rule:

Once the call to `Dispose` has returned, the source will make no further calls to the relevant observer. If you call `Dispose` on the object returned by `Subscribe`, then once that call returns you can be certain that the observer you passed in will receive no further calls to any of its three methods (`OnNext`, `OnError`, or `OnComplete`).

That might seem clear enough, but it leaves a grey area: what happens when you've called `Dispose` but it hasn't returned yet? The rules permit sources to continue to emit events in this case. In fact they couldn't very well require otherwise: it will invariably take some non-zero length of time for the `Dispose` implementation to make enough progress to have any effect, so in a multi-threaded world it it's always going to be possible that an event gets delivered in between the call to `Dispose` starting, and the call having any effect. The only situation in which you could depend on no further events emerging would be if your call to `Dispose` happened inside the `OnNext` handler—in this case the

source will already have noted a call to `OnNext` is in progress so further calls were already blocked before the call to `Dispose` started.

But assuming that your observer wasn't already in the middle of an `OnNext` call, any of the following would be legal:

- stopping calls to `IObserver<T>` almost immediately after `Dispose` begins, even when it takes a relatively long time to bring any relevant underlying processes to a halt, in which case your observer will never receive an `OnCompleted` or `OnError`
- producing notifications that reflect the process of shutting down (including calling `OnError` if an error occurs while trying to bring things to a neat halt, or `OnCompleted` if it halted without problems)
- producing a few more notifications for some time after the call to `Dispose` begins, but cutting them off at some arbitrary point, potentially losing track even of important things like errors that occurred while trying to bring things to a halt

As it happens, Rx has a preference for the first option. If you're using an `IObservable<T>` implemented by the `System.Reactive` library (e.g., one returned by a LINQ operator) it is highly likely to have this characteristic. This is partly to avoid tricky situations in which observers try to do things to their sources inside their notification callbacks—re-entrancy tends to be awkward to deal with, and Rx avoids ever having to deal with this particular form of re-entrancy by ensuring that it has already stopped delivering notifications to the observer before it begins the work of shutting down a subscription.

This sometimes catches people out. If you need to be able to cancel some process that you are observing but you need to be able to observe everything it does up until the point that it stops, then you can't use unsubscription as the shutdown mechanism. As soon as you've called `Dispose`, the `IObservable<T>` that returned that `IDisposable` is no longer under any obligation to tell you anything. This can be frustrating, because the `IDisposable` returned by `Subscribe` can sometimes seem like such a natural and easy way to shut something down. But basic truth is this: once you've initiated unsubscription, you can't rely on getting any further notifications associated with that subscription. You *might* receive some—the source is allowed to carry on supplying items until the call to `Dispose` returns. But you can't rely on it—the source is also allowed to silence itself immediately, and that's what most Rx-implemented sources will do.

One subtle consequence of this is that if an observable source reports an error after a subscriber has unsubscribed, that error might be lost. A source might call `OnError` on its observer, but if that's a wrapper provided by Rx relating to a subscription that has already been disposed, it just ignores the exception. So it's best to think of early unsubscription as inherently messy, a bit like aborting a thread: it can be done but information can be lost, and there are race conditions that will disrupt normal exception handling.

In short, if you unsubscribe, then a source is not obliged to tell you when things stop, and in most cases it definitely won't tell you.

Subscription Lifetime and Composition

We typically combine multiple LINQ operators to express our processing requirements in Rx. What does this mean for subscription lifetime.

For example, consider this:

```
IObservable<int> source = GetSource();
IObservable<int> filtered = source.Where(i => i % 2 == 0);
IDisposable subscription = filtered.Subscribe(
    i => Console.WriteLine(i),
    error => Console.WriteLine($"OnError: {error}"),
    () => Console.WriteLine("OnCompleted"));
```

We're calling `Subscribe` on the observable returned by `Where`. When we do that, it will in turn call `Subscribe` on the the `IObservable<int>` returned by `GetSource` (stored in the `source` variable). So there is in effect a chain of subscriptions here. (We only have access to the `IDisposable` returned by `filtered.Subscribe` but the object that returns will be storing the `IDisposable` that it received when it called `source.Subscribe`.)

If the source comes to an end all by itself (by calling either `OnCompleted` or `OnError`), this cascades through the chain. So `source` will call `OnCompleted` on the `IObserver<int>` that was supplied by the `Where` operator. And that in turn will call `OnCompleted` on the `IObserver<int>` that was passed to `filtered.Subscribe`, and that will have references to the three methods we passed, so it will call our completion handler. So you could look at this by saying that `source` completes, it tells `filtered` that it has completed, which invokes our completion handler. (In reality this is a very slight oversimplification, because `source` doesn't tell `filtered` anything; it's actually talking to the `IObserver<T>` that `filtered` supplied. This distinction matters if you have multiple subscriptions active simultaneously for the same chain of observables. But in this case, the simpler way of describing it is good enough even if it's not absolutely precise.)

In short, completion bubbles up from the source, through all the operators, and arrives at our handler.

What if we unsubscribe early by calling `subscription.Dispose()`? In that case it all happens the other way round—the `subscription` returned by `filtered.Subscribe` is the first to know that we're unsubscribing, but it will then call `Dispose` on the object that was returned when it called `source.Subscribe` for us.

Either way, everything from the source to the observer, including any operators that were sitting in between, gets shut down in either case.

Now that we understand the relationship between an `IObservable<T>` source and the `IObserver<T>` interface that received event notifications, we can look at how we might create an `IObservable<T>` instance to represent events of interest in our application.

Creating Observable Sequences

In the preceding chapter, we saw the two fundamental Rx interfaces, `IObservable<T>` and `IObserver<T>`. We also saw how to receive events by implementing `IObserver<T>`, and also by using implementations supplied by the `System.Reactive` package. In this chapter we'll see how to create `IObservable<T>` sources to represent source events of interest in your application.

We will begin by implementing `IObservable<T>` directly. In practice, it's relatively unusual to do that, so we'll then look at the various ways you can get `System.Reactive` to supply an implementation that does most of the work for you.

A Very Basic `IObservable<T>` Implementation

Here's an implementation of an `IObservable<int>` that produces a sequence of numbers:

```
public class MySequenceOfNumbers : IObservable<int>
{
    public IDisposable Subscribe(IObserver<int> observer)
    {
        observer.OnNext(1);
        observer.OnNext(2);
        observer.OnNext(3);
        observer.OnCompleted();
        return System.Reactive.Disposables.Disposable.Empty; // Handy
        ↪  do-nothing IDisposable
    }
}
```

We can test this by constructing an instance of it, and then subscribing to it:

```
var numbers = new MySequenceOfNumbers();
numbers.Subscribe(
    number => Console.WriteLine($"Received value: {number}"),
    () => Console.WriteLine("Sequence terminated"));
```

This produces the following output:

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

Although `MySequenceOfNumbers` is technically a correct implementation of `IObservable<int>`, it is a little too simple to be useful. For one thing, we typically use Rx when there are events of interest, but this is not really reactive at all—it just produces a fixed set of numbers immediately. Moreover, the implementation is blocking—it doesn't even return from `Subscribe` until after it has finished producing all of its values. This example illustrates the basics of how a source provides events to a subscriber, but if we just want to represent a predetermined sequence of numbers, we might as well use an `IEnumerable<T>` implementation such as `List<T>` or an array.

Representing Filesystem Events in Rx

Let's look at something a little more realistic. This is a wrapper around .NET's `FileSystemWatcher`, presenting filesystem change notifications as an `IObservable<FileSystemEventArgs>`. (Note: this is not necessarily the best design for an Rx `FileSystemWatcher` wrapper. The watcher provides events for several different types of change, and one of them, `Renamed`, provides details as an `RenamedEventArgs`. That derives from `FileSystemEventArgs` so collapsing everything down to a single event stream does work, but this would be inconvenient for applications that wanted access to the details of rename events. A more serious design problem is that this is incapable of reporting more than one event from `FileSystemWatcher.Error`. Such errors might be transient and recoverable, in which case an application might want to continue operating, but since this class chooses to represent everything with a single `IObservable<T>`, it reports errors by invoking the observer's `OnError`, at which point the rules of Rx oblige us to stop. It would be possible to work around this with Rx's `Retry` operator, which can automatically resubscribe after an error, but it might be better to offer a separate `IObservable<ErrorEventArgs>` so that we can report errors in a non-terminating way. However, the additional complication of that won't always be warranted—the simplicity of this design means it will be a good fit for some applications. As is often the way with software design, there isn't a one-size-fits-all approach.)

```
// Represents filesystem changes as an Rx observable sequence.
// NOTE: this is an oversimplified example for illustration purposes.
//       It does not handle multiple subscribers efficiently, it does not
//       use IScheduler, and it stops immediately after the first error.
public class RxFsEvents : IObservable<FileSystemEventArgs>
{
    private readonly string folder;
```

```
    public RxFsEvents(string folder)
    {
        this.folder = folder;
    }

    public IDisposable Subscribe(IObserver<FileSystemEventArgs> observer)
    {
        // Inefficient if we get multiple subscribers.
        FileSystemWatcher watcher = new(this.folder);

        // FileSystemWatcher's documentation says nothing about which
        ↪  thread
        // it raises events on (unless you use its SynchronizationObject,
        // which integrates well with Windows Forms, but is inconvenient
        ↪  for
        // us to use here) nor does it promise to wait until we've
        // finished handling one event before it delivers the next. The
        ↪  Mac,
        // Windows, and Linux implementations are all significantly
        ↪  different,
        // so it would be unwise to rely on anything not guaranteed by the
        // documentation. (As it happens, the Win32 implementation on .NET
        ↪  7
        // does appear to wait until each event handler returns before
        // delivering the next event, so we probably would get way with
        // ignoring this issue. For now. On Windows. And actually the Linux
        // implementation dedicates a single thread to this job, but
        ↪  there's
        // a comment in the source code saying that this should probably
        // change - another reason to rely only on documented behaviour.)
        // So it's our problem to ensure we obey the rules of IObserver<T>.
        // First, we need to make sure that we only make one call at a time
        // into the observer. A more realistic example would use an Rx
        // IScheduler, but since we've not explained what those are yet,
        // we're just going to use lock with this object.
        object sync = new();

        // More subtly, the FileSystemWatcher documentation doesn't make it
        // clear whether we might continue to get a few more change events
        // after it has reported an error. Since there are no promises
        ↪  about
```

```
// threads, it's possible that race conditions exist that would
↪  lead to
// us trying to handle an event from a FileSystemWatcher after it
↪  has
// reported an error. So we need to remember if we've already
↪  called
// OnError to make sure we don't break the IObserver<T> rules in
↪  that
// case.
bool onErrorAlreadyCalled = false;

void SendToObserver(object _, FileSystemEventArgs e)
{
    lock (sync)
    {
        if (!onErrorAlreadyCalled)
        {
            observer.OnNext(e);
        }
    }
}

watcher.Created += SendToObserver;
watcher.Changed += SendToObserver;
watcher.Renamed += SendToObserver;
watcher.Deleted += SendToObserver;

watcher.Error += (_, e) =>
{
    lock (sync)
    {
        // Maybe the FileSystemWatcher can report multiple errors,
        ↪  but
        // we're only allowed to report one to IObservable<T>.
        if (onErrorAlreadyCalled)
        {
            observer.OnError(e.GetException());
            onErrorAlreadyCalled = true;
            watcher.Dispose();
        }
    }
};
```

```
        watcher.EnableRaisingEvents = true;

        return watcher;
    }
}
```

That got more complex fast. This illustrates that `IObservable<T>` implementations are responsible for obeying the `IObserver<T>` rules. This is generally a good thing: it keeps the messy concerns around concurrency contained in a single place. Any `IObserver<FileSystemEventArgs>` that I subscribe to this `RxFsEvents` doesn't have to worry about concurrency, because it can count on the `IObserver<T>` rules, which guarantee that it will only have to handle one thing at a time. If I hadn't been required to enforce these rules in the source, it might have made my `RxFsEvents` class simpler, but all of that complexity of dealing with overlapping events would have spread out into the code that handles the events. Concurrency is hard enough to deal with when its effects are contained. Once it starts to spread across multiple types, it can become almost impossible to reason about. Rx's `IObserver<T>` rules prevent this from happening.

(Note: this is a significant feature of Rx. The rules keep things simple for observers. This becomes increasingly important as the complexity of your event sources or event process grows.)

There are a couple of issues with this code (aside from the API design issues already mentioned). One is that when `IObservable<T>` implementations produce events modelling real-life asynchronous activity (such as filesystem changes) applications will often want some way to take control over which threads notifications arrive on. For example, UI frameworks tend to have thread affinity requirements—you typically need to be on a particular thread to be allowed to update the user interface. Rx provides mechanisms for redirecting notifications onto different schedulers, so we can work around it, but we would normally expect to be able to provide this sort of observer with an `IScheduler`, and for it to deliver notifications through that. We'll discuss schedulers in later chapters.

The other issue is that this does not deal with multiple subscribers efficiently. You're allowed to call `IObservable<T>.Subscribe` multiple times, and if you do that with this code, it will create a new `FileSystemWatcher` each time. That could happen more easily than you might think. Suppose we had an instance of this watcher, and wanted to handle different events in different ways. We might use the `Where` operator to define observable sources that split events up in the way we want:

```
IObservable<FileSystemEventArgs> configChanges =
    fs.Where(e => Path.GetExtension(e.Name) == ".config");
IObservable<FileSystemEventArgs> deletions =
    fs.Where(e => e.ChangeType == WatcherChangeTypes.Deleted);
```

When you call `Subscribe` on the `IObservable<T>` returned by the `Where` operator, it will

call Subscribe on its input. So in this case, if we call Subscribe on both configChanges and deletions, that will result in *two* calls to Subscribe on rs. So if rs is an instance of our RxFsEvents type above, each one will construct its own FileSystemEventWatcher, which is inefficient.

Rx offers a few ways to deal with this. It provides operators designed specifically to take an IObservable<T> that does not tolerate multiple subscribers and wrap it in an adapter that can:

```
IObservable<FileSystemEventArgs> fs =
    new RxFsEvents(@"c:\temp")
    .Publish()
    .RefCount();
```

But this is leaping ahead. (These operators are described in the Publishing Operators chapter.) If you want to build a type that is inherently multi-subscriber-friendly, all you really need to do is keep track of all your subscribers and notify each of them in a loop. Here's a modified version of the filesystem watcher:

```
public class RxFsEventsMultiSubscriber : IObservable<FileSystemEventArgs>
{
    private readonly object sync = new();
    private readonly List<Subscription> subscribers = new();
    private readonly FileSystemWatcher watcher;

    public RxFsEventsMultiSubscriber(string folder)
    {
        this.watcher = new FileSystemWatcher(folder);

        watcher.Created += SendEventToObservers;
        watcher.Changed += SendEventToObservers;
        watcher.Renamed += SendEventToObservers;
        watcher.Deleted += SendEventToObservers;

        watcher.Error += SendErrorToObservers;
    }

    public IDisposable Subscribe(IObserver<FileSystemEventArgs> observer)
    {
        Subscription sub = new(this, observer);
        lock (this.sync)
        {
            this.subscribers.Add(sub);

            if (this.subscribers.Count == 1)
```

```
            {
                // We had no subscribers before, but now we've got one so
                ↪  we need
                // to start up the FileSystemWatcher.
                watcher.EnableRaisingEvents = true;
            }
        }

        return sub;
    }

    private void Unsubscribe(Subscription sub)
    {
        lock (this.sync)
        {
            this.subscribers.Remove(sub);

            if (this.subscribers.Count == 0)
            {
                watcher.EnableRaisingEvents = false;
            }
        }
    }

    void SendEventToObservers(object _, FileSystemEventArgs e)
    {
        lock (this.sync)
        {
            foreach (var subscription in this.subscribers)
            {
                subscription.Observer.OnNext(e);
            }
        }
    }

    void SendErrorToObservers(object _, ErrorEventArgs e)
    {
        Exception x = e.GetException();
        lock (this.sync)
        {
            foreach (var subscription in this.subscribers)
            {
                subscription.Observer.OnError(x);
```

```
            }

            this.subscribers.Clear();
        }
    }

    private class Subscription : IDisposable
    {
        private RxFsEventsMultiSubscriber? parent;

        public Subscription(
            RxFsEventsMultiSubscriber rxFsEventsMultiSubscriber,
            IObserver<FileSystemEventArgs> observer)
        {
            this.parent = rxFsEventsMultiSubscriber;
            this.Observer = observer;
        }

        public IObserver<FileSystemEventArgs> Observer { get; }

        public void Dispose()
        {
            this.parent?.Unsubscribe(this);
            this.parent = null;
        }
    }
}
```

This creates only a single `FileSystemWatcher` instance no matter how many times `Subscribe` is called. Notice that I've had to introduce a nested class to provide the `IDisposable` that `Subscribe` returns. I didn't need that with the very first `IObservable<T>` implementation in this chapter because it had already completed the sequence before returning, so it was able to return the `Disposable.Empty` property conveniently supplied by Rx. (This is handy in cases where you're obliged to supply an `IDisposable`, but you don't actually need to do anything when disposed.) And in my first `FileSystemWatcher` wrapper, `RxFsEvents`, I just returned the `FileSystemWatcher` itself from `Dispose`. (This works because `FileSystemWatcher.Dispose` shuts down the watcher, and each subscriber was given its own `FileSystemWatcher`.) But now that a single `FileSystemWatcher` supports multiple observers, we need to do a little more work when an observer unsubscribes.

One a `Subscription` instance that we returned from `Subscribe` gets disposed, it removes itself from the list of subscribers, ensuring that it won't receive any more notifications. It also sets

the `FileSystemWatcher`'s `EnableRaisingEvents` to false if there are no more subscribers, ensuring that this source does not do unnecessary work if nothing needs notifications right now.

This is looking more realistic than the first example. This is truly a source of events that could occur at any moment (making this exactly the sort of thing Rx does well) and it now handles multiple subscribers intelligently. However, we wouldn't often write things this way. We're doing all the work ourselves here—this code doesn't even require a reference to the `System.Reactive` package because the only Rx types it refers to are `IObservable<T>` and `IObserver<T>`, both of which are built into the .NET runtime libraries. In practice we typically defer to helpers in `System.Reactive` because they can do a lot of work for us.

For example, suppose we only cared about `Changed` events. We could write just this:

```
FileSystemWatcher watcher = new (@"c:\temp");
IObservable<FileSystemEventArgs> changes = Observable
    .FromEventPattern<FileSystemEventArgs>(watcher,
    ↪  nameof(watcher.Changed))
    .Select(ep => ep.EventArgs);
watcher.EnableRaisingEvents = true;
```

Here we're using the `FromEventPattern` helper from the `System.Reactive` library's `Observable` class, which can be used to build an `IObservable<T>` from any .NET event that conforms the the normal pattern (in which event handlers take two arguments: a sender of type `object`, and then some `EventArgs`-derived type containing information about the event). This is not as flexible as the earlier example—it reports only one of the events, and we have to manually start (and, if necessary stop) the `FileSystemWatcher`. But for some applications that will be good enough, and this is a lot less code to write. If we were aiming to write a fully-featured wrapper for `FileSystemWatcher` suitable for many different scenarios, it might be worth writing a specialized `IObservable<T>` implementation as shown earlier. (We could easily extend this last example to watch all of the events—we'd just use the `FromEventPattern` once for each event, and then use `Observable.Merge` to combine the four resulting observables into one. The only real benefit we're getting from a full custom implementation is that we can automatically start and stop the `FileSystemWatcher` depending on whether there are currently any observers.) But if we just need to represent some events as an `IObservable<T>` so that we can work with them in our application, we can just use this simpler approach.

In practice, we almost always get `System.Reactive` to implement `IObservable<T>` for us. Even if we want to take control of certain aspects (such as automatically starting up and shutting down the `FileSystemWatcher` in these examples) we can almost always find a combination of operators that enable this. The following code uses various methods from `System.Reactive` to return an `IObservable<FileSystemEventArgs>` that has all the same functionality as the fully-featured hand-written `RxFsEventsMultiSubscriber` above, but with considerably less code.

```
IObservable<FileSystemEventArgs> ObserveFileSystem(string folder)
{
    return
        // Observable.Defer enables us to avoid doing any work
        // until we have a subscriber.
        Observable.Defer(() =>
            {
                FileSystemWatcher fsw = new(folder);
                fsw.EnableRaisingEvents = true;

                return Observable.Return(fsw);
            })
        // Once the preceding parts emits the FileSystemWatcher
        // (which will happen when someone first subscribes), we
        // want to wrap all the events as IObservable<T>s, for which
        // we'll use a projection. To avoid ending up with an
        // IObservable<IObservable<FileSystemEventArgs>>, we use
        // SelectMany, which effectively flattens it by one level.
        .SelectMany(fsw =>
            Observable.Merge(new[]
                {
                    Observable.FromEventPattern<FileSystemEventHandler,
↪   FileSystemEventArgs>(
                        h => fsw.Created += h, h => fsw.Created -= h),
                    Observable.FromEventPattern<FileSystemEventHandler,
↪   FileSystemEventArgs>(
                        h => fsw.Changed += h, h => fsw.Changed -= h),
                    Observable.FromEventPattern<RenamedEventHandler,
↪   FileSystemEventArgs>(
                        h => fsw.Renamed += h, h => fsw.Renamed -= h),
                    Observable.FromEventPattern<FileSystemEventHandler,
↪   FileSystemEventArgs>(
                        h => fsw.Deleted += h, h => fsw.Deleted -= h)
                })
            // FromEventPattern supplies both the sender and the event
            // args. Extract just the latter.
            .Select(ep => ep.EventArgs)
            // The Finally here ensures the watcher gets shut down once
            // we have no subscribers.
            .Finally(() => fsw.Dispose()))
        // This combination of Publish and RefCount means that multiple
        // subscribers will get to share a single FileSystemWatcher,
        // but that it gets shut down if all subscribers unsubscribe.
```

```
        .Publish()
        .RefCount();
}
```

I've used a lot of methods there, most of which I've not talked about before. For that example to make any sense, I clearly need to start describing the numerous ways in which the `System.Reactive` package can implement `IObservable<T>` for you.

Simple factory methods

Due to the large number of methods available for creating observable sequences, we will break them down into categories. Our first category of methods create `IObservable<T>` sequences produce at most a single result.

Observable.Return

One of the simplest factory methods is `Observable.Return<T>(T value)`, which you've already seen in the `Quiescent` example in the preceding chapter. This method takes a value of type `T` and returns an `IObservable<T>` which will produce this single value and then complete. In a sense, this *wraps* a value in an `IObservable<T>`; it's conceptually similar to writing `new T[] { value }`, in that it's a sequence containing just one element.

```
var singleValue = Observable.Return<string>("Value");
```

I specified the type parameter for clarity, but this is not necessary as the compiler can inferred the type from argument provided:

```
var singleValue = Observable.Return("Value");
```

Observable.Empty

Sometimes it can be useful to have an empty sequence. .NET's `Enumerable.Empty<T>()` does this for `IEnumerable<T>`, and Rx has a direct equivalent in the form of `Observable.Empty<T>()`, which returns an empty `IObservable<T>`. We need to provide the type argument because there's no value from which the compiler can infer the type.

```
var empty = Observable.Empty<string>();
```

In practice, an empty sequence is one that immediately provides an `OnCompleted` notification to any subscriber.

Observable.Never

The `Observable.Never<T>()` method returns a sequence which, like `Empty`, does not produce any values, but unlike `Empty`, it never ends. In practice, that means that it never invokes any method (neither `OnNext`, `OnCompleted`, nor `OnError`) on subscribers. Whereas `Observable.Empty<T>()` completes immediately, `Observable.Never<T>` has infinite duration.

```
var never = Observable.Never<string>();
```

It might not seem obvious why this could be useful. It tends to be used in places where we use observables to represent time-based information. Sometimes we don't actually care what emerges from an observable; we might care only *when* something (anything) happens. For example, in the preceding chapter, the `Quiescent` example used the `Buffer` operator, which works over two observable sequences: the first contains the items of interest, and the second is used purely to determine how to cut the first into chunks. `Buffer` doesn't do anything with the values produced by the second observable: it pays attention only to *when* value emerge, completing the previous chunk each time the second observable produces a value. And if we're representing temporal information it can sometimes be useful to have a way to represent the idea that some event never occurs.

Observable.Throw

`Observable.Throw<T>(Exception)` returns a sequence that immediately reports an error to any subscriber. As with `Empty` and `Never`, we don't supply a value to this method (just an exception) so we need to provide a type parameter so that it knows what T to use in the `IObservable<T>` that it returns. (It will never actually a produce a T, but you can't have an instance of `IObservable<T>` without picking some particular type for T.)

```
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 more powerful than the other creation methods because it can be used to create any kind of sequence. You could implement any of the preceding four methods with `Observable.Create`. The method signature itself may seem more complex than necessary 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)
{...}
```

You provide this with a delegate that will be executed anytime a subscription is made. Your delegate will be passed an `IObserver<T>`. Logically speaking, this represents the observer passed to the `Subscribe` method, although in practice Rx puts a wrapper around that for various reasons. You can call the `OnNext`/`OnError`/`OnCompleted` methods as you need. This is one of the few scenarios where you will work directly with the `IObserver<T>` interface. Here's a simple example that produces two items:

```
private IObservable<string> SomeLetters()
{
    return Observable.Create<string>(
        (IObserver<string> observer) =>
        {
            observer.OnNext("a");
            observer.OnNext("b");
            observer.OnCompleted();

            return Disposable.Empty;
        });
}
```

Your delegate must return either an `IDisposable` or an `Action` to enable unsubscription. When the subscriber disposes their subscription in order to unsubscribe, Rx will invoke `Dispose()` on the `IDisposable` you returned, or in the case where you returned an `Action`, it will invoke that.

This example is reminiscent of the `MySequenceOfNumbers` example from the start of this chapter, in that it immediately produces a few fixed values. The main difference in this case is that Rx adds some wrappers that can handle awkward situations such as re-entrancy. Rx will sometimes automatically defer work to prevent deadlocks, so it's possible that code consuming the `IObservable<string>` returned by this method will see a call to `Subscribe` return before the callback in the code above runs, in which case it would be possible for them to unsubscribe inside their `OnNext` handler. (In the `MySequenceOfNumbers` case that can't actually happen because that doesn't return an `IDisposable` until after it has finished producing all values. This might look like it would do the same, but the `IDisposable` we return isn't necessarily the one the subscriber sees, and in cases

where deferred execution is occurring, the subscriber may well already have an `IDisposable` in hand by the time we call `OnNext`.)

You might be wondering how the `IDisposable` or callback can ever do anything useful, given that it's the return value of the callback, so we can only return it to Rx as the last thing our callback does. Won't we always have finished our work by the time we return, meaning there's nothing to cancel? Not necessarily—we might kick off some work that continues to run after we return. This next example does that, meaning that the unsubscription action it returns is able to do something useful—it sets a cancellation token that is being observed by the loop that generates our observable's output.

```
IObservable<char> KeyPresses() =>
    Observable.Create<char>(observer =>
    {
        CancellationTokenSource cts = new();
        Task.Run(() =>
        {
            while (!cts.IsCancellationRequested)
            {
                ConsoleKeyInfo ki = Console.ReadKey();
                observer.OnNext(ki.KeyChar);
            }
        });

        return () => cts.Cancel();
    });
```

This illustrates how cancellation won't necessarily take effect immediately. The `Console.ReadKey` API does not offer an overload accepting a `CancellationToken`, so this observable won't be able to detect that cancellation is requested until the user next presses a key, causing `ReadKey` to return.

Bearing in mind that cancellation might have been requested while we were waiting for `ReadKey` to return, you might think we should check for that after `ReadKey` returns and before calling `OnNext`. In fact it doesn't matter if we don't, because Rx ensures that you can't break the rule that says an observable source must not call into an observer after a call to `Dispose` on that observer's subscription returns. This is one reason why the `IObserver<T>` it passes to you is a wrapper: if you continue to call methods on that `IObserver<T>` after a request to unsubscribe, Rx just ignores the calls. This means there are two important things to be aware of: if you *do* ignore attempts to unsubscribe and continue to do work to produce items, you are just wasting time because nothing will receive those items; if you call `OnError` it's possible that nothing is listening and that the error will be completely ignored.

There are overloads of `Create` designed to support `async` methods. This methods exploits this to be able to the asynchronous `ReadLineAsync` method to present lines of text from a file as an

observable source.

```
IObservable<string> ReadFileLines(string path) =>
    Observable.Create<string>(async (observer, cancellationToken) =>
    {
        using (var reader = File.OpenText(path))
        {
            while (cancellationToken.IsCancellationRequested)
            {
                string? line = await
                ↪  reader.ReadLineAsync(cancellationToken);
                if (line is null)
                {
                    break;
                }

                observer.OnNext(line);
            }

            observer.OnCompleted();
        }
    });
```

Reading data from a storage device typically doesn't happen instantaneously (unless it happens to be in the filesystem cache already), so this source will provide data as quickly as it can be read from storage.

Notice that because this is an `async` method, it will typically return to its caller before it completes. (The first `await` that actually has to wait returns, and the remainder of the method runs via a callback when the work completes.) That means that subscribers will typically be in possession of the `IDisposable` representing their subscription before this method returns, so we're using a different mechanism to handle unsubscription here. This particular overload of `Create` passes its callback not just an `IObserver<T>` but also a `CancellationToken`, with which it will request cancellation when unsubscription occurs.

File IO can encounter errors. The file we're looking for might not exist, or we might be unable to open it due to security restrictions, or because some other application is using it. The file might be on a remote storage server, and we could lose network connectivity. For this reason, we must expect exceptions from such code. This example has done nothing to detect exceptions, and yet the `IObservable<string>` that this `ReadFileLines` method returns will in fact report any exceptions that occur. This is because the `Create` method will catch any exception that emerges from our callback and report it with `OnError`. (If our code already called `OnComplete` on the observer,

Rx won't call `OnError` because that would violate the rules. Instead it will silently drop the exception, so it's best not to attempt to do any work after you call `OnCompleted`.)

This automatic exception delivery is another example of why the `Create` factory method is the preferred way to implement custom observable sequences. It is almost always a better option than creating custom types that implement the `IObservable<T>` interface. This is not just because it saves you some time. It's also that Rx tackles the intricacies that you may not think of such as thread safety of notifications and disposal of subscriptions.

The `Create` method entails lazy evaluation, which 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. So in the `ReadFileLines` example, it won't attempt to open the file until you subscribe to the `IObservable<string>` that is returned. If you subscribe multiple times, it will execute the callback each time. (So if the file has changed, you can retrieve the latest contents by calling `Subscribe` again.)

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.

You completed that last step before moving onto this paragraph, right? Because you can now compare your versions with these 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.

Observable.Defer

One very useful aspect of `Observable.Create` is that it provides a place to put code that should run only when subscription occurs. Often, libraries will make `IObservable<T>` properties available that won't necessarily be used by all applications, so it can be useful to defer the work involved until you know you will really need it. This deferred initialization is inherent to how `Observable.Create` works, but what if the nature of our source means that `Observable.Create` is not a good fit? How can we perform deferred initialization in that case? Rx providers `Observable.Defer` for this purpose.

I've already used `Defer` once. The `ObserveFileSystem` method returned an `IObservable<FileSystemEver` reporting changes in a folder. It was not a good candidate for `Observable.Create` because it provided all the notifications we wanted as .NET events, so it made sense to use Rx's event adaptation features. But we still wanted to defer the creation of the `FileSystemWatcher` until the moment of subscription, which is why that example used `Observable.Defer`.

`Observable.Defer` takes a callback that returns an `IObservable<T>`, and `Defer` wraps this with an `IObservable<T>` that invokes that callback upon subscription. To show the effect, I'm first going to show an example that does not use `Defer`:

```
static IObservable<int> WithoutDeferal()
```

```
{
    Console.WriteLine("Doing some startup work...");
    return Observable.Range(1, 3);
}

Console.WriteLine("Calling factory method");
IObservable<int> s = WithoutDeferal();

Console.WriteLine("First subscription");
s.Subscribe(Console.WriteLine);

Console.WriteLine("Second subscription");
s.Subscribe(Console.WriteLine);
```

This produces the following output:

```
Calling factory method
Doing some startup work...
First subscription
1
2
3
Second subscription
1
2
3
```

As you can see, the `"Doing some startup work...` message appears when we call the factory method, and before we've subscribed. So if nothing ever subscribed to the `IObservable<int>` that method returns, the work would be done anyway, wasting time and energy. Here's the `Defer` version:

```
static IObservable<int> WithDeferal()
{
    return Observable.Defer(() =>
    {
        Console.WriteLine("Doing some startup work...");
        return Observable.Range(1, 3);
    });
}
```

If we were to use this with similar code to the first example, we'd see this output:

```
Calling factory method
First subscription
Doing some startup work...
1
2
3
Second subscription
Doing some startup work...
1
2
3
```

There are two important differences. First, the `"Doing some startup work..."` message does not appear until we first subscribe, illustrating that `Defer` has done what we wanted. However, notice that the message now appears twice: it will do this work each time we subscribe. If you want this deferred initialization but you'd also like once-only execution, you should look at the operators in the Publishing Operators chapter, which provide various ways to enable multiple subscribers to share a single subscription to an underlying source.

Sequence Generators

The creation methods we've looked at so far are straightforward in that they either produce very simple sequences (such as single-element, or empty sequences), or they rely on our code to tell them exactly what to produce. Now we'll look at some methods that can produce longer sequences.

Observable.Range

`Observable.Range(int, int)` returns an `IObservable<int>` that produces 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

Suppose you wanted to emulate the `Range` factory method using `Observable.Create`. You might try this:

```
// Not the best way to do it!
IObservable<int> Range(int start, int count) =>
    Observable.Create<int>(observer =>
        {
            for (int i = 0; i < count; ++i)
            {
                observer.OnNext(start + i);
            }

            return Disposable.Empty;
        });
```

This will work, but it does not respect request to unsubscribe. That won't cause direct harm, because Rx detects unsubscription, and will simply ignore any further values we produce. However, it's a waste of CPU time (and therefore energy, with consequent battery lifetime and/or environmental impact) to carry on generating numbers after nobody is listening. How bad that is depends on how long a range was requested. But imagine you wanted an infinite sequence? Perhaps it's useful to you to have an `IObservable<BigInteger>` that produces value from the Fibonacci sequence, or prime numbers. How would you write that with `Create`? You'd certainly want some means of handling unsubscription in that case. We need our callback to return if we're to be notified of unsubscription (or we need to supply an `async` method, which doesn't really seem suitable here).

There's a different approach that can work better here: `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)
```

This shows how you could use `Observable.Generate` to construct a `Range` method:

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

```
        start,
        value => value < max,
        value => value + 1,
        value => value);
}
```

The `Generate` method calls us back repeatedly until either our `condition` callback says we're done, or the observer unsubscribes. We can define an infinite sequence simply by never saying we are done:

```
IObservable<BigInteger> Fibonacci()
{
    return Observable.Generate(
        (v1: new BigInteger(1), v2: new BigInteger(1)),
        value => true, // It never ends!
        value => (value.v2, value.v1 + value.v2),
        value => value.v1); ;
}
```

Timed Sequence Generators

Most of the methods we've looked at so far have returned sequences that produce all of their values immediately. (The only exception is where we called `Observable.Create` and produced values when we were ready to.) However, Rx is able to generate sequences on a schedule.

As we'll see, operators that schedule their work do so through an abstraction called a *scheduler*. If you don't specify one, they will pick a default scheduler, but sometimes the timer mechanism is significant. For example, there are timers that integrate with UI frameworks, delivering notifications on the same thread that mouse clicks and other input are delivered on, and we might want Rx's time-based operators to use these. For testing purposes it can be useful to virtualize timings, so we can verify what happens in timing-sensitive code without necessarily waiting for tests to execute in real time.

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 ways of producing timed events.

Observable.Interval

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.

```
IObservable<long> 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, because `Interval` returns an infinite sequence. Rx presumes that you might have considerable patience, because the sequences returned by `Interval` are of type `IObservable<long>` (`long`, not `int`) meaning you won't hit problems if you produce more than a paltry 2.1475 billion event (i.e. more than `int.MaxValue`).

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 specified 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. It also shows how `Observable.Interval` is really just a special case of `Observable.Timer`—`Interval` could be implemented like this:

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

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 starts 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`.

Timed Observable.Generate

There's a more complex overload of `Observable.Generate` 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 as you've already seen, this in turn would enable us to write our own `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 for test purposes.

Adapting Common Type to IObservable<T>

Although we've now seen two very general ways to produce arbitrary sequences—`Create` and `Generate`—what if you already have an existing source of information in some other form that you'd like to make available as an `IObservable<T>`? Rx provides a few adapters for common source types.

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 represents the absence of information, so it's somewhat analogous to `void`, except you can have an instance of the `Unit` type. It's particularly useful in Rx because we often care only about when something has happened, and there might not be any information besides timing. In these cases, we often use an `IObservable<Unit>` so that it's possible to produce definite events even though there's no meaningful data in them. (The name comes from the world of functional programming, where this kind of construct is used a lot.) In this case `Unit` is used to publish an acknowledgement that the `Action` is complete, because an `Action` does not return any information. 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` invokes our callback only upon subscription, so it is an example of a 'lazy' operation. Conversely, `Return` requires us to supply the value up front.

The observable returned by `Start` may seem to have a superficial resemblance to `Task` or `Task<T>` (depending on whether you use the `Action` or `Func<T>` overload). Each represents work that may take some time before eventually completing, perhaps producing a result. However, there's a significant difference: `Start` doesn't begin the work until you subscribe to it. And it will re-execute the callback every time you subscribe to it. So it is more like a factory for a task-like entity.

From events

As we discussed early in the book, .NET has a model for events that is baked into its type system. This predates Rx (not least because Rx wasn't feasible until .NET got generics in .NET 2.0) so it's common for types to support events but not Rx. To be able to integrate with the existing event model, Rx provides methods to take an event and turn it into an observable sequence. I showed this briefly in the file system watcher example earlier, but let's look in a bit more detail. There are several different varieties you can use. This show the most succinct form:

```
FileSystemWatcher watcher = new (@"c:\temp");
IObservable<EventPattern<FileSystemEventArgs>> changeEvents = Observable
    .FromEventPattern<FileSystemEventArgs>(watcher,
    ↪  nameof(watcher.Changed));
```

If you have an object that provides an event you can use this overload of `FromEventPattern`, passing in the object and the name of the event that you'd like to use with Rx. There are a few problems with this.

Firstly, why do I need to pass the event name as a string? Identifying members with strings is an error-prone technique—the compiler won't notice if there's a mismatch between the first and second argument (e.g., if I passed the arguments `(somethingElse, nameof(watcher.Changed))` by mistake). Couldn't I just pass `watcher.Changed` itself? Unfortunately not—this is an example of the issue I mentioned in the first chapter: .NET events are not first class citizens. We can't use them in the way we can use other objects or values. For example, we can't pass an event as an argument to a method. In fact the only thing you can do with a .NET event is attach and remove event handlers. If I

want to get some other method to attach handlers to the event of my choosing (e.g., here I want Rx to handle the events), then the only way to do that is to specify the event's name so that the method (`FromEventPattern`) can then use reflection to attach its own handlers.

This is a problem for some deployment scenarios. It is increasingly common in .NET to do extra work at build time to optimize runtime behaviour, and reliance on reflection can compromise these techniques. For example, instead of relying on Just In Time (JIT) compilation of code, we might use Ahead of time (AoT) mechanisms. .NET's Ready to Run (R2R) system enables you to include pre-compiled code targeting specific CPU types alongside the normal IL, avoiding having to wait for .NET to compile the IL into runnable code. This can have a significant effect on startup times, making it an important technique both in client side applications, where it can fix problems where applications are sluggish when they first start up, and also in server-side applications, especially in cloud environments where code may be moved from one compute node to another fairly frequently, making it important to minimize cold start costs. There are also scenarios where JIT compilation is not even an option, in which case AoT compilation isn't simply an optimization: it's the only means by which code can run at all.

The problem with reflection is that it makes it difficult for the build tools to work out what code will execute at runtime. When they inspect this call to `FromEventPattern` they will just see arguments of type `object` and `string`. It's not self-evident that this is going to result in reflection-driven calls to the `add` and `remove` methods for `FileSystemWatcher.Changed` at runtime. There are attributes that can be used to provide hints, but there are limits to how well these can work. Sometimes the build tools will be unable to determine what code would need to be AoT compiled to enable this method to execute without relying on runtime JIT.

There's another, related problem. The .NET build tools support a feature called 'trimming', in which they remove unused code. The `System.Reactive.dll` file is about 1.3MB in size, but it would be a very unusual application that used every member of every type in that component. Basic use of Rx might need only a few tens of kilobytes. The idea with trimming is to work out which bits are actually in use, and produce a copy of the DLL that contains only that code. This can dramatically reduce the volume of code that needs to be deployed for an executable to run. This can be especially important in client-side Blazor applications, where .NET components end up being downloaded by the browser. Having to download an entire 1.3MB component might make you think twice about using it. But if trimming means that basic usage requires only a few tens of KB, and that the size would increase only if you were making more extensive use of the component, that can make it reasonable to use a component that would, without trimming, have imposed too large a penalty to justify its inclusion. But as with AoT compilation, trimming can only work if the tools can determine which code is in use. If they can't do that, it's not just a case of falling back to a slower path, waiting while the relevant code gets JIT compiler—if code has been trimmed, it will be unavailable at runtime, and your application might crash with a `MissingMethodException`.

So reflection-based APIs can be problematic if you're using any of these techniques. Fortunately, there's an alternative. We can use an overload that takes a couple of delegates, and Rx will invoke these when it wants to add or remove handlers for the event:

```
IObservable<EventPattern<FileSystemEventArgs>> changeEvents = Observable
    .FromEventPattern<FileSystemEventHandler, FileSystemEventArgs>(
        h => watcher.Changed += h,
        h => watcher.Changed -= h);
```

This is code that AoT and trimming tools can understand easily. We've written methods that explicitly add and remove handlers for the `FileSystemWatcher.Changed` event, so AoT tools can pre-compile those two methods, and trimming tools know that they cannot remove the add and remove handlers for those events.

The downside is that this is a pretty cumbersome bit of code to write. If you've not already bought into the idea of using Rx, this might well be enough to make you think "I'll just stick with ordinary .NET events, thanks. But the cumbersome nature is a symptom of what is wrong with .NET events. We wouldn't have had to write anything so ugly if events had been first class citizens in the first place.

Not only has that second-class status meant we couldn't just pass the event itself as an argument, it has also meant that we've had to state type arguments explicitly. The relationship between an event's delegate type (`FileSystemEventHandler` in this example) and its event argument type (`FileSystemEventArgs` here) is, in general, not something that C#'s type inference can determine automatically, which is why we've had to specify both types explicitly. (Events that use the generic `EventHandler<T>` type are more amenable to type inference, and can use a slightly less verbose version of `FromEventPattern`. Unfortunately, relatively few events actually use that. Some events provide information besides the fact that something just happened, and use the base `EventHandler` type, and for those kinds of events, you can in fact omit the type arguments completely, making the code slightly less ugly. You still need to provide the add and remove callbacks though.)

Notice that the return type of `FromEventPattern` in this example is `IObservable<EventPattern<FileSyst` The `EventPattern<T>` type encapsulates the information that the event passes to handlers. Most .NET events follow a common pattern in which handler methods take two arguments: an `object sender`, which just tells you which object raised the event (useful if you attach one event handler to multiple objects) and then a second argument of some type derived from `EventArgs` that provides information about the event. `EventPattern<T>` just packages these two arguments into a single object that offers `Sender` and `EventArgs` properties. In cases where you don't in fact want to attach one handler to multiple sources, you only really need that `EventArgs` property, which is why the earlier `FileSystemWatcher` examples went on to extract that just that, to get a simpler result of type `IObservable<FileSystemEventArgs>`. It did this with the `Select` operator, which we'll get to in more detail later:

```
IObservable<FileSystemEventArgs> changes = changeEvents.Select(ep =>
↪   ep.EventArgs);
```

It is very common to want to expose property changed events as observable sequences. The .NET runtime libraries define a .NET-event-based interface for advertising property changes, `INotifyPropertyChanged`, and some user interface frameworks have more specialized systems for this, such as WPF's `DependencyProperty`. If you are contemplating writing your own wrappers to do this sort of thing, I would strongly suggest looking at the Reactive UI libraries first. It has a set of features for wrapping properties as `IObservable<T>`.

From Task

The `Task` and `Task<T>` types are very widely used in .NET. Widely used .NET languages have built-in support for working with them (e.g., C#'s `async` and `await` keywords). There's some conceptual overlap between tasks an `IObservable<T>`: both represent some sort of work that might take a while to complete. There is a sense in which an `IObservable<T>` is a generalization of a `Task<T>`—both represent potentially long-running work, but an `IObservable<T>` can produce multiple results whereas `Task<T>` can produce just one.

Since `IObservable<T>` is the more general abstraction, we should be able to represent a `Task<T>` as an `IObservable<T>`. And Rx defines various extension methods for `Task` and `Task<T>` to do this. These methods are all called `ToObservable()`, and it offers various overloads offering control of the details where required, and simplicity for the most common scenarios.

Although they are conceptually similar, `Task<T>` does a few things differently in the details. For example, you can retrieve its `Status` property, which might report that it is in a cancelled or faulted state. `IObservable<T>` doesn't provide a way to ask a source for its state; it just tells you things. So `ToObservable` makes some decisions about how to present status in a way that makes makes sense in an Rx world:

- if the task is Cancelled, `IObservable<T>` invoke a subscriber's `OnError` passing a `TaskCanceledException`
- if the task is Faulted then the sequence will error with the task's inner exception
- if the task is not yet in a final state (neither Cancelled, Faulted, or RanToCompletion), the `IObservable<T>` will not produce any notifications until such time as the task does enter one of these final states

It does not matter whether the task is already in a final state at the moment that you call `ToObservable`. If it has finished, `ToObservable` will just return a sequence representing that state. (In fact, it uses either the `Return` or `Throw` creation methods you saw earlier.) If the task has

not yet finished, `ToObservable` will attach a continuation to the task to detect the outcome once it does complete.

Tasks come in two forms: `Task<T>`, which produces a result, and `Task`, which does not. But in Rx, there is only `IObservable<T>`—there isn't a no-result form. We've already seen this problem once before, when the `Observable.Start` method needed to be able to adapt a delegate as an `IObservable<T>` even when the delegate was an `Action` that produced no result. The solution was to return an `IObservable<Unit>`, and that's also exactly what you get when you call `ToObservable` on a `Task`.

The extension method is simple to use:

```
Task<string> t = Task.Run(() =>
{
    Console.WriteLine("Task running...");
    return "Test";
});
IObservable<string> source = t.ToObservable();
source.Subscribe(
    Console.WriteLine,
    () => Console.WriteLine("completed"));
source.Subscribe(
    Console.WriteLine,
    () => Console.WriteLine("completed"));
```

Here's the output.

```
Task running...
Test
completed
Test
completed
```

Notice that even with two subscribers, the task runs only once. That shouldn't be surprising since we only created a single task. If the task has not yet finished, then all subscribers will receive the result when it does. If the task has finished, the `IObservable<T>` effectively becomes a single-value cold observable. But there's a different way to wrap a task: `Observable.FromAsync`.

One Task per subscription There's a different way to get an `IObservable<T>` for a source. I can replace the first statement in the preceding example with this:

```
IObservable<string> source = Observable.FromAsync(() => Task.Run(() =>
{
    Console.WriteLine("Task running...");
    return "Test";
}));
```

I get slightly different output now when subscribing twice:

```
Task running...
Task running...
Test
Test
completed
completed
```

Notice that this executes the task twice, once for each call to `Subscribe`. `FromAsync` can do this because instead of passing a `Task<T>` we pass a callback that returns a `Task<T>`. It calls that when we call `Subscribe`, so each subscriber essentially gets their own task.

If I want to use `async` and `await` to define my task, then I don't need to bother with the `Task.Run` because an `async` lambda creates a `Func<Task<T>>`, which is exactly the type `FromAsync` wants:

```
IObservable<string> source = Observable.FromAsync(async () =>
{
    Console.WriteLine("Task running...");
    await Task.Delay(50);
    return "Test";
});
```

There is a subtle difference with this though. When I used `Task.Run` the lambda ran on a task pool thread from the start. But when I write it this way, the lambda will begin to run on whatever thread calls `Subscribe`. It's only when it hits the first `await` that it returns (and the call to `Subscribe` will then return), with the remainder of the method running on the thread pool.

From IEnumerable<T>

Rx defines another extension method called `ToObservable`, this time for `IEnumerable<T>`. In earlier chapters I described how `IObservable<T>` was designed to represent the same basic abstraction as `IEnumerable<T>`, with the only difference being the mechanism we use to obtain the elements in the sequence: with `IEnumerable<T>`, we write code that *pulls* values out of the

collection (e.g., a `foreach` loop), whereas `IObservable<T>` *pushes* values to us by invoking `OnNext` on our `IObserver<T>`.

We could write code that bridges from *pull* to *push*:

```
// Example code only - do not use!
public static IObservable<T> ToObservableOversimplified<T>(this
↪  IEnumerable<T> source)
{
    return Observable.Create<T>(o =>
    {
        foreach (var item in source)
        {
            o.OnNext(item);
        }

        o.OnComplete();

        // Incorrectly ignoring unsubscription.
        return Disposable.Empty;
    });
}
```

This crude implementation conveys the basic idea, but it is naive. It does not attempt to handle unsubscription, and it's not easy to fix that when using `Observable.Create` for this particular scenario. And as we will see later in the book, it does not have a very nice concurrency model. The implementation that Rx supplies does of course cater for all of these tricky details. That makes it rather more complex, but that's Rx's problem; you can think of it as being logically equivalent to the code shown above, but without the shortcomings.

When transitioning from `IEnumerable<T>` to `IObservable<T>`, you should carefully consider what you are really trying to achieve. Consider that the blocking synchronous (pull) nature of `IEnumerable<T>` does always not mix well with the asynchronous (push) nature of `IObservable<T>`. As soon as something subscribes to an `IObservable<T>` created in this way, it is effectively asking to iterate over the `IEnumerable<T>`, immediately producing all of the values. The call to `Subscribe` might not return until it has reached the end of the `IEnumerable<T>`, making it similar to the very simple example shown at the start of this chapter. (I say "might" because as we'll see when we get to schedulers, the exact behaviour depends on the context.) `ToObservable` can't work magic—something somewhere has to execute what amounts to a `foreach` loop.

So although this can be a convenient way to bring sequences of data into an Rx world, you should carefully test and measure the performance impact.

From APM

Rx provides support for the ancient .NET Asynchronous Programming Model (APM). Back in .NET 1.0, this was the only pattern for representing asynchronous operations. It was superseded in 2010 when .NET 4.0 introduced the Task-based Asynchronous Pattern (TAP). The old APM offers no benefits over the TAP. Moreover, C#'s `async` and `await` keywords (and equivalents in other .NET languages) only support the TAP, meaning that the APM is best avoided. However, the TAP was fairly new back in 2011 when Rx 1.0 was released, so it offered adapters for presenting an APM implementation as an `IObservable<T>`.

Nobody should be using the APM today, but for completeness (and just in case you have to use an ancient library that only offers the APM) I will provide a very brief explanation.

The result of the call to `Observable.FromAsyncPattern` does *not* return an observable sequence. It returns a delegate that returns an observable sequence. (So it is essentially a factory factory) 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. The following example wraps the `Stream` class's `BeginRead`/`EndRead` methods (which are an implementation of the APM).

Note: this is purely to illustrate how to wrap the APM. You would never do this in practice because `Stream` has supported the TAP for years.

```
Stream stream = GetStreamFromSomewhere();
var fileLength = (int) stream.Length;

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

Subjects

So far, this chapter has explored various factory methods that return `IObservable<T>` implementations. There is another way though: `System.Reactive` defines various types

that implement `IObservable<T>` that we can instantiate directly. But how do we determine what values these types produce? We're able to do that because they also implement `IObserver<T>`.

Types that implement both `IObservable<T>` and `IObserver<T>` are called *subjects* in Rx. There's an an `ISubject<T>` to represent this. (This is in the `System.Reactive` NuGet package, unlike `IObservable<T>` and `IObserver<T>`, which are both built into the .NET runtime libraries.) `ISubject<T>` looks like this:

```
public interface ISubject<T> : ISubject<T, T>
{
}
```

So it turns out there's also a two-argument `ISubject<TSource, TResult>` to accommodate the fact that something that is both an observer and an observable might transform the data that flows through it in some way, meaning that the input and output types are not necessarily the same. Here's the two-type-argument definition:

```
public interface ISubject<in TSource, out TResult> : IObserver<TSource>,
↪  IObservable<TResult>
{
}
```

As you can see the `ISubject` interfaces don't define any members of their own. They just inherit from `IObserver<T>` and `IObservable<T>`—these interfaces are nothing more than a direct expression of the fact that a subject is both an observer and an observable.

But what is this for? You can think of the `IObserver<T>` and the `IObservable<T>` as the 'reader' and 'writer' or, 'consumer' and 'publisher' interfaces respectively. A subject, then is both a reader and a writer, a consumer and a publisher. Data flows both into and out of a subject.

Rx offers a few subject implementations that can occasionally be useful in code that wants to make an `IObservable<T>` available. Although `Observable.Create` is usually the preferred way to do this, there's one important case where a subject might make more sense: if you have some code that discovers events of interest (e.g., by using the client API for some messaging technology) and wants to make them available through an `IObservable<T>`, subjects can sometimes provide a more convenient way to to this than wither `Observable.Create` or a custom implementation.

Rx offers a few subject types. We'll start with the most straightforward one to understand.

Subject

The `Subject<T>` type immediately forwards any calls to `IObserver<T>` methods on to all of the observers currently subscribed to it. This example shows its basic operation:

```
Subject<int> s = new();
s.Subscribe(x => Console.WriteLine($"Sub1: {x}"));
s.Subscribe(x => Console.WriteLine($"Sub2: {x}"));

s.OnNext(1);
s.OnNext(2);
s.OnNext(3);
```

I've created a `Subject<int>`. I've subscribed to it twice, and then called its `OnNext` method repeatedly. This produces the following output, illustrating that the `Subject<int>` forwards each `OnNext` call onto both subscribers:

```
Sub1: 1
Sub2: 1
Sub1: 2
Sub2: 2
Sub1: 3
Sub2: 3
```

We could use this as a way to bridge between some API from which we receive data into the world of Rx. You could imagine writing something of this kind:

```
public class MessageQueueToRx : IDisposable
{
    private readonly Subject<string> messages = new();

    public IObservable<string> Messages => messages;

    public void Run()
    {
        while (true)
        {
            // Receive a message from some hypothetical message queuing
            ↪ service
            string message = MqLibrary.ReceiveMessage();
            messages.OnNext(message);
        }
    }

    public void Dispose()
    {
        message.Dispose();
```

```
    }
}
```

It wouldn't be too hard to modify this to use `Observable.Create` instead. But where this approach can become easier is if you need to provide multiple different `IObservable<T>` sources. Imagine we distinguish between different message types based on their content, and publish them through different observables. That's hard to arrange with `Observable.Create` if we still want a single loop pulling messages off the queue.

`Subject<T>` also distributes calls to either `OnCompleted` or `OnError` to all subscribers. Of course, the rules of Rx require that once you have called either of these methods on an `IObserver<T>` (and any `ISubject<T>` is an `IObserver<T>`, so this rule applies to `Subject<T>`) you must not call `OnNext`, `OnError`, or `OnComplete` on that observer ever again. In fact, `Subject<T>` will tolerate calls that break this rule—it just ignores them, so even if your code doesn't quite stick to these rules internally, the `IObservable<T>` you present to the outside world will behave correctly, because Rx enforces this.

`Subject<T>` implements `IDisposable`. Disposing a `Subject<T>` puts it into a state where it will throw an exception if you call any of its methods. The documentation also describes it as unsubscribing all observers, but since a disposed `Subject<T>` isn't capable of producing any further notifications in any case, this doesn't really mean much. (Note that it does *not* call `OnCompleted` on its observers when you `Dispose` it.) The one practical effect is that its internal field that keeps track of observers is reset to a special sentinel value indicating that it has been disposed, meaning that the one externally observable effect of "unsubscribing" the observers is that if, for some reason, your code held onto a reference to a `Subject<T>` after disposing it, that would no longer keep all the subscribers reachable for GC purposes. If a `Subject<T>` remains reachable indefinitely after it is no longer in use, that in itself is effectively a memory leak, but disposal would at least limit the effects: only the `Subject<T>` itself would remain reachable, and not all of its subscribers.

`Subject<T>` is the most straightforward subject, but there are other, more specialized ones.

ReplaySubject

`Subject<T>` does not remember anything: it immediately distributes incoming values to subscribers. If new subscribers come along, they will only see events that occur after they subscribe. `ReplaySubject<T>`, on the other hand, can remember every value it has ever seen. If a new subject comes along, it will receive the complete history of events so far.

This is a variation on the first example in the preceding Subject section. It creates a `ReplaySubject<int>` instead of a `Subject<int>`. And instead of immediately subscribing twice, it creates an initial subscription, and then a second one only after a couple of values have been emitted.

```
ReplaySubject<int> s = new();
s.Subscribe(x => Console.WriteLine($"Sub1: {x}"));

s.OnNext(1);
s.OnNext(2);

s.Subscribe(x => Console.WriteLine($"Sub2: {x}"));

s.OnNext(3);
```

This produces the following output:

```
Sub1: 1
Sub1: 2
Sub2: 1
Sub2: 2
Sub1: 3
Sub2: 3
```

As you'd expect, we initially see output only from `Sub1`. But when we make the second call to subscribe, we can see that `Sub2` also received the first two values. And then when we report the third value, both see it. If this example had used `Subject<int>` instead, we would have seen just this output:

```
Sub1: 1
Sub1: 2
Sub1: 3
Sub2: 3
```

There's an obvious potential problem here: if `ReplaySubject<T>` remembers every value published to it, we mustn't use with endless event sources, because it will eventually cause us to run out of memory.

`ReplaySubject<T>` offers constructors that accept simple cache expiry settings that can limit memory consumption. One option is to specify the maximum number of item to remember. This next example creates a `ReplaySubject<T>` with a buffer size of 2:

```
ReplaySubject<int> s = new(2);
s.Subscribe(x => Console.WriteLine($"Sub1: {x}"));

s.OnNext(1);
s.OnNext(2);
```

```
s.OnNext(3);

s.Subscribe(x => Console.WriteLine($"Sub2: {x}"));

s.OnNext(4);
```

Since the second subscription only comes along after we've already produced 3 values, it no longer sees all of them—it only receives the last two values published prior to subscription (but the first subscription continues to see everything of course):

```
Sub1: 1
Sub1: 2
Sub1: 3
Sub2: 2
Sub2: 3
Sub1: 4
Sub2: 4
```

Alternatively, you can specify a time-based limit by passing a `TimeSpan` to the`ReplaySubject`' constructor.

BehaviorSubject

Like `ReplaySubject<T>`, `BehaviorSubject<T>` also has a memory, but it remembers exactly one value. However, it's not quite the same as a `ReplaySubject<T>` with a buffer size of 1, because a `ReplaySubject<T>` starts off in a state where it has nothing in its memory. But `BehaviorSubject<T>` always remembers *exactly* one item. How can that work before we've made our first call to `OnNext`? `BehaviorSubject<T>` enforces this by requiring us to supply the initial value when we construct it.

So you can think of `BehaviorSubject<T>` as a subject that *always* has a value available. If you subscribe to a `BehaviorSubject<T>` it will instantly produce a single value. (It may then go on to produce more values, but it always produces one right away.) As it happens, it also makes that value available through a property called `Value`, so you don't need to subscribe an `IObserver<T>` to it just to retrieve the value.

A `BehaviorSubject<T>` could be thought of an as observable property. Like a normal property, it can immediately supply a value whenever you ask it. The difference is that it can then go on to notify you every time its value changes.

This analogy falls down slightly when it comes to completion. If you call `OnCompleted`, it immediately calls `OnCompleted` on all of its observers, and if any new observers subscribe, they will also immediately be completed—it does not first supply the last value. (So this is another way in which it is different from a `ReplaySubject<T>` with a buffer size of 1.)

Similarly, if you call `OnError`, all current observers will receive an `OnError` call, and any subsequent subscribers will also receive nothing but an `OnError` call.

backing fields to properties.

AsyncSubject

`AsyncSubject<T>` provides all observers with the final value it receives. Since it can't know which is the final value until `OnCompleted` is called, it will not invoke any methods on any of its subscribers until either its `OnCompleted` or `OnError` method is called. (If `OnError` is called, it just forwards that to all current and future subscribers.)

If no calls were made to `OnNext` before `OnCompleted` then there was no final value, so it will just complete any observers without providing a value.

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

```
AsyncSubject<string> subject = new();
subject.OnNext("a");
subject.Subscribe(x => Console.WriteLine($"Sub1: {x}"));
subject.OnNext("b");
subject.OnNext("c");
```

In this example we invoke the `OnCompleted` method so there will be a final value ('c') for the subject to produce:

```
AsyncSubject<string> subject = new();

subject.OnNext("a");
subject.Subscribe(x => Console.WriteLine($"Sub1: {x}"));
subject.OnNext("b");
subject.OnNext("c");
subject.OnCompleted();
subject.Subscribe(x => Console.WriteLine($"Sub2: {x}"));
```

This produces the following output:

```
Sub1: c
```

```
Sub2: c
```

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, and are occasionally useful in production scenarios, but they are not recommended for most cases. An explanation is in the Usage Guidelines in the appendix. Instead of using subjects, favour the factory methods shown earlier in this chapter..

Summary

We have looked at the various eager and lazy ways to create a sequence. We have seen how to produce timer based sequences using the various factory methods. And we've also explored ways to transition from other synchronous and asynchronous representations.

As a quick recap:

- Factory Methods
  - Observable.Return
  - Observable.Empty
  - Observable.Never
  - Observable.Throw
  - Observable.Create
- Generative methods
  - Observable.Range
  - Observable.Interval

  - Observable.Timer
  - Observable.Generate

- Adaptation

  - 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 look in more detail at the operators that allow us to describe processing to be applied, to build up more complex observable sequences.

PART 2 - From Events to Insights

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 a fire hose. We need the ability to identify the important data, meaning we need ways to determine what is and is not relevant. We need to take groups of data and process them collectively to discover patterns or other information that might not be apparent from any individual raw input. Users, customers and managers need you do this with more data than ever before, while still delivering higher performance and tighter deadlines.

Rx provides some powerful mechanisms for extracting meaningful insights from raw data streams. This is one of the main reasons for representing information as `IObservable<T>` streams in the first place. The preceding chapter showed how to create an observable sequence, so now we will look at how to exploit the power this has unlocked using the the various Rx methods that can process and transform an observable sequence.

Rx supports most of the standard LINQ operators. It also defines numerous additional operators. These fall broadly into categories, and each of the following chapters tackles one category:

- Filtering
- Transformation
- Aggregation
- Partitioning
- [Combination]

- [Error Handling]
- [Timing]

Notes: * testing/discrimination/scrutinisation/picking * where, ignoreelements (segue by pointing out that Where might filter out everything but will still forward end/error) * OfType * Skip/Take[While/Until] * distinct * First(OrDefault), Last(OrDefault), Single(OrDefault) * ElementAt

- transformation
  - Select
  - SelectMany
  - Cast
  - Materialize/Dematerialize
- aggregation
  - to Boolean
    * Any
    * All
    * Contains
    * SequenceEqual
  - To numeric
    * Count
    * Sum, Average
    * Min, Max
  - Custom
    * Aggregate
    * Scan
- partitioning
  - GroupBy
  - Buffer
  - Window
- combination
  - Concat
  - DefaultIfEmpty
  - Repeat
  - Zip
  - CombineLatest

  - StartWith (prepend)
  - Append?
  - merge
  - join
  - Amb
  - Switch
  - TakeUntil (the flavour that accepts an IObservable as its cut-off)
  - And/Then/When

Does Part 3 start here? The existing book has the rather nondescriptive "Taming the sequence" title, and I'd prefer to come up with something a bit more meaningful. Maybe "Getting Pragmatic"

- Schedulers
- Leaving IObservable
  - Do
  - For
- Timing
  - TimeStamp TimeInterval
  - TakeUntil (time-based)
  - Delay
  - Sample
  - Throttle
  - Timeout
- Error Handling
  - Catch
  - Finally
  - Using
  - Retry
  - OnErrorResumeNext

Check these to see if there's anything in them not now covered earlier: * SequencesOfCoincidence

Filtering

Rx provides us with tools to take potentially vast quantities of events and process these to produce higher level insights. This can often involve a reduction in volume—a small number of events may be

more useful than a large number if the individual events in that lower-volume stream are, on average, more informative. The simplest mechanisms for achieving this involve simply filtering out events we don't want. Rx defines several operators that can do this.

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));
    }
}
```

Where

Applying a filter to a sequence is an extremely common exercise and the most straightforward filter in LINQ is the `Where` operator. As usual with LINQ, Rx provides its operators in the form of extension methods. If you are already familiar with LINQ, the signature of Rx's `Where` method will come as no surprise:

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

Note that the element type is the same for the `source` parameter as it is for the return type. This is because `Where` doesn't modify elements—it can filter some out, but those that it does not remove are passed through unaltered.

This example uses `Where` to filter out all even values produced from a `Range` sequence.

```
IObservable<int> xs = Observable.Range(0, 10); // The numbers 0-9

IObservable<int> oddNumbers = xs.Where(i => i % 2 == 0);

oddNumbers.Dump("Where");
```

Output:

```
Where-->0
```

```
Where-->2
Where-->4
Where-->6
Where-->8
Where 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, although there are some exceptions as we'll see later. 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 example, we could have written the first statement this way, and it would have compiled to effectively identical code:

```
IObservable<int> oddNumbers =
    from i in xs
    where i % 2 == 0
    select i;
```

For the examples in this book however, we will keep with using extension methods, partly because Rx implements some operators for which there is no corresponding query syntax, and partly because the method call approach can sometimes make it easier to see what is happening.

As with most Rx operators, `Where` does not subscribe immediately to its source. (Rx LINQ operators are much like those in LINQ to Objects: the `IEnumerable<T>` version of `Where` returns without attempting to enumerate its source. It's only when something attempts to enumerate the `IEnumerable<T>` that `Where` returns that it will in turn start enumerating the source.) Only when something calls `Susbcribe` on the `IObservable<T>` returned by `Where` will it call `Subscribe` on its source. And it will do so once for each such call to `Subscribe`. More generally, when you chain LINQ operators together, each `Subscribe` call on the resulting `IObservable<T>` results in a cascading series of calls to `Subscribe` all the way down the chain.

A side effect of this cascading `Subscribe` is that `Where` (and most other LINQ operators) is neither inherently *hot* or *cold*: since it just subscribes to its source, then it will be hot if its source is hot, and cold if its source is cold.

The `Where` operator passes on all elements for which its `predicate` callback returns `true`. To be more precise, `Where` will create its own `IObserver<T>` which it passes as the argument to `source.Subscribe`, and this observer invokes the `predicate` for each call to `OnNext`. If that predicate returns `true`, then and only then will the observer created by `Where` call `OnNext` on the observer that was passed to `Where`.

`Where` always passes the final call to either `OnComplete` or `OnError` through. That means that if you were to write this:

```
IObservable<int> dropEverything = xs.Where(_ => false);
```

then although this would filter out all elements (because the predicate ignores its argument and always returns `false`, instructing `Where` to drop everything), this won't filter out error or completion.

In fact if that's what you want—an operator that drops all the elements and just tells you when a source completes or fails—there's a simpler way.

IgnoreElements

The `IgnoreElements` extension method allows you to receive just the `OnCompleted` or `OnError` notifications. It is equivalent to using the `Where` operator with a predicate that always returns false, as this example illustrates:

```
IObservable<int> xs = Observable.Range(1, 3);
IObservable<int> dropEverything = xs.IgnoreElements();

xs.Dump("Unfiltered");
dropEverything.Dump("IgnoreElements");
```

As the output shows, the `xs` source produces the numbers 1 to 3 then completes, but if we run that through `IgnoreElements`, all we see is the `OnCompleted`.

```
Unfiltered-->1
Unfiltered-->2
Unfiltered-->3
Unfiltered completed
IgnoreElements completed
```

OfType

Some observable sequence produce items of various types. For example, consider an application that wants to keep track of ships as they move. This is possible with an AIS receiver. AIS is the Automatic Identification System, which most ocean-going ships use to report their location, heading, speed, and other information. There are numerous kinds of AIS message—some report a ship's location and speed, but its name is reported in a different kind of message. (This is because most ships move more often than they change their names, so they broadcast these two types of information at quite different frequencies.)

Imagine how this might look in Rx. Actually you don't have to imagine it. The open source Ais.Net project includes a `ReceiverHost` class that makes AIS messages available through Rx. The `ReceiverHost` defines a `Messages` property of type `IObservable<IAisMessage>`. Since AIS defines numerous message types, this observable source can produce many different kinds of objects. Everything it emits will implement the `IAisMessage` interface, which reports the ship's unique identifier, but not much else. But the `Ais.Net.Models` library defines numerous other interfaces, including `IVesselNavigation`, which reports location, speed, and heading, and `IVesselName`, which tells you the vessel's name.

Suppose you are interested only in the locations of vessels in the water, and you don't care about the vessels' names. You will want to see all messages that implement the `IVesselNavigation` interface, and to ignore all those that don't. You could try to achieve this with the `Where` operator:

```
// Won't compile!
IObservable<IVesselNavigation> vesselMovements = receiverHost.Messages
    .Where(m => m is IVesselNavigation);
```

However, that won't compile. You will get this error:

```
Cannot implicitly convert type 'System.IObservable<Ais.Net.Models.Abstractions.I
```

Remember that the return type of `Where` is always the same as its input. Since `receiverHost.Messages` is of type `IObservable<IAisMessage>`, that's is also the type that `Where` will return. It so happens that our predicate ensures that only those messages that implement `IVesselNavigation` make it through, but there's no way for the C# compiler to understand the relationship between the predicate and the output. (For all it knows, `Where` might do the exact opposite, including only those elements for which the predicate returns `false`. In fact the compiler can't guess anything about how `Where` might use its predicate.)

Fortunately, Rx provides an operator specialized for this case. `OfType` filters items down to just those that are of a particular type—items must be either the exact type specified, or if it's an interface they must implement it, or otherwise they must inherit from it. This enables us to fix the last example:

```
IObservable<IVesselNavigation> vesselMovements = receiverHost.Messages
    .OfType<IVesselNavigation>();
```

Positional Filtering

Sometimes, we don't care about what an element is, so much as where it is in the sequence.

FirstAsync and FirstOrDefaultAsync

LINQ providers typically implement a `First` operator that provides the first element of a sequence. Rx is no exception, but the nature of Rx means we typically need this to work slightly differently. With providers for data at rest (such as LINQ to Objects or Entity Framework Core) the source elements already exist, so retrieving the first item is just a matter of reading it. But with Rx, sources produce data when they choose, so there's no way of knowing when the first item will become available.

So with Rx, we typically use `FirstAsync`. This returns an `IObservable<T>` that will produce the first value that emerges from the source sequence and will then complete. (There is also a `First` method, but it can be problematic. See the **Blocking Versions of First/Last/Single[OrDefault]** section later for details.)

For example, this code uses the AIS.NET source introduced earlier to report the first time a particular boat (the aptly named HMS Example, as it happens) reports that it is moving:

```
uint exampleMmsi = 235009890;
IObservable<IVesselNavigation> moving = receiverHost.Messages
    .Where(v => v.Mmsi == exampleMmsi)
    .OfType<IVesselNavigation>()
    .Where(vn => vn.SpeedOverGround > 1f)
    .FirstAsync();
```

As well as using `FirstAsync`, this also uses a couple of the other filter elements already described. It starts with a `Where` step that filters message down to those from the one boat we happen to be interested in. (Specifically, we filter based on that boat's Maritime Mobile Service Identity, or MMSI.) Then we use `OfType` so that we are looking only at those messages that report movement. Then we use another `Where` clause so that we can ignore messages indicating that the boat is not actually moving, finally, we use `FirstAsync` so that we get only the first message indicating movement. As soon as the boat moves, this `moving` source will emit a single `IVesselNavigation` event and will then immediately complete.

We can simplify that query slightly, because `FirstAsync` optionally takes a predicate. This enables us to collapse the final `Where` and `FirstAsync` into a single operator:

```
IObservable<IVesselNavigation> moving = receiverHost.Messages
    .Where(v => v.Mmsi == exampleMmsi)
    .OfType<IVesselNavigation>()
    .FirstAsync(vn => vn.SpeedOverGround > 1f);
```

What if the input to `FirstAsync` is empty? If its completes without ever producing an item, `FirstAsync` invokes its subscriber's `OnError` with an `InvalidOperationException` with an error message reporting that the sequence contains no elements. The same is true if we're using

the form that takes a predicate (as in this second example), and no elements matching the predicate emerged. This is consistent with the LINQ to Objects `First` operator. (Note that we wouldn't expect this to happen with the examples just shown, because the source will continue to report AIS messages for as long as the application is running, meaning there's no reason for it ever to complete.)

Sometimes, we might want to tolerate this kind of absence of events. Most LINQ providers offer not just `First` but `FirstOrDefault`. We can use this by modify the preceding example. This uses the `TakeUntil` operator to introduce a cut-off time: this example is prepared to wait for 5 minutes, but gives up after that. (So although the AIS receiver can produce messages endlessly, this example has decided it won't wait forever.) And since that means we might complete without ever seeing the boat move, we've replaced `FirstAsync` with `FirstOrDefaultAsync`:

```
IObservable<IVesselNavigation?> moving = receiverHost.Messages
    .Where(v => v.Mmsi == exampleMmsi)
    .OfType<IVesselNavigation>()
    .TakeUntil(DateTimeOffset.Now.AddMinutes(5))
    .FirstOrDefaultAsync(vn => vn.SpeedOverGround > 1f);
```

If, after 5 minutes, we've not seen a message from the boat indicating that it's moving at 1 knot or faster, `TakeUntil` will unsubscribe from its upstream source and will call `OnCompleted` on the observer supplied by `FirstOrDefaultAsync`. Whereas `FirstAsync` would treat this as an error, `FirstOrDefaultAsync` will produce the default value for its element type (`IVesselNavigation` in this case; the default value for an interface type is `null`), pass that to its subscriber's `OnNext` and then call `OnCompleted`.

In short, this `moving` observable will always produce exactly one item. Either it will produce an `IVesselNavigation` indicating that the boat has moved, or it will produce `null` to indicate that this didn't happen in the 5 minutes that this code has allowed.

This production of a `null` might be an OK way to indicate that something didn't happen, but there's something slightly clunky about it: anything consuming this `moving` source now has to work out whether a notification signifies the event of interest, or the absence of any such event. If that happens to be convenient for your code, then great, but Rx provides a more direct way to represent the absence of an event: an empty sequence.

You could imagine a *first or empty* operator that worked this way. This wouldn't make sense for LINQ providers that return an actual value. For example, as LINQ to Objects' `First` returns `T`, not `IEnumerable<T>`, so there's no way for it to return an empty sequence. But because Rx's offers `First`-like operators that return `IObservable<T>`, it would be technically possible to have an operator that returns either the first item or no items at all. There is no such operator built into Rx, but we can get exactly the same effect by using a more generalised operator, `Take`.

Take

`Take` is a standard LINQ operator that takes the first few items from a sequence and then discards the rest.

In a sense, `Take` is a generalization of `First`: `Take(1)` returns only the first item, so you could think of LINQ's `First` as being a special case of `Take`. That's not strictly correct because these operators respond differently to missing elements: as we've just seen, `First` (and Rx's `FirstAsync`) insists on receiving at least one element, producing an `InvalidOperationException` if you supply it with an empty sequence. Even the more existentially relaxed `FirstOrDefault` still insists on producing something. `Take` works slightly differently.

If the input to `Take` completes before producing as many elements as have been specified, `Take` does not complain—it just forwards whatever the source has provided. If the source did nothing other than call `OnCompleted`, then `Take` just calls `OnCompleted` on its observer. If we used `Take(5)`, but the source produced three items and then completed, `Take(5)` will forward those three items to its subscriber, and will then complete. This means we could use `Take` to implement the hypothetical `FirstOrEmpty` discussed in the preceding section:

```
public static IObservable<T> FirstOrEmpty<T>(this IObservable<T> src) =>
↪  src.Take(1);
```

Now would be a good time to remind you that most Rx operators (and all the ones in this chapter) are not intrinsically either hot or cold. They defer to their source. Given some hot `source`, `source.Take(1)` is also hot. The AIS.NET `receiverHost.Messages` source I've been using in these examples is hot (because it reports live message broadcasts from ships), so observable sequences derived from it are also hot. Why is now a good time to discuss this? Because it enables me to make the following absolutely dreadful pun:

```
IObservable<IAisMessage> hotTake = receiverHost.Messages.Take(1);
```

Thank you. I'm here all week.

The `FirstAsync` and `Take` operators work from the start of the sequence. What if we're interested only in the tail end?

LastAsync and LastOrDefaultAsync

LINQ providers typically provide `Last` and `LastOrDefault`. These do almost exactly the same thing as `First` or `FirstOrDefault` except, as the name suggests, they return the final element instead of the first one. As with `First`, the nature of Rx means that unlike with LINQ providers working with data at rest, the final element might not be just sitting there waiting to be fetched. So just as Rx

offers `FirstAsync` and `FirstOrDefault`, it offers `LastAsync` and `LastOrDefaultAsync`. (It does also offer `Last`, but again, as the Blocking Versions of First/Last/Single[OrDefault] section discusses, this can be problematic.)

There is also `PublishLast`. This has similar semantics to `LastAsync` but it handles multiple subscriptions differently. Each time you subscribe to the `IObservable<T>` that `LastAsync` returns, it will subscribe to the underlying source, but `PublishLast` makes only a single `Subscribe` call to the underlying source (which happens when you call `Connect`), and then all subscribers receive the same result. This is one of a family of operators based on the `Multicast` operator described in more detail in later chapters.

The distinction between these two operators is the same as with `FirstAsync` and `FirstOrDefaultAsync`. If the source completes having produced nothing, `LastAsync` reports an error, whereas `LastOrDefaultAsync` emits the default value for its element type and then completes.

Reporting the final element of a sequence entails a challenge that `First` does not face. It's very easy to know when you've received the first element from a source: if the source produces an element, and it hasn't previously produced an element, then that's the first element right there. This means that operators such as `FirstAsync` can report the first element immediately. But `LastAsync` and `LastOrDefaultAsync` don't have that luxury.

If you receive an element from a source, how do you know that it is the last element? In general, you can't know this at the instant that you receive it. You will only know that you have received the last element when the source goes on to invoke your `OnCompleted` method. This won't necessarily happen immediately. An earlier example used `TakeUntil(DateTimeOffset.Now.AddMinutes(5))` to bring a sequence to an end after 5 minutes, and if you do that, it's entirely possible that a significant amount of time might elapse between the final element being emitted, and `TakeUntil` shutting things down. In the AIS scenario, boats might only emit messages once every few minutes, so it's quite plausible that we could end up with `TakeUntil` forwarding a message, and then discovering a few minutes later that the cutoff time has been reached without any further messages coming in. Several minutes could have elapsed between the final `OnNext` and the `OnComplete`.

Because of this. `LastAsync` and `LastOrDefaultAsync` emit nothing at all until their source completes.

This has an important consequence: there might be a significant delay between `LastAsync` receiving the final element from the source, and it forwarding that element to its subscriber.

TakeLast

Earlier we saw that Rx implements the standard `Take` operator, which forwards up to a specified number of elements from the start of a sequence and then stops. `TakeLast` forwards the elements

at the end of a sequence. For example, `TakeLast(3)` asks for the final 3 elements of the source sequence. As with `Take`, `TakeLast` is tolerant of sources that produce too few items—if a source produces fewer than 3 items, `TaskLast(3)` will just forward the entire sequence.

`TaskLast` faces the same challenge as `Last`: it doesn't know when it is near the end of the sequence. It therefore has to hold onto copies of the most recently seen values. Note that it will need enough memory to hold onto however many values you've specified. If you write `TakeLast(1_000_000)`, it will need to allocate a buffer large enough to store 1,000,000 values. It doesn't know if the first element it receives will be one of the final million. It can't know that until either the source completes, or the source has emitted more than 1,000,000 items. When the source finally does complete, `TakeLast` will now know what the final million elements were and will need to pass all of them to its subscriber's `OnNext` method one after another.

SingleAsync and SingleOrDefaultAsync

LINQ operators typically provide a `Single` operator, for use when a source should provide exactly one item, and it would be an error for it to contain more, or for it to be empty. The same Rx considerations apply here as for `First` and `Last`, so you will probably be unsurprised to learn that Rx offers a `SingleAsync` method that returns an `IObservable<T>` that will either call its observer's `OnNext` exactly once, or will call its `OnError` to indicate either that the source reported an error, or that the source did not produce exactly one item.

With `SingleAsync`, you will get an error if the source is empty, just like with `FirstAsync` and `LastAsync`, but you will also get an error if the source contains multiple items. There is a `SingleOrDefault` which, like its first/last counterparts, tolerates an empty input sequence, generating a single element with the element type's default value in that case.

`Single` and `SingleAsync` share with `Last` and `LastAsync` the characteristic that they know when they receive an item from the source whether it should be the output. That may seem odd: since `Single` requires the source stream to provide just one item, surely it must know that the item it will deliver to its subscriber will be the first item it receives. This is true, but the thing it doesn't yet know when it receives the first item is whether the source is going to produce a second one. It can't forward the first item unless and until the source completes. We could say that `SingleAsync`'s job is to first verify that the source contains exactly one item, and then to forward that item if it does, but to report an error if it does not. In the error case, `SingleAsync` will know it has gone wrong if it ever receives a second item, so it can immediately call `OnError` on its subscriber at that point. But in the success scenario, it can't know that all is well until the source confirms that nothing more is coming by completing. Only then will `SingleAsync` emit the result.

Skip and SkipLast

What if we want the exact opposite of the `Take` or `TakeLast` operators? Instead of taking the first 5 items from a source, maybe I want to discard the first 5 items instead? Perhaps I have some `IObservable<float>` taking readings from a sensor, and I have discovered that the sensor produces garbage values for its first few readings, so I'd like to ignore those, and only start listening once it has settled down. I can achieve this with `Skip(5)`.

`SkipLast` does the same thing at the end of the sequence—it omits the specified number of elements at the tail end. As with some of the other operators we've just been looking at, this has to deal with the problem that it can't tell when it's near the end of the sequence. It only gets to discover which were the last (say) 4 elements after the source has emitted all of them, followed by an `OnComplete`. So `SkipLast` will introduce a delay. If you use `SkipLast(4)`, it won't forward the first element that the source produces until the source produces a 5th element. So it doesn't need to wait for `OnCompleted` or `OnError` before it can start doing things, it just has to wait until its certain that an element is not one of the ones we want to discard.

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.

Blocking Versions of First/Last/Single[OrDefault]

Several of the operators described in the preceding sections end in the name `Async`. This is a little strange because normally, .NET methods that end in `Async` return a `Task` or `Task<T>`, and yet these all return an `IObservable<T>`. Also, as already discussed, each of these methods corresponds to a standard LINQ operator which does not generally end in `Async`. (And to further add to the confusion, some LINQ providers such as Entity Framework Core do include `Async` versions of some of these operators, but they are different. Unlike Rx, these do in fact return a `Task<T>`, so they still produce a single value, and not an `IQueryable<T>` or `IEnumerable<T>`.) This naming arises from an unfortunate choice early in Rx's design.

If Rx were being designed from scratch today, the relevant operators in the preceding section would just have the normal names: `First`, and `FirstOrDefault`, and so on. The reason they all end with `Async` is that these were added in Rx 2.0, and Rx 1.0 had already defined operators with those names. This example shows the `First` operator:

```
int v = Observable.Range(1, 10).First();
Console.WriteLine(v);
```

This prints out the value 1, which is the first item returned by Range here. But look at the type of that variable v. It's not an IObservable<int>, it's just an int. What would happen if we used this on an Rx operator that didn't immediately produce values upon subscription? Here's one example:

```
long v = Observable.Timer(TimeSpan.FromSeconds(2)).First();
Console.WriteLine(v);
```

If you run this, you'll find that the call to First doesn't return until a value is produced. It is a *blocking* operator. We typically avoid blocking operators in Rx, because it's easy to create deadlocks with them. The whole point of Rx is that we can create code that reacts to events, so to just sit and wait until a specific observable source produces a value is not really in the spirit of things. If you find yourself wanting to do that, there are often better ways to achieve the results you're looking for. (Or perhaps Rx isn't good model for whatever you're doing.)

If you really do need to wait for a value like this, it might be better to use the Async forms in conjunction with Rx's integrated support for C#'s async/await:

```
long v = await Observable.Timer(TimeSpan.FromSeconds(2)).FirstAsync();
Console.WriteLine(v);
```

This logically has the same effect, but because we're using await, this won't block the calling thread while it waits for the observable source to produce a value. This might reduce the chances of deadlock.

The fact that we're able to use await makes some sense of the fact that these methods end with Async, but you might be wondering what's going on here. We've seen that these methods all return IObservable<T>, not Task<T>, so how are we able to use await? There's a full explanation in the Leaving Rx's World chapter, but the short answer is that Rx provides extension methods that enable this to work. When you await an observable sequence, the await will complete once the source completes, and it will return the final value that emerges from the source. This works well for operators such as FirstAsync and LastAsync that produce exactly one item.

Note that there are occasionally situations in which values are available immediately. For example, the BehaviourSubject<T> section in chapter 3, showed that the defining feature of BehaviourSubject<T> is that it always has a current value. That means that Rx's First method won't actually block—it will subscribe to the BehaviourSubject<T>, and BehaviourSubject<T>.Subscribe calls OnNext on its subscriber's observable before returning. That enables First to return a value immediately without blocking. (Of course, if you use the overload of First that accepts a predicate, and if the BehaviourSubject<T>'s value doesn't satisfy the predicate, First will then block.)

ElementAt

There is yet another standard LINQ operator for selecting one particular element from the source: `ElementAt`. You provide this with a number indicating the position in the sequence of the element you require. In data-at-rest LINQ providers, this is logically equivalent to accessing an array element by index. Rx implements this operator, but whereas most LINQ providers' `ElementAt<T>` implementation returns a `T`, Rx's returns an `IObservable<T>`. Unlike with `First`, `Last`, and `Single`, Rx does not provide a blocking form of `ElementAt<T>`. But since you can await any `IObservable<T>`, you can always do this:

```
IAisMessage fourth = await recieverHost.Message.ElementAt(4);
```

If your source sequence only produces five values and we ask for `ElementAt(5)`, the sequence that `ElementAt` returns will report an `ArgumentOutOfRangeException` error to its subscriber when the source completes. There are three ways we can deal with this:

- 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.

Temporal Filtering

The `Take` and `TakeLast` operators let us filter out everything except elements either at the very start or very end (and `Skip` and `SkipLast` let see everything but those), but these all require us to know the exact number of elements. What if we want to specify the cut-off not in terms of an element count, but in terms of a particular instant in time?

In fact you've already seen one example: earlier I used `TakeUntil` to convert an endless `IObservable<T>` into one that would complete after five minutes. This is one of a family of operators.

SkipWhile and TakeWhile

In the `Skip` and `SkipLast` section, I described a sensor that produces garbage values for its first few readings. This is quite common. For example, gas monitoring sensors often need to get some component up to a correct operator temperature before they can produce accurate readings. In the

example in that section, I used `Skip(5)` to ignore the first few readings, but that is a crude solution. How do we know that 5 is enough? Or might it be ready sooner, in which case 5 is too few.

What we really want to do is discard readings until we know the readings will be valid. And that's exactly the kind of scenario that `SkipWhile` can be useful for. Suppose we have a gas sensor that reports concentrations of some particular gas, but which also reports the temperature of the sensor plate that is performing the detection. Instead of hoping that 5 readings is a sensible number to skip, we could express the actual requirement:

```
const int MinimumSensorTemperature = 74;
IObservable<SensorReading> readings = sensor.RawReadings
    .SkipUntil(r => r.SensorTemperature >= MinimumSensorTemperature);
```

This directly expresses the logic we require: this will discard readings until the device is up to its minimum operating temperature.

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
```

SkipUntil and TakeUntil

In addition to `SkipWhile` and `TakeWhile`, Rx defines `SkipUntil` and `TakeUntil`. These may sound like nothing more than an alternate expression of the same idea: you might expect `SkipUntil` to do almost exactly the same thing as `SkipWhile`, with the only difference being that `SkipWhile` runs for as long as its predicate returns `true`, whereas `SkipUntil` runs for as long as its predicate returns `false`. And there is an overload of `SkipUntil` that does exactly that (and a corresponding one for `TakeUntil`). If that's all these were they wouldn't be interesting. However, there are overloads of `SkipUtil` and `TakeUntil` that enable us to do things we can't do with `SkipWhile` and `TakeWhile`.

You've already seen one example. The `FirstAsync` and `FirstOrDefaultAsync` included an example that used an overload of `TakeUntil` that accepted a `DateTimeOffset`. This wraps any `IObservable<T>`, returning an `IObservable<T>` that will forward everything from the source until the specified time, at which point it will immediately complete (and will unsubscribe from the underlying source).

We couldn't have achieved this with `TakeWhile`, because that consults its predicate only when the source produces an item. If we want the source to complete at a specific time, the only way we could do that with `TakeWhile` is if its source happens to produce an item at the exact moment we wanted to finish. `TakeWhile` will only ever complete as a result of its source producing an item. `TakeUntil` can complete asynchronously—if we specified a time 5 minutes into the future, it doesn't matter if the source is completely idle when that time arrives. `TakeUntil` will complete anyway. (It relies on Schedulers to be able to do this.)

We don't have to use a time, `TakeUntil` offers an overload that accept a second `IObservable<T>`. This enables us to tell it to stop when something interesting happens, without needing to know in advance exactly when that will occur. This overload of `TakeUntil` forwards items from the source until that second `IObservable<T>` produces a value. `SkipUntil` offers a similar overload in which the second `IObservable<T>` determines when it should start forwarding items from the source.

Note: these overloads require the second observable to produce a value in order to trigger the start or end. If that second observable completes without producing a single notification, then it has no effect—`TakeUntil` will continue to take items indefinitely; `SkipUntil` will never produce anything. In other words, these operators would treat `Observable.Empty<T>()` as being effectively equivalent to `Observable.Never<T>()`.

Distinct and DistinctUntilChanged

`Distinct` is yet another standard LINQ operator. It removes duplicates from a sequence. To do this, it needs to remember all the values that its source has ever product, so that it can filter out any items that it has seen before. Rx includes an implementation of `Distinct`, and this example uses it to display the unique identifier of vessels generating AIS messages, but ensuring that we only display each such identifier the first time we see it:

```
IObservable<uint> newIds = receiverHost.Messages
    .Select(m => m.Mmsi)
    .Distinct();

newIds.Subscribe(id => Console.WriteLine($"New vessel: {id}"));
```

(This is leaping ahead a little—it uses `Select`, which we'll get to in Chapter XXX. However, this is a very widely used LINQ operator, so you are probably already familiar with it. I'm using it here to extract just the MMSI—the vessel identifier—from the message.)

This example is fine if we are only interested in vessels' identifiers. But what if we want to inspect the detail of these messages? How can we retain the ability to see messages only for vessels we've never previously heard of, but still be able to look at the information in those message? The use of `Select`

to extract the id stops us from doing this. Fortunately, `Distinct` provides an overload enabling us to change how it determines uniqueness. Instead of getting `Distinct` to look at the values it is processing, we can provide it with a function that lets us pick whatever characteristics we like. So instead of filtering the stream down to values that have never been seen before, we can instead filter the stream down to values that have some particular property of combination of properties we've never seen before. Here's a simple example:

```
IObservable<IAisMessage> newVesselMessages = receiverHost.Messages
    .Distinct(m => m.Mmsi);
```

Here, the input to `Distinct` is now an `IObservable<IAisMessage>`. (In the preceding example it was actually `IObservable<uint>`, because the `Select` clause picked out just the MMSI.) So `Distinct` now receives the entire `IAisMessage` each time the source emits one. But because we've supplied a callback, it's not going try and compare entire `IAisMessage` messages with one another. Instead, each time it receives one, it passes that to our callback, and then looks at the value our callback returns, and compares that with the values the callback returned for all previously seen messages, and lets the message through only if that's new.

So the effect is similar to before—messages will be allowed through only if they have an MMSI not previously seen. But the difference is that the `Distinct` operator's output here is `IObservable<IAisMessage>`, so when `Distinct` lets an item through, the entire original message remains available.

In addition to the standard LINQ `Distinct` operator, Rx also provides `DistinctUntilChanged`. This only lets through notifications when something has changed, which it achieved by filtering out only adjacent duplicates. For example, given the sequence `1,2,2,3,4,4,5,4,3,3,2,1,1` it would produce `1,2,3,4,5,4,3,2,1`. Whereas `Distinct` remembers every value ever produced, `DistinctUntilChanged` remembers only the most recently emitted value, and filters out new values if and only if they match that most recent value.

This example uses `DistinctUntilChanged` to detect when a particular vessel reports a change in `NavigationStatus`.

```
uint exampleMmsi = 235009890;
IObservable<IAisMessageType1to3> statusChanges = receiverHost.Messages
    .Where(v => v.Mmsi == exampleMmsi)
    .OfType<IAisMessageType1to3>()
    .DistinctUntilChanged(m => m.NavigationStatus)
    .Skip(1);
```

For example, if the vessel had repeatedly been reporting a status of `AtAnchor`, `DistinctUntilChanged` would drop each such message because the status was the same as it had previously been. But if the status were to change to `UnderwayUsingEngine`, that would cause `DistinctUntilChanged`

to let the first message reporting that status through. It would then not allow any further messages through until there was another change in value, either back to `AtAnchor`, or to something else such as `Moored`. (The `Skip(1)` on the end is there because `DistinctUntilChanged` always lets through the very first message it sees. We have no way of knowing whether that actually represents a change in status, but it is very likely not to: ships report their status every few minutes, and change their status much less often, so the first time we receive a report of a ship's status, it probably doesn't represent a change of status. By dropping that first item, we ensure that `statusChanges` provides notifications only when we can be certain that the status changed.)

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

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 operators that can transform data.

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. In this chapter we will look at *transforming* sequences.

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")
```

This uses the Dump extension method we defined at the start of the Filtering Chapter. It produces the following 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.

SelectMany

Whereas `Select` produces one output for each input, `SelectMany` enables each input element to be transformed into any number of outputs. To see how this can work, let's first look at an example that uses just `Select`:

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

which produces this output:

```
strings-->A
strings-->BB
strings-->CCC
strings-->DDDD
strings-->EEEEE
strings completed
```

As you can see, for each of the numbers produced by `Range`, our output contains a string whose length is that many characters. What if, instead of transforming each number into a string, we transformed it into an `IObservable<char>`. We can do that just by adding `.ToObservable()` after constructing the string:

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

(Alternatively, we could have replaced the selection expression with `i => Observable.Repeat((char)(i+64), i)`. Either has exactly the same effect.) The output isn't terribly useful:

```
strings-->System.Reactive.Linq.ObservableImpl.ToObservableRecursive`1[System.Cha
strings-->System.Reactive.Linq.ObservableImpl.ToObservableRecursive`1[System.Cha
strings-->System.Reactive.Linq.ObservableImpl.ToObservableRecursive`1[System.Cha
strings-->System.Reactive.Linq.ObservableImpl.ToObservableRecursive`1[System.Cha
strings-->System.Reactive.Linq.ObservableImpl.ToObservableRecursive`1[System.Cha
strings completed
```

We have an observable sequence of observable sequences. But look at what happens if we now replace that `Select` with a `SelectMany`:

```
Observable
    .Range(1, 5)
    .SelectMany(i => new string((char)(i+64), i).ToObservable())
    .Dump("chars");
```

This gives us an `IObservable<char>`, with this output:

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

The order has become a little scrambled, but if you look carefully you'll see that the number of occurrences of each letter is the same as when we were emitting strings. There is just one A, for example, but C appears three times, and E five times.

`SelectMany` expects the transformation function to return an `IObservable<T>` for each input, and it then combines the result of those back into a single result. The LINQ to Objects equivalent is a little less chaotic. If you were to run this:

```
Enumerable
    .Range(1, 5)
    .SelectMany(i => new string((char)(i+64), i))
    .ToList()
```

it would produce a list with these elements:

```
[ A, B, B, C, C, C, D, D, D, D, E, E, E, E, E ]
```

The order is less odd. It's worth exploring the reasons for this in a little more detail.

IEnumerable vs. IObservable SelectMany

`IEnumerable<T>` is pull based—sequences produce elements only when asked. `Enumerable.SelectMany` pulls items from its sources in a very particular order. It begins by asking its source `IEnumerable<int>` (the one returned by `Range` in the preceding example), and then retrieves the first value. `SelectMany` then invokes our callback, passing this first item, and then enumerates everything in the `IEnumerable<char>` our callback returns. Only when it has exhausted this does it ask the source (`Range`) for a second item. Again, it passes that second item to our callback and then fully enumerates the `IEnumerable<char>`, we return, and so on. So we get everything from the first nested sequence first, then everything from the second, etc.

`Enumerable.SelectMany` is able to proceed in this way for two reasons. First, the pull-based nature of `IEnumerable<T>` enables it to decide on the order in which it processes things. Second, with `IEnumerable<T>` it is normal for operations to block, i.e., not to return until they have something for us. When the preceding example calls `ToList`, it won't return until it has fully populated a `List<T>` with all of the results.

Rx is not like that. First, consumers don't get to tell sources when to produce each item—sources emit items when they are ready to. Second, Rx typically models ongoing processes, so we don't expect method calls to block until they are done. There are some cases where Rx sequences will naturally produce all of their items very quickly and complete as soon as they can, but kinds of information sources that we tend to want model with Rx typically don't behave that way. So most operations in Rx do

not block—they immediately return something (such as an `IObservable<T>`, or an `IDisposable` representing a subscription) and will then produce values later.

The Rx version of the example we're currently examining is in fact one of these unusual cases where each of the sequences emits items as soon as it can. Logically speaking, all of the nested `IObservable<char>` sequences are in progress concurrently. The result is a mess because each of the observable sources here attempts to produce every element as quickly as the source can consume them. The fact that they end up being interleaved has to do with the way these kinds of observable sources use Rx's *scheduler* system, which we will describe in chapter XXX. Schedulers ensure that even when we are modelling logically concurrent processes, the rules of Rx are maintained, and observers of the output of `SelectMany` will only be given one item at a time. The following marble diagram shows the events that lead to the scrambled output we see:

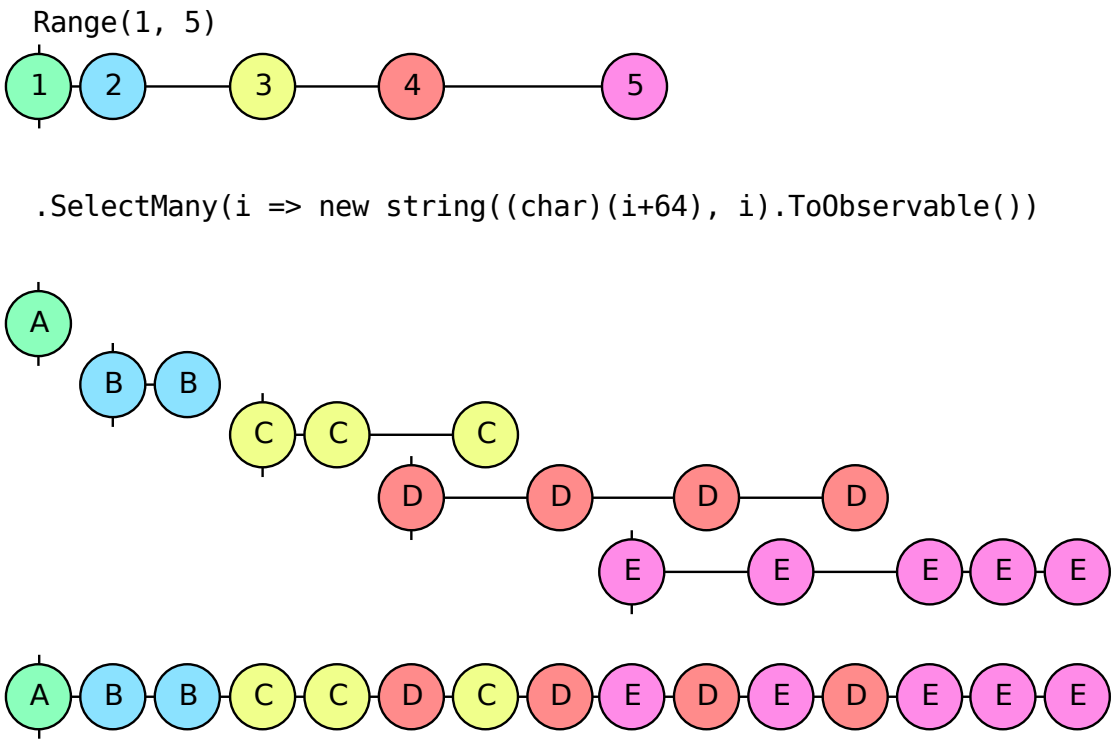


Figure 1: An Rx marble diagram showing 7 observables. The first illustrates the Range operator producing the values 1 through 5. These are colour coded as follows: green, blue, yellow, red, and pink. These colour correspond to observables further down in the diagram, as will be described shortly. The items in this first observable are not evenly spaced—the 2nd value immediately follows the 1st, but there are gaps before the 3rd, 4th, and 5th items. These gaps correspond with activity shown further down in the diagram. Beneath the first observable is code invoking the SelectMany operator, passing this lamba: "i => new string((char)(i+64),i).ToObservable()". Beneath this are 6 more observables. The first 5 show the individual observables that the lambda produces for each of the inputs. These are colour coded to show how they correspond—the first observable's item is green, to indicate that this observable corresponds to the first item emitted by Range, the second observable's items are blue showing that it corresponds to the second item emitted by Range, and so on with the same colour sequence as described earlier for Range. Each observable's first item is aligned horizontally with the corresponding item from Range, signifying the fact that each one of these observables starts when the Range observable emits a value. These 5 observables show the values produced by the observable returned by the lambda for each of the 5 values from Range. The first of these child observables shows a single item with value 'A', vertically aligned with the value 1 from the Range observable to indicate that this item is produced immediately when Range produces its first value. This child observable then immediately ends, indicating that only one item was produced. The second child observable contains two 'B' values, the third three 'C' values, the fourth four 'D' values and the fifth give 'E' values. The horizontal positioning of these items indicates that all of first 6 observables in the diagram (the Range observable, and the 5 observables produced by the lambda) overlap to some extent. Initially this overlap is minimal: the first of the lambda-produced observables starts at the same time the Range produces its first value so these two observables overlap, but since this first child completes immediately it overlaps with nothing else. The second child starts when Range produces its second value, and manages to produce two values and then completes before anything else happens, so thus far, the child observables produced by the lambda overlap only with the Range one, and not with each

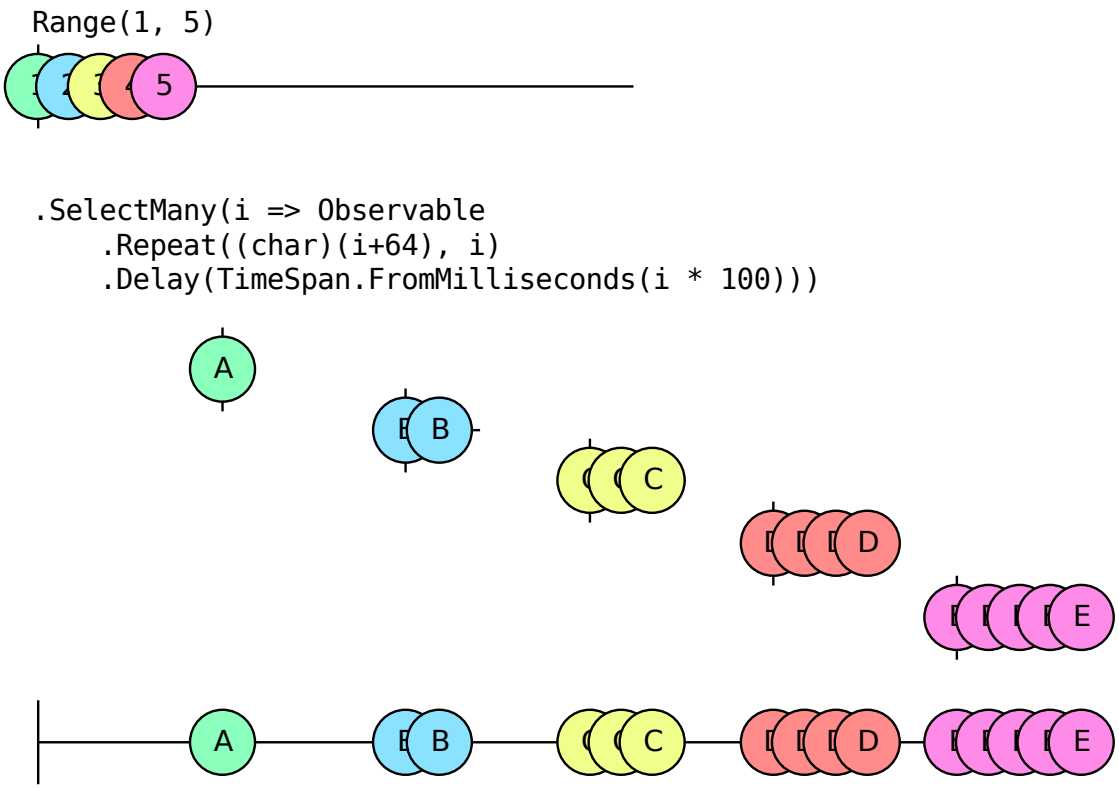
We can make a small tweak to prevent the child sequences all from trying to run at the same time. (This also uses `Observable.Repeat` instead of the rather indirect route of constructing a `string` and then calling `ToObserable` on that. I did that in earlier examples to emphasize the similarity with the LINQ to Objects example, but you wouldn't really do it that way in Rx.)

```
Observable
    .Range(1, 5)
    .SelectMany(i => Observable.Repeat((char)(i+64),
    ↪  i).Delay(TimeSpan.FromMilliseconds(i * 100)))
    .Dump("chars");
```

Now we get output consistent with the `IEnumerable<T>` version:

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

This clarifies that `SelectMany` lets you produce a sequence for each item that the source produces, and to have all of the items from all of those new sequences flattened back out into one sequence that contains everything. While that might make it easier to understand, you wouldn't want to introduce this sort of delay in reality purely for the goal of making it easier to understand. These delays mean it will take about a second and a half for all the elements to emerge. This marble diagram shows that the code above produces a sensible-looking ordering by making each child observable produce a little bunch of items, and we've just introduced dead time to get the separation:



I introduced these gaps purely to provide a slightly less confusing example, but if you really wanted this sort of strictly-in-order handling, you wouldn't use `SelectMany` in this way in practice. For one thing, it's not completely guaranteed to work. (If you try this example, but modify it to use shorter and shorter timespans, eventually you reach a point where the items start getting jumbled up again. And since .NET is not a real-time programming system, there's actually no safe timespan you can specific here that absolutely guarantees the ordering.) If you absolutely need all the items from the first child sequence before seeing any from the second, there's actually a robust way to ask for that:

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

However, that would not have been a good way to show what `SelectMany` does, since this no longer uses it. (It uses `Concat`, which will be discussed in Chapter XXX.) We use `SelectMany` either when we know we're unwrapping a single-valued sequence, or when we don't have specific ordering requirements, and want to take elements as and when they emerge from child observables.

The Significance of SelectMany

If you've been reading this book's chapters in order, you had already seen two examples of `SelectMany` in earlier chapters. The first example in the **LINQ Operators and Composition** section of Chapter 2 used it. Here's the relevant code:

```
IObservable<int> onoffs =
    from _ in src
    from delta in Observable.Return(1,
↪   scheduler).Concat(Observable.Return(-1,
↪   scheduler).Delay(minimumInactivityPeriod, scheduler))
    select delta;
```

(If you're wondering where the call to `SelectMany` is in that, remember that if a Query Expression contains two `from` clauses, the C# compiler turns those into a call to `SelectMany`.) This illustrates a common pattern in Rx, which might be described as fanning out, and then back in again.

As you may recall, this example worked by creating a new, short-lived `IObservable<int>` for each item produced by `src`. (These child sequences, represented by the `delta` range variable in the example, produce the value `1`, and then after the specified `minimumActivityPeriod`, they produce `-1`. This enabled us to keep count of the number of recent events emitted.) This is the *fanning out* part, where items in a source sequence produce new observable sequences. `SelectMany` is crucial in these scenarios because it enables all of those new sequences to be flattened back out into a single output sequence.

The second place I used `SelectMany` was slightly different: it was the final example of the **Representing Filesystem Events in Rx** section in Chapter 3. Although that example also combined multiple observable sources into a single observable, that list of observables was fixed: there was one for each of the different events from `FileSystemWatcher`. It used a different operator `Merge` (which we'll get to in XXX), which was simpler to use in that scenario because you just pass it the list of all the observables you'd like to combine. However, because of a few other things this code wanted to do (including deferred startup, automated disposal, and sharing a single source when multiple subscribers were active), the particular combination of operators used to achieve this meant our merging code that returned an `IObservable<FileSystemEventArgs>`, needed to be invoked as a transforming step. If we'd just used `Select`, the result would have been an `IObservable<IObservable<FileSystemEventArgs>>`. The structure of the code meant that it would only ever produce a single `IObservable<FileSystemEventArgs>`, so the double-wrapped type would be rather inconvenient. `SelectMany` is very useful in these scenarios—if composition of operators has introduced an extra layer of observables-in-observables that you don't want, `SelectMany` can unwrap one layer for you.

These two cases—fanning out then back in, and removing or avoiding a layer of observables of

observables—come up quite often, which makes `SelectMany` an important method. (It's not surprising that I was unable to avoid using it in earlier examples.)

As it happens, `SelectMany` is also a particularly important operator in the mathematical theory that Rx is based on. It is a fundamental operator, in the sense that it is possible to build many other Rx operators with it. Section XXX in Appendix C shows how you can implement `Select` and `Where` using `SelectMany`.

Cast

C#'s type system is not omniscient. Sometimes we might know something about the type of the values emerging from an observable source that is not reflected in that source's type. This might be based on domain-specific knowledge. For example, with the AIS message broadcast by ships, we might know that if the message type is 3, it will contain navigation information. That means we could write this:

```
IObservable<IVesselNavigation> type3 = receiverHost.Messages
    .Where(v => v.MessageType == 3)
    .Cast<IVesselNavigation>();
```

This uses `Cast`, a standard LINQ operator that we can use whenever we know that the items in some collection are of some more specific type than the type system ahs been able to deduce.

The difference between `Cast` and the `OfType` operator shown in Chapter 5 is the way in which they handle items that are not of the specified type. `OfType` is a filtering operator, so it just filters out any items that are not of the specified type. But with `Cast` (as with a normal C# cast expression) we are asserting that we expect the source items to be of the specified type, so the observable returned by `Cast` will invoke its subscriber's `OnError` if its source produces an item that is not compatible with the specified type.

This distinction might be easier to see if we recreate the functionality of `Cast` and `OfType` using other more fundamental operators.

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

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

Materialize and Dematerialize

The `Materialize` operator transforms a source of `IObservable<T>` into one of type `IObservable<Notification<T>>`. It will provide one `Notification<T>` for each item the

source produces, and, if the sourced terminates, it will produce one final `Notification<T>` indicating whether it completed successfully or with an error.

This can be useful because it produces objects that describe a whole sequence. If you wanted to record the output of an observable in a way that could later be replayed…well you'd probably use a `ReplaySubject<T>` because it is designed for precisely that job. But if you wanted to be able to inspect or modify the items in the sequence once they had been collectors, you might want to write your own code to store items. `Notification<T>` can be helpful because it enables you to represent everything a source does in a uniform way. You don't need to store information about whether or how the sequence terminates separately—this information is just the final `Notification<T>`.

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>>`.

This completes our tour of the transformation operators. Their common characteristic is that they produce an output (or, in the case of `SelectMany`, a set of outputs) for each input item. Next we will look at the operators that can combine information from multiple items in their source.

Aggregation

Data is not always tractable is its raw form. Sometimes we need to consolidate, collate, combine or condense the mountains of data we receive. This might just be a case of reducing the volume of data to a manageable level. For example, 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 for individual sources, and much higher rates if we're observing multiple sources. Can a person actually consume this? For human consumption, aggregate values like averages, minimums and maximums can be of more use.

We can often achieve more than this. The way in which we combine and correlate may enable us to reveal patterns, providing insights that would not be available from any individual message, or from simple reduction to a single statistical measure. Rx's composability enables us to express complex and subtle computations over streams of data enabling us not just to reduce the volume of messages that users have to deal with, but to increase the amount of value in each message a human receives.

We will start with the simplest aggregation functions, which reduce an observable sequence to a sequence with a single value in some specific way. We then move on to more general-purpose operators that enable you to define your own aggregation mechanisms.

Simple Numeric Aggregation

Rx supports various standard LINQ operators that reduce all of the values in a sequence down to a single numeric result.

Count

`Count` tells you how many elements a sequence contains. Although this is a standard LINQ operator, Rx's version deviates from the `IEnumerable<T>` version as Rx will return an observable sequence, not a scalar value. As usual, this is because of the push-related nature of Rx. Rx's `Count` can't demand that its source supply all elements immediately, so it just has to wait until the source says that it has finished.

The sequence that `Count` returns will always be of type `IObservable<int>`, regardless of the source's element type. This will produce nothing until the source completes, at which point it will produce a single value reporting how many elements the source produced, and then it will in turn immediately complete. This example uses `Count` with `Range`, because `Range` produces all of its values as quickly as possible and then completes, meaning we get a result from `Count` immediately:

```
IObservable<int> 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 signed integer can count, you can use the `LongCount` operator instead. This is just the same as `Count` except it returns an `IObservable<long>`.

Sum

The `Sum` operator adds together all the values in its source, producing the total as its only output. As with `Count`, Rx's `Sum` differs from most other LINQ providers in that it does not produce a scalar as its output—it produces an observable sequence that does nothing until its source completes. When the source completes, the observable returned by `Sum` produces a single value and then immediately completes. This example shows it in use:

```
IObservable<int> numbers = Observable.Range(1,5);
numbers.Dump("numbers");
numbers.Sum().Dump("sum");
```

The output shows the numbers produced by the source, and also the single result produced by `Sum`:

```
numbers-->1
numbers-->2
numbers-->3
numbers-->4
numbers-->5
numbers completed
sum-->15
sum completed
```

`Sum` is only able to work with values of type `int`, `long`, `float`, `double` `decimal`, or the nullable versions of these. This means that there are types you might expect to be able to `Sum` that you can't. For example the `BigInteger` type in the `System.Numerics` namespace represents integer values whose size is limited only by available memory, and how long you're prepared to wait for it to perform calculations. (Even basic arithmetic gets very slow on numbers with millions of digits.) You can use + to add these together because the type defines an overload for that operator. But

Sum has historically had no way to find that. The introduction of generic math in C# 11.0 means that it would technically be possible to introduce a version of Sum that would work for any type T that implemented `IAdditionOperators<T, T, T>`. However, that would mean a dependency on .NET 7.0 (because generic math is not available in order versions), and at the time of writing this, Rx supports .NET 7.0 through its `net6.0` target. It could need to introduce a separate `net7.0` target to enable this, but has not yet done so. (To be fair, Sum in LINQ to Objects also doesn't support this yet.)

If you supply Sum with the nullable versions of these types (e.g., your source is an `IObservable<int?>`) then Sum will also return a sequence with a nullable item type, and it will produce `null` if any of the input values is `null`.

Although Sum can work only with a small, fixed list of numeric types, your source doesn't necessarily have to produce values of those types. Sum offers overloads that accept a lambda that extracts a suitable numeric value from each input element. For example, suppose you wanted to answer the following unlikely question: if the next 10 ships that happen to broadcast descriptions of themselves over AIS were put side by side, would they all fit in a channel of some particular width? We could do this by filtering the AIS messages down to those that provide ship size information, using `Take` to collect the next 10 such messages, and then using Sum. The Ais.NET library's `IVesselDimensions` interface does not implement addition (and even if it did, we already just saw that Rx wouldn't be able to exploit that), but that's fine: all we need to do is supply a lambda that can take an `IVesselDimensions` and return a value of some numeric type that Sum can process:

```
IObservable<IVesselDimensions> vesselDimensions = receiverHost.Messages
    .OfType<IVesselDimensions>();

IObservable<int> totalVesselWidths = vesselDimensions
    .Take(10)
    .Sum(dimensions => checked((int)(dimensions.DimensionToPort +
    ↪  dimensions.DimensionToStarboard)));
```

(If you're wondering what's with cast and the `checked` keyword here, AIS defines these values as unsigned integers, so the Ais.NET library reports them as `uint`, which is not a type Rx's Sum supports. In practice, it's very unlikely that a vessel will be wide enough to overflow a 32-bit signed integer, so we just cast it to `int`, and the `checked` keyword will throw an exception in the unlikely event that we encounter ship more than 2.1 billion metres wide.)

Average

The standard LINQ operator `Average` effectively calculates the value that Sum would calculate, and then divides it by the value that `Count` would calculate. And once again, whereas most LINQ

implementations would return a scalar, Rx's `Average` produces an observable.

Although `Average` can process values of the same numeric types as `Sum`, the output type will be different in some cases. If the source is `IObservable<int>`, or if you use one of the overloads that takes a lambda that extracts the value from the source, and that lambda returns an `int`, the result will be a `double`. This is because the average of a set of whole numbers is not necessarily a whole number. Likewise, averaging `long` values produces a `double`. However, inputs of type `decimal` produce outputs of type `decimal`, and likewise `float` inputs produce a 'float output.

As with `Sum`, if the inputs to `Average` are nullable, the output will be too.

Min and Max

Rx implements the standard LINQ `Min` and `Max` operators which find the element with the highest or lowest value. As with all the other operators in this section, these do not return scalars, and instead return an `IObservable<T>` that produces a single value.

Rx defines specialized implementations for the same numeric types that `Sum` and `Average` support. However, unlike those operators it also defines an overload that will accept source items of any type. When you use `Min` or `Max` on a source type where Rx does not define a specialized implementation, it uses `Comparer<T>.Default` to compare items. There is also an overload enabling you to pass a comparer.

As with `Sum` and `Average` there are overloads that accept a callback. If you use these overloads, `Min` and `Max` will invoke this callback for each source item, and will look for the lowest or highest value that your callback returns. Note that the single output they eventually produce will be a value returned by the callback, and not the original source item from which that value was derived. To see what that means, look at this example:

```
IObservable<int> widthOfWidestVessel = vesselDimensions
    .Take(10)
    .Max(dimensions => checked((int)(dimensions.DimensionToPort +
    ↪  dimensions.DimensionToStarboard)));
```

`Max` returns an `IObservable<int>` here, which will be the width of the widest vessel out of the next 10 messages that report vessel dimensions. But what if you didn't just want to see the width? What if you wanted the whole message?

MinBy and MaxBy

Rx offers two subtle variations on `Min` and `Max`: `MinBy` and `MaxBy`. These are similar to the callback-based `Min` and `Max` we just saw that enable us to work with sequences of elements that are not

numeric values, but which may have numeric properties. The difference is that instead of returning the minimum or maximum value, `MinBy` and `MaxBy` tell you which source element produced that value. For example, suppose that instead of just discovering the width of the widest ship, we wanted to know what ship that actually was:

```
IObservable<IVesselDimensions> widthOfWidestVessel = vesselDimensions
    .Take(10)
    .MaxBy(dimensions => checked((int)(dimensions.DimensionToPort +
    ↪  dimensions.DimensionToStarboard)));
```

This is very similar to the example in the preceding section. We're working with a sequence where the element type is `IVesselDimensions`, so we've supplied a callback that extracts the value we want to use for comparison purposes. (The same callback as last time, in fact.) Just like `Max`, `MaxBy` is trying to work out which element produces the highest value when passed to this callback. It can't know which that is until the source completes—if the source hasn't completed yet, all it can know is the highest *yet*, but that might be exceeded by a future value. So as with all the other operators we've looked at in this chapter, this produces nothing until the source completes, which is why I've put a `Take(10)` in there.

However, the type of sequence we get is different. `Max` returned an `IObservable<int>`, because it invokes the callback for every item in the source, and then produces the highest of the values that our callback returned. But with `MaxBy`, we get back an `IObservable<IVesselDimensions>` because `MaxBy` tells us which source element produced that value.

Of course, there might be more than one item that has the highest width—there might be three equally large ships, for example. With `Max` this doesn't matter because it's only trying to return the actual value: it doesn't matter how many source items had the maximum value, because it's the same value in all cases. But with `MaxBy` we get back the original items that produce the maximum, and if there were three that all did this, we wouldn't want Rx to pick just one arbitrarily.

So unlike the other aggregation operators we've seen so far, an observable returned by `MinBy` or `MaxBy` doesn't necessarily produce just a single value. It might produce several. You might ask whether it really is an aggregation operator, since it's not reducing the input sequence to one output. But it is reducing it to a single value: the minimum (or maximum) returned by the callback. It's just that it presents the result slightly differently—it produces a sequence based on the result of the aggregation process. You could think of it as a combination of aggregation and filtering: it performs aggregation to determine the minimum or maximum, and then filters the source sequence down just to the elements for which the callback produces that value.

Note: LINQ to Objects also defines `MinBy` and `MaxBy` methods, but they are slightly different. These LINQ to Objects versions do in fact arbitrarily pick a single source element—if there are multiple source values all producing the minimum (or maximum) result, LINQ to Objects gives you just one whereas Rx

gives you all of them. Rx defined its versions of these operators years before .NET 6.0 added their LINQ to Objects namesakes, so if you're wondering why Rx does it differently, the real question is why did LINQ to Objects not follow Rx's precedent. Presumably the .NET runtime library implementors decided they wanted to make `Min/MaxBy` feel as similar as possible to `Min` and `Max`.

Simple Boolean Aggregation

LINQ defines several standard operators that reduce entire sequences to a single boolean value.

Any

There are two ways to use `Any`. First we can look at the parameterless overload for the extension method `Any`. This effectively asks the question "are there any elements in this sequence?" It returns an observable sequence that will produce a single value of `false` if the source completes without emitting any values. If the source does produce a value however, then when the first value is produced, the result sequence will immediately produce `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
```

In the case where the source does produce a value, Any immediately unsubscribes from it. So if the source wants to report an error, Any will only see this if that is the first notification it produces.

```
var subject = new Subject<int>();
subject.Subscribe(Console.WriteLine,
    ex => Console.WriteLine("subject OnError : {0}", ex),
    () => Console.WriteLine("Subject completed"));
IObservable<bool> 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 asks a slightly different question: "are there any elements in this sequence that meet these criteria?" The effect is similar to using `Where` followed by the no-arguments form of Any.

```
IObservable<bool> any = subject.Any(i => i > 2);
// Functionally equivalent to
IObservable<bool> longWindedAny = 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` operator is similar to the `Any` method that takes a predicate, 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 reaches its end without producing any elements that do not satisfy the predicate, then `All` will push `true` as its value. (A consequence of this is that if you use `All` on an empty sequence, the result will be a sequence that produces `true`. This is consistent with how `All` works in other LINQ providers, but it might be surprising for anyone not familiar with the formal logic convention known as vacuous truth.)

Once `All` decides to produce a `false` value, it immediately unsubscribes from the source (just like `Any` does as soon as it determines that it can produce `true`.) If the source produces an error before this happens, the error 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
```

IsEmpty

The LINQ `IsEmpty` operator is logically the opposite of the no-arguments `Any` method. It returns `true` if and only if the source completes without producing any elements, and if the source produces an item, `IsEmpty` produces `false` and immediately unsubscribes.

Contains

The `Contains` operator determines whether a particular element is present in a sequence. You could implement it using `Any`, just supplying a callback that compares each item with the value you're looking for. However, it will typically be slightly more succinct, and may be a more direct expression of intent to write `Contains`.

```
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.

Build your own aggregations

If the built-in aggregations described in the preceding sections do not meet your needs, you can build your own. Rx provides two different ways to do this.

Aggregate

The `Aggregate` method is very flexible: it is possible to build any of the operators shown so far in this chapter with it. You supply it with a function, and it invokes that function once for every element. But it doesn't just pass the element into your function: it also provides a way for your function to collect—to *aggregate*—information. As well as the current element, it also passes in an *accumulator*. The accumulator can be any type you like—it will depend on what sort of information you're looking to collect. `Aggregate` passes in the current accumulator value, and then whatever value your function returns becomes the new accumulator value, and it will pass that into the function along with the next element from the source. There are a few variations on this, but the simplest overload looks like this:

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

If you wanted to produce your own version of `Count` for `int` values, you could do so by providing a function that just adds 1 for each value the source produces:

```
IObservable<int> sum = source.Aggregate((acc, element) => acc + 1);
```

To understand exactly what this is doing, let's look at how `Aggregate` will call this lambda. To make that slightly easier to see, suppose we put that lambda in its own variable:

```
Func<int, int, int> c = (acc, element) => acc + 1;
```

Now suppose the source produces an item with the value 100. `Aggregate` will invoke our function:

```
c(0, 100) // returns 1
```

The first argument is the current accumulator. `Aggregate` has used `default(int)` for the initial accumulator value, which is 0. Our function returns 1, which becomes the new accumulator value. So if the source produces a second value, say, 200, `Aggregate` will pass the new accumulator, along with the second value from the source:

```
c(1, 200) // returns 2
```

This particular function completely ignores its second argument (the element from the source). It just adds one to the accumulator each time. So the accumulator is nothing more than a record of the number of times our function has been called.

Now let's look at how we might implement `Sum` using `Aggregate`:

```
Func<int, int, int> s = (acc, element) => acc + element
IObservable<int> sum = source.Aggregate(s);
```

For the first element, `Aggregate` will again pass the default value for our chosen accumulator type, `int`: 0. And it will pass the first element value. So again if the first element is 100 it does this:

```
s(0, 100) // returns 100
```

And then if the second element is 200, `Aggregate` will make this call:

```
s(100, 200) // returns 300
```

Notice that this time, the first argument was 100, because that's what the previous invocation of `s` returned. So in this case, after seeing elements 100 and 200, the accumulator's value is 300, which is the sum of all the elements.

What if we want the initial accumulator value to be something other than `default(TAccumulator)`? There's an overload for that. For example, here's how we could implement something like `All` with `Aggregate`:

```
IObservable<bool> all = source.Aggregate(true, (acc, element) => acc &&
↪  element);
```

This isn't exactly equivalent to the real `All` by the way: it handles errors differently. `All` instantly unsubscribes from its source if it sees a single element that is `false`, because it knows that nothing else the source produces can possibly change the outcome. That means that if the source had been about to produce an error, it will no longer have the opportunity to do so because `All` unsubscribed. But `Aggregate` has no way of knowing that the accumulator has entered a state from which it can never leave, so it will remain subscribed to the source until the source completes (or until whichever code subscribed to the `IObservable<T>` returned by `Aggregate` unsubscribes). This means that if the source were to produce `true`, then `false`, `Aggregate` would, unlike `All`, remain subscribed to the source, so if the source goes on to call `OnError`, `Aggregate` will receive that error, and pass it on to its subscriber.

Here's a way to think about `Aggregate` that some people find helpful. If your source produces the values 1 through 5, and if the function we pass to `Aggregate` is called `f`, then the value that `Aggregate` produces once the source completes will be this:

```
T result = f(f(f(f(f(default(T), 1), 2), 3), 4), 5);
```

So in the case of our recreation of `Count`, the accumulator type was `int`, so that becomes:

```
int sum = s(s(s(s(s(0, 1), 2), 3), 4), 5);
// Note: Aggregate doesn't return this directly -
// it returns an IObservable<int> that produces this value.
```

Rx's `Aggregate` doesn't perform all those invocations at once: it invokes the function as each element occurs. So the calculations will be spread out over time. (If your callback is a *pure function*—one that is unaffected by global variables and other environmental factors, and which will always return the same result for any particular input—this doesn't matter. The result of `Aggregate` will be the same as if it had all happened in one big expression like the preceding example. But if your callback's behaviour is affected by, say, a global variable, or by the current contents of the filesystem, then the fact that it will be invoked when the source produces each value may be more significant.)

`Aggregate` has other names in some programming systems by the way. It is often referred to as a *fold*. (Specifically a *left fold*. A right fold proceeds in reverse. Conventionally its function takes arguments in the reverse order, so it would look like `s(1, s(2, s(3, s(4, s(5, 0)))))`. Rx does not offer a built-in right fold. It would not be a natural fit because it would have to wait until it received the final element before it could begin, meaning it would need to hold onto every element in the entire sequence, and then evaluate the entire fold at once when the sequence completes.) Some systems call it `reduce`.

You might have spotted that in my quest to re-implement some of the built-in aggregation operators, I went straight from `Sum` to `Any`. What about `Average`? It turns out we can't do that with the overloads I've shown you so far. And that's because `Average` needs to accumulate two pieces of information—the

running total and the count—and it also needs to perform once final step right at the end: it needs to divide the total by the count. With the overloads shown so far, we can only get part way there:

```
IObservable<int> nums = Observable.Range(1, 5);

IObservable<(int Count, int Sum)> avgAcc = nums.Aggregate(
    (Count: 0, Sum: 0),
    (acc, element) => (Count: acc.Count + 1, Sum: acc.Sum + element));
```

This uses a tuple as the accumulator, enabling it to accumulate two values: the count and the sum. But the final accumulator value becomes the result, and that's not what we want. We're missing that final step that calculates the average by dividing the sum by the count. Fortunately, `Aggregate` offers a 3rd overload that enables us to provide this final step. We pass a second callback which will be invoked just once when the source completes. `Aggregate` passes the final accumulator value into this lambda, and then whatever it returns becomes the single item produced by the observable that `Aggregate` returns.

```
IObservable<double> avg = nums.Aggregate(
    (Count: 0, Sum: 0),
    (acc, element) => (Count: acc.Count + 1, Sum: acc.Sum + element),
    acc => ((double) acc.Sum) / acc.Count);
```

I've been showing how `Aggregate` can re-implement some of the built-in aggregation operators to illustrate that it is a powerful and very general operator. However, that's not what we use it for. `Aggregate` is useful precisely because it lets us define custom aggregation.

For example, suppose I wanted to build up a list of the names of all the ships that have broadcast their details over AIS. Here's one way to do that:

```
IObservable<IReadOnlySet<string>> allNames = vesselNames
    .Take(10)
    .Aggregate(
        ImmutableHashSet<string>.Empty,
        (set, name) => set.Add(name.VesselName));
```

I've used `ImmutableHashSet<string>` here because its usage patterns happen to fit `Aggregate` neatly. An ordinary `HashSet<string>` would also have worked, but is a little less convenient because its Add method doesn't return the set, so our function needs an extra statement to return the accumulated set:

```
IObservable<IReadOnlySet<string>> allNames = vesselNames
    .Take(10)
    .Aggregate(
        new HashSet<string>(),
```

```
        (set, name) =>
        {
            set.Add(name.VesselName);
            return set;
        });
```

With either of these implementations, `vesselNames` will produce a single value that is a `IReadOnlySet<string>` containing each vessel name seen in the first 10 messages that report a name.

I've had to fudge an issue in these last two examples. I've made them work over just the first 10 suitable messages to emerge. Think about what would happen if I didn't have the `Take(10)` in there. The code would compile, but we'd have a problem. The AIS message source I've been using in various examples is an endless source. Ships will continue to move around the oceans for the foreseeable future. Ais.NET does not contain any code designed to detect either the end of civilisation, or the invention of technologies that render the use of ships obsolete, so it will never call `OnCompleted` on its subscribers. The observable returned by `Aggregate` reports nothing until its source either completes or fails. So if we remove that `Take(10)`, the behaviour would be identical `Observable.Never<IReadOnlySet<string>>`. I had to force the input to `Aggregate` to come to an end to make it produce something. But there is another way.

Scan

While `Aggregate` allows us to reduce complete sequences to a single, final value, sometimes this is not what we need. If we are dealing with an endless source, we might want something more like a running total, updated each time we receive a value. The `Scan` operator is designed for exactly this requirement. The signatures for both `Scan` and `Aggregate` are the same; the difference is that `Scan` doesn't wait for the end of its input. It produces the aggregated value after every item.

We can use this to build up a set of vessel names as in the preceding section, but with `Scan` we don't have to wait until the end. This will report the current list every time it receives a message containing a name:

```
IObservable<IReadOnlySet<string>> allNames = vesselNames
    .Scan(
        ImmutableHashSet<string>.Empty,
        (set, name) => set.Add(name.VesselName));
```

Note that this `allNames` observable will produce elements even if nothing has changed. If the accumulated set of names already contained the name that just emerged from `vesselNames`, the

call to `set.Add` will do nothing, because that name will already be in the set. But `Scan` scan produces one output for each input, and doesn't care if the accumulator didn't change. Whether this matters will depend on what you are planning to do with this `allNames` observable, but if you need to, you can fix this easily with the `DistinctUntilChanged` operator shown in Chapter 5.

You could think of `Scan` as being a version of `Aggregate` that shows its working. If we wanted to see how the process of calculating an average aggregates the count and sum, we could write this:

```
IObservable<int> nums = Observable.Range(1, 5);

IObservable<(int Count, int Sum)> avgAcc = nums.Scan(
    (Count: 0, Sum: 0),
    (acc, element) => (Count: acc.Count + 1, Sum: acc.Sum + element));

avgAcc.Dump("acc");
```

That produces this output:

```
acc-->(1, 1)
acc-->(2, 3)
acc-->(3, 6)
acc-->(4, 10)
acc-->(5, 15)
acc completed
```

You can see clearly here that `Scan` is emitting the current accumulated values each time the source produces a value.

Unlike `Aggregate`, `Scan` doesn't offer an overload taking a second function to transform the accumulator into the result. So we can see the tuple containing the count and sum here, but not the actual average value we want. But we can achieve that by using the `Select` operator described in the Transformation chapter:

```
IObservable<double> avg = nums.Scan(
    (Count: 0, Sum: 0),
    (acc, element) => (Count: acc.Count + 1, Sum: acc.Sum + element))
    .Select(acc => ((double) acc.Sum) / acc.Count);

avg.Dump("avg");
```

Now we get this output:

```
avg-->1
```

```
avg-->1.5
avg-->2
avg-->2.5
avg-->3
avg completed
```

Scan is a more generalised operator than Aggregate. You could implement Aggregate by combining Scan with the TakeLast() operator described in the Filtering chapter.

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

Aggregation is useful for reducing volumes of data or combining multiple elements to produce averages, or other measure that incorporate information from multiple elements. But to perform some kinds of analysis we will also need to slice up or otherwise restructure our data before calculating aggregated values. So in the next chapter we'll look at the various mechanisms Rx offers for partitioning data.

Partitioning

Rx can split a single sequence into multiple sequences. This can be useful for taking a single sequence and fanning out to many subscribers or perhaps taking aggregates on partitions. You may already be familiar with the standard LINQ operator GroupBy. Rx supports this, and also defines some of its own.

GroupBy

The GroupBy operator allows you to partition your sequence just as IEnumerable<T>'s GroupBy operator does. Once again, the open source Ais.Net project can provide a useful example. Its ReceiverHost class makes AIS messages available through Rx, defining a Messages property of type IObservable<IAisMessage>. This is a very busy source, because it reports every message it is able to access. For example, if you connect the receiver to the AIS message source generously provided by the Norwegian government, it produces a notification every time *any* ship broadcasts an AIS message anywhere on the Norwegian coast. There are a lot of ships moving around Norway, so this is a bit of a firehose.

If we know exactly which ships we're interested in, you saw how to filter this stream in the Filtering chapter. But what if we don't, and yet we still want to be able to perform processing relating to individual ships? For example, perhaps we'd like to discover any time any ship changes its

`NavigationStatus` (which reports values such as `AtAnchor`, or `Moored`). The `Distinct` and `DistinctUntilChanged` section of the Filtering chapter showed how to do exactly that, but it began by filtering the stream down to message from a single ship. If we tried to use `DistinctUntilChanged` directly on the all-ships stream it will not produce meaningful information. If ship A is moored and ship B is at anchor, and if we receive alternative status messages from ship A and ship B, `DistinctUntilChanged` would report each message as a change in status, even though neither ship's status has changed.

We can fix this by splitting the "all the ships" sequence into lots of little sequences:

```
IObservable<IGroupedObservable<uint, IAisMessage>> perShipObservables =
↪   receiverHost.Messages
    .GroupBy(message => message.Mmsi);
```

This `perShipObservables` is an observable sequence of observable sequences. More specifically, it's an observable sequence of grouped observable sequences, but as you can see from the definition of `IGroupedObservable<TKey, T>`, a grouped observable is just a specialized kind of observable:

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

Each time `receiverHost.Message` reports an AIS message, the `GroupBy` operator will invoke the callback to find out which group this item belongs to. We refer to the value returned by the callback as the *key*, and `GroupBy` remembers each key it has already seen. If this is a new key, `GroupBy` creates a new `IGroupedObservable` whose `Key` property will be the value just returned by the callback. It emits this `IGroupedObservable` from the outer observable (the one we put in `perShipObservables`) and then immediately causes that new `IGroupedObservable` to emit the element (an `IAisMessage` in this example) that produced that key. But if the callback produces a key that `GroupBy` has seen before, it finds the `IGroupedObservable` that it already produced for that key, and causes that to emit the value.

So in this example, the effect is that any time the `receiverHost` reports a message from a ship with we've not previously heard from, `perShipObservables` will emit a new observable that reports message just for that ship. We could use this to report each time we learn about a new ship:

```
perShipObservables.Subscribe(m => Console.WriteLine($"New ship! {m.Key}"));
```

But that doesn't do anything we couldn't have achieved with `Distinct`. The power of `GroupBy` is that we get an observable sequence for each ship here, so we can go on to set up some per-ship processing:

```
IObservable<IObservable<IAisMessageType1to3>> shipStatusChangeObservables =
    perShipObservables.Select(shipMessages => shipMessages
        .OfType<IAisMessageType1to3>()
        .DistinctUntilChanged(m => m.NavigationStatus)
        .Skip(1));
```

This uses `Select` (introduced in the Transformation chapter) to apply processing to each group that comes out of `perShipObservables`. Remember, each such group represents a distinct ship, so the callback we've passed to `Select` here will be invoked exactly once for each ship. This means it's now fine for us to use `DistinctUntilChanged`—the input this example supplies to `DistinctUntilChanged` is a sequence representing the messages from just one ship, so this will tell us when that ship changes it status. This is now able to do what we want because each ship gets its own instance of `DistinctUntilChanged`.

At this point we have an observable sequence of observable sequences. The outer sequence produces a nested sequence for each distinct ship that it sees, and that nested sequence will report `NavigationStatus` changes for that particular ship.

I'm going to make a small tweak:

```
IObservable<IAisMessageType1to3> shipStatusChanges =
    perShipObservables.SelectMany(shipMessages => shipMessages
        .OfType<IAisMessageType1to3>()
        .DistinctUntilChanged(m => m.NavigationStatus)
        .Skip(1));
```

I've replaced `Select` with `SelectMany`, also described in the Transformation chapter. As you may recall, `SelectMany` flattens nested observables back into a single flat sequence. You can see this reflected in the return type—now we've got just an `IObservable<IAisMessageType1to3>` instead of a sequence of sequences.

Wait a second! Haven't I just undone the work that `GroupBy` did? I asked it to partition the events by vessel id, so why am I now recombining it back into a single, flat stream? Isn't that what I started with?

It's true that the stream type has the same shape as my original input: this will be a single observable sequence of AIS messages. (It's a little more specialized—the element type is `IAisMessageType1to3`, because that's where I can get `NavigationStatus` from, but these all still implement `IAisMessage`.) And all the different vessels will be mixed together in this one stream. But I've not actually negated the work that `GroupBy` did. A quick look at some output shows that this is very different from the raw `receiverHost.Messages`. First, I need to attach a subscriber:

```
shipStatusChanges.Subscribe(m =>
    Console.WriteLine($"Ship {((IAisMessage)m).Mmsi} changed status to
↪   {m.NavigationStatus} at {DateTimeOffset.UtcNow}"));
```

If I then let the receiver run for about ten minutes, I see this output:

```
Ship 257076860 changed status to UnderwayUsingEngine at 23/06/2023 06:42:48 +00:
Ship 257006640 changed status to UnderwayUsingEngine at 23/06/2023 06:43:08 +00:
Ship 259005960 changed status to UnderwayUsingEngine at 23/06/2023 06:44:23 +00:
Ship 259112000 changed status to UnderwayUsingEngine at 23/06/2023 06:44:33 +00:
Ship 259004130 changed status to Moored at 23/06/2023 06:44:43 +00:00
Ship 257076860 changed status to NotDefined at 23/06/2023 06:44:53 +00:00
Ship 258024800 changed status to Moored at 23/06/2023 06:45:24 +00:00
Ship 258006830 changed status to UnderwayUsingEngine at 23/06/2023 06:46:39 +00:
Ship 257428000 changed status to Moored at 23/06/2023 06:46:49 +00:00
Ship 257812800 changed status to Moored at 23/06/2023 06:46:49 +00:00
Ship 257805000 changed status to Moored at 23/06/2023 06:47:54 +00:00
Ship 259366000 changed status to UnderwayUsingEngine at 23/06/2023 06:47:59 +00:
Ship 257076860 changed status to UnderwayUsingEngine at 23/06/2023 06:48:59 +00:
Ship 257020500 changed status to UnderwayUsingEngine at 23/06/2023 06:50:24 +00:
Ship 257737000 changed status to UnderwayUsingEngine at 23/06/2023 06:50:39 +00:
Ship 257076860 changed status to NotDefined at 23/06/2023 06:51:04 +00:00
Ship 259366000 changed status to Moored at 23/06/2023 06:51:54 +00:00
Ship 232026676 changed status to Moored at 23/06/2023 06:51:54 +00:00
Ship 259638000 changed status to UnderwayUsingEngine at 23/06/2023 06:52:34 +00:
```

The critical thing to understand here is that in the space of ten minutes, `receiverHost.Messages` produced *thousands* of messages. (The rate varies by time of day, but it's typically over a thousand messages a minute. The code would have processed roughly ten thousand messages when I ran to produce that output.) But as you can see, `shipStatusChanges` produced just 19 messages.

This shows how Rx can tame high volume event sources in ways that are much more powerful than mere aggregation. We've not just reduced the data down to some statistical measure that can only provide an overview. Statistical measures such as averages or variance wouldn't be able to tell us anything about any particular ship. But here, every message tells us something about a particular ship. We've been able to retain that level of detail, despite the fact that we are looking at every ship. We've been able to instruct Rx to tell us any time any ship changes its status.

It probably seems like I'm making too big a deal of this, but it took so little effort to achieve this result that it can be easy to miss just how much work Rx is doing for us here. This code does all of the following:

- monitors every single ship operating in Norwegian waters
- provides per-ship information
- reports events at a rate that a human could reasonable cope with

It can take thousands of messages and perform the necessary processing to find the handful that really matter to us.

This is an example of the "fanning out, and then back in again" technique I described in 'The Significance of SelectMany' in the Transformation chapter. This code uses `GroupBy` to fan out from a single observable to multiple observables. The key to this step is to create nested observables that provide the right level of detail for the processing we want to do—in this example that level of detail was "one specific ship" but it wouldn't have to be. You could imagine wanting to group messages by region—perhaps we're interesting in comparing different ports, so we'd want to partition the source based on whichever port a vessel is closest to, or perhaps by its destination port. (AIS provides a way for vessels to broadcast their intended destination.) Having partitioned the data by whatever criteria we require, we then define the processing to be applied for each group. In this case, we just watched for changes to `NavigationStatus`. This step will typically be where the reduction in volume happens. For example, most vessels will only change their `NavigationStatus` a few times a day at most. Having then reduced the notification stream to just those events we really care about, we can combine it back into a single stream that provides the high-value notifications we want.

Now that we've seen an example, let's look at `GroupBy` in a bit more detail. It comes in a few different flavours. We just used this overload:

```
public static IObservable<IGroupedObservable<TKey, TSource>>
↪   GroupBy<TSource, TKey>(
    this IObservable<TSource> source,
    Func<TSource, TKey> keySelector)
```

That overload uses whatever the default comparison behaviour is for your chosen key type. In our case we used `uint` (the type of the `Mmsi` property that uniquely identifies a vessel in an AIS message), which is just a number, so it's an intrinsically comparable type. In some cases you might want non-standard comparison—for example, if you use `string` as a key, you might want to be able to specify a locale-specific case-insensitive comparison. For these scenarios, there's an overload that takes a comparer:

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

There are two more overloads that extend the preceding two with an `elementSelector` argument:

```
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)
{...}
```

This is functionally equivalent to using the `Select` operator after `GroupBy`.

By the way, when using `GroupBy` you might be tempted `Subscribe` directly to the nested observables:

```
// Don't do it this way. Use the earlier example.
perShipObservables.Subscribe(shipMessages =>
    shipMessages
        .OfType<IAisMessageType1to3>()
        .DistinctUntilChanged(m => m.NavigationStatus)
        .Skip(1)
        .Subscribe(m =>
            Console.WriteLine(
                $"Ship {((IAisMessage)m).Mmsi} changed status to
↪   {m.NavigationStatus} at {DateTimeOffset.UtcNow}")));
```

This may seem to have the same effect: `perShipObservables` here is the sequence returned by `GroupBy`, so it will produce a observable stream for each distinct ship. This example subscribes to that, and then uses the same operators as before on each nested sequence, but instead of collecting the results out into a single output observable with `SelectMany`, this explicitly calls `Subscribe` for each nested stream.

This might seem like a more natural way to work if you're unfamiliar with Rx. But although this will seem to produce he same behaviour, it introduces a problem: Rx doesn't understand that these nested subscriptions are associated with the outer subscription. That won't necessarily cause a problem in

this simple example, but it could if we start using additional operators. Consider this modification:

```
IDisposable sub = perShipObservables.Subscribe(shipMessages =>
    shipMessages
        .OfType<IAisMessageType1to3>()
        .DistinctUntilChanged(m => m.NavigationStatus)
        .Skip(1)
        .Finally(() => Console.WriteLine($"Nested sub for
        ↪  {shipMessages.Key} ending"))
        .Subscribe(m =>
            Console.WriteLine(
                $"Ship {((IAisMessage)m).Mmsi} changed status to
↪  {m.NavigationStatus} at {DateTimeOffset.UtcNow}")));
```

I've added a `Finally` operator for the nested sequence. This enables us to invoke a callback when a sequence comes to an end for any reason. But even if we unsubscribe from the outer sequence (by calling `sub.Dispose();`) this `Finally` will never do anything. That's because Rx has no way of knowing that these inner subscriptions are part of the outer one.

If we made the same modification to the earlier version, in which these nested sequences were collected into one output sequence by `SelectMany`, Rx understands that subscriptions to the inner sequence exist only because of the subscription to the sequence returned by `SelectMany`. (In fact, `SelectMany` is what subscribes to those inner sequences.) So if we unsubscribe from the output sequence in that example, it will correctly run any `Finally` callbacks on any inners sequences.

More generally, if you have lots of sequences coming into existence as part of a single processing chain, it is usually better to get Rx to manage the process from end to end.

Buffer

The `Buffer` operator is useful if you need to deal with events in batches. This can be useful for performance, especially if you're storing data about events. Take the AIS example. If you wanted to log notifications to a persistent store, the cost of storing a single record is likely to be almost identical to the cost of storing several. Most storage devices operate with blocks of data often several kilobytes in size, so the amount of work required to store a single byte of data is often identical to the amount of work required to store several thousand bytes. The pattern of buffering up data until we have a reasonably large chunk of work crops up all the time in programming—the .NET runtime library's `Stream` class has built-in buffering for exactly the reason, so it's no surprise that it's built into Rx.

Efficiency concerns are not the only reason you might want to process multiple events in once batch instead of individual ones. Suppose you wanted to generate a stream of continuously updated statistics

about some source of data. By carving the source into chunks with `Buffer`, you can calculate, say, an average over the last 10 events.

`Buffer` can partition the elements from a source stream, so it's a similar kind of operator to `GroupBy`, but there are a couple of significant differences. First, `Buffer` doesn't inspect the elements to determine how to partition them—it partitions purely based on the order in which elements emerge. Second, `Buffer` waits until it has completely filled a partition, and then presents all of the elements as an `IList<T>`. This can make certain tasks a lot easier because everything in the partition is available for immediate use—values aren't buried in a nested `IObservable<T>`. Third, `Buffer` offers some overloads that make it possible for a single element to turn up in more than one 'partition'. (In this case, `Buffer` is no longer strictly partitioning the data, but as you'll see, it's just a small variation on the other behaviours.)

The simplest way to use `Buffer` is to gather up adjacent elements into chunks. (LINQ to Objects now has an equivalent operator that it calls `Chunk`. The reason Rx didn't use the same name is that Rx introduced this operator over 10 years before LINQ to Objects did. So the real question is why LINQ to Objects chose a different name, but you'd need to ask the .NET runtime library team.) This overload of `Buffer` takes a single argument, indicating the chunk size you would like:

```
public static IObservable<IList<TSource>> Buffer<TSource>(
    this IObservable<TSource> source,
    int count)
{...}
```

This example uses it to split navigation messages into chunks of 4, and then goes on to calculate the average speed across those 4 readings:

```
IObservable<IList<IVesselNavigation>> navigationChunks =
↪   receiverHost.Messages
    .Where(v => v.Mmsi == 235009890)
    .OfType<IVesselNavigation>()
    .Where(n => n.SpeedOverGround.HasValue)
    .Buffer(4);

IObservable<float> recentAverageSpeed = navigationChunks
    .Select(chunk => chunk.Average(n => n.SpeedOverGround.Value));
```

If the source completes, and has not produced an exact multiple of the chunk size, the final chunk will be smaller. We can see this with the following more artificial example:

```
Observable
    .Range(1, 5)
    .Buffer(2)
```

```
    .Select(chunk => string.Join(", ", chunk))
    .Dump("chunks");
```

As you can see from this output, the final chunk has just a single item, even though we asked for 2 at a time:

```
chunks-->1, 2
chunks-->3, 4
chunks-->5
chunks completed
```

`Buffer` had no choice here because the source completed, and if it hadn't produced that final undersized chunk, we would never have seen the final item. But apart from this end-of-source case, this overload of `Buffer` waits until it has collected enough elements to fill a buffer of the specified size before passing it on. That means that `Buffer` introduces a delay. If source items are quite far apart (e.g., when a ship is not moving it might only report AIS navigation data every few minutes) this can lead to long delays.

In some cases, we might want to handle multiple events in a batch when a source is busy without having to wait a long time when the source is operating more slowly. This would be useful in a user interface—if you want to provide fresh information, it might be better to accept an undersized chunk so that you can provide more timely information. For these scenarios, `Buffer` offers overloads that accept a `TimeSpan`:

```
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 first of these partitions the source based on nothing but timing. This will emit one chunk every second no matter the rate at which `source` produces value:

```
IObservable<IList<string>> output = source.Buffer(TimeSpan.FromSeconds(1));
```

If `source` happened to emit no values during any particular chunk's lifetime, `output` will emit an empty list.

The second overload, taking both a `timespan` and a `count`, essentially imposes two upper limits: you'll never have to wait longer than `timespan` between chunks, and you'll never receive a chunk with more than `count` elements. As with the `timespan`-only overload, this can deliver under-full and even empty chunks if the source doesn't produce elements fast enough to fill the buffer within the time specified.

Overlapping buffers

In the preceding section, I showed an example that collected chunks of 4 `IVesselNavigation` entries for a particular vessel, and calculated the average speed. This sort of averaging over multiple samples can be a useful way of smoothing out slight random variations in readings. So the goal in this case wasn't to process items in batches for efficiency—it was to enable a particular kind of calculation.

But there was a problem with the example: because it was averaging 4 readings, it produced an output only once every 4 input messages. And since vessels might report their speed only once every few minutes if they are not moving, we might be waiting a very long time.

There's an overload of `Buffer` that enables us to do a little better: instead of averaging the first 4 readings, and then the 4 readings after that, and then the 4 after that, and so on, we might want to calculate the average of the last 4 readings _every time the vessel reports a new reading.

This is sometimes called a sliding window. We want to process readings 1, 2, 3, 4, then 2, 3, 4, 5, then 3, 4, 5, 6, and so on. There's an overload of buffer that can do this. This example shows the first statement from the earlier average speed example, but with one small modification:

```
IObservable<IList<IVesselNavigation>> navigationChunks =
↪   receiverHost.Messages
    .Where(v => v.Mmsi == 235009890)
    .OfType<IVesselNavigation>()
    .Where(n => n.SpeedOverGround.HasValue)
    .Buffer(4, 1);
```

This calls an overload of `Buffer` that takes two `int` arguments. The first does the same thing as before: it indicates that we want 4 items in each chunk. But the second argument indicates how often to produce a buffer. This says we want a buffer for every `1` element (i.e., every single element) that the source produces. (The overload that accepts just a `count` is equivalent to passing the same value for both arguments to this overload.)

So this will wait until the source has produce 4 suitable messages (i.e., messages that satisfy the `Where` and `OfType` operators here) and will then report those first four readings in the first `IList<VesselNavigation>` to emerge from `navigationChunks`. But the source only has to

produce one more suitable message, and then this will emit another `IList<VesselNavigation>`, containing 3 of the same value as were in the first chunk, and then the new value. When the next suitable message emerges, this will emit another list with the 3rd, 4th, 5th, and 6th messages, and so on.

This marble diagram illustrates the behaviour for `Buffer(4, 1)`.

TODO: MARBLE DIAGRAM!

If we fed this into the same `recentAverageSpeed` expression as the earlier example, we'd still get no output until the 4th suitable message emerges from the source, but from then on, every single suitable message to emerge from the source will emit a new average value. These average values will still always report the average of the 4 most recently reported speeds, but we will now get these averages four times as often.

We could also use this to improve the example earlier that reported when ships changed their `NavigationStatus`. The last example told you what state a vessel had just entered, but this raises an obvious question: what state was it in before? We can use `Buffer(2, 1)` so that each time we see a message indicating a change in status, we also have access to the preceding change in status:

```
IObservable<IList<IAisMessageType1to3>> shipStatusChanges =
    perShipObservables.SelectMany(shipMessages => shipMessages
        .OfType<IAisMessageType1to3>()
        .DistinctUntilChanged(m => m.NavigationStatus)
        .Buffer(2, 1));

IDisposable sub = shipStatusChanges.Subscribe(m => Console.WriteLine(
    $"Ship {((IAisMessage)m[0]).Mmsi} changed status from" +
    $" {m[1].NavigationStatus} to {m[1].NavigationStatus}" +
    $" at {DateTimeOffset.UtcNow}"));
```

As the output shows, we can now report the previous state as well as the state just entered:

```
Ship 259664000 changed status from UnderwayUsingEngine to Moored at 30/06/2023
 13:36:39 +00:00
Ship 257139000 changed status from AtAnchor to UnderwayUsingEngine at 30/06/20
23 13:38:39 +00:00
Ship 257798800 changed status from UnderwayUsingEngine to Moored at 30/06/2023
 13:38:39 +00:00
```

This change enabled us to remove the `Skip`—the earlier example had that because we can't tell whether the first message we receive from any particular ship after startup represents a change. But

since we're telling `Buffer` we want pairs of messages, it won't give us anything for any single ship until it has seen messages with two different statuses.

You can also ask for a sliding window defined by time instead of counts using this overload:

```
public static IObservable<IList<TSource>> Buffer<TSource>(
    this IObservable<TSource> source,
    TimeSpan timeSpan,
    TimeSpan timeShift)
{...}
```

The `timeSpan` determines the length of time covered by each window, and the `timeShift` determines the interval at which new windows are started.

Window

The `Window` operator is very similar to the `Buffer`. It can split the input into chunks based either on element count or time, and it also offers support for overlapping windows. However, it has a different return type. Whereas using `Buffer` on an `IObservable<T>` will return an `IObservable<IList<T>>`, `Window` will return an `IObservable<IObservable<T>>`. This means that `Window` doesn't have to wait until it has filled a complete buffer before producing anything.

Because `Buffer` returns an `IObservable<IList<T>>`, it can't produce a chunk until it has all of the elements that will go into that chunk. `IList<T>` supports random access—you can ask it how many elements it has, and you can retrieve any element by numeric index, and we expect these operations to complete immediately. (It would be technically possible to write an implementation of `IList<T>` representing as yet unreceived data, and to make its `Count` and indexer properties block if you try to use them before that data is available, but this would be a strange thing to do. Developers expect lists to return information immediately.) So if you write, say, `Buffer(4)`, it can't produce anything until it has all 4 items that will constitute the first chunk.

But because `Window` returns an observable that produces a nested observable to represent each chunk, it can emit that before necessarily having all of the elements. In fact, it emits a new window as soon as it knows it will need one. If you use `Window(4, 1)` for example, the observable it returns emits its first nested observable immediately. And then as soon as the source produces its first element, that nested observable will emit that element, and then the second nested observable will be produced. We passed `1` as the 2nd argument to `Window`, so we get a new window for every element the source produces. As soon as the first element has been emitted, the next item the source emits will appear in the second window (and also the first, since we've specified overlapping windows in this case), so the second window is effectively *open* from immediately after the emergence of the first element. So the

`IObservable<IObservable<T>>` that `Window` return produces a new `IObservable<T>` at that point.

Nested observables produce their items as and when they become available. They complete once `Window` knows there will be no further items in that window (i.e., at exactly the same point `Buffer` would have produced the completed `IList<T>` for that window.)

`Window` can seem like it is better than `Buffer` because it lets you get your hands on the individual items in a chunk the instant they are available. However, if you were doing calculations that required access to every single item in the chunk, this doesn't necessarily help you. You're not going to be able to complete your processing until you've received every item in the chunk, so you're not going to produce a final result any earlier, and your code might be more complicated because it can no longer count on having an `IList<T>` conveniently making all of the items available at once. However, if you're calcluating some sort of aggregation over the items in a chunk, `Window` might be more efficient because it enables you to process each item as it emerges and then discard it. If a chunk is very large, `Buffer` would have to hold onto every item until the chunk completes, which might use more memory. Moreover, in cases where you don't necessarily need to see every item in a chunk before you can do something useful with those items, `Window` might enable you to avoid introducing processing delays.

`Window` doesn't help us in the AIS `NavigationStatus` example, because the goal there was to report both the *before* and *after* status for each change. We can't do that until we know what the *after* value is, so we would get no benefit from receiving the *before* value earlier. We need the second value to do what we're trying to do, so we might as well use `Buffer` because it's easier.

TODO: need a good example for when you might actually use `Window`.

TODO: this next bit was imported from Sequences of Coincidence (now dropped), and needs to be edited

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`.

TODO: summary for this chapter.

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: price feeds, sensor networks, news feeds,

social media aggregators, file watchers, multi touch surfaces, 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.

Earlier chapters have also shown some examples of the *fan out and back in* style of data processing, where we partition data, and perform processing on each partition to convert high-volume data into lower-volume higher-value events before recombining. This ability to restructure streams greatly enhances the benefits of operator composition. If Rx only enabled us to apply composition as a simple linear processing chain, it would be a good deal less powerful. Being able to pull streams apart gives us much more flexibility. So even when there is a single source of events, we often still need to combine multiple observable streams as part of our processing. Sequence composition enables you to create complex queries across multiple data sources. This unlocks the possibility to write some very powerful and succinct code.

We've already used `SelectMany` in earlier chapters. This is one of the fundamental operators in Rx—as we saw in the Transformation chapter, it's possible to build several other operators from `SelectMany`, and its ability to combine streams is part of what makes it powerful. But there are several more specialized combination operators available, which make it easier to solve certain problems than it would be using `SelectMany`. Also, some operators we've seen before (including `TakeUntil` and `Buffer`) have overloads we've not yet explored that can combine multiple sequences.

Sequential Combination

We'll start with the simplest kind of combining operator, which do not attempt concurrent combination. They deal with one source sequence at a time.

Concat

`Concat` is arguably the simplest way to combine sequences. It does the same thing as its namesake in other LINQ providers: it concatenates two sequences. The resulting sequence produces all of the elements from the first sequence, followed by all of the elements from the second sequence. The simplest signature for `Concat` is as follows.

```
public static IObservable<TSource> Concat<TSource>(
    this IObservable<TSource> first,
    IObservable<TSource> second)
```

Since of `Concat` is an extension method, we can invoke it as a method on any sequence, passing the second sequence in as the only argument:

```
IObservable<int> s1 = Observable.Range(0, 3);
IObservable<int> s2 = Observable.Range(5, 5);
IObservable<int> c = s1.Concat(s2);
IDisposable sub = c.Subscribe(Console.WriteLine, x =>
↪   Console.WriteLine("Error: " + x));
```

This marble diagram shows the items emerging from the two sources, `s1` and `s2`, and how `Concat` combines them into the result, `c`:

TODO: draw properly

```
s1 0-1-2|
s2       5-6-7-8-9|
c  0-1-2-5-6-7-8-9|
```

Rx's `Concat` does nothing with its sources until something subscribes to the `IObservable<T>` it returns. So in this case, when we call `Subscribe` on `c` (the source returned by `Concat`) it will subscribe to its first input, `s1`, and each time that produces a value, the `c` observable will emit that same value to its subscriber. If we went on to call `sub.Dispose()` before `s1` completes, `Concat` would unsubscribe from the first source, and would never subscribe to `s2`. If `s1` were to report an error, `c` would report that same error to is subscriber, and again, it will never subscribe to `s2`. Only if `s1` completes will the `Concat` operator subscribe to `s2`, at which point it will forward any items that second input produces until either the second source completes or fails, or the application unsubscribes from the concatenated observable.

Although Rx's `Concat` has the same logical behaviour as the LINQ to Objects `Concat`, there are some Rx-specific details to be aware of. In particular, timing is often more significant in Rx than with other LINQ implementations. For example, in Rx we distinguish between *hot* and *cold* source. With a cold source it typically doesn't matter exactly when you subscribe, but hot sources are essentially live, so you only get notified of things that happen while you are subscribed. This can mean that hot sources might not be a good fit with `Concat` The following marble diagram illustrates a scenario in which this produces results that have the potential to surprise:

TODO: create marble diagram like this:

```
cold               |--0--1--2-|
hot                |---A---B---C---D---E-|
Concat(cold, hot)  |--0--1--2--C---D---E-|
```

Since `Concat` doesn't subscribe to its second input until the first has finished, it won't see the first couple of items that the `hot` source would deliver to any subscribers that been listening from the start. This might not be the behaviour you would expect: it certainly doesn't look like this concatenated all of the items from the first sequence with all of the items from the second one. It looks like it missed out A and B from `hot`.

Marble Diagram Limitations This last example reveals that marble diagrams gloss over a detail: they show when a source starts, when it produces values, and when it finishes, but they ignore the fact that to be able to produce items at all, an observable source needs a subscriber. If nothing subscribes to an `IObservable<T>`, then it doesn't really produce anything. `Concat` doesn't subscribe to its second input until the first completes, so arguably instead of the diagram above, it would be more accurate to show this:

```
cold                |--0--1--2-|
hot                             |C---D---E-|
Concat(cold, hot)   |--0--1--2--C---D---E-|
```

This makes it easier to see why `Concat` produces the output it does. But since `hot` is a hot source here, this diagram fails to convey the fact that `hot` is producing items entirely on its own schedule. In a scenario where `hot` had multiple subscribers, then the first diagram would arguably be better because it correctly reflects every event coming out of `hot` (regardless of however many listeners might be subscribed at any particular moment). But although this convention works for hot sources, it doesn't work for cold ones, which typically start producing items upon subscription. A source returned by `Timer` produces items on a regular schedule, but that schedule starts at the instant when subscription occurs. That means that if there are multiple subscriptions, there are multiple schedules. Even if I have just a single `IObservable<long>` returned by `Observable.Timer`, each distinct subscriber will get items on its own schedule—subscribers receive events at a regular interval *starting from whenever they happened subscribe*. So for cold observables, it typically makes sense to use the convention used by this second diagram, in which we're looking at the events received by one particular subscription to a source.

Most of the time we can get away with ignoring this subtlety, quietly using whichever convention suits us. To paraphrase Humpty Dumpty: when I use a marble diagram, it means just what I choose it to mean—neither more nor less. But when you're combining hot and cold sources together, there might not be one obviously best way to represent this in a marble diagram. We could even do something like this, where we describe the events that `hot` represents separately from the events seen by a particular subscription to `hot`.

```
Concat subscription to cold  |--0--1--2-|
Events available through hot  ---A---B---C---D---E-
Concat subscription to hot                |C---D---E-|
Concat(cold, hot)            |--0--1--2--C---D---E-|
```

We're using a distinct 'lane' in the marble diagram to represent the events seen by a particular subscription to a source. With this technique, we can also show what would happen if you pass the same cold source into `Concat` twice:

```
Concat 1st subscription to cold |--0--1--2-|
Concat 2nd subscription to cold            |--0--1--2-|
Concat(cold, cold)              |--0--1--2----0--1--2-|
```

This highlights the fact that that being a cold source, `cold` provides items separately to each subscription. We see the same three values emerging from the same source, but at different times.

Concatenating Multiple Sources What if you wanted to concatenate more than two sequences? `Concat` has an overloads accepting multiple observable sequences as an array. This is annotated with the `params` keyword, so you don't need to construct the array explicitly—you can just pass any number of arguments, and the C# compiler will generate the code to create the array for you. There's also an overload taking an `IEnumerable<IObservable<T>>`, in case the observables you want to concatenate are already in some collection.

```
public static IObservable<TSource> Concat<TSource>(
    params IObservable<TSource>[] sources)

public static IObservable<TSource> Concat<TSource>(
    this IEnumerable<IObservable<TSource>> sources)
```

The `IEnumerable<IObservable<T>>` overload evaluates `sources` lazily. It won't begin to ask it for source observables until someone subscribes to the observable that `Concat` returns, and it only calls `MoveNext` again on the resulting `IEnumerator<IObservable<T>>` when the current source completes meaning it's ready to start on the text. To illustrate this, the following example is an iterator 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)
            .Finally(() => Console.WriteLine("1st finished"))
            .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)
            .Finally(() => Console.WriteLine("2nd finished"))
            .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)
            .Finally(() => Console.WriteLine("3rd finished"))
            .Subscribe(o);
    });

    Console.WriteLine("GetSequences() complete");
}
```

We can call this `GetSequences` method and pass the results to `Concat`, and then use our `Dump` extension method to watch what happens:

```
GetSequences().Concat().Dump("Concat");
```

Here's the output:

```
GetSequences() called
Yield 1st sequence
1st subscribed to
Concat-->1
1st finished
Yield 2nd sequence
2nd subscribed to
Concat-->2
2nd finished
Yield 3rd sequence
3rd subscribed to
Concat-->3
3rd finished
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 once the iterator has executed its first `yield return` to return the first sequence, the iterator does not continue until the first sequence has completed. The iterator calls `Console.WriteLine` to display the text `Yield 2nd sequence` immediately after that first `yield return`, but you can see that message doesn't appear in the output until after we see the `Concat-->1` message showing the first output from `Concat`, and also the `1st finished` message, produced by the `Finally` operator, which runs only after that first sequence has completed. (The code also includes a 500ms delay so that if you run this, you can see that everything stops for a bit until that first source produces its single value then completes.) Once the first source completes, the `GetSequences` method continues (because `Concat` will ask it for the next item once the first observable source completes). When `GetSequences` provides the second sequence with another `yield return`, `Concat` subscribes to that, and again `GetSequences` makes no further progress until that second observable sequence completes. The third sequence is processed in the same fashion.

Prepend

There's one particular scenario that `Concat` supports, but in a slightly cumbersome way. It can sometimes be useful to make a sequence that always emits some initial value immediately. Take the example I've been using a lot in this book, where ships transmit AIS messages to report their location and other information: in some applications you might not want to wait until the ship happens next to transmit a message. You could imagine an application that records the last known location of any vessel. This would make it possible for the application to offer, say, an `IObservable<IVesselNavigation>` which instantly reports the last known information upon subscription, and which then goes on to supply any newer messages if the vessel produces any.

How would we implement this? We want initially cold-source-like behaviour, but transitioning into hot. So we could just concatenate two sources. We could use `Observable.Return` to create a single-element cold source, and then concatenate that with the live stream:

```
IVesselNavigation lastKnown = ais.GetLastReportedNavigationForVessel(mmsi);
IObservable<IVesselNavigation> live =
↪   ais.GetNavigationMessagesForVessel(mmsi);

IObservable<IVesselNavigation> lastKnownThenLive = Observable.Concat(
    Observable.Return(lastKnown), live);
```

This is a common enough requirement that Rx supplies `Prepend` that has a similar effect. We can replace the final line with:

```
IObservable<IVesselNavigation> lastKnownThenLive = live.Prepend(lastKnown);
```

This observable will do exactly the same thing: subscribers will immediately receive the `lastKnown`, and then if the vessel should emit further navigation messages, they will receive those too. By the way, for this scenario you'd probably also want to ensure that the look up of the "last known" message happens as late as possible. We can delay this until the point of subscription by using `Defer`:

```
public static IObservable<IVesselNavigation>
↪   GetLastKnownAndSubsequenceNavigationForVessel(uint mmsi)
{
    return Observable.Defer<IVesselNavigation>(() =>
    {
        // This lambda will run each time someone subscribes.
        IVesselNavigation lastKnown =
↪   ais.GetLastReportedNavigationForVessel(mmsi);
        IObservable<IVesselNavigation> live =
↪   ais.GetNavigationMessagesForVessel(mmsi);
```

```
            return live.Prepend(lastKnown);
        }
    }
```

`StartWith` might remind you of `BehaviorSubject<T>`, because that also ensures that consumers receive a value as soon as they subscribe. It's not quite the same—`BehaviorSubject<T>` caches the last value its own source emits. You might thank that would make it a better way to implement this vessel navigation example. However, since this example is able to return a source for any vessel (the `mmsi` argument is a Maritime Mobile Service Identity uniquely identifying a vessel) it would need to keep a `BehaviorSubject<T>` running for every single vessel you were interested in, which might be impractical.

`BehaviorSubject<T>` can hold onto only one value, which is fine for this AIS scenario, and `Prepend` shares this limitation. But what if you need a source to begin with some particular sequence?

StartWith

`StartWith` is a generalization of `Prepend` that enables us to provide any number of values to emit immediately upon subscription. As with `Prepend`, it will then go on to forward any further notifications that emerge from the source.

As you can see from its signature, this method takes a `params` array of values so you can 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)
```

There's also an overload that accepts an `IEnumerable<T>`. Note that Rx will *not* defer its enumeration of this. `StartWith` immediately converts the `IEnumerable<T>` into an array before returning.

`StartsWith` is not a common LINQ operator, and its existence is peculiar to Rx. If you imagine what `StartsWith` would look like in LINQ to Objects, it would not be meaningfully different from `Concat`. There's a difference in Rx because `StartsWith` effectively bridges between *pull* and *push* worlds. It effectively converts the items we supply into an observable, and it then concatenates the `source` argument onto that.

Append

The existence of `Prepend` might lead you to wonder whether there is an `Append` for adding a single item onto the end of any `IObservable<T>`. After all, this is a common LINQ operator; LINQ to Objects has an `Append` implementation, for example. And Rx does indeed supply such a thing:

```
IObservable<string> oneMore = arguments.Append("And another thing...");
```

There is no corresponding `EndWith`. Apparently there's not much demand—the Rx repository has not yet had a feature request. So although the symmetry of `Prepend` and `Append` does suggest that there could be a similar symmetry between `StartWith` and an as-yet-hypothetical `EndWith`, the absence of this counterpart doesn't seem to have caused any problems. There's an obvious value to being able to create observable sources that always immediately produce a useful output; it's not clear what `EndWith` would be useful for beside satisfying a craving for symmetry.

DefaultIfEmpty

The next operator we'll examine doesn't strictly performs sequential combination. However, it's a very close relative of `Append` and `Prepend`. Like those operators, this will emit everything their source does. And like those operators, `DefaultIfEmpty` takes one additional item. The difference is that it won't always emit that additional item.

Whereas `Prepend` emits its additional item at the start, and `Append` emits its additional item at the end, `DefaultIfEmpty` emits the additional item only if the source completes without producing anything. So this provides a way of guaranteeing that an observable will not be empty.

You don't have to supply `DefaultIfEmpty` with a value. If you use the overload in which you supply no such value, it will just use `default(T)`. This will be a zero-like value for *struct* types and `null` for reference types.

Repeat

The final operator that combines sequences sequentially is `Repeat`. It allows you to simply repeat a sequence. It offers overloads where you can specify the number of times to repeat the input, and one that repeats infinitely:

```
//Repeats the observable sequence a specified number of times.
public static IObservable<TSource> Repeat<TSource>(
    this IObservable<TSource> source,
    int repeatCount)
```

```
// Repeats the observable sequence indefinitely and sequentially.
public static IObservable<TSource> Repeat<TSource>(
    this IObservable<TSource> source)
```

`Repeat` resubscribes to the source for each repetition.

If you use the overload that repeats 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
```

Concurrent sequences

We'll now move on to operators for combining observable sequences that are producing values concurrently.

Amb

Amb is a strangely named operator. It's short for *ambiguous*, but that doesn't tell us much more than Amb. If you're curious about the name you can read about the origins of Amb in Appendix C, but for

now, let's look at what it actually does. Rx's Amb takes any number of `IObservable<T>` sources as inputs, and waits to see which, if any, first produces some sort of output. As soon as this happens, it immediately unsubscribes from all of the other sources, and forwards all notifications from the source that reacted first.

Why is that useful?

A common use case for Amb is when you want to produce some sort of result as quickly as possible, and you have multiple options for obtaining that result but you don't know in advance which will be fastest. Perhaps there are multiple servers that could all potentially give you the answer you want, and it's impossible to predict which will have the lowest response time. You could send requests to all of them, and then just use the first to respond. If you model each individual request as its own `IObservable<T>`, Amb can handle this for you. Note that this isn't very efficient: you're asking several servers all to do the same work, and you're going to discard the results from most of them. (Since Amb unsubscribes from all the sources it's not going to use as soon as the first reacts, it's possible that you might be able to send a message to all the other servers to cancel the request. But this is still somewhat wasteful.) But there may be scenarios in which timeliness is crucial, and for those cases it might be worth tolerating a bit of wasted effort to produce faster results.

To illustrate Amb's behaviour, here's a marble diagram showing three sequences, `s1`, `s2`, and `s3`, each able to produce a sequence values. The line labelled `r` shows the result of passing all three sequences into Amb. As you can see, `r` provides exactly the same notifications as `s1`. This is because in this example, `s1` was the first sequence to produce a value.

TODO: draw properly.

```
s1 -1---2----3--4|
s2 --99--88-|
s3 ----8---7--6---|
r  -1---2----3--4|
```

This code creates exactly the situation described in that marble diagram, to verify that this is indeed how Amb behaves:

```
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(99);
s3.OnNext(8);
s1.OnNext(2);
s2.OnNext(88);
s3.OnNext(7);
s2.OnCompleted();
s1.OnNext(3);
s3.OnNext(6);
s1.OnNext(4);
s1.OnCompleted();
s3.OnCompleted();
```

Output:

```
1
2
3
4
Completed
```

If we changed the order so that `s2.OnNext(99)` came before the call to `s1.OnNext(1);` then s2 would produce values first and the marble diagram would look like this.

```
s1 --1--2----3--4|
s2 99----88--|
s3 ---8-----7--6--|
r  99----88-|
```

There are a few overloads of Amb. The preceding example used the overload that takes a `params` array of sequences. There's also an overload that takes exactly two sources, avoiding the array allocation that occurs with `params`. Finally, you could pass in an `IEnumerable<IObservable<T>>`. (Note that there are no overloads that take an `IObservable<IObservable<T>>`. Amb requires all of the source observables it monitors to be supplied up front.)

```
// 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
```

Here is the marble diagram illustrating how this code behaves:

```
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 takes multiple sequences as its input. Any time any of those input sequences produces a value, the observable returned by `Merge` produces that same value. If the input

sequences produce values at the same time on different threads, `Merge` handles this safely, ensuring that it delivers items one at a time.

Since `Merge` returns a single observable sequence that includes all of the values from all of its input sequences, there's a sense in which it is similar to `Concat`. But whereas `Concat` waits until each input sequence completes before moving onto the next, `Merge` supports concurrently active sequences. As soon as you subscribe to the observable returned by `Merge`, it immediately subscribes to all of its inputs, forwarding everything any of them produces. This marble diagram shows two sequences, `s1` and `s2`, running concurrently and `r` shows the effect of combining these with `Merge`—the values from both source sequences emerge from the merged sequence.

TODO: draw.

```
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. However, the `Merge` operator will error if any of the input sequences terminates erroneously (at which point it will unsubscribe from all its other inputs).

If you read the Creating Observables chapter, you've already seen one example of `Merge`. I used it to combine the individual sequences representing the various events provided by a `FileSystemWatcher` into a single stream at the end of the 'Representing Filesystem Events in Rx' section. As another example, let's look at AIS once again. There is no publicly available single global source that can provide all AIS messages across the entire globe as an `IObservable<IAisMessage>`. Any single source is likely to cover just one area, or maybe even just a single AIS receiver. With `Merge`, it's straightforward to combine these into a single source:

```
IObservable<IAisMessage> station1 =
↪   aisStations.GetMessagesFromStation("AdurStation");
IObservable<IAisMessage> station2 =
↪   aisStations.GetMessagesFromStation("EastbourneStation");

IObservable<IAisMessage> allMessages = station1.Merge(station2);
```

If you want to combine more than two sources, you have a few options::

- 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,`
- Apply the `Merge` operator to an `IEnumerable<IObservable<T>>`.
- Apply the `Merge` operator to an `IObservable<IObservable<T>>`.

The overloads look like this:

```
/// 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)
{...}
```

When you know at compile time exactly how many sequences you will be merging, choosing between the first two operators really is a matter of your preferred style: 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. That last `Merge` overload that takes a sequence of sequences is particularly interesting, because it makes it possible for the set of sources being merged to grow over time. With that last overload, `Merge` will remain subscribed to `sources` for as long as your code remains subscribed to the `IObservable<T>` that `Merge` returns. So if `sources` emits more and more `IObservable<T>`s over time, these will all be included by `Merge`.

That might sound familiar. In the Transformation chapter, we looked at the `SelectMany` operator, which is able to flatten multiple observable sources back out into a single observable source. This is just another illustration of why I've described `SelectMany` as a fundamental operator in Rx: strictly speaking we don't need a lot of the operators that Rx gives us because we could build them using

`SelectMany`. Here's a simple re-implementation of that last `Merge` overload using `SelectMany`:

```
public static IObservable<T> MyMerge<T>(this IObservable<IObservable<T>>
↪  sources) =>
    sources.SelectMany(source => source);
```

As well as illustrating that we don't technically need Rx to provide that last `Merge` for us, it's also a good illustration of why it's helpful that it does. It's not immediately obvious what this does—why are we passing a lambda that just returns its argument? Unless you've seen this before, it can take some thought to work out that `SelectMany` expects us to pass a callback that it invokes for each incoming item, but that our input items are already nested sequences, so we can just return each item directly, and `SelectMany` will then take that and merge everything it produces into its output stream. And even if you have internalized `SelectMany` so completely that you know right away that this will just flatten `sources`, you'd still probably find `Observable.Merge(sources)` a more direct expression of intent.

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|
```

For each of the `Merge` overloads that accept variable numbers of sources (either via an array, an `IEnumerable<IObservable<T>>`, or an `IObservable<IObservable<T>>`) there's an additional overload adding a `maxconcurrent` parameter. For example:

```
public static IObservable<TSource> Merge<TSource>(this
↪  IEnumerable<IObservable<TSource>> sources, int maxConcurrent)
```

This enables you to limit the number of sources that `Merge` accepts inputs from at any single time. If the number of sources available exceeds `maxConcurrent` (either because you passed in a collection with more sources, or because you used the `IObservable<IObservable<T>>`-based overload and the source emitted more nested sources than `maxConcurrent`) `Merge` will wait for existing sources to complete before moving onto new ones. A `maxConcurrent` of 1 makes `Merge` behave in the same way as `Concat`.

Switch

Rx's `Switch` operator takes an `IObservable<IObservable<T>>`, and produces notifications from the most recent nested observable. Each time its source produces a new nested `IObservable<T>`, `Switch` unsubscribes from the previous nested source (unless this is the first source, in which case there won't be a previous one) and subscribes to the latest one.

`Switch` can be used in a 'time to leave' type application. In fact you can see the source code for a modified version of how Bing provides (or at least provided; the implementation may have changed) notifications telling you that it's time to leave for an appointment. Since that's derived from a real example, it's a little complex, so I'll describe just the essence here.

The basic idea with a 'time to leave' notification is that we using map and route finding services to work out the expected journey time to get to wherever the appointment is, and to use the `Timer` operator to create an `IObservable<T>` that will produce a notification when it's time to leave. (Specifically this code produces an `IObservable<TrafficInfo>` which reports the proposed route for the journey, and expected travel time.) However, there are two things that can change, rendering the initial predicted journey time useless. First, traffic conditions can change. When the user created their appointment, we have to guess the expected journey time based on how traffic normally flows at the time of day in question. However, if there turns out to be really bad traffic on the day, the estimate will need to be revised upwards, and we'll need to notify the end user earlier.

The other thing that can change is the user's location. This will also obviously affect the predicted journey time.

To handle this, the system will need observable sources that can report changes in the user's location, and changes in traffic conditions affecting the proposed journey. Every time either of these reports a change, we will need to produce a new estimated journey time, and a new `IObservable<TrafficInfo>` that will produce a notification when it's time to leave.

Every time we revise our estimate, we want to abandon the previously created `IObservable<TrafficInfo>`. (Otherwise, the user will receive a bewildering number of notifications telling them to leave, one for every time we recalculated the journey time.) We just want to use the latest one. And that's exactly what `Switch` does.

You can see the example for that scenario in the Reaqtor repo. Here, I'm going to present a different, simpler scenario: 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. I really only want results corresponding to the latest search text. 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 (Result Merge) 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 (Result Switch) 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 (with various strategies for deciding what to include and what to discard). The operators in this section 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.

Zip

`Zip` combines pairs of items from two sequences. So its first output is created by combining the first item from one input with the first item from the other. The second output combines the second item from each input. And so on. The name is meant to evoke a zipper on clothing or a bag, which brings the teeth on each half of the zipper together one pair at a time.

Since `Zip` combines pairs of item in strict order, it will complete when the first of the sequences complete. If one of the sequence has reached its end, then even if the other continues to emit values, there will be nothing to pair any of these values with, so `Zip` just unsubscribes at this point and reports completion.

If either of the sequences produces an error, the sequence returned by `Zip` will report that same error.

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, because it can only emit an item when it has one from each source.

Here's an example:

```
// 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!

SequenceEqual

There's another operator that processes pairs of items from two source: `SequenceEqual`. But instead of producing an output for each pair of inputs, this compares each pair, and ultimately produces a single value indicating whether every pair of inputs was equal or not.

In the case where the sources produce different values, `SequenceEqual` produces a single `false` value as soon as it detects this. But if the sources are equal, it can only report this when both have completed because until that happens, it doesn't yet know if there might a difference coming later.

```
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
```

CombineLatest

The `CombineLatest` operator is similar to `Zip` in that it combines pairs of items from its sources. However, instead of pairing the first items, then the second, and so on, `CombineLatest` produces an output any time *either* of its inputs produces a new value. For each new value to emerge from an input, `CombineLatest` uses that along with the most recently seen value from the other input. 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 `DistinctUntilChanged` 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);
```

TODO: these next two sequences were relocated from the now-dropped Sequences of Coincidence chapter. They need editing

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

Right Sequence

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.

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

We could switch it around and have the left expired immediately and the right never expire. The result would then look like this:

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.

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.

Summary

This chapter covered a set of methods that allow us to combine observable sequences. This brings us to a close on Part 2. We've looked at the operators that are mostly concerned with defining the computations we want to perform on the data. In Part 3 we will move onto practical concerns such as managing side effects, error handling, and scheduling. The

PART 3 - Getting pragmatic.

The first part of this book focused on the basic ideas and types of Rx. In the second part, I showed the operators Rx offers, enabling us to define the transformations and computations we want to apply to our source data. This second part was essentially functional programming—Rx's operators are mostly like mathematical functions, in that they will invariably behave in the same way for particular inputs.

They are unaffected by the state of the world around them, and they also do nothing to change its state. In functional programming, such mechanisms are sometimes described as *pure*.

This *purity* can help us understand what our code will do. It means we don't need to know about the state of the rest of our program in order to understand how one particular part functions. However, code that is completely detached from the outside world is unlikely to achieve anything useful. In practice, we need to connect these pure computations with more pragmatic concerns. The Creating Observable Sequences chapter already showed how to define observable streams, so we've already looked at how to connect real world inputs into the world of Rx. But what about the other end? How do we do something useful with the results of our processing?

In some cases, it might be enough to do work inside `IObserver` implementations, or using the callback-based subscription mechanisms you've already seen. However, some situations will demand something more sophisticated. So in this third part of the book, we will look at some of the features Rx offers to help connect processes of the kind we looked at in part 2 with the rest of the world.

Scheduling and Threading

Rx is primarily a system for working with *data in motion* asynchronously. To effectively provide the level of asynchrony that developers require, some level of concurrency control is required. If we are dealing with multiple information sources, they may well generate data concurrently. We may want some degree of parallelism when processing data to achieve our scalability targets, but we will need control over this.

So far, we have managed to avoid any explicit usage of threading or concurrency. There are some methods that we have had to deal with timing to perform their jobs. (For example, `Buffer`, `Delay`, `Sample` must arrange for work to happen on a particular schedule.) Most of this however, has been kindly abstracted away from us. This chapter will look the Rx's scheduling system which offers an elegant system for managing these concerns.

Rx, Threads and Concurrency

Rx does not impose constraints on which threads we use. An `IObservable<T>` is free to invoke its subscribers' `OnNext`/`Completed`/`Error` methods on any thread, perhaps a different thread for each call. Despite this free-for-all, there is one aspect of Rx that prevents chaos: observable sources must obey the Fundamental Rules of Rx Sequences under all circumstances.

When we first explored these rules, we focused on how they determine the ordering of calls into any single observer—there can be any number of calls to `OnNext`, but once either `OnError` or

OnCompleted have been invoked, there must be no further calls. But now that we're looking at concurrency, a different aspect of these rules becomes more important: for any single subscription, an observable source must not make concurrent calls into that subscription's observer. So if a source calls OnNext, it must wait until that call returns before either calling OnNext again, or calling OnError or OnComplete.

The upshot for observers is that as long as your observer is involved in just one subscription, it will only ever be asked to deal with one thing at a time. If doesn't matter if the source to which it is subscribed is a long and complex processing chain involving many different operators. Even if you build that source by combining multiple inputs (e.g., using Merge), the fundamental rules require that if you called Subscribe just once on a single IObservable<T>, that source is never allowed to make multiple concurrent calls into your IObserver<T> methods.

So although each call might come in on a different thread, the calls are strictly sequential (unless a single observer is involved in multiple subscriptions).

Rx operators that receive incoming notifications as well as producing them will notify their observers on whatever thread the incoming notification happened to arrive on. Suppose you have a sequence of operators like this:

```
source
    .Where(x => x.MessageType == 3)
    .Buffer(10)
    .Take(20)
    .Subscribe(x => Console.WriteLine(x));
```

When that call to Subscribe happens, we end up with a chain of observers—the Rx-supplied observer that will invoke our callback was passed to the observable returned by Take, which will in turn create an observer that subscribed to the observable returned by Buffer, which will in turn create an observer subscribed to the Where observable, which will have created yet another observer which is subscribed to source.

So when source decides to produce an item, it will invoke the Where operator's observer's OnNext. That will invoke the predicate, and if the MessageType is indeed 3, the Where observer will call OnNext on the Buffer's observer, and it will do this on the same thread. The Where observer's OnNext isn't going to return until the Buffer observer's OnNext returns. Now if the Buffer observer determines that it has completely filled a buffer (e.g., it just received its 10th item), then it is also not going to return yet—it's going to invoke the Take observer's OnNext, and as long as Take hasn't already received 20 items, it's going to call OnNext on the Rx-supplied observer that will invoke our callback.

So for the source notifications that make it all the way through to that Console.WriteLine in the callback passed to subscribe, we end up with a lot of nested calls on the stack:

```
`source` calls:
  `Where` observer, which calls:
    `Buffer` observer, which calls:
      `Take` observer, which calls:
        `Subscribe` observer, which calls our lambda
```

This is all happening on one thread. Most Rx operators don't have any one particular thread that they call home. They just do their work on whatever thread the call comes in on. This makes Rx pretty efficient—passing data from one operator to the next merely involves a method call, and those are pretty fast. (In fact, there are typically a few more layers. Rx tends to add a few wrappers to handle errors and early unsubscription. So the call stack will look a bit more complex than what I've just shown. But it's still typically all just method calls.)

You will sometimes hear Rx described as having a *free threaded* model. All that means is that operators don't generally care what thread they use. As we will see, there are exceptions, but this direct calling by one operator of the next is the norm.

An upshot of this is that it's typically the original source that determine which thread is used. This next example verifies by creating a subject, then calling `OnNext` on various threads and reporting the thread id.

```
Console.WriteLine($"Main thread: {Environment.CurrentManagedThreadId}");
var subject = new Subject<string>();

subject.Subscribe(
    m => Console.WriteLine($"Received {m} on thread:
↪  {Environment.CurrentManagedThreadId}"));

object sync = new();
ParameterizedThreadStart notify = arg =>
{
    string message = arg?.ToString() ?? "null";
    Console.WriteLine(
        $"OnNext({message}) on thread:
↪  {Environment.CurrentManagedThreadId}");
    lock (sync)
    {
        subject.OnNext(message);
    }
};

notify("Main");
```

```
new Thread(notify).Start("First worker thread");
new Thread(notify).Start("Second worker thread");
```

Output:

```
Main thread: 1
OnNext(Main) on thread: 1
Received Main on thread: 1
OnNext(First worker thread) on thread: 10
Received First worker thread on thread: 10
OnNext(Second worker thread) on thread: 11
Received Second worker thread on thread: 11
```

Note that the handler passed to `Subscribe` was called back on the same thread that made the call to `OnNext`. This is straightforward and efficient. However, things are not always this simple.

Timed invocation

Some notifications will not be the immediate result of a source providing an item. For example, Rx offers a [`Delay`] operator, which time shifts the delivery of items. This next example is based on the preceding one, with the main difference being that we no longer subscribe directly to the source. We go via `Delay`:

```
Console.WriteLine($"Main thread: {Environment.CurrentManagedThreadId}");
var subject = new Subject<string>();

subject
    .Delay(TimeSpan.FromSeconds(0.25))
    .Subscribe(
    m => Console.WriteLine($"Received {m} on thread:
↪   {Environment.CurrentManagedThreadId}"));

object sync = new();
ParameterizedThreadStart notify = arg =>
{
    string message = arg?.ToString() ?? "null";
    Console.WriteLine(
        $"OnNext({message}) on thread:
↪   {Environment.CurrentManagedThreadId}");
    lock (sync)
    {
```

```
        subject.OnNext(message);
    }
};

notify("Main 1");
Thread.Sleep(TimeSpan.FromSeconds(0.1));
notify("Main 2");
Thread.Sleep(TimeSpan.FromSeconds(0.3));
notify("Main 3");
new Thread(notify).Start("First worker thread");
Thread.Sleep(TimeSpan.FromSeconds(0.1));
new Thread(notify).Start("Second worker thread");

Thread.Sleep(TimeSpan.FromSeconds(2));
```

This also waits for a while between sending source items, so we can see the effect of `Delay`. Here's the output:

```
Main thread: 1
OnNext(Main 1) on thread: 1
OnNext(Main 2) on thread: 1
Received Main 1 on thread: 12
Received Main 2 on thread: 12
OnNext(Main 3) on thread: 1
OnNext(First worker thread) on thread: 13
OnNext(Second worker thread) on thread: 14
Received Main 3 on thread: 12
Received First worker thread on thread: 12
Received Second worker thread on thread: 12
```

Notice that in this case every `Received` message is on thread id 12, which is different from any of the three threads on which the notifications were raised.

This shouldn't be entirely surprising. The only way Rx could have used the original thread here would be for `Delay` to block the thread for the specified time (a quarter of a second here) before forwarding the call. This would be unacceptable for most scenarios, so instead, the `Delay` operator arranges for a callback to occur after a suitable delay. As you can see from the output, these all seems to happen on one particular thread. No matter which thread calls `OnNext`, the delayed notification arrives on thread id 12. But this is not a thread created by the `Delay` operator. This is happening because `Delay` is using a *scheduler*.

Schedulers

Schedulers do three things:

- determining the context in which to execute work (e.g., a certain thread)
- deciding when to execute work (e.g., immediately, or deferred)
- keeping track of time

Here's a simple example to explore the first two of those:

```
Console.WriteLine($"Main thread: {Environment.CurrentManagedThreadId}");

Observable
    .Range(1, 5)
    .Subscribe(
    m => Console.WriteLine($"Received {m} on thread:
↪   {Environment.CurrentManagedThreadId}"));

Console.WriteLine("Subscribe returned");
Console.ReadLine();
```

It might not be obvious that this has anything to do with scheduling, but in fact, `Range` always uses a scheduler to do its work. We've just let it use its default scheduler. Here's the output:

```
Main thread: 1
Received 1 on thread: 1
Received 2 on thread: 1
Received 3 on thread: 1
Received 4 on thread: 1
Received 5 on thread: 1
Subscribe returned
```

Looking at the first two items in our list of what schedulers do, we can see that the context in which this has executed the work is the thread on which I called `Subscribe`. And as for when it has decided to execute the work, it has decided to do it all before `Subscribe` returns. So you might think that `Range` immediately produces all of the items we've asked for and then returns. However, it's not quite as simple as that. Let's look at what happens if we have multiple `Range` instances running simultaneously. This introduces an extra operator—a `SelectMany` that calls `Range` again:

```
Observable
    .Range(1, 5)
    .SelectMany(i => Observable.Range(i * 10, 5))
```

```
    .Subscribe(
    m => Console.WriteLine($"Received {m} on thread:
↪   {Environment.CurrentManagedThreadId}"));
```

The output shows that Range doesn't in fact necessarily produce all of its items immediately:

```
Received 10 on thread: 1
Received 11 on thread: 1
Received 20 on thread: 1
Received 12 on thread: 1
Received 21 on thread: 1
Received 30 on thread: 1
Received 13 on thread: 1
Received 22 on thread: 1
Received 31 on thread: 1
Received 40 on thread: 1
Received 14 on thread: 1
Received 23 on thread: 1
Received 32 on thread: 1
Received 41 on thread: 1
Received 50 on thread: 1
Received 24 on thread: 1
Received 33 on thread: 1
Received 42 on thread: 1
Received 51 on thread: 1
Received 34 on thread: 1
Received 43 on thread: 1
Received 52 on thread: 1
Received 44 on thread: 1
Received 53 on thread: 1
Received 54 on thread: 1
Subscribe returned
```

The first nested Range produces by the SelectMany callback produces a couple of values (10 and 11) but then the second one manages to get its first value out (20) before the first one produces its third (12). You can see there's some interleaving of progress here. So although the context in which work is executed continues to be the thread on which we invoked Subscribe, the second choice the scheduler has to make—when to execute the work—is more subtle than it first seems. This tells us that Range is not as simple as this naive implementation:

```
public static IObservable<int> NaiveRange(int start, int count)
{
    return System.Reactive.Linq.Observable.Create<int>(obs =>
    {
        for (int i = 0; i < count; i++)
        {
            obs.OnNext(start + i);
        }

        return Disposable.Empty;
    });
}
```

If Range worked like that, this code would produce all of the items from the first range returned by the `SelectMany` callback before moving on to the next. In fact, Rx does provide a scheduler that would give us that behaviour if that's what we want. This example passes `ImmediateScheduler.Instance` to the nested `Observable.Range` call:

```
Observable
    .Range(1, 5)
    .SelectMany(i => Observable.Range(i * 10, 5,
    ↪  ImmediateScheduler.Instance))
    .Subscribe(
    m => Console.WriteLine($"Received {m} on thread:
↪   {Environment.CurrentManagedThreadId}"));
```

Here's the outcome:

```
Received 10 on thread: 1
Received 11 on thread: 1
Received 12 on thread: 1
Received 13 on thread: 1
Received 14 on thread: 1
Received 20 on thread: 1
Received 21 on thread: 1
Received 22 on thread: 1
Received 23 on thread: 1
Received 24 on thread: 1
Received 30 on thread: 1
Received 31 on thread: 1
Received 32 on thread: 1
Received 33 on thread: 1
```

```
Received 34 on thread: 1
Received 40 on thread: 1
Received 41 on thread: 1
Received 42 on thread: 1
Received 43 on thread: 1
Received 44 on thread: 1
Received 50 on thread: 1
Received 51 on thread: 1
Received 52 on thread: 1
Received 53 on thread: 1
Received 54 on thread: 1
Subscribe returned
```

By specifying `ImmediateScheduler.Instance` we've asked for a particular policy: this invokes all work on the caller's thread, and it always does so immediately, avoiding introducing any concurrency. There are a couple of reasons this is not `Range`'s default. (Its default is `Scheduler.CurrentThread`, which always returns an instance of `CurrentThreadScheduler`.) First, `ImmediateScheduler.Instance` can end up causing fairly deep call stacks—most of the other schedulers maintain work queues, so if one operator decides it has new work to do while another is in the middle of doing something (e.g., a nested `Range` operator decides to start emitting its values), instead of starting that work immediately (which will involve invoking the method that will do the work) that work can be put on a queue instead, enabling the work already in progress to finish before starting on the next thing. Using the immediate scheduler everywhere can cause stack overflows when queries become complex. The second reason `Range` does not use the immediate scheduler is so that when multiple observables are all active at once, they can all make some progress—`Range` produces all of its items as quickly as it can, so it could end up starving other operators of CPU time if it didn't use a scheduler that enabled operators to take it in turns.

Notice that the `Subscribe returned` message appears last. So although the `CurrentThreadScheduler` isn't quite as eager as the immediate scheduler, it still won't return to its called until it has completed all outstanding work. It maintains a work queue, enabling slightly more fairness, and avoiding stack overflows, but as soon as anything asks the `CurrentThreadScheduler` to do something, it won't return until it has drained its queue.

Not all schedulers have this characteristic. Here's a variation on the earlier example in which we have just a single call to `Range`, without any nested observables. This time I'm asking it to use the `TaskPoolScheduler`.

```
Observable
    .Range(1, 5, TaskPoolScheduler.Default)
```

```
    .Subscribe(
    m => Console.WriteLine($"Received {m} on thread:
↪   {Environment.CurrentManagedThreadId}"));
```

This makes a different decision about the context in which to run work from the immediate and current thread schedulers, as we can see from its output:

```
Main thread: 1
Subscribe returned
Received 1 on thread: 12
Received 2 on thread: 12
Received 3 on thread: 12
Received 4 on thread: 12
Received 5 on thread: 12
```

Notice that the notifications all happened on a different thread (with id 12) than the thread on which we invoked `Subscribe` (id 1). That's because the `TaskPoolScheduler`'s defining feature is that it invokes all work through the Task Parallel Library's (TPL) task pool. That's why we see a different thread id: the task pool doesn't own our application's main thread. In this case, it hasn't seen any need to spin up multiple threads. That's reasonable, there's just a single source here providing item one at a time. It's good that we didn't get more threads in this case—the thread pool is at its most efficient when a single thread processes work items sequentially, because it avoids context switching overheads, and since there's no actual scope for concurrent work here, we would gain nothing if it had created multiple threads in this case.

There's one other very significant difference with this scheduler: notice that the call to `Subscribe` returned before *any* of the notifications were made it through to our observer. That's because this is the first scheduler we've looked at that will introduce real parallelism. The `ImmediateScheduler` and `CurrentThreadScheduler` will never spin up new threads by themselves, no matter how much the operators executing might want to perform concurrent operations. And although the `TaskPoolScheduler` determined that there's no need for it to create multiple threads, the one thread it did create is a different thread from the application's main thread, meaning that the main thread can continue to run in parallel with this subscription. Since `TaskPoolScheduler` isn't going to do any work on the thread that initiated the work, it can return as soon as it has queued the work up, enabling the `Subscribe` method to return immediately.

What if we use the `TaskPoolScheduler` in the example with nested observables? This uses it just on the inner call to `Range`, so the outer one will still use the default `CurrentThreadScheduler`:

```
Observable
    .Range(1, 5)
    .SelectMany(i => Observable.Range(i * 10, 5,
    ↪  TaskPoolScheduler.Default))
```

```
    .Subscribe(
    m => Console.WriteLine($"Received {m} on thread:
↪   {Environment.CurrentManagedThreadId}"));
```

Now we can see a few more threads getting involved:

```
Received 10 on thread: 13
Received 11 on thread: 13
Received 12 on thread: 13
Received 13 on thread: 13
Received 40 on thread: 16
Received 41 on thread: 16
Received 42 on thread: 16
Received 43 on thread: 16
Received 44 on thread: 16
Received 50 on thread: 17
Received 51 on thread: 17
Received 52 on thread: 17
Received 53 on thread: 17
Received 54 on thread: 17
Subscribe returned
Received 14 on thread: 13
Received 20 on thread: 14
Received 21 on thread: 14
Received 22 on thread: 14
Received 23 on thread: 14
Received 24 on thread: 14
Received 30 on thread: 15
Received 31 on thread: 15
Received 32 on thread: 15
Received 33 on thread: 15
Received 34 on thread: 15
```

Since we have only a single observer in this example, the rules of Rx require it to be given items one at a time, so in practice there wasn't really any scope for parallelism here, but the more complex structure would have resulted in more work items initially going into the scheduler's queue than in the preceding example, which is probably why the work got picked up by more than one thread this time. In practice most of these threads would have spent most of their time blocked in the code inside `SelectMany` that ensures that it delivers one item at a time to its target observer. It's perhaps a little

surprising that the items are not more scrambled. The subranges themselves seem to have emerged in a random order, but it has almost produced the items sequentially within each subrange (with item 14 being the one exception to that). This is a quirk relating to the way in which `Range` interacts with the `TaskPoolScheduler`.

I've not yet talked about the scheduler's third job: keeping track of time. This doesn't arise with `Range` because it attempts to produce all of its items as quickly as it can. But for the `Delay` operator I showed in the Timed Invocation section, timing is obviously a critical element. In fact this would be a good point to show the API that schedulers offer:

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

You can see that all but one of these is concerned with timing. Only the first `Schedule` overload is not—operators call this when they want to schedule work to run as soon as the scheduler will allow. That's the overload used by `Range`. (Strictly speaking, `Range` interrogates the scheduler to find out whether it supports long-running operations, in which an operator can temporary control of a thread for an extended period. It prefers to use that when it can because it tends to be more efficient than submitting work to the scheduler for every single item it wishes to produce. The `TaskPoolScheduler` does support long running operations, which explains the slightly surprising output we saw earlier, but the `CurrentThreadScheduler`, `Range`'s default choice, does not. So by default, `Range` will invoke that first `Schedule` overload once for each item it wishes to produce.)

`Delay` uses the second overload. The exact implementation is quite complex (mainly because of how it catches up efficiently when a busy source causes it to fall behind) but in essence, each time a new item arrives into the `Delay` operator, it schedules a work item to run after the configured delay, so that it can supply that item to its subscriber with the expected time shift.

Schedulers have to be responsible for managing time, because .NET has several different timer mechanisms, and the choice of timer is often determined by the context in which you want to handle a timer callback. Since schedulers determine the context in which work runs, that means them must also choose the timer type. For example, UI frameworks typically provide timers that invoke their callbacks in a context suitable for making updates to the user interface. Rx provides some UI-framework-specific schedulers that use these timers, but these would be inappropriate choices for other scenarios. So

each scheduler uses a timer suitable for the context in which it is going to run work items.

There's a useful upshot of this: because `IScheduler` provides an abstraction for timing-related details, it is possible to virtualize time. This is very useful for testing. If you look at the extensive test suite in the Rx repository you will find that there are many tests that verify timing-related behaviour. If these ran in real-time, the test suite would take far too long to run, and would also be likely to produce the odd spurious failure, because background tasks running on the same machine as the tests will occasionally change the speed of execution in a way that might confuse the test. Instead, these tests use a specialized scheduler that provides complete control over the passage of time.

Notice that all three `IScheduler.Schedule` methods require a callback. A scheduler will invoke this at the time and in the context that it chooses. A scheduler callback takes another `IScheduler` as its first argument. This enables

Rx supplies several schedulers. The following sections describe the most widely used ones.

ImmediateScheduler

`ImmediateScheduler` is the simplest scheduler Rx offers. As you saw in the preceding sections, whenever it is asked to schedule some work, it just runs it immediately. It does this inside its `IScheduler.Schedule` method.

This is a very simple strategy, and it makes `ImmediateScheduler` very efficient. For this reason, many operators default to using `ImmediateScheduler`. However, it can be problematic with operators that instantly produce multiple items, especially when the number of items might be large. For example, Rx defines the `ToObservable` extension method for `IEnumerable<T>`. When you subscribe to an `IObservable<T>` returned by this, it will start iterating over the collection immediately, and if you were to tell it to use the `ImmediateScheduler`, `Subscribe` would not return until it reached the end of the collection. That would obviously be a problem for an infinite sequence, and it's why operators of this kind do not use `ImmediateScheduler` by default.

The `ImmediateScheduler` also has potentially surprising behaviour when you invoke the `Schedule` overload that takes a `TimeSpan`. This asks the scheduler to run some work after the specified length of time. The way it achieves this is to call `Thread.Sleep`. With most of Rx's schedulers, this overload will arrange for some sort of timer mechanism to run the code later, enabling the current thread to get on with its business, but `ImmediateScheduler` is true to its name here, in that it refuses to engage in such deferred execution. It just blocks the current thread until it is time to do the work. This means that time-based observables like those returned by `Interval` would work if you specified this scheduler, but at the cost of preventing the thread from doing anything else.

The `Schedule` overload that takes a `DateTime` is slightly different. If you specify a time less than 10 seconds into the future, it will block the calling thread like it does when you use `TimeSpan`. But if you

pass a DateTime that is further into the future, it gives up on immediate execution, and falls back to using a timer.

CurrentThreadScheduler

The CurrentThreadScheduler is very similar to the ImmediateScheduler. The difference is how it handles requests to schedule work when an existing work item is already being handled on the current thread. This can happen if you chain together multiple operators that use schedulers to do their work.

To understand what happens, it's helpful to know how sources that produce multiple items in quick succession, such as the ToObservable extension method for IEnumerable<T> or Observable.Range, use schedulers. These kinds of operators do not use normal for or foreach loops. They typically schedule a new work item for each iteration. Whereas the ImmediateScheduler will run such work immediately, the CurrentThreadScheduler checks to see if it is already processing a work item. That happens with this example from earlier:

```
Observable
    .Range(1, 5)
    .SelectMany(i => Observable.Range(i * 10, 5))
    .Subscribe(
        m => Console.WriteLine($"Received {m} on thread:
↪   {Environment.CurrentManagedThreadId}"));
```

Let's follow exactly what happens here. First, assume that this code is just running normally and not in any unusual context—perhaps inside the Main entry point of a program. When this code calls Subscribe on the IObservable<int> returned by SelectMany, that will in turn will call Subscribe on the IObservable<int> returned by the first Observable.Range, which will in turn schedule a work item for the generation of the first value in the range (1).

The Range operator uses the CurrentThreadScheduler by default, and that will ask itself "Am I already in the middle of handling some work item on this thread?" In this case the answer will be no, so it will run the work item immediately (before returning from the Schedule call made by the Range operator). The Range operator will then produce its first value, calling OnNext on the IObserver<int> that the SelectMany operator provided when it subscribed to the range.

The SelectMany operator's OnNext method will now invoke its lambda, passing in the argument supplied (the value 1 from the Range operator). You can see from the example above that this lambda calls Observable.Range again, returning a new IObservable<int>. SelectMany will immediately subscribe to this (before returning from its OnNext). This is the second time this code has ended up calling Subscribe on an IObservable<int> returned by a Range

(but it's a different instance than the last time), and Range will once again default to using the CurrentThreadScheduler, and will once again schedule a work item to perform the first iteration.

So once again,the CurrentThreadScheduler will ask itself "Am I already in the middle of handling some work item on this thread?" But this time, the answer will be yes. And this is where the behaviour is different than ImmediateScheduler. The CurrentThreadScheduler maintains a queue of work for each thread that it gets used on, and in this case it just adds the newly scheduled work to the queue, and returns back to the SelectMany operators OnNext.

SelectMany has now completed its handling of this item (the value 1) from the first Range, so its OnNext returns. At this point, this outer Range operator schedules another work item. Again, the CurrentThreadScheduler will detect that it is currently running a work item, so it just adds this to the queue.

Having scheduled the work item that is going to generate its second value (2), the Range operator returns. Remember, the code in the Range operator that was running at this point was the callback for the first scheduled work item, so it's returning to the CurrentThreadScheduler—we are back inside its Schedule method (which was invoked by the range operator's Subscribe method).

At this point, the CurrentThreadScheduler does not return from Schedule because it checks its work queue, and will see that there are now two items in the queue. (There's the work item that the nested Range observable scheduled to generate its first value, and there's also the work item that the top-level Range observable just scheduled to generate its second value.) The CurrentThreadScheduler will now execute the first of these—the nested Range operator now gets to generate its first value (which will be 10), so it calls OnNext on the observer supplied by SelectMany, which will then call its observer, which was supplied thanks to the top-level call to Subscribe in the example. And that observer will just call the lambda we passed to Subscribe, causing our Console.WriteLine to run. After that returns, the nested Range operator will schedule another work item to generate its second item. Again, the CurrentThreadScheduler will realise that it's already in the middle of handling a work item on this thread, so it just puts it in the queue and then returns immediately from Schedule. The nested Range operator is now done for this iteration so it returns back to the scheduler. The scheduler will now pick up the next item in the queue, which in this case it the work item added by the top-level Range to produce the second item.

And so it continues. This queuing of work items when work is already in progress is what enables multiple observable sources to make progress in parallel.

By contrast, the ImmediateScheduler runs new work items immediately, which is why we don't see this parallel progress.

(To be strictly accurate, there are certain scenarios in which ImmediateScheduler can't run work

immediately. In these iterative scenarios, it actually supplies a slightly different scheduler that the operators use to schedule all work after the first item, and this checks whether it's being asked to process multiple work items simultaneously. If it is, it falls back to a queuing strategy similar to `CurrentThreadScheduler`, except it's a queue local to the initial work item, instead of a per-thread queue. This prevents problems due to multithreading, and it also avoids stack overflows that would otherwise occur when an iterative operator schedules a new work item inside the handler for the current work item. Since the queue is not shared across all work in the thread, this still has the effect of ensuring that any nested work queued up by a work item completes before the call to `Schedule` returns. So even when this queueing kicks in, we typically don't see interleaving of work from separate source like we do with `CurrentThreadScheduler`. For example, if we told the nested `Range` to use `ImmediateScheduler`, this queueing behaviour would kick in as `Range` starts to iterate, but because the queue is local to initial work item executed by that nested `Range`, it will end up producing all of the nested `Range` items before returning.)

DefaultScheduler

The `DefaultScheduler` is intended for work that may need to be spread out over time, or where you are likely to want concurrent execution. These features mean that this can't guarantee to run work on any particular thread, and in practice it schedulers work via the CLR's thread pool. This is the default scheduler for all of Rx's time-based operators, and also for the `Observable.ToAsync` operator that can wrap a .NET method as an `IObservable<T>`.

Although this scheduler is useful if you would prefer work not to happen on your current thread—perhaps you're writing an application with a user interface and your code is running on the thread responsible for updating the UI and responding to user input—the fact that it can end up running work on any thread may make like complicated. What if you want all the work to happen on one thread, just not the thread you're on now? There's another scheduler for that.

EventLoopScheduler

The `EventLoopScheduler` provides one-at-a-time scheduling, queuing up newly scheduled work items. This is similar to how the `CurrentThreadScheduler` operates if you use it from just one thread. The difference is that `EventLoopScheduler` creates a dedicated thread for this work instead of using whatever thread you happen to schedule the work from.

Unlike the schedulers we've examined so far, there is no static property for obtaining an `EventLoopScheduler`. That's because each one has its own thread, so you need to create one explicitly. It offers two constructors:

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

The first creates a thread for you. The second lets you control the thread creation process—it invokes the callback you supply, and it will pass this its own callback that you are required to run on the newly created thread.

The `EventLoopScheduler` implements `IDisposable`, and calling Dispose will allow the thread to terminate. This can work nicely with the `Observable.Using` method. The following example shows how to use an `EventLoopScheduler` to iterate over all contents of an `IEnumerable<T>` on a dedicated thread, ensuring that the thread exits once we have finished:

```
IEnumerable<int> xs = GetNumbers();
Observable
    .Using(
        () => new EventLoopScheduler(),
        scheduler => xs.ToObservable(scheduler))
    .Subscribe(...);
```

NewThreadScheduler

The `NewThreadScheduler` creates a new thread to execute every work item it is given. This is unlikely to make sense in most scenarios. However, it might be useful in cases where you want to execute some long running work, and represent its completion through an `IObservable<T>`. The `Observable.ToAsync` does exactly this, and will normally use the `DefaultScheduler`, meaning it will run the work on a thread pool thread. But if the work is likely to take more than second or two, the thread pool may not be a good choice, because it is optimized for short execution times, and its heuristics for managing the size of the thread pool are not designed with long-running operations in mind. The `NewThreadScheduler` may be a better choice in this case.

Although each call to `Schedule` creates a new thread, the `NewThreadScheduler` passes a different scheduler into work item callbacks, meaning that anything that attempts to perform iterative work will not create a new thread for every iteration. For example, if you use `NewThreadScheduler` with `Observable.Range`, you will get a new thread each time you subscribe to the resulting `IObservable<int>`, but you won't get a new thread for each item, even though `Range` does schedule a new work item for each value it produces. It schedules these per-value work items through the nested scheduler supplied to the work item callback, and the nested scheduler that `NewThreadScheduler` supplies in these cases invokes all such nested work items on the same thread.

SynchronizationContextScheduler

This invokes all work through a `SynchronizationContext`. This is useful in user interface scenarios—most .NET client-side user interface frameworks make a `SynchronizationContext` available that can be used to invoke callbacks in a context suitable for making changes to the UI. (Typically this involves invoking them on the correct thread, but individual implementations can decide what constitutes the appropriate context.)

TaskPoolScheduler

Invokes all work through the TPL task pool. The TPL task pool is newer than the CLR thread pool. Rx's DefaultScheduler uses the older CLR thread pool for backwards compatibility reasons, but the TPL task pool can offer performance benefits in some scenarios.

ThreadPoolScheduler

Invokes all work through the thread pool. This type is a historical artifact, dating back to when not all platforms offered the same kind of thread pool. In almost all cases, you should use the `DefaultScheduler`. The only scenario in which using `ThreadPoolScheduler` makes any difference is when writing UWP applications. The UWP target of `System.Reactive` v6.0 provides a different implementation of this class than you get for all other targets. It uses `Windows.System.Threading.ThreadPool` whereas all other targets use `System.Threading.ThreadPoo` The UWP version provides properties letting you configure some features specific to the UWP thread pool.

In practice it's best to avoid this class in new code. The only reason the UWP target had a different implementation was that UWP used not to provide `System.Threading.ThreadPool`. But that changed when UWP added support for .NET Standard 2.0 in Windows version 10.0.19041. There is no longer any good reason for there to be a UWP-specific `ThreadPoolScheduler`, and it's a source of confusion that this type is quite different in the UWP target but it has to remain for backwards compatibility purposes. (It may well be deprecated because Rx 7 will be addressing some problems arising from the fact that the `System.Reactive` component currently has direct dependencies on UI frameworks.) If you use the `DefaultScheduler` you will be using the `System.Threading.ThreadPool` no matter which platform you are running on.

UI Framework Schedulers: ControlScheduler, DispatcherScheduler and CoreDispatcherScheduler

Although the `SynchronizationContextScheduler` will work for all widely used client-side UI frameworks in .NET, Rx offers more specialized schedulers. `ControlScheduler` is for Windows Forms applications, `DispatcherScheduler` for WPF, and `CoreDispatcherScheduler` for UWP.

These more specialized types offer two benefits. First, you don't necessarily have to be on the target UI thread to obtain an instance of these schedulers. Whereas with `SynchronizationContextScheduler` the only way you can generally obtain the `SynchronizationContext` this requires is by retrieving `SynchronizationContext.Current` while running on the UI thread. But these other UI-framework-specific schedulers can be passed a suitable `Control`, `Dispatcher` or `CoreDispatcher`, which it's possible to obtain from a non-UI thread. Second, `DispatcherScheduler` and `CoreDispatcherScheduler` provide a way to use the prioritisation mechanism supported by the `Dispatcher` and `CoreDispatcher` types.

Test Schedulers

The Rx libraries define several schedulers that virtualize time, including `HistoricalScheduler`, `TestScheduler`, `VirtualTimeScheduler`, and `VirtualTimeSchedulerBase`. We will look at this sort of scheduler in the Testing chapter.

SubscribeOn and ObserveOn

So far, I've talked about why some Rx sources need access to schedulers—this is necessary for timing-related behaviour, and also for sources that produce items as quickly as possible. But remember, schedulers control three things:

- determining the context in which to execute work (e.g., a certain thread)
- deciding when to execute work (e.g., immediately, or deferred)
- keeping track of time

The discussion so far as mostly focused on the 2nd and 3rd features. When it comes to our own application code, we are most likely to use schedulers to control that first aspect. Rx defines two extension methods to `IObservable<T>` for this: `SubscribeOn` and `ObserveOn`. Both methods take an `IScheduler` and return an `IObservable<T>` so you can chain more operators downstream of these.

These methods do what their names suggest. If you use `SubscribeOn`, then when you call `Subscribe` on the resulting `IObservable<T>` it arranges to call the original `IObservable<T>`'s `Subscribe` method via the specified scheduler. Here's an example:

```
Console.WriteLine($"Main thread: {Environment.CurrentManagedThreadId}");

Observable
    .Interval(TimeSpan.FromSeconds(1))
    .SubscribeOn(new EventLoopScheduler((start) =>
    {
        Thread t = new(start) { IsBackground = false };
        Console.WriteLine($"Created thread for EventLoopScheduler:
↪   {t.ManagedThreadId}");
        return t;
    }))
    .Subscribe(
        tick => Console.WriteLine($"{DateTime.Now}-
↪   {Environment.CurrentManagedThreadId}: Tick
↪   {tick}"));

Console.WriteLine($"{DateTime.Now}-{Environment.CurrentManagedThreadId}:
↪   Main thread exiting");
```

This calls `Observable.Interval` (which uses `DefaultScheduler` by default), but instead of subscribing directly to this, it first takes the `IObservable<T>` returned by `Interval` and invokes `SubscribeOn`. I've used an `EventLoopScheduler`, and I've passed it a factory callback for the thread that it will use to ensure that it is a non-background thread. (By default `EventLoopScheduler` creates itself a background thread, meaning that the thread won't force the process to stay alive. Normally that's what you'd want but I'm changing that in this example to show what's happening.)

When I call `Subscribe` on the `IObservable<long>` returned by `SubscribeOn`, it calls `Schedule` on the `EventLoopScheduler` that I supplied, and in the callback for that work item, it then calls `Subscribe` on the original `Interval` source. So the effect is that the subscription to the underlying source doesn't happen on my main thread, it happens on the thread created for my `EventLoopScheduler`. Running the program produces this output:

```
Main thread: 1
Created thread for EventLoopScheduler: 12
21/07/2023 14:57:21-1: Main thread exiting
21/07/2023 14:57:22-6: Tick 0
21/07/2023 14:57:23-6: Tick 1
```

```
21/07/2023 14:57:24-6: Tick 2
...
```

Notice that my application's main thread exits before the source begins producing notifications. But also notice that the thread id for the newly created thread is 12, and yet my notifications are coming through on a different thread, with id 6! What's happening?

This often catches people out. The scheduler on which you subscribe to an observable source doesn't necessarily have any impact on how the source behaves once it is up and running. Remember earlier that I said `Observable.Interval` uses `DefaultScheduler` by default? Well we've not specified a scheduler for the `Interval` here, so it will be using that default. It doesn't care what context we invoke its `Subscribe` method from. So really, the only effect of introducing the `EventLoopScheduler` here has been to keep the process alive even after its main thread exits. That scheduler thread never actually gets used again after it makes its initial `Subscribe` call into the `IObservable<long>` returned by `Observable.Interval`. It just sits patiently waiting for further calls to `Schedule` that never come.

Not all sources are completely unaffected by the context in which their `Subscribe` is invoked, though. If I were to replace this line:

```
    .Interval(TimeSpan.FromSeconds(1))
```

with this:

```
    .Range(1, 5)
```

then we get this output:

```
Main thread: 1
Created thread for EventLoopScheduler: 12
21/07/2023 15:02:09-12: Tick 1
21/07/2023 15:02:09-1: Main thread exiting
21/07/2023 15:02:09-12: Tick 2
21/07/2023 15:02:09-12: Tick 3
21/07/2023 15:02:09-12: Tick 4
21/07/2023 15:02:09-12: Tick 5
```

Now all the notifications are coming in on thread 12, the thread created for the `EventLoopScheduler`. Note that even here, `Range` isn't using that scheduler. The difference is that `Range` defaults to `CurrentThreadScheduler`, so it will generate its outputs from whatever thread you happen to call it from. So even though it's not actually using the `EventLoopScheduler`, it does end up using that scheduler's thread, because we used that scheduler to subscribe to the `Range`.

So this illustrates that SubscribeOn is doing what it promises: it does determine the context from which Subscribe is invoked. It's just that it doesn't always matter what context that is. If Subscribe does non-trivial work, it can matter. For example, if you use Observable.Create to create a custom sequence, SubscribeOn determines the context in which the callback you passed to Create is invoked. But Rx doesn't have a concept of a 'current' scheduler—there's no way to ask "which scheduler was I invoked from?"—so Rx operators don't just their scheduler from the context on which they were subscribed.

When it comes to emitting items, most of the sources Rx supplies fall into one of three categories. First, operators that produce outputs in response to inputs from an upstream source (e.g., Where, Select, or GroupBy) generally call their observers methods from inside their own OnNext. So whatever context their source observable was running in when it called OnNext, that's the context the operator will use when calling its observer. Second, operators that produce items either iteratively, or based on timing will use a scheduler (either explicitly supplied, or a default type when none is specified). Third, some sources just produce items from whatever context they like. For example, if an async method uses await and specifies ConfigureAwait(false) then it could be on more or less any thread and in any context after the await completes, and it might then go on to invoke OnNext on an observer.

As long as a source follows the fundamental rules of Rx sequences, it's allowed to invoke its observer's methods from any context it likes. It can choose to accept a scheduler as input and to use that, but it's under no obligation to. And if you have such an unruly source that you'd like to tame, that's where the ObserveOn extension method comes in. Consider the following rather daft example:

```
Observable
    .Interval(TimeSpan.FromSeconds(1))
    .SelectMany(tick => Observable.Return(tick,
    ↪  NewThreadScheduler.Default))
    .Subscribe(
        tick => Console.WriteLine($"{DateTime.Now}-
↪   {Environment.CurrentManagedThreadId}: Tick
↪   {tick}"));
```

This deliberately causes every notification to arrive on a different thread, as this output shows:

```
Main thread: 1
21/07/2023 15:19:56-12: Tick 0
21/07/2023 15:19:57-13: Tick 1
21/07/2023 15:19:58-14: Tick 2
21/07/2023 15:19:59-15: Tick 3
...
```

(It's achieving this by calling `Observable.Return` for every single tick that emerges from `Interval`, and telling `Return` to use the `NewThreadScheduler`. Each such call to `Return` will create a new thread. This is a terrible idea, but it is an easy way to get a source that calls from a different context every time.) If I want to impose some order, I can add a call to `ObserveOn`:

```
Observable
    .Interval(TimeSpan.FromSeconds(1))
    .SelectMany(tick => Observable.Return(tick,
    ↪  NewThreadScheduler.Default))
    .ObserveOn(new EventLoopScheduler())
    .Subscribe(
        tick => Console.WriteLine($"{DateTime.Now}-
↪   {Environment.CurrentManagedThreadId}: Tick
↪   {tick}"));
```

I've created an `EventLoopScheduler` here because it creates a single thread, and runs every scheduled work item on that thread. The output now shows the same thread id (13) every time:

```
Main thread: 1
21/07/2023 15:24:23-13: Tick 0
21/07/2023 15:24:24-13: Tick 1
21/07/2023 15:24:25-13: Tick 2
21/07/2023 15:24:26-13: Tick 3
...
```

So although each new observable created by `Observable.Return` creates a brand new thread, `ObserveOn` ensures that my observer's `OnNext` (and `OnCompleted` or `OnError` in cases where those are called) is invoked via the specified scheduler.

SubscribeOn and ObserveOn in UI applications

If you're using Rx in a user interface, `ObserveOn` is useful when you are dealing with information sources that don't provide notifications on the UI thread. You can wrap any `IObservable<T>` with `ObserveOn`, passing a `SynchronizationContextScheduler` (or a framework-specific type such as `DispatcherScheduler`), to ensure that your observer receives notifications on the UI thread, making it safe to update the UI.

`SubscribeOn` can also be useful in user interfaces as a way to ensure that any initialization work that an observable source does to get started does not happen on the UI thread.

Most UI frameworks designate one particular thread for receiving notifications from the user and also for updating the UI, for any one window. It is critical to avoid blocking this UI thread, as doing

so leads to a poor user experience—if you are doing work on the UI thread, it will be unavailable for responding to user input until that work is done. As a general rule, if you cause a user interface to become unresponsive for longer than 100ms, users will become irritated, so you should not be perform any work that will take longer than this on the UI thread. When Microsoft first introduced its application store (which came in with Windows 8) they specified an even more stringent limit—if your application blocked the UI thread for longer than 50ms, it might not be allowed into the store. With the processing power offered by modern processors, you can achieve a lot of processing 50ms—even on the relatively low-powered processors in mobile devices that's long enough to execute millions of instructions. However, anything involving I/O (reading or writing files, or waiting for a response from any kind of network service) should not be done on the UI thread. The general pattern for creating responsive UI applications is:

- respond to some sort of user action
- if slow work is required, do this 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 could use `SubscribeOn` to ensure that the main work was not done on the UI thread, followed by `ObserveOn` to ensure you were notified back on the correct thread. If you failed to use the `ObserveOn` method, then your `OnNext` handlers would be invoked on the same thread that raised the notification. In most UI frameworks, this would cause some sort of not-supported/cross-threading exception. In this example, we subscribe to a sequence of `Customers`. I'm using `Defer` so that if `GetCustomers` does any slow initial work before returning its `IObservable<Customer>`, that won't happen until we subscribe. We then use `SubscribeOn` to call that method and perform the subscription on a task pool thread. Then we ensure that as we receive `Customer` notifications, we add them to the `Customers` collection on the `Dispatcher`.

```
Observable
    .Defer(() => _customerService.GetCustomers())
    .SubscribeOn(TaskPoolScheduler.Default)
    .ObserveOn(DispatcherScheduler.Instance)
    .Subscribe(Customers.Add);
```

Rx also offers `SubscribeOnDispatcher()` and `ObserveOnDispatcher()` extension methods to `IObservable<T>`, that automatically use the current thread's `Dispatcher` (and equivalents for `CoreDispatcher`). While these might be slightly more convenient they can make it harder to

test your code. We explain why in the Testing Rx 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 deadlocks, livelocks and corrupted state.

A particularly significant danger of introducing concurrency to an application, is that you can silently introduce bugs. Bugs arising from unpredictable timing are notoriously difficult to detect, making easy for these kinds of defects to 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

Rx can simplify handling of concurrency, but it is not immune deadlock. Some calls (like `First`, `Last`, `Single` and `ForEach`) are blocking—they do not return until something that they are waiting for occurs. The following example shows that this makes it very easy for deadlock to occur:

```
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....");
```

The `First` method will not return until its source emits a sequence. But the code that causes this source to emit sequence is on the line *after* the call to `First`. So the source can't emit a sequence until `First` returns. This style of deadlock, with two parties, each unable to proceed until the other proceeds, is often known as a *deadly embrace*. As this code shows, it's entirely possible for a deadly embrace to occur even in single threaded code. In fact, the single threaded nature of this code is what enables deadlock: we have two operations (waiting for the first notification, and sending the first notification) and only a single thread. That doesn't have to be a problem—it we'd used `FirstAsync` and attached an observer to that, `FirstAsync` would have executed its logic when the source `Subject<int>` invoked its `OnNext`. But that is more complex than just calling `First` and assigning the result into a variable.

This is an oversimplified example to illustrate the behaviour, and we would never write such code in production. (And even if we did, it fails so quickly and consistency that we would immediately become aware of a problem.) But in real application code, these kinds of problems can be harder to spot. Race conditions often slip into the system at integration points, so the problem isn't necessarily evidence in any one piece of code: timing problems can emerge as a result of how we plug multiple pieces of code together.

The next example may be a little harder to detect, but is only small step away from our first, unrealistic example. The basic idea is that we've got a subject that represents button clicks in a user interface. Event handlers representing user input are invoked by the UI framework—we just provide the framework with event handler methods, and it calls them for us whenever the event of interest, such as a button being clicked, occurs. This code calls `First` on the subject representing clicks, but it's less obvious that this might cause a problem here than it was in the preceding example:

```
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 have the intended effect, but because it does not block,
    // we can call this on the UI thread without deadlocking.
    //_subject.FirstAsync(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;
        PropertyChanged?.Invoke(this, new
↪   PropertyChangedEventArgs("Value"));
    }
}
```

The earlier example called the subject's `OnNext` after `First` returned, making it relatively straightforward to see that if `First` didn't return, then the subject wouldn't emit a notification. But that's not as obvious here—the `MyButton_Click` event handler will be set up inside the call to `InitializeComponent` (as is normal in WPF code), so apparently we've done the necessary setup to enable events to flow. By the time we reach this call to `First`, the UI framework already knows that if the user clicks `MyButton`, it should call `MyButton_Click`, and that method is going to cause the subject to emit a value.

There's nothing intrinsically wrong with that use of `First`. (Risky, yes, but there are scenarios in which that exact code would be absolutely fine.) The problem is the context in which we've used it. This code is in the constructor of a UI element, and these always run on a particular thread associated with that window's UI elements. (This happens to be a WPF example, but other UI frameworks work the same way.) And that's the same thread that the UI framework will use to deliver notifications about user input. If we block this UI thread, we prevent the UI framework from invoking our button click event handler. So this blocking call is waiting for an event that can only be raised from the very thread that it is blocking, thus creating a deadlock.

You might be starting to get the impression that we should try to avoid blocking calls in Rx. This is a good rule of thumb. We can fix the code above by commenting out the line that uses `First`, and uncommenting the one below it containing this code:

```
_subject.FirstAsync(1).Subscribe(value => Value = value);
```

This uses `FirstAsync` which does the same job, but with a different approach. It implements the same logic but it returns an `IObservable<T>` to which we must subscribe if we want to receive the first value whenever it does eventually appear. It is more complex than the just assigning the result of `First` into the `Value` property, but it is better adapted to the fact that we can't know when that source will produce a value.

If you do a lot of UI development, that last example might have seemed obviously wrong to you: we had code in the constructor for a window that wouldn't allow the constructor to complete until the user clicked a button in that window. The window isn't even going to appear until construction is complete so it makes no sense to wait for the user to click a button. That button's not even going to be visible on screen until after our constructor completes. Moreover, seasoned UI developers know that you don't just stop the world and wait for a specific action from the user. (Even modal dialogs, which effectively do demand a response before continuing, don't block the UI thread.) But as the next example shows, it's easy for problems to be harder to see. In this example, a 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();
    }
```

```
}
```

Unlike the earlier example, this does not attempt to block progress in the constructor. The blocking call to `First` occurs here in a button click handler (the `MyButton2_Click` method near the end). This example is more interesting because this sort of thing isn't necessarily wrong. Applications often perform blocking operations in click handlers: when we click a button to save a copy of a document, we expect the application to perform all necessary IO work to write our data out to storage. With modern solid state storage devices, this often happens so quickly as to appear instantaneous, but back in the days when mechanical hard drives were the norm, it was not unusual for an application to become briefly unresponsive while it saved our document. This can happen even today if your storage is remote, and networking issues are causing delays.

So even if we've learned to be suspicious of blocking operations such as `First`, it's possible that it's OK in this example. It's not possible to tell for certain by looking at this code alone. It all depends on what sort of an observable `GetTemperature` returns, and the manner in which it produces its items. That call to `First` will block on the UI thread until a first item becomes available, so this will produce a deadlock if the production of that first item requires access to the UI thread. Here's a slightly contrived way to create that problem:

```
class MyService : IMyService
{
    public IObservable<int> GetTemperature()
    {
        return Observable.Create<int>(
            o =>
            {
                o.OnNext(27);
                o.OnNext(26);
                o.OnNext(24);
                return () => { };
            })
            .SubscribeOnDispatcher();
    }
}
```

This fakes up behaviour intended to simulate an actual temperature sensor by making a series of calls to `OnNext`. But it does some odd explicit scheduling: it calls `SubscribeOnDispatcher`. That's an extension method that effectively calls `SubscribeOn(DispatcherScheduler.Current.Dispatcher)`. This effectively tells Rx that when something tries to subscribe to the `IObservable<int>` that `GetTemperature` returns, that subscription call should be done through a WPF-specific scheduler that runs its work items on the UI thread. (Strictly, speaking, WPF does allow multiple UI threads, so to more precise, this code only works if you call it on a UI thread, and if you do so, the scheduler will

ensure that work items are scheduled onto the same UI thread.)

The effect is that when our click handler calls `First`, that will in turn subscribe to the `IObservable<int>` returned by `GetTemperature`, and because that used `SubscribeOnDispatcher`, this does not invoke the callback passed to `Obserable.Create` immediately. Instead, it schedules a work item that will do that when the UI thread (i.e., the thread we're running on) becomes free. It's not considered to be free right now, because it's in the middle of handling the button click. Having handed this work item to the scheduler, the `Subscribe` call returns back to the `First` method. And the `First` method now sits and waits for the first item to emerge. Since it won't return until that happens, the UI thread will not be considered to be available until that happens, meaning that the scheduled work item that was supposed to produce that first item can never run, and we have deadlock.

This boils down to the same basic problem as the first of these `First`-related deadlock examples. We have two processes: the generation of items, and waiting for an item to occur. These need to be in progress concurrently—we need the "wait for first item" logic to be up and running at the point when the source emits its first item. These examples all use just a single thread, which makes it a bad idea to use a single blocking call (`First`) both to set up the process of watching for the first item, and also to wait for that to happen. But even though it was the same basic problem in all three cases, it became harder to see as the code became more complex. With real application code, it's often a lot harder than this to see the root causes of deadlocks.

So far, this chapter may seem to say that concurrency is all doom and gloom by focusing on the problems you could face, and the fact that they are often hard to spot in practice; this is not the intent though. Although adopting Rx can't magically avoid classic concurrency problems, Rx can make it easier to get it right, provided you follow these two simple rules.

- Only the top-level subscriber should make scheduling decisions
- Avoid using blocking calls: e.g. `First`, `Last` and `Single`

The last example came unstuck with one simple problem; the `GetTemperature` service was dictating the scheduling model when, really, it had no business doing so. Code representing a temperature sensor shouldn't need to know that I'm using a particular UI framework, and certainly shouldn't be unilaterally deciding that it is going to run certain work on a WPF user interface thread. When getting started with Rx, it can be easy to convince yourself that baking scheduling decisions into lower layers is somehow being 'helpful'. "Look!" you might say. "Not only have I provided temperature readings, I've also made this automatically notify you on the UI thread, so you won't have to bother with `ObserveOn`." The intentions may be good, but it's all too easy to create a threading nightmare. Only the code that sets up a subscription and consumes its results can have a complete overview of the concurrency requirements, so that is the right level at which to choose which schedulers to use. Lower levels of code should not try to get involved; they should just do what they are told. (Rx arguably

breaks this rule slightly itself by choosing default schedulers where they are needed. But it makes very conservative choices designed to minimize the chances of deadlock, and always allows applications to take control by specifying the scheduler.)

Note that following either one of the two rules above would have been sufficient to prevent deadlock in this example. But it is best to follow both rules.

This does leave one question unanswered: *how* should the top-level subscriber make scheduling decisions? I've identified the area of the code that needs to make the decision, but what should the decision be? It will depend on the kind of application you are writing. For UI code, this pattern generally works well: "Subscribe on a background thread; Observe on the UI thread". With UI code, the risk of deadlock arises in because the UI thread is effectively a shared resource, and contention for that resource can produce deadlock. So the strategy is to avoid requiring that resource as much as possible: work that doesn't need to be on the thread should not be on that thread, which is why performing subscription on a worker thread (e.g., by using the `TaskPoolScheduler`) reduces the risk of deadlock. It follows that if you have observable sources that decide when to produce events (e.g., timers, or sources representing inputs from external information feeds or devices) you would also want those to schedule work on worker threads. It is only when we need to update the user interface that we need our code to run on the UI thread, and so we defer that until the last possible moment by using `ObserveOn` in conjunction with a suitable UI-aware scheduler (such as the WPF `DispatcherScheduler`). If we have a complex Rx query made up out of multiple operators, this `ObserveOn` should come right at the end, just before we call `Subscribe` to attach the handler that will update the UI. This way, only the final step, the updating of the UI, will need access to the UI thread. By the time this runs, all complex processing will be complete, and so this should be able to run very quickly, relinquishing control of the UI thread almost immediately, improving application responsiveness, and lowering the risk of deadlock.

Other scenarios will require other strategies, but the general principle with deadlocks is always the same: understand which shared resources require exclusive access. For example, if you have a sensor library, it might create a dedicated thread to monitor devices and report new measurements, and if it were to stipulate that certain work had to be done on that thread, this would be very similar to the UI scenario: there is a particular thread that you will need to avoid blocking. The same approach would likely apply here. But this is not the only kind of scenario.

You could imagine a data processing application in which certain data structures are shared. It's quite common in these cases to be allowed to access such data structures from any thread, but to be required to do so one thread at a time. Typically we would use thread synchronization primitives to protect against concurrent use of these critical data structures. In these cases, the risks of deadlock do not arise from the use of particular threads. Instead, they arise from the possibility that one thread can't progress because some other threads is using a shared data structure, but that other thread is waiting

for the first thread to do something, and won't relinquish its lock on that data structure until that happens. The simplest way to avoid problems here is to avoid blocking wherever possible. Avoid methods like `First`, preferring their non-blocking equivalents such as `FirstAsync`. (If there are cases where you can't avoid blocking, try to avoid doing so while in possession of locks that guard access to shared data. And if you really can't avoid that either, then there are no simple answers—you'll now have to start thinking about lock hierarchies to systematically avoid deadlock, just as you would if you weren't using Rx.) The non-blocking style is the natural way to do things with Rx, and that's the main way Rx can help you avoid concurrency related problems in these cases.

Advanced features of schedulers

Schedulers provide some features that are mainly of interest when writing observable sources that need to interact with a schedule. The most common way to use schedulers is when setting up a subscription, either supplying them as arguments when creating observable sources, or passing them to `SubscribeOn` and `ObserveOn`. But if you need to write an observable source that produces items on some schedule of its own choosing (e.g., suppose you are writing a library that represents some external data source and you want to present that as an `IObservable<T>`), you might need to use some of these more advanced features.

Passing state

All of the methods defined by `IScheduler` take a `state` argument. Here's the interface definition again:

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

The scheduler does not care what is in this `state` argument. It just passes it unmodified into your callback when it executes your work item. This provides one way to provide context for that callback. It's not strictly necessary: the delegate we pass as the `action` can incorporate whatever state we need. The easiest way to do that is to capture variables in a lambda. However, if you look at the Rx source

code you will find that it typically doesn't do that. For example, the heart of the Range operator is a method called LoopRec and if you look at the source for LoopRec you'll see that it includes this line:

```
var next = scheduler.Schedule(this, static (innerScheduler, @this) =>
↪  @this.LoopRec(innerScheduler));
```

Logically, Range is just a loop that executes once for each item it produces. But to enable concurrent execution and to avoid stack overflows, it implements this by scheduling each iteration of the loop as an individual work item. (The method is called LoopRec because it is logically a recursive loop: we kick it off by calling Schedule, and each time the scheduler calls this method, it calls Schedule again to ask for the next item to run. This doesn't actually cause recursion with any of Rx's built-in schedulers, even the ImmediateScheduler, because they all detect this and arrange to run the next item after the current one returns. But if you wrote the most naive scheduler possible, this would actually end up recursing at runtime, likely leading to stack overflows if you tried to create a large sequence.)

Notice that the lambda passed to Schedule has been annotated with static. This tells the C# compiler that it is our intention *not* to capture any variables, and that any attempt to do so should cause a compiler error. The advantage of this is that the compiler is able to generate code that reuses the same delegate instance for every call. The first time this runs, it will create a delegate and store it in a hidden field. On every subsequent execution of this (either in future iterations of the same range, or for completely new range instances) it can just use that same delegate again and again and again. This is possible because the delegate captures no state. This avoids allocating a new object each time round the loop.

Couldn't the Rx library have used this more straightforward approach?

```
var next = scheduler.Schedule(() => LoopRec(scheduler)); // Don't do this.
↪  (See below.)
```

This uses an extension method that Rx supplies for cases where you have no need for the state argument. In fact it's the wrong thing to use here, because this could end up passing the wrong scheduler. If you look at the IScheduler interface, you'll see that the work item callback's first argument is an IScheduler, and you're supposed to use that scheduler for any logically recursive work. Some schedulers (e.g., the ImmediateScheduler) supply a special type of scheduler specifically to avoid logical recursion turning into actual recursion that could end up going so deep that the application crashes with a stack overflow. This particular extension method isn't meant for these logically recursive scenarios, so we can't quite make it this simple. But perhaps we could avoid that weirdness of passing our own this argument:

```
var next = scheduler.Schedule<object?>(null, (innerScheduler, _) =>
↪  LoopRec(innerScheduler));
```

This fixes the problem with the preceding example: it correctly uses the correct scheduler for any further logically recursive calls. And now we're just invoking the `LoopRec` instance member in the ordinary way: we're implicitly using the `this` reference that is in scope. So this will create a delegate the captures that implicit `this` reference. But that's inefficient: it will force the compiler to generate code that allocates two objects each time this line runs. (And the same would happen with the preceding example—not only is it broken, it's also inefficient.) It creates one object that has a field holding onto the captured `this`, and then it needs to create a distinct delegate instance that has a reference to that capture object.

The more complex code that is actually in the `Range` implementation avoids this. It disables capture by annotating the lambda with `static`. That prevents code from relying on the implicit `this` reference. So it has had to arrange for the `this` reference to be available to the callback. And that's exactly the sort of thing the `state` argument is there for. It provides a way to pass in some per-work-item state so that you can avoid the overhead of capturing variables on each iteration.

Future scheduling

I talked earlier about time-based operators, and also about the two time-based members of `ISchedule` that enable this, but I've not yet shown how to use it. These enable you 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 by calling the overload of `Schedule` that takes a `DateTimeOffset`, or you can specify the period of time to wait until the action is invoked with the `TimeSpan`-based overload.

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
```

This illustrates that scheduling was non-blocking here, because the 'before' and 'after' calls are very close together in time. (It will be this way for most schedulers, but as discussed earlier,

`ImmediateScheduler` works differently. In this case, you would see the After message after the Inside one. that's why none of the timed operators will use `ImmediateScheduler` by default.) 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. Be aware that changes in the system clock complicate matters. Rx's schedulers do make some accommodations to deal with clock drift, but sudden large changes to the system clock can cause some short term chaos.

Cancellation

Each of the overloads to `Schedule` returns an `IDisposable`, and calling `Dispose` on this will 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 return value.

```
var delay = TimeSpan.FromSeconds(1);
Console.WriteLine("Before schedule at {0:o}", DateTime.Now);

var workItem = scheduler.Schedule(delay, () => Console.WriteLine("Inside
↪  schedule at {0:o}", DateTime.Now));
Console.WriteLine("After schedule at  {0:o}", DateTime.Now);

workItem.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. It's possible to cancel scheduled work that is already running, and this is why the work item callback is required to return `IDisposable`: if work has already begun when you try to cancel the work item, Rx calls `Dispose` on the `IDisposable` that your work item callback returned. 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.

You may be wondering how this mechanism can be any use: the work item callback needs to have returned already for Rx to be able to invoke the `IDisposable` that it returns. This mechanism can

only be used in practice if work continues after returning to the scheduler. You could fire up another thread so the work happens concurrently, although we generally try to avoid creating threads in Rx. Another possibility would be it the scheduled work item invoked some asynchronous API and returned without waiting for it to complete—if that API offered cancellation, you could return an `IDisposable` that cancelled it.

To illustrate cancellation in operation, this slightly unrealistic example runs some work as a `Task` to enable it to continue after our callback returns. It just fakes some work by performing a spin wait and adding values to the `list` argument. The key here is that create a `CancellationToken` to be able to tell the task we want it to stop, and we return an `IDisposable` that puts this token in to a cancelled state.

```
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("Cancellation 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
Cancellation requested
```

The problem here is that we have introduced explicit use of `Task` so we are increasing concurrency in a way that is outside of the control of the scheduler. The Rx library generally allowed control over the way in which concurrency is introduced by accepting a scheduler parameter. If the goal is to enable long-running iterative work, we can avoid having to spin up new threads or tasks but using Rx recursive scheduler features instead. I already talked a bit about this in the Passing state section, but there are a few ways to go about it.

Recursion

In addition to the `IScheduler` methods, Rx defines various overloads of `Schedule` in the form of extension methods. Some of these 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 allows us to achieve a similar logically recursive iterative approach as you saw in Passing state section, but in a potentially simpler way.

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. Since each individual iteration was scheduled as a separate work item, there are no long-running jobs, so it's enough to let the scheduler deal entirely with cancellation.

The main difference between these overloads, and using the `IScheduler` methods directly, is that you don't need to pass another callback directly into the scheduler. You just invoke the supplied `Action` and it schedules another call to your method. They also enable you not to pass a state argument if you don't have any use for one.

As mentioned in the earlier section, although this logically represents recursion, Rx protects us from stack overflows. The schedulers implement this style of recursion by waiting for the method to return before performing the recursive call. (So it is always what's called "tail recursion" where the recursive call occurs right at the end of the current method.)

This concludes our tour of scheduling and threading. Next, we will look at the boundary between Rx and the rest of the world.

Leaving Rx's World

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 we must leave the `IObservable<T>` paradigm. This is necessary when we need to integrate with an existing non-Rx-based API (e.g. one that uses 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.

Rx's compositional nature is the key to its power, but it can look like a problem when you need to integrate with a component that doesn't understand Rx. Most of the Rx library features we've looked at so far express their inputs and outputs as observables. How are you supposed to take some real world source of events and turn that into an observable? How are you supposed to do something meaningful with the output of an observable?

You've already seen some answer to these questions. The Creating Observable Sequences chapter showed various ways to create observable sources. But when it comes to handling the items that emerge from an `IObservable<T>`, all we've really seen is how to implement `IObserver<T>`, and how to use the callback based `Subscribe` extension methods to subscribe to an `IObservable<T>`.

In this chapter, we will look at the methods in Rx which allow you to leave the `IObservable<T>` world, so you can do some useful work with the notifications that emerge from a source.

Integration with `async` and `await`

You can use C#'s `await` keyword with any `IObservable<T>`. We saw this earlier with `FirstAsync`:

```
long v = await Observable.Timer(TimeSpan.FromSeconds(2)).FirstAsync();
Console.WriteLine(v);
```

Although `await` is most often used with `Task`, `Task<T>`, or `ValueTask<T>`, it is actually an extensible language feature. It's possible to make `await` work for more or less any type by defining some suitable extension methods and some supporting types. And that's precisely what Rx does. If your source file includes a `using System.Reactive.Linq;` directive, a suitable extension method will be available, so you can `await` any task.

The way this actually works is that the relevant `GetAwaiter` extension method wraps the `IObservable<T>` in an `AsyncSubject<T>`, which provides everything that C# requires to support `await`. These wrappers work in such a way that there will be a call to `Subscribe` each time you execute an `await` against an `IObservable<T>`.

If a source reports an error by invoking its observer's `OnError`, Rx's `await` integration handles this by putting the task into a faulted state, so that the `await` will rethrow the exception.

Sequence can be empty—they call `OnCompleted` without ever having called `OnNext`. If you `await` a sequence that does this, it will throw an `InvalidOperationException` reporting that the sequence contains no elements.

As you may recall from the `AsyncSubject<T>` section of chapter 3, `AsyncSubject<T>` reports only the final value to emerge from its source. So if you `await` a sequence that reports multiple items, all but the final item will be ignored. What if you want to see all of the items, but you'd still like to use `await` to handle completion and errors?

ForEachAsync

The `ForEachAsync` method supports `await`, but it provides a way to process each element. You could think of it as a hybrid of the `await` behaviour described in the preceding section, and the callback-based `Subscribe`. We can still use `await` to detect completion and errors, but we supply a callback enabling us to handle every item:

```
IObservable<long> source = Observable.Interval(TimeSpan.FromSeconds(1))
                        .Take(5);
await source.ForEachAsync(i => Console.WriteLine("received {0} @ {1}", i,
↪  DateTime.Now));
Console.WriteLine("finished @ {0}", DateTime.Now);
```

Output:

```
received 0 @ 02/08/2023 07:53:46
received 1 @ 02/08/2023 07:53:47
received 2 @ 02/08/2023 07:53:48
received 3 @ 02/08/2023 07:53:49
received 4 @ 02/08/2023 07:53:50
finished @ 02/08/2023 07:53:50
```

Note that the `finished` line is last, as you would expect. Let's compare this with the `Subscribe` extension method, which also lets us provide a single callback for handling items:

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

As the output shows, `Subscribe` returned immediately. Our per-item callback was invoked just like before, but this all happened later on:

```
finished @ 02/08/2023 07:55:42
received 0 @ 02/08/2023 07:55:43
received 1 @ 02/08/2023 07:55:44
received 2 @ 02/08/2023 07:55:45
received 3 @ 02/08/2023 07:55:46
received 4 @ 02/08/2023 07:55:47
```

This can be useful in batch-style programs that perform some work and then exit. The problem with using `Subscribe` in that scenario is that our program could easily exit without having finished the work it started. This is easy to avoid with `ForEachAsync` because we just use `await` to ensure that our method doesn't complete until the work is done.

When we use `await` either directly against an `IObservable<T>`, or through `ForEachAsync`, we are essentially choosing to handle sequence completion in a conventional way. Error and completion handling are no longer callback driven—Rx supplies the `OnCompleted` and `OnError` handlers for us, and instead represents these through the awaiter mechanism C# recognizes. (Specifically, when we `await` a source directly, Rx supplies a custom awaiter, and when we use `ForEachAsync` it just returns a `Task`.)

Note that there are some circumstances in which `Subscribe` will block until its source completes. `Observable.Return` will do this by default, as will `Observable.Range`. We could try to make the last example do it by specifying a different scheduler:

```
// Don't do this!
IObservable<long> source = Observable.Interval(TimeSpan.FromSeconds(1),
↪  ImmediateScheduler.Instance)
                      .Take(5);
source.Subscribe(i => Console.WriteLine("received {0} @ {1}", i,
↪  DateTime.Now));
Console.WriteLine("finished @ {0}", DateTime.Now);
```

However, this highlights the dangers of non-async blocking calls: although this looks like it should work, in practice it deadlocks in the current version of Rx. Rx doesn't consider the `ImmediateScheduler` to be suitable for timer-based operations, which is why it's not the default, and this scenario is a good example of why that is. (The fundamental issue is that the only way to cancel a scheduled work item is to call `Dispose` on the object returned by the call to `Schedule`. `ImmediateScheduler` by definition doesn't return until after it has finished the work, meaning it effectively can't support

cancellation. So the call to `Interval` effectively creates a periodically scheduled work item that can't be cancelled, and which is therefore doomed to run forever.)

This is why we need `ForEachAsync`. It might look like we can get the same effect through clever use of schedulers, but in practice if you need to wait for something asynchronous to happen, it's always better to use `await` than to use an approach that entails blocking the calling thread.

ToEnumerable

The two mechanisms we've explored so far convert completion and error handling from Rx's callback mechanism to a more conventional approach enabled by `await`, but if we wanted to handle each item from the source, each still used a callback for that. But the `ToEnumerable` extension method goes a step further: it enables the entire sequence to be consumed with a conventional `foreach` loop:

```
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). If no elements are available yet, or you have consumed all elements produced so far, the call that `foreach` makes to the enumerator's `MoveNext` will block until the source produces an element. So this approach relies on the source being able to generate elements from some other

thread. (In this example, `Timer` defaults to the `DefaultScheduler`, which runs work on the thread pool.) If the sequence produces values faster than you consume them, they will be queued for you. (This means that it is technically possible to consume and generate items on the same thread when using `ToEnumerable` but this would rely on the producer always remaining ahead. This would be a dangerous approach because if the `foreach` loop ever caught up, it would then deadlock.)

As with `await` and `ForEachAsync`, if the source reports an error, this will be thrown, so you can use ordinary C# exception handling as this example shows:

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

To a single collection

Sometimes you will want all of the items a source produces as a single list. For example, perhaps you can't just process the elements in order—perhaps you will sometimes need to refer back to elements received earlier. The four operations described in following sections gather all of the items into a single collection. They all still produce an `IObservable<T>` (e.g., an `IObservable<int[]>` or an `IObservable<Dictionary<string, long>>`), but these are all single-element observables, and as you've already seen, you can use the `await` keyword to get hold of this single output.

ToArray and ToList

`ToArray` and `ToList` take an observable sequence and package it into an array or an instance of `List<T>` respectively. As with all of the single collection operations, these return an observable source that waits for their input sequence to completes, and then produces the array or list as the single value, after which they immediately complete. This example uses `ToArray` to collect all 5 elements from a source sequence into an array, and uses the `await` to extract that array from the sequence that `ToArray` returns:

```
TimeSpan period = TimeSpan.FromMilliseconds(200);
IObservable<long> source = Observable.Timer(TimeSpan.Zero, period).Take(5);
```

```
IObservable<long[]> resultSource = source.ToArray();

long[] result = await resultSource;
foreach (var value in result)
{
    Console.WriteLine(value);
}
```

Output:

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

As these methods still return observable sequences, you can also use the normal Rx `Subscribe` mechanisms, or use these as inputs to other operators.

If the source produces values and then errors, you will not receive any of those values—all values received up to that point will be discarded, and the operator will invoke its observer's `OnError` (and in the example above, that will result in the exception being thrown from the `await`). All four operators (`ToArray`, `ToList`, `ToDictionary` and `ToLookup`) handle errors like this.

ToDictionary and ToLookup

Rx can package an observable sequence into a dictionary or lookup with the `ToDictionary` and `ToLookup` methods. Both methods take the same basic approach as the `ToArray` and `ToList` methods: they return a single-element sequence that produces the collection once the input source completes.

`ToDictionary` provides four overloads that correspond directly to the `ToDictionary` extension methods for `IEnumerable<T>` defined by LINQ to Objects:

```
// 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 offers near-identical-looking overloads, the difference being the return type (and the name, obviously). They all return an `IObservable<ILookup<TKey, TElement>>`. As with LINQ to Objects, the distinction between a dictionary and a lookup is that the `ILookup<TKey, TElement>>` interface allows each key to have any number of values, whereas a dictionary maps each key to one value.

ToTask

Although Rx provides direct support for using `await` with an `IObservable<T>`, it can sometimes be useful to obtain a `Task<T>` representing an `IObservable<T>`. This is useful because some APIs expect a `Task<T>`. You can call `ToTask()` on any `IObservable<T>`, and this will subscribe to that observable, returning a `Task<T>` that will complete when the task completes, producing the sequence's final output as the task's result. If the source completes without producing an element, the task will enter a faulted state, with an `InvalidOperation` exception complaining that the inputs sequence contains no elements.

You can optionally pass a cancellation token. If you cancel this before the observable sequence completes, Rx will unsubscribe from the source, and put the task into a cancelled 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.

```
IObservable<long> source = Observable.Interval(TimeSpan.FromSeconds(1))
                        .Take(5);
Task<long> resultTask = source.ToTask();
long result = await resultTask; // Will take 5 seconds.
Console.WriteLine(result);
```

Output:

```
4
```

If the source sequence calls `OnError`, Rx puts the task in a faulted state, using the exception supplied.

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

ToEvent

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)
{...}
```

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
```

This does not follow the standard .NET event pattern. We use a slightly different approach if we want that.

ToEventPattern

Normally, .NET events supply `sender` and `EventArgs` arguments to their handlers. 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`.

```
// 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 `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;
}
```

To use this, we will need a suitable `EventArgs` type. You might be able to use one supplied by the .NET runtime libraries, but if not you can easily write your own:

The `EventArgs` type:

```
public class MyEventArgs : EventArgs
{
    private readonly long _value;

    public MyEventArgs(long value)
    {
        _value = value;
    }

    public long Value
    {
        get { return _value; }
    }
}
```

We can then use this from Rx by applying a simple transform using `Select`:

```
IObservable<EventPattern<MyEventArgs>> 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.

```
IEventPatternSource<MyEventArgs> 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 syntax. 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 do not have a standard pattern for reporting errors
- Events do not have a standard pattern for indicating the end of a sequence of values
- 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 Sequences chapter. We now have the means to enter and leave Rx's world. Take care when opting in and out of the `IObservable<T>`. It's best not to transition back and forth—having a bit of Rx-based processing, then some more conventional code, and then plumbing the results of that back into Rx can quickly make a mess of your code base, and may indicate a design flaw. Typically it is better to keep all of your Rx logic together, so you only need to integrating with the outside world twice: once for input and once for output.

Do

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. To enable this it can often be useful to 'tap into' your Rx queries, making them deliver monitoring and diagnostic information as a side effect of their normal operation.

The Do extension method allows you to inject side effect behaviour. From an Rx perspective, Do doesn't appear to do anything: you can apply it to any `IObservable<T>`, and it returns another `IObservable<T>` that reports exactly the same elements and error or completion as its source.

However, its various overloads takes callback arguments that look just like those for `Subscribe`: you can provide callbacks for individual items, completion, and errors. Unlike `Subscribe`, Do is not the final destination—everything the Do callbacks see will also be forwarded onto Do's subscribers. This makes it useful for logging and similar instrumentation because you can use it to report how information is flowing through an Rx query without changing that query's behaviour.

You have to be a bit careful of course. Using Do will have a performance impact. And if the callback you supply to Do performs any operations that could change the inputs to the Rx query it is part of, you will have created a feedback loop making the behaviour altogether harder to understand.

Let's first define some logging methods that we can go on to use in an example:

```
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 part of the query, it necessarily sees the values earlier than the `Subscribe`, which is the final link in the chain. That's why the log messages appear before the lines produced by the `Subscribe` callbacks. 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 modifying it.

As with `Subscribe`, instead of passing callbacks, you can pass Do 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.

Encapsulating with AsObservable

Poor encapsulation is a way developers can leave the door open for bugs. 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.

TODO: swap this and the next chapter over? timing follows naturally from scheduling.

Time-based sequences

With event sources, timing is often important. In some cases, the only information of interest about some event might the time at which it occurred. The only way in which the core `IObservable<T>` and `IObserver<T>` interfaces address time is that a source can decide when it calls an observer's `OnNext` method. A subscriber knows when an event occurred because it is occurring right now. This isn't always the most convenient way in which to work with timing, so the Rx library provides some timing-related operators. We've already seen a couple of operators that offer optional time-based operation: `Buffer` and [`Window`])(08_Partitioning#window). This chapter looks at the various operators that are all about timing.

Timestamp and TimeInterval

As observable sequences are asynchronous it can be convenient to know timings for when elements are received. Obviously, a subscriber can always just use `DateTimeOffset.Now`, but if you want to refer to the arrival time as part of a larger query, the `Timestamp` extension method is a handy convenience method that attaches a timestamp to each element. It wraps elements from its source 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 also a `DateTimeOffset` indicating when `Timestamp` operator received it.

In this example we create a sequence of three values, one second apart, and then transform it to a time stamped sequence.

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

As you can see, `Timestamped<T>`'s implementation of `ToString()` gives us a readable output.

```
TimeStamp-->0@07/08/2023 10:03:58 +00:00
TimeStamp-->1@07/08/2023 10:03:59 +00:00
TimeStamp-->2@07/08/2023 10:04:00 +00:00
TimeStamp completed
```

We can see that the values 0, 1 & 2 were each produced one second apart.

Rx also offers `TimeInterval`. Instead of reporting the time at which items arrived, it reports the interval between items (or, in the case of the first element, how long it took for that to emerge

after subscription). Similarly to the `Timestamp` method, elements are wrapped in a light weight structure. But whereas `Timestamped<T>` adorned each item with the arrival time, `TimeInterval` wraps each element with the `TimeInterval<T>` type which adds a `TimeSpan`. We can modify the previous example to use `TimeInterval`:

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

As you can see, the output now reports the time between elements instead of the time of day at which they were received:

```
TimeStamp-->0@00:00:01.0183771
TimeStamp-->1@00:00:00.9965679
TimeStamp-->2@00:00:00.9908958
TimeStamp completed
```

As you can see from the output, the timings are not exactly one second but are pretty close. Some of this will be measurement noise in the `TimeInterval` operator, but most of this variability is likely to arise from the `Observable.Interval` class. There will always be a limit to the precision with which a scheduler can honour the timing request of it. Some scheduler introduce more variation than others—the schedulers that deliver work via a UI thread are ultimately limited by how quickly that thread's message loop responds. But even in the most favourable condition, schedulers are limited by the fact that .NET is not built for use in real-time systems (and nor are most of the operating systems Rx can be used on). So with all of the operators in this section, you should be aware that timing is always a *best effort* affair in Rx.

Delay

The `Delay` extension method time-shifts an entire sequence. `Delay` attempts to preserve the relative time intervals between the values. There is inevitably a limit to the precision with which it can do this—it won't recreate timing down to the nearest nanosecond. The exact precision is determined by the scheduler you use, and will typically get worse under heavy load, but it will typically reproduce timings to within a few milliseconds.

There are overloads of `Delay` accepting a `TimeSpan`, and an optional scheduler, which will delay the sequence by the specified amount. And there are also delays that accept a `DateTimeOffset` (and also, optionally, a scheduler) which will wait until the specified time occurs, and then start replaying the input. (This second, absolute time based approach is essentially equivalent to the `TimeSpan`

overloads—you would get more or less the same effect by subtracting the current time from the target time to get a `TimeSpan`, except the `DateTimeOffset` version attempts to deal with changes in the system clock that occur between `Delay` being called, and the specified time arriving.)

To show the `Delay` method in action, this example creates a sequence of values one second apart and timestamps 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.

```
IObservable<Timestamped<long>> source = Observable
    .Interval(TimeSpan.FromSeconds(1))
    .Take(5)
    .Timestamp();

IObservable<Timestamped<long>> 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"));
```

If you look at the timestamps in the output, you can see that both the immediate and the delayed subscriptions subscribed to the source at the same time: both report the first timestamp as `10:15:46`. But you can also see that the `delay` subscription did not start receiving those events until 2 seconds later.

```
source : 0@07/08/2023 10:15:46 +00:00
source : 1@07/08/2023 10:15:47 +00:00
source : 2@07/08/2023 10:15:48 +00:00
delay : 0@07/08/2023 10:15:46 +00:00
source : 3@07/08/2023 10:15:49 +00:00
delay : 1@07/08/2023 10:15:47 +00:00
delay : 2@07/08/2023 10:15:48 +00:00
source : 4@07/08/2023 10:15:50 +00:00
source Completed
delay : 3@07/08/2023 10:15:49 +00:00
delay : 4@07/08/2023 10:15:50 +00:00
delay Completed
```

Note that `Delay` will not time-shift `OnError` notifications. These will be propagated immediately.

Sample

The `Sample` method produces items whatever interval you ask. Each time it produces a value, it reports the last value that emerged from your source. If you have a source that produces data at a higher rate than you need (e.g. suppose you have an accelerometer that reports 100 measurements per second, but you only need to take a reading 10 times a second), `Sample` provides an easy way to reduce the data rate. This example shows `Sample` in action.

```
IObservable<long> interval =
↪   Observable.Interval(TimeSpan.FromMilliseconds(150));
interval.Sample(TimeSpan.FromSeconds(1)).Subscribe(Console.WriteLine);
```

Output:

```
5
12
18
```

If you looked at these numbers closely, you might have noticed that the interval between the values is not the same each time. I chose a source interval of 150ms and a sample interval of 1 second to highlight an aspect of sampling that can require careful handling: if the rate at which a source produces items doesn't line up neatly with the sampling rate, this can mean that `Sample` introduces irregularities that weren't present in the source. If we list the times at which the underlying sequence produces values, and the times at which `Sample` takes each value, we can see that with these particular timings, the sample intervals only line up with the source timings every 3 seconds.

Relative time (ms)	Source value	Sampled value
0		
50		
100		
150	0	
200		
250		
300	1	
350		
400		

Relative time (ms)	Source value	Sampled value
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		

Relative time (ms)	Source value	Sampled value
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

Since the first sample is taken after the source emits five, and two thirds of the way into the gap after which it will produce six, there's a sense in which the "right" current value is something like 5.67,

but `Sample` doesn't attempt any such interpolation. It just reports the last value to emerge from the source. A related consequence is that if the sampling interval is short enough that you're asking `Sample` to report values faster than they are emerging from the source, it will just repeat values.

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)
{...}
```

We could apply `Throttle` to use a live search feature that makes suggestions as you type. We would typically want to wait until the user has stopped typing for a bit before searching for suggestions, because otherwise, we might end up kicking off several searches in a row, cancelling the last one each time the user presses another key. Only once there is a pause should we can execute a search with what they have typed so far. `Throttle` fits well with this scenario, because it won't allow any events through at all if the source is producing values faster than the specified rate.

Note that the RxJS library decided to make their version of throttle work differently, so if you ever find yourself using both Rx.NET and RxJS, be aware that they don't work the same way. In RxJS, throttle doesn't shut off completely when the source exceeds the specified rate: it just drops enough items that the output never exceeds the specified rate. So RxJS's throttle implementation is a kind of rate limiter,

whereas Rx.NET's `Throttle` is more like a self-resetting circuit breaker that shuts off completely during an overload.

Timeout

The `Timeout` operator method allows us terminate a sequence with an error if the source does not produce 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.
```

Alternatively, we can pass `Timeout` an absolute time; if the sequence does not completed by that time, it produces an error.

```
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.
```

There are other `Timeout` overloads enabling us to substitute an alternative sequence when a timeout occurs.

```
// 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)
{...}
```

As we've now seen, Rx provides features to manage timing in a reactive paradigm. Data can be timed, throttled, or sampled 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.

Exceptions happen. Some exceptions are inherently avoidable, occurring only because of bugs in our code. For example, we shouldn't normally put the CLR into a situation where it has to raise a `DivideByZeroException`. But there are plenty of exceptions that cannot be prevented with defensive coding. For example, exceptions relating to I/O or networking failures such as like `FileNotFoundException` or `TimeoutException` can be caused by environmental factors outside of our code's control. In these cases, we need to handle the exception gracefully. The kind of handling will depend on the context—it might be appropriate to provide some sort of error message to the user; in some scenarios logging the error might be a more appropriate response. If the failure is likely to be transient, we could try to recover by retrying the operation that failed.

The `IObserver<T>` interface defines the `OnError` method so that a source can report an error, but since this terminates the sequence, it provides no direct means of working out what to do next.

However, Rx provides operators that provide a variety of error handling mechanisms.

Catch

Rx defines a `Catch` operator. The name is deliberately reminiscent of C#'s `try/catch` syntax because it lets you handle errors from an Rx source in a similar way to exceptions that emerge from normal execution of code. It gives you the option of swallowing an exception, wrapping it in another exception or performing some other logic.

Swallowing exceptions

The most basic (although rarely the best) way to handle an exception is to swallow it. In C#, we could write a `try` block with an empty `catch` block. We can achieve something similar with Rx's `Catch` operator. The basic idea with swallowing exceptions is that the process that caused the exception stops, but we act as though nothing had happened—we handle it in the same way as it the process had naturally reached an end. We can represent an exception being swallowed like this with a marble diagram.

TODO: make this a proper diagram.

```
S1--1--2--3--X
S2            -|
R --1--2--3----|
```

Here S1 represents the first sequence that ends with an error (X). If we're swallowing the exception, we want to make it look like the sequence just came to a normal halt. So S2 here is an empty sequence we will substitute when the first throws an exception. R is the result sequence which starts as S1, then continues with S2 when S1 fails. This code creates the scenario described in that marble diagram:

```
var source = new Subject<int>();
IObservable<int> result = source.Catch(Observable.Empty<int>());

result.Dump("Catch"););

source.OnNext(1);
source.OnNext(2);
source.OnNext(3);
source.OnError(new Exception("Fail!"));
```

Output:

```
Catch-->1
Catch-->2
Catch-->3
Catch completed
```

This is conceptually similar to the following code:

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

This kind of catch-and-ignore everything handling is generally discouraged in C#, and you probably also want to limit your use of its equivalent in Rx.

Swallowing only specific exception types

It's much more common to want to handle a specific exception like this:

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

`Catch` has an overload that enables you specify the type of exception:

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

This enables us to write the Rx equivalent to the more selective `catch`, where we only swallow a `TimeoutException`:

```
IObservable<int> result = source.Catch<int, TimeoutException>(_ =>
↪   Observable.Empty());
```

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.

Examining the exception

Notice that with the overload in the preceding example, we supplied a callback. If an exception of the specified type emerges, this overload of `Catch` will pass it to our callback so that if necessary, we can decide exactly what to return based on information in the exception. If you were to decide that, having inspected the exception, you don't want to swallow it after all, you can use `Observable.Throw` to return an observable that rethrows the exception. (This is effectively the Rx equivalent to a `throw;` statement inside a C# `catch` block.) The following example uses this to swallow all IO exceptions of type IOException or any type derived from that except for `FileNotFoundException`.

```
IObservable<int> result = source.Catch<int, IOException>(
    x => x is FileNotFoundException ? Observable.Throw(tx) :
↪   Observable.Empty());
```

Replacing an exception

So far all the examples using `Catch` have either swallowed the exception by returning `Observable.Empty` or rethrown it with `Observable.Throw`. We can supply any observable we want to `Catch` (as long as its item type matches the source item type). We could use this to replace an exception with, say, a message:

```
IObservable<string> messages = stringSource.Catch<string, IOException>(
    x => Observable.Return(
        x is FileNotFoundException fnf
        ? $"Did not find {fnf.FileName}"
        : $"Was not expecting exception {tx.GetType().Name}"));
```

Alternatively, we could handle certain source exceptions by throwing a new exception, with the original one as an inner exception:

```
IObservable<int> result = source.Catch<int, IOException>(
    x => Observable.Throw<int>(
        x is FileNotFoundException fnf
        ? new InvalidOperationException("Config not available", fnf)
        : x));
```

No matter what your `Catch` does, remember that you'll see no new items from the source after it produces an error. The basic rules of Rx mean that once a source has called `OnError` on its subscriber, it must not make any further calls.

Finally

Similar to a `finally` block in C#, Rx enables us to execute some code on completion of a sequence, regardless of whether it runs to completion naturally or fails. The `Finally` extension method accepts an `Action` as a parameter. This `Action` will be invoked when the sequence terminates, regardless of whether `OnCompleted` or `OnError` was called. It will also invoke the action if the subscription is disposed of before it completes.

```
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 if the subscriber's `OnError` throws an exception, and if the source calls `OnNext` without a `try`/`catch` block, the CLR's unhandled exception reporting mechanism kicks in, and in some circumstances this can result in the application shutting down before the `Finally` operator has had an opportunity to invoke the callback. We can create this scenario with the following code:

```
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 might not be called.
source.OnError(new Exception("Fail"));
```

If you run this directly from the program's entry point, without wrapping it in a `try`/`catch`, you may or may not see the `Finally` message displayed, because exception handling works subtly differently in the case an exception reaches all the way to the top of the stack without being caught. (Oddly, it usually does run, but if you have a debugger attached, the program usually exits without running the `Finally` callback.)

This is mostly just a curiosity: application frameworks such as ASP.NET Core or WPF typically install their own top-of-stack exception handlers, and in any case you shouldn't be subscribing to a source that you know will call `OnError` without supplying an error callback. This problem only emerges because

the delegate-based Subscribe overload in use here supplies an IObserver<T> implementation that throws in its OnError. However, if you're building console applications to experiment with Rx's behaviour you are quite likely to run into this. In practice, Finally will do the right thing in more normal situations. (But in any case, you shouldn't throw exceptions from an OnError handler.)

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 callbacks; one to create the disposable resource and one to provide the sequence. This allows everything to be lazily evaluated—these callbacks are invoked when code calls Subscribe on the IObservable<T> this method returns.

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

The resource will be disposed of when the sequence terminates either with OnCompleted or OnError, or when the subscription is disposed.

OnErrorResumeNext

Just the title of this section will send a shudder down the spines of old VB developers! (For those not familiar with this murky language feature, the VB language lets you instruct it to ignore any errors that occur during execution, and to just continue with the next statement after any failure.) 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 failures, you might simply want to retry. For example, if you are running in a cloud environment, it's very common for operations to fail occasionally for no obvious reason. Cloud platforms often relocate services on a fairly regular basis for operational reasons, which means it's not unusual for operations to fail—you might make a request to a service just before the cloud provider decided to move that service to a different compute node—but for the exact same operation to succeed if you immediately retry it (because the retry request gets routed to the new node). Rx's `Retry` extension method offers the ability to retry on failure a specified number of times or until it succeeds. This works by resubscribing to the source if it reports an error.

This example uses the simple overload, which 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 a source that produces the values 0, 1, and 2, and then calls `OnError`, the output would be the numbers 0, 1, 2 repeating over an over endlessly. This output would continue forever because this example never unsubscribes, and `Retry` will retry forever if you don't tell it otherwise.

We can specify the maximum number of retries. In this next 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 we tell `Retry` the maximum number of attempts, so if we want it to retry once, you pass a value of 2—that's the initial attempt plus one retry.

```
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.

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 respond to error scenarios with something more constructive than just terminating.

Publishing operators

Hot sources need to be able to deliver events to multiple subscribers. While we can implement the subscriber tracking ourselves, it can be easier to write a degenerate source that works only for a single subscriber, and then to use one of Rx's *multicast* operators to publish it as a multi-subscriber hot source. The example in "Representing Filesystem Events in Rx" used this trick, but as you'll see in this chapter there are a few variations on the theme.

Multicast

Rx offers three operators enabling us to support multiple subscribers using just a single subscription to some underlying source: `Publish`, `PublishLast`, and `Replay`. All three of these are wrappers around Rx's `Multicast` operator, which provides the common mechanism at the heart of all of them.

`Multicast` turns any `IObservable<T>` into a `IConnectableObservable<T>` which, as you can see, just adds a `Connect` method:

```
public interface IConnectableObservable<out T> : IObservable<T>
{
    IDisposable Connect();
}
```

Since it derives from `IObservable<T>`, you can call `Subscribe` on an `IConnectableObservable<T>`, but the implementation returned by `Multicast` won't call `Subscribe` on the underlying source when you do that. It only calls `Subscribe` on the underlying source when you call `Connect`. So that we can see this in action, let's define a source that prints out a message each time `Subscribe` is called:

```
IObservable<int> src = Observable.Create<int>(obs =>
{
    Console.WriteLine("Create callback called");
    obs.OnNext(1);
    obs.OnNext(2);
    obs.OnCompleted();
```

```
    return Disposable.Empty;
});
```

Since this is only going to be invoked once no matter how many observers subscribe, `Multicast` can't pass on the `IObserver<T>`s handed to its own `Subscribe` method, because there could be any number of them. It uses a Subject as the single `IObserver<T>` that is passes to the underlying source, and this subject is also responsible for keeping track of all subscribers. If we call `Multicast` directly, we are required to pass in the subject we want to use:

```
IConnectableObservable<int> m = src.Multicast(new Subject<int>());
```

We can now subscribe to this a few times over:

```
m.Subscribe(x => Console.WriteLine($"Sub1: {x}"));
m.Subscribe(x => Console.WriteLine($"Sub2: {x}"));
m.Subscribe(x => Console.WriteLine($"Sub3: {x}"));
```

None of these subscribers will receive anything unless we call `Connect`:

```
m.Connect();
```

Note: `Connect` returns an `IDisposable`. Calling `Dispose` on that unsubscribes from the underlying source.

This call to `Connect` causes the following output:

```
Create callback called
Sub1: 1
Sub2: 1
Sub3: 1
Sub1: 2
Sub2: 2
Sub3: 2
```

As you can see, the method we passed to `Create` runs only once, confirming that `Multicast` did only subscribe once, despite us calling `Subscribe` three times over. But each item went to all three subscriptions.

The way `Multicast` works is fairly straightforward: it gets the subject do most of the work. Whenever you call `Subscribe` on an observable returned by `Multicast`, it just calls `Subscribe` on the subject. And when you call `Connect`, it just passes the subject into the underlying source's `Subscribe`. So this code would have had the same effect:

```
var s = new Subject<int>();

s.Subscribe(x => Console.WriteLine($"Sub1: {x}"));
```

```
s.Subscribe(x => Console.WriteLine($"Sub2: {x}"));
s.Subscribe(x => Console.WriteLine($"Sub3: {x}"));

src.Subscribe(s);
```

However, an advantage of `Multicast` is that it returns `IConnectableObservable<T>`, and as we'll see later, some other parts of Rx know how to work with this interface.

`Multicast` offers an overload that works in a quite different way: it is intended for scenarios where you want to write a query that uses its source observable twice. For example, we might want to get adjacent pairs of items using `Zip`:

```
IObservable<(int, int)> ps = src.Zip(src.Skip(1));
ps.Subscribe(ps => Console.WriteLine(ps));
```

(Although `Buffer` might seem like a more obvious way to do this, one advantage of this `Zip` approach is that it will never give us half of a pair. When we ask `Buffer` for pairs, it will give us a single-item buffer when we reach the end, which can require extra code to work around.)

The problem with this approach is that the source will see two subscriptions: one directly from `Zip`, and then a second one through `Skip`. If we were to run the code above, we'd see this output:

```
Create callback called
Create callback called
(1, 2)
```

Our `Create` callback ran twice. The second `Multicast` overload lets us avoid that:

```
IObservable<(int, int)> ps = src.Multicast(() => new Subject<int>(), s =>
↪  s.Zip(s.Skip(1)));
ps.Subscribe(ps => Console.WriteLine(ps));
```

As the output shows, this avoids the multiple subscriptions:

```
Create callback called
(1, 2)
```

This overload of `Multicast` returns a normal `IObservable<T>`. This means we don't need to call `Connect`. But it also means that each subscription to the resulting `IObservable<T>` causes a subscription to the underlying source. But for the scenario it is designed for this is fine: we're just trying to avoid getting twice as many subscriptions to the underlying source.

The remaining operators defined in this section, `Publish`, `PublishLast`, and `Replay`, are all wrappers around `Multicast`, each supplying a specific type of subject for you.

Publish

The `Publish` operator calls `Multicast` with a `Subject<T>`. The effect of this is that once you have called `Connect` on the result, any items produced by the source will be delivered to all subscribers. This enables me to replace this earlier example:

```
IConnectableObservable<int> m = src.Multicast(new Subject<int>());
```

with this:

```
IConnectableObservable<int> m = src.Publish();
```

These are exactly equivalent.

Because `Subject<T>` forwards all incoming `OnNext` calls to each of its subscribers immediately, and because it doesn't store any previously made calls, the result is a hot source. If you attach some subscribers before calling `Connect`, and then you attached more subscribers after calling `Connect`, those later subscribers will only receive events that occurred after they subscribed. This example demonstrates that:

```
IConnectableObservable<long> pticks = Observable
    .Interval(TimeSpan.FromSeconds(1))
    .Take(4)
    .Publish();

pticks.Subscribe(x => Console.WriteLine($"Sub1: {x} ({DateTime.Now})"));
pticks.Subscribe(x => Console.WriteLine($"Sub2: {x} ({DateTime.Now})"));

pticks.Connect();
Thread.Sleep(2500);
Console.WriteLine();
Console.WriteLine("Adding more subscribers");
Console.WriteLine();

pticks.Subscribe(x => Console.WriteLine($"Sub3: {x} ({DateTime.Now})"));
pticks.Subscribe(x => Console.WriteLine($"Sub4: {x} ({DateTime.Now})"));
```

The following output shows that we only see output for the Sub3 and Sub4 subscriptions for the final 2 events:

```
Sub1: 0 (10/08/2023 16:04:02)
Sub2: 0 (10/08/2023 16:04:02)
Sub1: 1 (10/08/2023 16:04:03)
Sub2: 1 (10/08/2023 16:04:03)
```

```
Adding more subscribers

Sub1: 2 (10/08/2023 16:04:04)
Sub2: 2 (10/08/2023 16:04:04)
Sub3: 2 (10/08/2023 16:04:04)
Sub4: 2 (10/08/2023 16:04:04)
Sub1: 3 (10/08/2023 16:04:05)
Sub2: 3 (10/08/2023 16:04:05)
Sub3: 3 (10/08/2023 16:04:05)
Sub4: 3 (10/08/2023 16:04:05)
```

As with `Multicast`, `Publish` offers an overload that provides per-top-level-subscription multicast. This lets us simplify the example from the end of that section from this:

```
IObservable<(int, int)> ps = src.Multicast(() => new Subject<int>(), s =>
↪  s.Zip(s.Skip(1)));
ps.Subscribe(ps => Console.WriteLine(ps));
```

to this:

```
IObservable<(int, int)> ps = src.Publish(s => s.Zip(s.Skip(1)));
ps.Subscribe(ps => Console.WriteLine(ps));
```

`Publish` offers overloads that let you specify an initial value. These use `BehaviorSubject<T>` instead of `Subject<T>`. The difference here is that all subscribers will immediately receive a value as soon as they subscribe. If the underlying source hasn't yet produced an item (or if `Connect` hasn't been called, meaning we've not even subscribed to the source yet) they will receive the initial value. And if at least one item has been received from the source, any new subscribers will instantly receive the latest value the source produced, and will then go on to receive any further new values.

PublishLast

The `PublishLast` operator calls `Multicast` with an `AsyncSubject<T>`. The effect of this is that the final item produced by the source will be delivered to all subscribers. You still need to call `Connect`—this determines when subscription to the underlying source occurs. But all subscribers will receive the final event regardless of when they subscribe, because `AsyncSubject<T>` remembers the final result. We can see this in action with the following example:

```
IConnectableObservable<long> pticks = Observable
    .Interval(TimeSpan.FromSeconds(0.1))
```

```
    .Take(4)
    .PublishLast();

pticks.Subscribe(x => Console.WriteLine($"Sub1: {x} ({DateTime.Now})"));
pticks.Subscribe(x => Console.WriteLine($"Sub2: {x} ({DateTime.Now})"));

pticks.Connect();
Thread.Sleep(1000);
Console.WriteLine();
Console.WriteLine("Adding more subscribers");
Console.WriteLine();

pticks.Subscribe(x => Console.WriteLine($"Sub3: {x} ({DateTime.Now})"));
pticks.Subscribe(x => Console.WriteLine($"Sub4: {x} ({DateTime.Now})"));
```

This creates a source the produces 4 values in the space of 0.4 seconds. It attaches a couple of subscribers to the `IConnectableObservable<T>` returned by `PublishLast` and then immediately calls `Connect`. Then it sleeps for 1 second, which gives the source time to complete. This means that those first two subscribers will receive the one and only value they will ever receive (the last value in the sequence) before that call to `Thread.Sleep` returns. But we then go on to attach two more subscribers. As the output shows, these also receive that same final event. They receive it later because they subscribed later, but the `AsyncSubject<T>` created by `PublishLast` is just replaying the final value it received to these late subscribers.

Replay

The `Replay` operator calls `Multicast` with a `ReplaySubject<T>`. The effect of this is that any subscribers attached before calling `Connect` just receive all events as the underlying source produces them, but any subscribers attached later effectively get to 'catch up', because the `ReplaySubject<T>` remembers events it has already seen, and replays them to new subscribers.

This example is very similar to the one used for `Publish`:

```
IConnectableObservable<long> pticks = Observable
    .Interval(TimeSpan.FromSeconds(1))
    .Take(4)
    .Replay();

pticks.Subscribe(x => Console.WriteLine($"Sub1: {x} ({DateTime.Now})"));
pticks.Subscribe(x => Console.WriteLine($"Sub2: {x} ({DateTime.Now})"));

pticks.Connect();
```

```
Thread.Sleep(2500);
Console.WriteLine();
Console.WriteLine("Adding more subscribers");
Console.WriteLine();

pticks.Subscribe(x => Console.WriteLine($"Sub3: {x} ({DateTime.Now})"));
pticks.Subscribe(x => Console.WriteLine($"Sub4: {x} ({DateTime.Now})"));
```

It creates a source that will produce items regularly for 4 seconds. It attaches two subscribers before calling `Connect`. It then waits long enough for the first two events to emerge before attaching two more subscribers. But unlike with `Publish`, those late subscribers will see the events that happened before they subscribed:

```
Sub1: 0 (10/08/2023 16:18:22)
Sub2: 0 (10/08/2023 16:18:22)
Sub1: 1 (10/08/2023 16:18:23)
Sub2: 1 (10/08/2023 16:18:23)

Adding more subscribers

Sub3: 0 (10/08/2023 16:18:24)
Sub3: 1 (10/08/2023 16:18:24)
Sub4: 0 (10/08/2023 16:18:24)
Sub4: 1 (10/08/2023 16:18:24)
Sub1: 2 (10/08/2023 16:18:24)
Sub2: 2 (10/08/2023 16:18:24)
Sub3: 2 (10/08/2023 16:18:24)
Sub4: 2 (10/08/2023 16:18:24)
Sub1: 3 (10/08/2023 16:18:25)
Sub2: 3 (10/08/2023 16:18:25)
Sub3: 3 (10/08/2023 16:18:25)
Sub4: 3 (10/08/2023 16:18:25)
```

They receive them late of course, because they subscribed late. So we see a quick flurry of events reported as `Sub3` and `Sub4` catch up, but once they have caught up, they then receive all further events immediately.

The `ReplaySubject<T>` that enables this behaviour will consume memory to store events. As you may recall, this subject type can be configured to store only a limited number of events, or not to

hold onto events older than some specified time limit. The `Replay` operator provides overloads that enable you to configure these kinds of limits.

`Replay` also supports the per-subscription-multicast model I showed for the other `Multicast`-based operators in this section.

RefCount

We saw in the preceding section that `Multicast` (and also its various wrappers) supports two usage models:

- returning an `IConnectableObservable<T>` to allow top-level control of when subscription to the underlying source occurs
- returning an ordinary `IObservable<T>`, enabling us to avoid unnecessary multiple subscriptions to the source when using a query that uses the source in multiple places (e.g., `s.Zip(s.Take(1))`), but still making one `Subscribe` call to the underlying source for each top-level `Subscribe`

`RefCount` offers a slightly different model. It enables subscription to the underlying source to be triggered by an ordinary `Subscribe`, but can still make just a single call to the underlying source. To be able to observe this, I'm going to use a modified version of the source that reports when subscription occurs:

```
IObservable<int> src = Observable.Create<int>(async obs =>
{
    Console.WriteLine("Create callback called");
    obs.OnNext(1);
    await Task.Delay(250).ConfigureAwait(false);
    obs.OnNext(2);
    await Task.Delay(250).ConfigureAwait(false);
    obs.OnNext(3);
    await Task.Delay(250).ConfigureAwait(false);
    obs.OnNext(4);
    await Task.Delay(100).ConfigureAwait(false);
    obs.OnCompleted();
});
```

Unlike the earlier example, this uses `async` and delays between each `OnNext` to ensure that the main thread has time to set up multiple subscriptions before all the items are produced. We can then wrap this with `RefCount`:

```
IObservable<int> rc = src
    .Publish()
    .RefCount();
```

Notice that I have to call `Publish` first. This is because `RefCount` expects an `IConnectableObservable<T>`. It wants to start the source only when something first subscribes. It will call `Connect` as soon as there's at least one subscriber. Let's try it:

```
rc.Subscribe(x => Console.WriteLine($"Sub1: {x} ({DateTime.Now})"));
rc.Subscribe(x => Console.WriteLine($"Sub2: {x} ({DateTime.Now})"));
Thread.Sleep(600);
Console.WriteLine();
Console.WriteLine("Adding more subscribers");
Console.WriteLine();
rc.Subscribe(x => Console.WriteLine($"Sub3: {x} ({DateTime.Now})"));
rc.Subscribe(x => Console.WriteLine($"Sub4: {x} ({DateTime.Now})"));
```

Here's the output:

```
Create callback called
Sub1: 1 (10/08/2023 16:36:44)
Sub1: 2 (10/08/2023 16:36:45)
Sub2: 2 (10/08/2023 16:36:45)
Sub1: 3 (10/08/2023 16:36:45)
Sub2: 3 (10/08/2023 16:36:45)

Adding more subscribers

Sub1: 4 (10/08/2023 16:36:45)
Sub2: 4 (10/08/2023 16:36:45)
Sub3: 4 (10/08/2023 16:36:45)
Sub4: 4 (10/08/2023 16:36:45)
```

Notice that only `Sub1` receives the very first event. That's because the callback passed to `Create` produces that immediately. Only when it invokes its first `await` does it return to the caller, enabling us to attach the second subscriber. That has already missed the first event, but as you can see it receives the 2nd and 3rd. The code waits long enough for the first three events to occur before attaching two more subscribers, and you can see that all four subscribers receive the final event.

As the name suggests `RefCount` counts the number of active subscribers. If this ever drops to 0, it will call `Dispose` on the object that `Connect` returned, shutting down the subscription. If further subscribers attach, it will restart. This example shows that:

```
IDisposable s1 = rc.Subscribe(x => Console.WriteLine($"Sub1: {x}
↪  ({DateTime.Now})"));
IDisposable s2 = rc.Subscribe(x => Console.WriteLine($"Sub2: {x}
↪  ({DateTime.Now})"));
Thread.Sleep(600);

Console.WriteLine();
Console.WriteLine("Removing subscribers");
s1.Dispose();
s2.Dispose();
Thread.Sleep(600);
Console.WriteLine();

Console.WriteLine();
Console.WriteLine("Adding more subscribers");
Console.WriteLine();
rc.Subscribe(x => Console.WriteLine($"Sub3: {x} ({DateTime.Now})"));
rc.Subscribe(x => Console.WriteLine($"Sub4: {x} ({DateTime.Now})"));
```

We get this output:

```
Create callback called
Sub1: 1 (10/08/2023 16:40:39)
Sub1: 2 (10/08/2023 16:40:39)
Sub2: 2 (10/08/2023 16:40:39)
Sub1: 3 (10/08/2023 16:40:39)
Sub2: 3 (10/08/2023 16:40:39)

Removing subscribers


Adding more subscribers

Create callback called
Sub3: 1 (10/08/2023 16:40:40)
Sub3: 2 (10/08/2023 16:40:40)
Sub4: 2 (10/08/2023 16:40:40)
Sub3: 3 (10/08/2023 16:40:41)
Sub4: 3 (10/08/2023 16:40:41)
Sub3: 4 (10/08/2023 16:40:41)
Sub4: 4 (10/08/2023 16:40:41)
```

This time, the `Create` callback ran twice. That's because the number of active subscribers dropped to 0, so `RefCount` called `Dispose` to shut things down. When new subscribers came along, it called `Connect` again to start things back up. There are some overloads enabling you to specify a `disconnectDelay`—this tells it to wait for the specified time after the number of subscribers drops to zero before disconnecting, to see if any new subscribers come along. But it will still disconnect if the specified time elapses. If that's not what you want, the next operator might be for you.

AutoConnect

The `AutoConnect` operator behaves in much the same way as `RefCount`, in that it calls `Connect` on its underlying `IConnectableObservable<T>` when the first subscriber subscribers. The difference is that it doesn't attempt to detect when the number of active subscribers has dropped to zero: once it connects, it remains connected indefinitely, even if it has no subscribers.

Although `AutoConnect` can be convenient, you need to be a little careful, because it can cause leaks: it will never disconnect automatically. It is still possible to tear down the connection it creates: `AutoConnect` accepts an optional argument of type `Action<IDiposable>`. It invokes this when it first connects to the source, passing you the `IDisposable` returned by the source's `Connect` method. You can shut it down by calling `Dispose`.

The operators in this chapter can be useful whenever you have a source that is not well suited do dealing with multiple subscribers. It provides various ways to attach multiple subscribers while only triggering a single `Subscribe` to the underlying source.

Testing Rx

Modern quality assurance standards demand a level of automated testing that can help evaluate and prevent bugs. It is good practice to have a suite of tests that verify correct behaviour and to run this as part of the build process to detect regressions early.

The `System.Reactive` source code includes a comprehensive tests suite. Testing Rx-based code presents some challenges, especially when time-sensitive operators are involved. Rx.NET's test suite includes many tests designed to exercise awkward edge cases to ensure predictable behaviour under load. This is only possible because Rx.NET was designed to be testable.

In this chapters, we'll show how you can take advantage of Rx's testability in your own code.

Virtual Time

It's common to deal with timing in Rx. As you've seen, it offers several operators that take time into account, and this presents a challenge. We don't want to introduce slow tests, because that can make test suites take too long to execute, but how might we test an application that waits for the user to stop typing for half a second before submitting a query? Non-deterministic tests can also be a problem: when there are race conditions it can be very hard to recreate these reliably.

The Scheduling and Threading chapter described how schedulers use a virtualized representation of time. This is critical for enabling tests to validate time-related behaviour. It enables us to control Rx's perception of the progression of time, enabling us to write tests that logically take seconds, but which execute in microseconds.

Consider this example, where we create a sequence that publishes values every second for five seconds.

```
IObservable<long> interval = Observable
    .Interval(TimeSpan.FromSeconds(1))
    .Take(5);
```

A naive a test to ensure that this produces five values at one second intervals would take five seconds to run. That would be no good; we 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);
```

It looks like we would have no choice but to make our test wait for a minute before running that assert. In practice, we'd want to wait a little over a minute, because if the computer running the test is busy, it might run the relevant code a bit later than we've asked. This kind of scenario is notorious for causing tests to fail occasionally even when there's no real problem in the code being tested.

Nobody wants slow, inconsistent tests. So let's look at how Rx helps us to avoid these problems.

TestScheduler

The Scheduling and Threading chapter explained that schedulers determine when and how to execute code, and that they keep track of time. Most of the schedulers we looked at in that chapter addressed various threading concerns, and when it came to timing, they all attempted to run work at the time requested. But Rx provides `TestScheduler`, which handles time completely differently. It takes advantage of the fact that schedulers control all time-related behaviour to allow us to emulate and control time.

Note: `TestScheduler` is not in the main `System.Reactive` package. You will need to add a reference to `Microsoft.Reactive.Testing` to use it.

Any scheduler maintains a queue of actions to be executed. Each action is assigned a point in time when it should be executed. (Sometimes that time is "as soon as possible" but time-based operators will often schedule work to run at some specific time in the future.) If we use the `TestScheduler` it will effectively act as though time stands still until we tell it we want time to move on.

In this example, we schedule a task to be run immediately by using the simplest `Schedule` overload. Even though this effectively asks for the work to be run as soon as possible, the `TestScheduler` always waits for us to tell it we're ready before processing newly queued work. We advance the virtual clock forward by one tick, at which point it will execute that queued work. (It runs all newly-queued "as soon as possible" work any time we advance the virtual time. If we advance the time far enough to mean that work that was previously logically in the future is now runnable, it runs that too.)

```
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);
```

The `TestScheduler` implements the `IScheduler` interface and also defines methods allowing us to control and monitor virtual time. This shows these additional methods:

```
public class TestScheduler : // ...
{
    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)
```

```
    ...
}
```

`TestScheduler` works in the same units as `TimeSpan.Ticks`. If you want to move time forward by 1 second, you can call `scheduler.AdvanceBy(TimeSpan.FromSeconds(1).Ticks)`. One tick corresponds to 100ns, so 1 second is 10,000,000 ticks.

AdvanceTo

The `AdvanceTo(long)` method sets the virtual time to the specified number of ticks. This will execute all the actions that have been scheduled up to that 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 (1 and 2 microseconds respectively).

```
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 runs everything that has been scheduled, gradually advancing virtual time as necessary if any of the work was queued for a specific time. 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()` (or `AdvanceTo` or `AdvanceBy`) again.

Stop

There is a `Stop()` method whose name seems to imply some symmetry with `Start()`. This sets the scheduler's `IsEnabled` property to false, and if `Start` is currently running, this means that it will stop inspecting the queue for further work, and will return as soon as the work item currently being processed completes.

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.

Since `Start` automatically stops when it has drained the work queue, you're under no obligation to call `Stop`. It's there only if you want to call `Start` but then pause processing part way through the test.

Schedule collision

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 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. You will need some way of enabling tests to supply different schedulers during testing. In this example, we're going to define an interface for this purpose:

```
public interface ISchedulerProvider
{
    IScheduler CurrentThread { get; }
    IScheduler Dispatcher { get; }
    IScheduler Immediate { get; }
    IScheduler NewThread { get; }
    IScheduler ThreadPool { get; }
    IScheduler TaskPool { get; }
}
```

The default implementation that we would run in production is implemented as follows:

```
public sealed class SchedulerProvider : ISchedulerProvider
{
    public IScheduler CurrentThread => Scheduler.CurrentThread;
    public IScheduler Dispatcher => DispatcherScheduler.Instance;
    public IScheduler Immediate => Scheduler.Immediate;
    public IScheduler NewThread => Scheduler.NewThread;
    public IScheduler ThreadPool => Scheduler.ThreadPool;
    public IScheduler TaskPool => Scheduler.TaskPool;
}
```

We can substitute implementations of `ISchedulerProvider` to help with testing. For example:

```
public sealed class TestSchedulers : ISchedulerProvider
{
    // Schedulers available as TestScheduler type
    public TestScheduler CurrentThread { get; }  = new TestScheduler();
    public TestScheduler Dispatcher { get; }  = new TestScheduler();
    public TestScheduler Immediate { get; }  = new TestScheduler();
    public TestScheduler NewThread { get; }  = new TestScheduler();
    public TestScheduler ThreadPool { get; }  = new TestScheduler();
```

```
    // ISchedulerService needs us to return IScheduler, but we want the
    ↪  properties
    // to return TestScheduler for the convenience of test code, so we
    ↪  provide
    // explicit implementations of all the properties to match
    ↪  ISchedulerService.
    IScheduler ISchedulerProvider.CurrentThread => CurrentThread;
    IScheduler ISchedulerProvider.Dispatcher => Dispatcher;
    IScheduler ISchedulerProvider.Immediate => Immediate;
    IScheduler ISchedulerProvider.NewThread => NewThread;
    IScheduler ISchedulerProvider.ThreadPool => ThreadPool;
}
```

Note that `ISchedulerProvider` is implemented explicitly because that interface requires each property to return an `IScheduler`, but our tests will need to access the `TestScheduler` instances directly. 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 to provide a suitable fake implementation of 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:

- That the `Price` property has prices added to it as the model produces them
- That the sequence is subscribed to on the ThreadPool
- That the `Price` property is updated on the Dispatcher i.e. the sequenceis observed on the Dispatcher
- That a timeout of 10 seconds between prices will set the ViewModel to disconnected
- 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, a modern CPU should be able to execute thousands of unit tests per second.

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 => Scheduler.Immediate;
    public IScheduler Dispatcher => Scheduler.Immediate;
    public IScheduler Immediate => Scheduler.Immediate;
    public IScheduler NewThread => Scheduler.Immediate;
    public IScheduler ThreadPool => Scheduler.Immediate;
}
```

Advanced features - ITestableObserver

The `TestScheduler` provides further advanced features. These can be useful when parts of your test setup need to run at particular virtual times.

Start(Func<IObservable>)

There are three overloads to `Start`, which are used to start an observablesequence 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:

- an observable sequence factory delegate
- the point in time to invoke the factory
- the point in time to subscribe to the observable sequence returned from the factory
- 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. These default values provide short gaps before creation and between creation and subscription, giving enough space to configure other things to happen between them. And then the disposal happens a bit later, allowing a little longer for the thing to run. There's nothing particularly magical about these default values, but if you value a lack of clutter over it being completely obvious what happens when, and are happy to rely on the invisible effects of convention, then you might prefer this. The Rx source code itself contains thousands of tests, and a very large number of them use the simplest `Start` overload, and developers working in the code base day in, day out soon get used to the idea that creation occurs at time 100, and subscription at time 200, and that test everything you need to before 1000.

CreateColdObservable

Just as we can record an observable sequence, we can also use `CreateColdObservable` to play back 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)
{...}
```

The `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.

Appendix A: What's Wrong with Classic IO Streams

In the Key Types chapter, I stated that `System.IO.Stream` is not a good fit for modelling the kinds of event streams we work with in Rx. This appendix explains why.

The abstraction that `System.IO.Stream` represents was designed as a way for an operating system to enable application code to communicate with devices that could receive and/or produce streams of bytes. This makes them a good model for the reel to reel tape storage devices that were commonplace back when this kind of stream was designed, but unnecessarily cumbersome if you just want to represent a sequence of values. Over the years, streams have been co-opted to represent an increasingly diverse range of things, including files, keyboards, network connections, and OS status information, meaning that by the time .NET came along in 2002, its `Stream` type needed a mixture of features to accommodate some quite diverse scenarios. And since not all streams are alike, it's quite common for some of these features to not to work on some streams.

IO streams were designed to support efficient delivery of fairly high volumes of byte data, often with devices that inherently work with data in big chunks. In the main scenarios for which they were designed, read and write operations would involve calls into operating system APIs, which are typically relatively expensive, so the basic read and write operations expect to work with arrays of bytes. (If you make one system call to deliver thousands of bytes, the overhead of that single call is far lower than if you work one byte at a time.) While that's good for efficiency, it can be inconvenient for developers (and irksome if you were hoping to use streams purely to represent in-process event

streams that don't actually need to make system calls, and therefore don't get to enjoy the upside of this performance/convenience trade off).

There is a standard band-aid kind of a fix for this: libraries that present streams to application code often don't represent the underlying OS stream directly. Instead, they are often *buffered*, meaning that the library will perform reads fairly large chunks, and hold recently-fetched bytes in memory until the application code asks for them. This can enable methods like .NET's single-byte `Stream.ReadByte` method to work reasonably efficiently: several thousand calls to that might correspond to only one call to the operating system API that provides access to whatever physical device the stream represents. Likewise, if you're sending data into an IO stream, a buffered stream will wait until you've supplied some minimum quantity of data (4096 bytes is a common default with certain .NET `Streams`) before it actually sends any data to its destination.

But this could be a serious problem for the kinds of event sources we represent in Rx. If an IO stream deliberately insulates you from the real movement of data, that could introduce delays that might be disastrous in a financial application where delays in delivery and receipt of information can have enormous financial consequences. And even if there aren't direct financial implications, this kind of buffering would be unhelpful in representing events in a user interface—nobody wants to have to click a button several thousand times before the application starts to act on that input.

There's also the problem that you don't always know which kind of stream you've been given. If you know for a fact that you've got an unbuffered stream representing a file on disk (because you created that stream yourself) you'd typically right quite different code than you would if you knew you had a buffered stream. But if you've written a method that takes a `Stream` argument, it's not clear what you've got, so you don't necessarily know which coding strategy is best.

Another problem is that because they are byte-oriented, there's no such thing as a `System.IO.Stream` that produces more complex values. If you want a stream of `int` values (which isn't a *much* more complex idea than a stream of *byte* values) `System.IO.Stream` does nothing to help you, and until very recently it might even hinder you. If you use the normal `Read` or `ReadAsync` methods, you can try reading four bytes at a time but a `System.IO.Stream` is at liberty to decide that it's only going to return three. (The reason streams are allowed to be petty in this way is that the original design presumes that a stream represents some underlying device that might inherently work with fixed size units of data. Disk drives and SSDs are incapable of reading or writing individual bytes; instead, each operation works with some whole number of 'sectors' each of which are hundreds or thousands of bytes long. So a read operation might simply be unable to give you exactly as many bytes as you asked for. This can also come into play for a stream that represents data coming in over the network: such streams might already have received some data, but less than you've asked for, and they might decide to return what they've already got instead of making you wait until the next network message arrives.) It's now the consuming code's problem to work out how to deal with that.

.NET 7.0 finally fixed this problem (only about two decades after `Stream` first appeared) by adding the `ReadExactly` and `ReadExactlyAsync` methods, but if you have to target .NET Framework, these methods are unavailable and you still have to solve this entirely yourself.

Even if you use the new methods (or you write wrappers to deal with these issues caused by `Stream`'s origins as an abstraction for a magnetic tape storage device) there are still shortcomings. If you want the type system to help you to distinguish between a stream of `int` values and a stream of `float` values, `Stream` won't help you. You'll end up needing some different abstraction that has a type parameter. Something like `IObservable<T>` in fact. The fact that we know exactly what shape of data to expect from `IObservable<T>` is critical to making many of the LINQ operators it supports practical.

Another potential source of confusion is Unix's "everything is a file" design feature. The operating system represents all manner of things through the same OS abstractions as files, and this simplifies the OS design, and in some cases enables you to apply tools originally designed for files in creative ways. But the downside is that some streams are finicky. It's possible to end up with a stream that looks like any other from a .NET type system point of view, but which only works if you read or write in blocks of some particular size.

Conversely, Rx's strictly defined rules for how observable sources interact with their subscribers means we know exactly where we stand.

There isn't a clear model for how streams might support multiple subscribers. Programs such as the Unix `tail` command are able to 'follow' changes to a file, but the way they achieve this is nothing like as simple as two observers both calling `Subscribe`.

And these are just the problems on the consumer side. It's not much fun if you want to implement a source of events as a `Stream` either. To implement your own type that derives from `Stream`, you'll need to implement all ten of the abstract members it defines: 5 properties and 5 methods. This is a far cry from the simple ways `System.Reactive` provides to implement an Rx event source.

Appendix B: 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.

TODO: we recently made SingleAssignmentDisposableValue public after a request to do so.

TODO: ICancelable?

TODO: StableCompositeDisposable?

TODO: fit this in?

```
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.

Appendix C: 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 only if you need to unsubscribe from them early.
- Always provide an `OnError` handler.
- Avoid blocking operators such as `First`, `FirstOrDefault`, `Last`, `LastOrDefault`, `Single`, `SingleOrDefault` and `ForEach`.; use the non-blocking alternative such as `FirstAsync`.
- Avoid switching back and forth between `IObservable<T>` and `IEnumerable<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` keyword

- Name your observables well, i.e. avoid using variable names like `query`, `q`, `xs`, `ys`, `subject` etc.
- Avoid creating side effects. If you really can't avoid it, don't bury the side effects in callbacks for operators designed to be use functionally such as `Select` or `Where`. be explicit by using the Do operator.
- Where possible, prefer `Observable.Create` to subjects as a means of defining new Rx sources.
- Avoid creating your own implementations of the `IObservable<T>` interface. Use `Observable.Create` (or subjects if you really need to).
- Avoid creating your own implementations of the `IObserver<T>` interface. Favor using the `Subscribe` extension method overloads instead.
- The application should define the concurrency model.

    - If you need to schedule deferred work, use schedulers
    - The `SubscribeOn` and `ObserveOn` operators should always be right before a `Subscribe` method. (So don't sandwich it, e.g. `source.SubscribeOn(s).Where(x => x.Foo)`.)

Appendix C: Rx's Algebraic Underpinnings

Rx operators can be combined together in more or less any way you can imagine, and they generally just work. The fact that this works is not merely a happy accident. In general, integration between software components is often one of the largest sources of pain in software development, so the fact that it works so well is remarkable. This is in large part thanks to the fact that Rx relies on some underlying theory. Rx has been designed so that you don't need to know these details to use it, but curious developers typically want to know these things.

The earlier sections of the book have already talked about one formal aspect of Rx: the contract between observable sources and their observables. There is a clearly defined grammar for what constitutes acceptable use of `IObservable<T>`. This goes beyond what the .NET type system is able to enforce, so we are reliant on code doing the right thing. However, the `System.Reactive` library does always adhere to this contract, and it also has some guard types in place that detect when application code has not quite played by the rules, and to prevent this from wreaking havoc.

The `IObservable<T>` grammar is important. Components rely on it to ensure correct operation. Consider the `Where` operator, for example. It provides its own `IObserver<T>` interface with which it subscribes to the underlying source, and this receives items from that source, and then decides which to forward to the observer that subscribed to the `IObservable<T>` presented by `Where`. You could imagine it looking something like this:

```
public class OverSimplifiedWhereObserver<T> : IObserver<T>
{
    private IObserver<T> downstreamSubscriber;
    private readonly Func<T, bool> predicate;

    public OverSimplifiedWhereObserver(IObserver<T> downstreamSubscriber,
    ↪  Func<T, bool> predicate)
    {
        this.downstreamSubscriber = downstreamSubscriber;
        this.predicate = predicate;
    }

    public void OnNext(T value)
    {
        if (this.predicate(value))
        {
            this.downstreamSubscriber.OnNext(value);
        }
    }
```

```
    public void OnCompleted()
    {
        this.downstreamSubscriber.OnCompleted();
    }

    public void OnError(Exception x)
    {
        this.downstreamSubscriber.OnCompleted(x);
    }
}
```

This does not take any explicit steps to follow the `IObserver<T>` grammar. But it doesn't need to if the source to which it is subscriber obeys those rules. Since this only ever calls its subscriber's `OnNext` in its own `OnNext`, and likewise for `OnCompleted` and `OnError`, then as long as the underlying source to which this operator is subscribed obeys the rules for how to call those three methods, this class will in turn also follow those rules automatically.

In fact, `System.Reactive` is not quite that trusting. It does have some code that detects certain violations of the grammar, but even these measure just ensure that the grammar is adhered to once execution enters Rx. There are some checks at the boundaries of the system, but Rx's innards rely heavily on the fact that upstream sources will abide by the rules.

However, the grammar for `IObservable<T>` is not the only place where Rx relies on formalism to ensure correct operation. It also depends on a particular set of mathematical concepts:

- Monads
- Catamorphisms
- Anamorphisms

Standard LINQ operators can be expressed purely in terms of these three ideas.

These concepts come out of a branch of mathematics called category theory, a pretty abstract branch of mathematics concerned with mathematical structures. In the late 1980s, a few computer scientists were exploring this area of maths with a view to using them to model the behaviour of programs. Eugenio Moggi (an Italian computer scientist who was, at the time, working at the University of Edinburgh) is generally credited for realising that monads in particular are well suited to describing computations, as his 1991 paper, Notions of computations and monads explains. These theoretical ideas and were incorporated into the Haskell programming language, primarily by Philip Wadler and Simon Peyton Jones, who published a proposal for monadic handling of IO in 1992, which, by 1996, had been fully incorporated into Haskell in its v1.3 release to enable programs' handling of input and output (e.g., handling user input, or writing data to files) to to work in a way that was underpinned by strong mathematical foundations. This has widely been recognized as a significant improvement on Haskell's earlier attempts to model the messy realities of IO in a purely functional language.

Why does any of this matter? These mathematical foundations are exactly why LINQ operators can be freely composed.

The mathematical discipline of category theory has developed a very deep understanding of various mathematical structures, and one of the most useful upshots for programmers is that it offers certain rules which, if followed, can ensure that software elements will behave well when combined together. This is, admittedly, a rather hand-wavey explanation. If you'd like a detailed explanation of exactly how category theory can be applied to programming, and why it is useful to do so, I can highly recommend Bartosz Milewski's 'Category Theory for Programmers'. The sheer volume of information available there should make it clear why I'm not about to attempt a full explanation in this appendix. Instead, my goal is just to outline the basic concepts, and explain how they correspond to features of Rx.

Monads

Monads are the most important mathematical concept underpinning LINQ's (and therefore Rx's) design. It's not necessary to have the faintest idea of what a monad is to be able to use Rx. The most important fact is that their mathematical characteristics (and in particular, their support for composition) are what enable Rx operators to combine together freely. From a practical perspective, all that really matters is that it just works, but if you've read this far, that probably won't satisfy you.

It is often hard to describe precisely what mathematical objects really are, because they are inherently abstract. So before I get to the definition of a monad, it may be helpful to understand how LINQ uses this concept. LINQ treats a monad as a general purpose representation of a container of items. As developers, we know that there are many kinds of things that can contain items. There are arrays, and other collection types such as `IList<T>`. There are also databases, and although there are many ways in which a database table is quite different from an array, there are also some ways in which they are similar. The basic insight underpinning LINQ is that there is a mathematical abstraction that captures the essence of what containers have in common. If we determine that some .NET type represents a monad, then all of the work that mathematicians have done over the years to understand the characteristics and behaviours of monads will be applicable to that .NET type.

For example, `IEnumerable<T>` is a monad, as is `IQueryable<T>`. And crucially for Rx, `IObservable<T>` is as well. LINQ's design relies on the properties of monads, so if you can determine that some .NET type is a monad, then it is a candidate for a LINQ implementation. (Conversely, if you try to create a LINQ provider for a type that is not a monad, you are likely to have problems.)

So what are these characteristics that LINQ relies on? The first relates directly to containment: it must be possible to take some value and put it inside your monad. You'll notice that all the examples I've given so far are generic types, and that's no coincidence: monads are essentially type constructors,

and the type argument indicates the kind of thing you want the monad to contain. So given some value of type T, it must be possible to wrap that in a monad for that type. Given an `int` we can get an `IEnumerable<int>`, and if we couldn't do that, `IEnumerable<T>` would not be monadic. The second characteristic is slightly harder to pin down without getting lost in high abstraction, but it essentially boils down to the idea that if we have functions that we can apply to individual contained items, and if those functions compose in useful ways, we can create new functions that operate not on individual values but on the containers, and crucially, those functions can also be composed in the same ways.

This enables us to work with entire containers as freely as we can work with individual values.

The monadic operations: return and bind

We've just seen that monads aren't just a type. They need to supply certain operations. This first operation, the ability to wrap a value in the monad, is sometimes called *unit* in mathematical texts, but in a computing context it is more often known as *return*. This is how `Observable.Return` got its name.

There doesn't technically need to be an actual function. The monadic laws are satisfied as long as some mechanism is available to put a value into the monad. For example, unlike `Observable`, the `Enumerable` type does *not* define a `Return` method, but it doesn't matter. You can just write `new[] { value }`, and that's enough.

Monads are required to provide just one other operation. The mathematical literature calls it *bind*, some programming systems call it `flatMap`, and LINQ refers to it as `SelectMany`. This is the one that tends to cause the most head scratching, because although it has a clear formal definition, it's harder to say what it really does than with *return*. However, we're looking at monads through their ability to represent containers, and this offers a fairly straightforward way to understand bind/`SelectMany`: it lets us take a container where every item is a nested container (e.g., an array of arrays, or an `IEnumerable<IEnumerable<T>>`) and flatten it out. For example, a list of lists would become one list, containing every item from every list. As we'll soon see, this is not obviously related to the formal mathematical definition of bind, which is altogether more abstract, but it is compatible with it, which is all that's needed for us to enjoy the fruits of the mathematicians' labours.

Critically, to qualify as a monad, the two operations just described (return and bind) must conform to certain rules, or *laws* as they are often described in the literature. There are three laws, and all of them govern how the bind operation works, and two of these are concerned with how return and bind interact with one another. These laws are the foundation of the composability of operations based on monads. The laws are somewhat abstract, so it isn't exactly obvious *why* they enable this, but they are non-negotiable. If your type and operations don't follow these laws, then you don't have a monad, so

you can't rely on the characteristics monads guarantee.

So what does bind actually look like? Here's how it looks for `IEnumerable<T>`:

```
public static IEnumerable<TResult> SelectMany<TSource,TCollection,TResult>
↪   (
    this IEnumerable<TSource> source,
    Func<TSource,IEnumerable<TResult>> selector);
```

So it is a function that takes two inputs. The first is an `IEnumerable<TSource>`. The second input is itself a function which, when supplied with a `TSource` produces an `IEnumerable<TResult>`. And when you invoke `SelectMany` (aka *bind*) with these two arguments, you get back an `IEnumerable<TResult>`. Although formal definition of bind requires it to have this shape, it doesn't dictate any particular behaviour—anything that conforms to the laws is acceptable. But in the context of LINQ, we do expect a specific behaviour: this will invoke the function (the second argument) once for every `TSource` in the source enumerable (the first argument), and then collect all of the `TResult` values produced by all of the `IEnumerable<TResult>` collections returned by all of the invocations of that function, wrapping them as a one big `IEnumerable<TResult>`. In this specific case of `IEnumerable<T>` we could describe `SelectMany` as getting one output collection for each input value, and then concatenating all of those output collections.

But we've now got a little too specific. Even if we're looking specifically at LINQ's use of monads to represent generalised containers, `SelectMany` doesn't necessarily entail concatenation. It merely requires that the container returned by `SelectMany` contains all of the items produced by the function. Concatenation is one strategy, but Rx does something different. Since observables tend to produce values as and when they want to, the `IObservable<TResult>` returned by `Observable.SelectMany` just produces a value each time any of the individual per-`TSource` `IObservable<TResult>`s produced by the function produces a value. (It performs some synchronization to ensure that it follows Rx's rules for calls into `IObserver<T>`, so if one of these observables produces a value while a call to the subscriber's `OnNext` is in progress, it will wait for that to return before pushing the next value. But other than that, it just pushes all values straight through.) So the source values are essentially interleaved here, instead of being concatenated. But the broader principle—that the result is a container with every value produced by the callback for the individual inputs—applies.

The mathematical definition of a monadic bind has the same essential shape, it just doesn't dictate a particular behaviour. So any monad will have a bind operation that takes two inputs: an instance of the monadic type constructed for some input value type (`TSource`), and a function that takes a `TSource` as its input and produces an instance of the monadic type constructed for some output value type (`TResult`). When you invoke bind with these two inputs the result is an instance of the monadic type constructed for the output value type. We can't precisely represent this in C#'s type

system, but this sort of gives the broad flavour:

```
// An impressionistic sketch of the general form of a monadic bind
public static M<TResult> SelectMany<TSource,TCollection,TResult> (
    this M<TSource> source,
    Func<TSource,M<TResult>> selector);
```

Substitute your chosen monadic type (`IObservable<T>`, `IEnumerable<T>`, `IQueryable<T>`, or whatever) for `M<T>`, and that tells you what bind should look like for that particular type.

But it's not enough to provide the two functions, return and bind. Not only must they have the correct shape, they must also abide by the laws.

The monadic laws

So a monad consists of a type constructor (e.g., `IObservable<T>`) and two functions, `Return` and `SelectMany`. (From now on I'm just going to use these LINQy names.) But to qualify as a monad, these features must abide by three "laws" (given in a very compact form here, which I'll explain in the following sections):

1. `Return` is a 'left-identity' for `SelectMany`
2. `Return` is a 'right-identity' for `SelectMany`
3. `SelectMany` should be, in effect, associative

Let's look at each of these in a bit more detail

Monadic law 1: `Return` is a 'left-identity' for `SelectMany` This law means that if you pass some value `x` into `Return` and then pass the result as one of the inputs to `SelectMany` where the other input is a function `SomeFunc`, then the result should be identical to just passing `x` directly into `SomeFunc`. For example:

```
// Given a function like this:
//  IObservable<bool> SomeFunc(int)
// then these two should be identical.
IObservable<bool> o1 = Observable.Return(42).SelectMany(SomeFunc);
IObservable<bool> o2 = SomeFunc(42);
```

Here's an informal way to understand this. `SelectMany` pushes every item in its input container through `SomeFunc`, and each such call produces a container of type `IObservable<bool>`, and it collects all these containers together into one big `IObservable<bool>` that contains item from all of the individual `IObservable<bool>` containers. But in this example, the input we provide to `SelectMany` contains just a single item, meaning that there's no collection work to be

done—SelectMany is going to invoke our function just once with that one and only input, and that's going to produce just one output IObservable<bool>. SelectMany is obliged to return an IObservable<bool> that contains everything in the single IObservable<bool> it got from that single call to SomeFunc. There's no actual further processing for it to do in this case—since there was only one call to SomeFunc it doesn't need to combine items from multiple containers in this case: that single output produced by the single call to SomeFunc contains everything that should be in the container that SelectMany is going to return. We can therefore just invoke SomeFunc directly with the single input item.

It would be odd if SelectMany did anything else. If o1 were different in some way, that would mean one of three things:

- o1 would contain items that aren't in o2 (meaning it had somehow included items *not* produced by SomeFunc)
- o2 would contain items that aren't in o1 (meaning that SelectMany had omitted some of the items produced by SomeFunc)
- o1 and o2 contain the same items but are in some detectable sense specific to the monad type in use different (e.g., the items came out in a different order)

So this law essentially formalizes the idea that SelectMany shouldn't add or remove items, or fail to preserve characteristics that the monad in use would normally preserve such as ordering.

Monadic law 2: Return is a 'left-identity' for SelectMany This law means that if you pass Return as the function input to SelectMany, and then pass some value of the constructed monadic type in as the other argument, you should get that same value as the output. For example:

```
// These two should be identical.
IObservable<int> o1 = GetAnySource();
IObservable<int> o2 = o1.SelectMany(Observable.Return);
```

By using Return as the function for SelectMany, we are essentially asking to take every item in the input container and to wrap it in its very own container (Return wraps a single item) and then to flatten all of those containers back out into a single container. We are adding a layer of wrapping and then removing it again, so it makes sense that this should have no effect.

Monadic law 3: SelectMany should be, in effect, associative Suppose we have two functions, Tx1 and Tx2, each of form suitable for passing as the argument to SelectMany. There are two ways we could apply these:

```
// These two should be identical.
IObservable<int> o1 = source.SelectMany(x => Tx1(x).SelectMany(Tx2));
IObservable<int> o2 = source.SelectMany(x => Tx1(x)).SelectMany(Tx2);
```

The difference here is just a slight change in the placements of the parentheses: is the second call to `SelectMany` invoked inside the function passed to the first `SelectMany`, or is it invoked on the result of the first `SelectMany`. This code adjusts the layout, and also replaces the lambda `x => Tx1(x)` with the exactly equivalent `Tx1`, which might make the difference in structure a bit easier to see:

```
IObservable<int> o1 = source
    .SelectMany(x => Tx1(x).SelectMany(Tx2));
IObservable<int> o2 = source
    .SelectMany(Tx1)
    .SelectMany(Tx2);
```

The third law says that either of these should have the same effect. It shouldn't matter whether the second `SelectMany` call (for `Tx2`) happens "inside" or after the first `SelectMany` call.

An informal way to think about this is that `SelectMany` effectively applies two operations: a transformation and an unwrap. The transformation is defined by whatever function you pass to `SelectMany`, but because that function returns the monad type (in LINQ terms it returns a container which may contain any number of items) `SelectMany` unwraps each container returned when it passes an item to the function, in order to collect all the items together into the single container it ultimately returns. When you nest this sort of operation, it doesn't matter which order that unwrapping occurs in. For example, consider these functions:

```
IObservable<int> Tx1(int i) => Observable.Range(1, i);
IObservable<string> Tx2(int i) => Observable.Return(i.ToString());
```

The first converts a number into a range of numbers of the same length. 1 becomes `[1]`, 3 becomes `[1,2,3]` and so on. Before we get to `SelectMany`, imagine what will happen if we use this with `Select` on an observable source that produces a range of numbers:

```
IObservable<int> input = Observable.Range(1, 3); // [1,2,3]
IObservable<IObservable<int>> expandTx1 = input.Select(Tx1);
```

We get a sequence of sequences—`expand2` is effectively this:

```
[
    [1],
    [1,2],
    [1,2,3],
]
```

If instead we had used `SelectMany`:

```
IObservable<int> expandTx1Collect = input.SelectMany(Tx1);
```

it would apply the same transformation, but then flatten the results back out into a single list:

```
[
    1,
    1,2,
    1,2,3,
]
```

I've kept the line breaks to emphasize the connection between this and the preceding output, but I could just have written `[1,1,2,1,2,3]`.

If we then want to apply the second transform, we could use `Select`:

```
IObservable<IObservable<string>> expandTx1CollectExpandTx2 =
↪   expandTx1Collect
    .SelectMany(Tx1)
    .Select(Tx2);
```

This passes each number in `expandTx1Collect` to `Tx2`, which converts it into a sequence containing a single string:

```
[
    ["1"],
    ["1"],["2"],
    ["1"],["2"],["3"]
]
```

But if we use `SelectMany` on that final position too:

```
IObservable<string> expandTx1CollectExpandTx2Collect = expandTx1Collect
    .SelectMany(Tx1)
    .SelectMany(Tx2);
```

it flattens these back out into just the strings:

```
[
    "1",
    "1","2",
    "1","2","3"
]
```

The associative-like requirement says it shouldn't matter if we apply `Tx1` inside the function passed to the first `SelectMany` instead of applying it to the result of that first `SelectMany`. So instead of starting with this:

```
IObservable<IObservable<int>> expandTx1 = input.Select(Tx1);
```

we might write this:

```
IObservable<IObservable<IObservable<string>>> expandTx1ExpandTx2 =
    input.Select(x => Tx1(x).Select(Tx2));
```

That's going to produce this:

```
[
    [["1"]],
    [["1"],["2"]],
    [["1"],["2"],["3"]]
]
```

If we change that to use `SelectMany` for the nested call:

```
IObservable<IObservable<string>> expandTx1ExpandTx2Collect =
    input.Select(x => Tx1(x).SelectMany(Tx2));
```

That's going to flatten out the inner items (but we're still using `Select` on the outside, so we still get a list of lists) producing this:

```
[
    ["1"],
    ["1","2"],
    ["1","2","3"]
]
```

And then if we change that first `Select` to `SelectMany`:

```
IObservable<string> expandTx1ExpandTx2CollectCollect =
    input.SelectMany(x => Tx1(x).SelectMany(Tx2));
```

it will flatten that outer layer of lists, giving us:

```
[
    "1",
```

```
    "1","2",
    "1","2","3"
]
```

That's the same final result we got earlier, as the 3rd monad law requires.

To summarize, the two processes here were:

- expand and transform Tx1, flatten, expand and transform Tx2, flatten
- expand and transform Tx1, expand and transform Tx2, flatten, flatten

Both of these apply both transforms, and flatten out the extra layers of containment added by these transforms, and so although the intermediate steps looked different, we ended up with the same result, because it doesn't matter whether you unwrap after each transform, or you perform both transforms before unwrapping.

Why these laws matter These three laws directly reflect laws that hold true for composition of straightforward functions over numbers. If we have two functions, f, and g, we could write a new function *h*, defined as $g(f(x))$. This way of combining function is called *composition*, and is often written as $g \circ f$. If the identity function is called *id*, then the following statements are true:

- $id \circ f$ is equivalent to just f
- $f \circ id$ is equivalent to just f
- $(f \circ g) \circ s$ is equivalent to $f \circ (g \circ s)$

These correspond directly to the three monad laws. Informally speaking, this reflects the fact that the monadic bind operation (`SelectMany`) has deep structurally similarity to function composition. This is why we can combine LINQ operators together freely.

Recreating other operators with `SelectMany`

Remember that there are three mathematical concepts at the heart of LINQ: monads, anamorphisms and catamorphisms. So although the preceding discussion has focused on `SelectMany`, the significance is much wider because we can express other standard LINQ operators in terms of these primitives. For example, this shows how we could implement `Where` using just `Return` and `SelectMany`:

```
public static IObservable<T> Where<T>(this IObservable<T> source, Func<T,
↪  bool> predicate)
{
    return source.SelectMany(item =>
```

```
        predicate(item)
            ? Observable.Return(item)
            : Observable.Empty<T>());
}
```

This implements `Select`:

```
public static IObservable<TResult> Select<TSource, TResult>(
    this IObservable<TSource> source, Func<TSource, TResult> f)
{
    return source.SelectMany(item => Observable.Return(f(item)));
}
```

Some operators require anamorphisms or catamorphisms, so let's look at those now.

Catamorphisms

A catamorphism is essentially a generalization of any kind of processing that takes every item in a container into account. In practice in LINQ, this typically means processes that inspect all of the values, and produce a single value as a result, such as Observable.Sum. More generally, aggregation of any kind constitutes catamorphism. The mathematical definition of catamorphism is more general than this—it doesn't necessarily have to reduce things all the way down to a single value for example—but for the purposes of understanding LINQ, this container-oriented viewpoint is the most straightforward way to think about this construct.

Catamorphisms are one of the fundamental building blocks of LINQ because you can't construct catamorphisms out of the other elements. But there are numerous LINQ operators that can be built out of LINQ's most elemental catamorphism, the `Aggregate` operator. For example, here's one way to implement `Count` in terms of `Aggregate`:

```
public static IObservable<int> MyCount<T>(this IObservable<T> items)
    => items.Aggregate(0, (total, _) => total + 1);
```

We could implement Sum thus:

```
public static IObservable<T> MySum<T>(this IObservable<T> items)
    where T : INumber<T>
    => items.Aggregate(T.Zero, (total, x) => x + total);
```

This is more flexible than the similar sum example I showed in the Aggregation chapter, because that worked only with an `IObservable<int>`. Here I'm using the *generic math* feature added in C# 11.0 and .NET 7.0 there to enable `MySum` to work across any number-like type. But the basic principle of operation is the same

If you came here for the theory, it probably won't be enough for you just to see that the various aggregating operators are all special cases of `Aggregate`. What really is a catamorphism? One definition is as "the unique homomorphism from an initial algebra into some other algebra" but as is typical with category theory, that's one of those explanations that's easiest to understand if you already understand the concepts it's trying to describe. If you try to understand this description in terms of the school mathematics form of algebra, in which we write equations where some values are represented by letters, it's hard to make sense of this definition. That's because catamorphisms take a much more general view of what constitutes "algebra," meaning essentially some system by which expressions of some kind can be constructed and evaluated.

To be more precise, Catamorphisms are described in relation to something called an F-algebra. That's a combination of three things:

1. a Functor, *F*, that defines some sort of structure over some category *C*
2. some object *A* in the category *C*
3. a morphism from *F A* to *A* that effectively evaluates the structure

But that opens up more questions than it answers. So let's start with the obvious one: what's a Functor? From a LINQ perspective, it's essentially anything that implements `Select`. (Some programming systems call this `fmap`.) From our container-oriented viewpoint it's two things: 1) a type constructor that is container-like (e.g. something like `IEnumerable<T>` or `IObservable<T>`) and 2) some means of applying a function to everything in the container. So if you have a function that converts from `string` to `int`, a Functor lets you apply that in one operation to everything it contains.

The combination of `IEnumerable<T>` and its `Select` extension method is a Functor. You can use `Select` to convert an `IEnumerable<string>` to an `IEnumerable<int>`. `IObservable<T>` and its `Select` form another Functor—we can use these to get from an `IObservable<string>` to an `IObservable<int>`. What about that "over some category *C*" part? That alludes to the fact that the mathematical description of a Functor is rather broader. When developers use category theory, we generally stick to a category that represents types (as in programming language types like `int`) and functions. (Strictly speaking a Functor maps from one category to another, so in the most general case, a Functor maps objects and morphisms in some category *C* into objects and morphisms in some category *D*. But since for programming purposes, we are always using the category representing types, so for the Functors we use *C* and *D* will be the same thing. Strictly speaking this means we should be calling them Endofunctors, but nobody seems to bother—in practice we use the name for the more general form, Functor, and it's just taken as read that we mean an Endofunctor over the category of types and functions.)

So, that's the Functor part. Let's move onto 2, "some object *A* in the category *C*." Well *C* is the Functor's category, and we just established that objects in that category are types, so *A* here might be the `string` type. If our chosen Functor is the combination of `IObservable<T>` and its `Select` method, then *F*

A would be `IObservable<string>`.

So what about the "morphisms" in 3? Again, for our purposes we're just using Endofunctors over types and functions, so in this context, morphisms are just functions. So we could recast this as:

1. some container-like generic type such as `IObservable<T>`
2. an item type A (e.g., `string`, or `int`)
3. a function that takes an `IObservable<A>` and returns a value of type A (e.g. `Observable.Aggregate<A>`)

This is a good deal more specific—category theory is typically concerned with capturing the most general truths about mathematical structures, and this reformulation throws that generality away. However, from the perspective of a programmer looking to lean on mathematical theory, this is fine—as long as what we're doing fits the F-algebra mould, all the general results that mathematicians have derived will apply to our more specialized application of the theory.

Nonetheless, to give you an idea of the sorts of things the general concept of F-algebras can enable, it's possible for the Functor to be a type that represents expressions in a programming language, and you could create a F-algebras that evaluates those expressions. That's a similar idea to LINQ's `Aggregate`, in that it walks over the entire structure represented by the Functor (every element in a list if it's an `IEnumerable<T>`; every subexpression if you're representing an expression) and reduces the whole thing to a single value, but instead of our Functor representing a sequence of things, it has a more complex structure.

So that's an F-algebra. And from a theory point of view, it's important that the third part doesn't necessarily have to reduce things. Theoretically, the types can be recursive, with the item type *A* being *F A*. (This is important for inherently recursive structures such as expressions.) And there is typically a maximally general F-algebra in which the function (or morphism) in 3 only deals with the structure, and which doesn't actually perform any reduction at all. (E.g., given some expression syntax, you could imagine code that embodies all of the knowledge required to walk through every single subexpression of an expression, but which has no particular opinion on what processing to apply.) The idea of a catamorphism is that there are less general F-algebras available for the same Functor that are less general.

For example, with `IObservable<T>` the general purpose notion is that every item produced by some source can be processed by repeatedly applying some function of two arguments, one of which is a value of type T from the container, and the other of which is some sort of accumulator, representing all information aggregated so far. And this function would return the updated accumulator, ready to be passed into the function again along with the next T. And then there are more specific forms in which specific accumulation logic (e.g., summation, or determination of a maximum value) is applied. Technically, the catamorphism here is the connection from the general form to the more specialized form. But in practice it's common to refer to the specific specialized forms (such as `Sum` or `Average`) as catamorphisms.

Remaining inside the container

Although in general a catamorphism can strip off the container (e.g., Sum for `IEnumerable<int>` produces an `int`), this isn't absolutely necessary, and with Rx most catamorphisms don't do this. As described in the threading and scheduling chapter's Lock-ups section, blocking some thread while waiting for a result that will only occur once an `IObservable<T>` has done something in particualr (e.g., if you want to calculate the sum of items, you have to wait until you've seen all the items) is a recipe for deadlock in practice.

For this reason, most of the catamorphisms perform some sort of reduction but continue to produce a result wrapped in an `IObservable<T>`.

Anamorphisms

Anamorphisms are, roughly speaking, the opposite of catamorphisms. While catamorphisms essentially collapse some sort of structure down to something simpler, an anamorphism expands some input into a more complex structure. For example, given some number (e.g., 5) we could imagine a mechanism for turning that into a sequence with the specified number of elements in it (e.g., [0,1,2,3,4]).

In fact we don't have to imagine such a thing: that's what `Observable.Range` does.

We could think of the monadic `Return` operation as a very simple anamorphism. Given some value of type `T`, `Observable.Return` expands this into an `IObservable<T>`. Anamorphisms are essentially the generalization of this sort of idea.

The mathematical definition of an anamorphism is "the assignment of a coalgebra to its unique morphism to the final coalgebra of an endofunctor." This is the "dual" of the definition of a catamorphism, which from a category theory point of view essentially means that you reverse the direction of all of the morphisms. In our not-completely-general application of category theory, the morphisms in question here are the reduction of items to some output in a catamorphism, and so with an anamorphism this turns into the expansion of some value into the some instance of the container type (e.g., from an `int` to an `IObservable<int>`).

I'm not going to go into as much detail as with catamorphisms. Instead, I'm going to point out the key part at the heart of this: the most general F-algebra for a Functor embodies some understanding of the essential structure of the Functor, and catamorphisms make use of that to define various reductions. Similarly, the most general coalgebra for a Functor also embodies some understanding of the essential structure of the Functor and anamorphisms make use of that to define various expansions.

`Observable.Generate` represents this most general capability: it has the capability to produce an `IObservable<T>` but needs to be supplied with some specialized expansion function to generate

any particular observable.

So much for theory

Now we've reviewed the theoretical concepts behind LINQ, let's step back and look at how we use them. We have three kinds of operations:

- Anamorphisms enter the sequence: `T1 --> IObservable<T2>`
- Bind modifies the sequence. `IObservable<T1> --> IObservable<T2>`
- Catamorphisms leave the sequence. Logically `IObservable<T1> --> T2`, but in practice typically `IObservable<T1> --> IObservable<T2>` where the output observable produces just a single value

As an aside, bind and catamorphism were made famous by Google's MapReduce framework from Google. Here Google refer to Bind and Catamorphism by names more commonly used in some functional languages, Map and Reduce.

Most Rx operators are actually specializations of the higher order functional concepts. To give a few examples:

- Anamorphisms:
  - `Generate`
  - `Range`
  - `Return`
- Bind:
  - `SelectMany`
  - `Select`
  - `Where`
- Catamorphism:
  - `Aggregate`
  - `Sum`
  - `Min` and `Max`

Amb

The `Amb` method was a new concept to me when I started using Rx. This function was first introduced by John McCarthy, in his 1961 paper 'A Basis for a Mathematical Theory of Computation' in the Proceedings

of the Western Joint Computer Conference. (A digital copy of this is hard to find, but a later version was published in 1963 in 'Computer Programming and Format Systems'.) It is an abbreviation of the word *Ambiguous*. Rx diverges slightly from normal .NET class library naming conventions here in using this abbreviation, partly because amb is the established name for this operator, but also as a tribute to McCarthy, whose work was an inspiration for the design of Rx.

But what does Amb do? The basic idea of an *ambiguous function* is that we are allowed to define multiple ways to produce a result, and that some or all of these might in practice prove unable to produce a result. Suppose we've defined some ambiguous function called `equivocate`, and perhaps that for some particular input value, all of `equivocate`'s component parts—all the different ways we gave it of calculating a result—are unable to process the value. (Maybe every one of them divides a number by the input. If we supply an input of `0`, then none of the components can produce a value for this input because they would all attempt to divide by 0.) In cases such as these where none of `equivocate`'s component parts is able to produce a result, `equivocate` itself is unable to produce a result. But suppose we supply some input where exactly one of its component parts is able to produce a result. In that case this result becomes the result of `equivocate` for that input.

So in essence, we're supplying a bunch of different ways to process the input, and if exactly one of those is able to produce a result, we select that result. And if none of the ways of processing the input produces anything, then our ambiguous function also produces nothing.

Where it gets slightly more weird (and where Rx departs from the original definition of amb) is when more than one of an ambiguous function's constituents produces a result. In McCarthy's theoretical formulation, the ambiguous function effectively produces all of the results as possible outputs. (This is technically known as *nondeterministic* computation. It is as though the computer evaluating the ambiguous function clones itself, producing a copy for each possible result, continuing to execute every single copy. You could imagine an multithreaded implementation of such a system, where every time an ambiguous function produces multiple possible results, we create that many new threads so as to be able to evaluate all possible outcomes. This is a reasonable mental model for nondeterministic computation, but it's not what actually happens with Rx's Amb operator.) In the kinds of theoretical work ambiguous functions were introduced for, the ambiguity often vanishes in the end—there may have been an enormous number of ways in which a computation could have proceeded, but they might all, finally, produce the same result. However, such theoretical concerns are taking us away from what Rx's Amb does, and how we might use it in practice.

Rx's Amb provides the behaviour described in the cases where either none of the inputs produces anything, or exactly one of them does. Howeever, it makes no attempt to support non-deterministic computation, so its handling of the case where multiple constituents are able to produce value is oversimplified, but then McCarthy's amb was first and foremost an analytical construct, so any practical implementation of it is always going to fall short.

Staying inside the monad

It can be tempting to flip between programming styles when using Rx. For the parts where it's easy to see how Rx applies, then we will naturally use Rx. But when things get tricky, it might seem easiest to change tracks. It might seem like the easiest thing to do would be to `await` an observable, and then proceed with ordinary sequential code. Or maybe it might seem simplest to make callbacks passed to operators like `Select` or `Where` perform operations in addition to their main jobs—to have side effects that do useful things.

Although this can sometimes work, switching between paradigms should be done with caution, as this is a common root cause for concurrency problems such as deadlock and scalability issues. The basic reason for this is that for as long as you remain within Rx's way of doing things, you will benefit from the basic soundness of the mathematical underpinnings. But for this to work, you need to use a functional style—functions should process their inputs and deterministically produce outputs based on those inputs, and they should neither depend on external state nor change it. This can be a tall order, and it won't always be possible, but a lot of the theory falls apart if you break these rules. Composition doesn't work as reliably as it can. So using a functional style, and keeping your code within Rx's idiom will tend to improve reliability.

Issues with side effects

Since programs always have to have some side effects if they are to do anything useful—if the world is no different as a result of a program having run, then you may as well not have run it—it can be useful to explore the issues with side effects, so that we can know how best to deal with them when they are necessary. So we will now 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, or updating a display
- causing physical activity, such as when a vending machine dispenses an item, or directs a coin into its coin box

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.