

A large, stylized C# logo is positioned on the right side of the cover. It features a large, thick, black circle with a small gap at the top. Above the circle, there are two vertical black lines of equal height, and two diagonal black lines crossing them, forming a hash symbol (#).

Functional Programming in

How to write better C# code

Enrico Buonanno

 **MANNING**

SAMPLE CHAPTER



Functional Programming in C#

by Enrico Buonanno

Chapter 14

Copyright 2017 Manning Publications

brief contents

PART 1 CORE CONCEPTS1

- 1 ■ Introducing functional programming 3
- 2 ■ Why function purity matters 31
- 3 ■ Designing function signatures and types 52
- 4 ■ Patterns in functional programming 80
- 5 ■ Designing programs with function composition 102

PART 2 BECOMING FUNCTIONAL121

- 6 ■ Functional error handling 123
- 7 ■ Structuring an application with functions 149
- 8 ■ Working effectively with multi-argument functions 177
- 9 ■ Thinking about data functionally 202
- 10 ■ Event sourcing: a functional approach to persistence 229

PART 3 ADVANCED TECHNIQUES255

- 11 ■ Lazy computations, continuations, and the beauty of monadic composition 257
- 12 ■ Stateful programs and stateful computations 279

13	■	Working with asynchronous computations	295
14	■	Data streams and the Reactive Extensions	320
15	■	An introduction to message-passing concurrency	345

14

Data streams and the Reactive Extensions

This chapter covers

- Using `IObservable` to represent data streams
- Creating, transforming, and combining `IObservables`
- Knowing when you should use `IObservable`

In chapter 13, you gained a good understanding of asynchronous values—values that are received at some point in the future. What about a *series* of asynchronous values? For example, say you have an event-sourced system like the one in chapter 10; how can you model the stream of events that are produced and define downstream processing of those events? For example, say you want to recompute an account's balance with every transaction, and send a notification if it becomes negative?

The `IObservable` interface provides an abstraction to represent such event streams. And not just event streams, but more generally *data streams*, where the values in the stream could be, say, stock quotes, byte chunks being read from a file, successive states of an entity, and so on. Really, anything that constitutes a sequence of logically related values in time can be thought of as a data stream.

In this chapter, you'll learn what IObservables are, and how to use the *Reactive Extensions* (Rx) to create, transform, and combine IObservables. We'll also discuss what sort of scenarios benefit from using IObservable.

Rx is a set of libraries for working with IObservables—much like LINQ provides utilities for working with IEnumerable. Rx is a very rich framework, so thorough coverage is beyond the scope of this chapter; instead, we'll just look at some basic features and applications of IObservable and at how it relates to other abstractions we've covered so far.

14.1 Representing data streams with IObservable

If you think of an array as a sequence of values in space (space in memory, that is), then you can think of IObservable as a sequence of values in time:

- With an IEnumerable, you can enumerate its values at your leisure.
- With an IObservable, you can observe the values as they come.

Table 14.1 shows how IObservable relates to other abstractions.

Table 14.1 How IObservable compares with other abstractions

	Synchronous	Asynchronous
Single value	T	Task<T>
Multiple values	IEnumerable<T>	IObservable<T>

IObservable is like an IEnumerable, in that it contains several values, and it's like a Task, in that values are delivered asynchronously. IObservable is therefore more general than both: you can view IEnumerable as a special case of IObservable that produces all its values synchronously; you can think of Task as a special case of IObservable that produces a single value.

14.1.1 A sequence of values in time

The easiest way to develop an intuition about IObservable is through *marble diagrams*, a few examples of which are shown in figure 14.1. Marble diagrams represent the values in the stream. Each IObservable is represented with an arrow, representing time, and marbles, representing values that are produced by the IObservable.

The image illustrates that an IObservable can actually produce three different kinds of messages:

- `OnNext` signals a new value, so if your IObservable represents a stream of events, `OnNext` will be fired when an event is ready to be consumed. This is an IObservable's most important message, and often the only one you'll be interested in.
- `OnCompleted` signals that the IObservable is done and will signal no more values.
- `OnError` signals that an error has occurred and provides the relevant Exception.

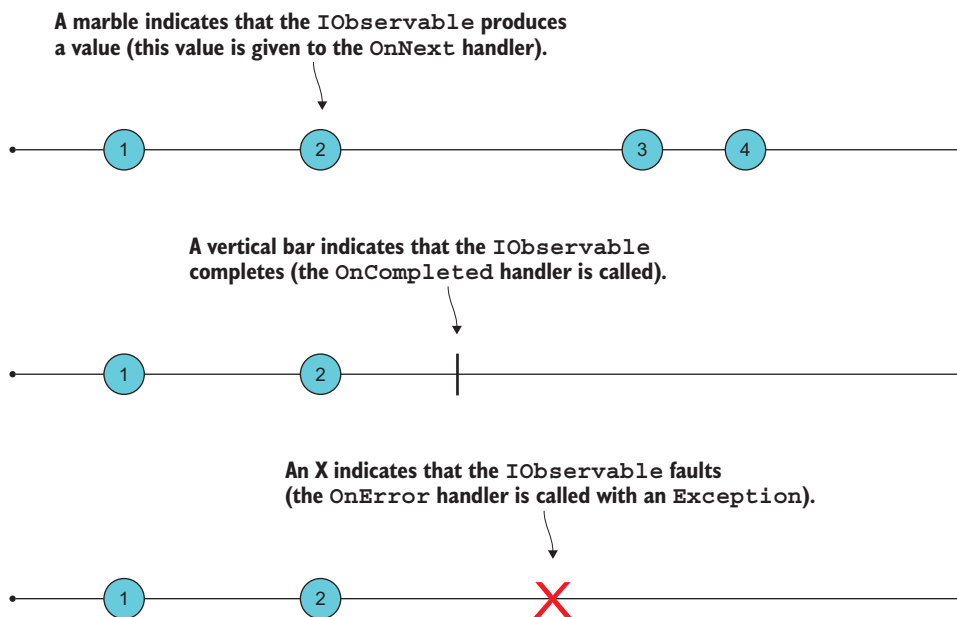


Figure 14.1 Marble diagrams provide an intuitive way to understand `IObservables`.

The `IObservable` contract

The `IObservable` contract specifies that an `IObservable` should produce messages according to the following grammar:

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

That is, an `IObservable` can produce an arbitrary number of `T`'s (`OnNext`), possibly followed by a single value indicating either successful completion (`OnCompleted`) or an error (`OnError`).

This means that there are three possibilities in terms of completion. An `IObservable` can

- Never complete
- Complete normally, with a completion message
- Complete abnormally, in which case it produces an `Exception`

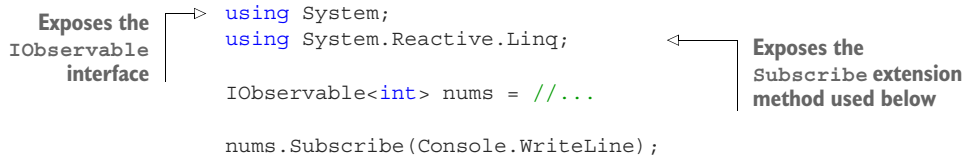
An `IObservable` *never* produces any values after it's completed, *regardless* of whether it completes normally or with an error.

14.1.2 Subscribing to an `IObservable`

Observ-ables work in tandem with observ-ers. Simply put,

- Observables produce values
- Observers consume them

If you want to consume the messages produced by an `IObservable`, you can create an observer and associate it with an `IObservable` via the `Subscribe` method. The simplest way to do this is by providing a callback that will handle the values produced by the `IObservable`, like so:



```

using System;
using System.Reactive.Linq;

IObservable<int> nums = //...

nums.Subscribe(Console.WriteLine);

```

So when I say that `nums` “produces” an `int` value, all I really mean is that it calls the given function (in this case, `Console.WriteLine`) with the value. The result of the preceding code is that whenever `nums` produces an `int`, it’s printed out.

I find the naming a bit confusing; you’d expect an `IObservable` to have an `Observe` method, but instead it’s called `Subscribe`. Basically, you can think of the two as synonyms: an “observer” is a subscriber, and in order to “observe” an observable, you subscribe to it.

What about the other types of messages an `IObservable` can produce? You can provide handlers for those as well. For instance, the following listing shows a convenience method that attaches an observer to an `IObservable`; this observer will simply print some diagnostic messages whenever the `IObservable` signals. We’ll use this method later for debugging.

Listing 14.1 Subscribing to the messages produced by an `IObservable`

```

using static System.Console;

public static IDisposable Trace<T>
    (this IObservable<T> source, string name)
    => source.Subscribe(
        onNext: t => WriteLine($"{name} -> {t}"),
        onError: ex => WriteLine($"{name} ERROR: {ex.Message}"),
        onCompleted: () => WriteLine($"{name} END"));

```

`Subscribe` actually takes three handlers (all are optional arguments), to handle the different messages that an `IObservable<T>` can produce. It should be clear why the handlers are optional: if you don’t expect an `IObservable` to ever complete, there’s no point providing an `onComplete` handler.

A more OO option for subscribing is to call `Subscribe` with an `IObserver`,¹ an interface that, unsurprisingly, exposes `OnNext`, `OnError`, and `OnCompleted` methods.

Also notice that `Subscribe` returns an `IDisposable` (the subscription). By disposing it, you unsubscribe.

¹ This is the method defined on the `IObservable` interface. The overload that takes the callbacks is an extension method.

In this section you’ve seen some of the basic concepts and terminology around `IObservable`. It’s a lot to absorb, but don’t worry; things will become clearer as you see some examples. These are the basic ideas to keep in mind:

- Observables produce values; observers consume them.
- You associate an observer with an observable by using `Subscribe`.
- An observable produces a value by calling the observer’s `OnNext` handler.

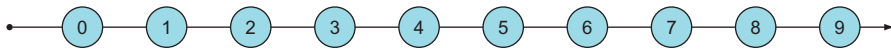
14.2 Creating IObservables

You now know how to consume the data in a stream by subscribing to an `IObservable`. But how do you get an `IObservable` in the first place? The `IObservable` and `IObserver` interfaces are included in .NET Standard, but if you want to create or perform many other operations on `IObservables`, you’ll typically use the Reactive Extensions (Rx) by installing the `System.Reactive` package.²

The recommended way to create `IObservables` is by using several dedicated methods included in the static `Observable`, and we’ll explore them next. I recommend you follow along in the REPL when possible.

14.2.1 Creating a timer

A timer can be modeled with an `IObservable` that signals at regular intervals. We can represent it with a marble diagram as follows:



This is a good way to start experimenting with `IObservables` because it’s simple but does include the element of time.

You create a timer with `Observable.Interval`.

Listing 14.2 Creating an `IObservable` that signals every second

```
using System.Reactive.Linq;

var oneSec = TimeSpan.FromSeconds(1);
IObservable<long> ticks = Observable.Interval(oneSec);
```

Here we define `ticks` as an `IObservable` that will begin signaling after one second, producing a long counter value that increments every second, starting at 0. Notice I said “will begin” signaling? The resulting `IObservable` is lazy, so unless there’s a subscriber, nothing will actually be done. Why talk, if nobody’s listening?

² Rx includes several libraries. The main library, `System.Reactive`, bundles the packages you’ll most commonly need: `System.Reactive.Interfaces`, `System.Reactive.Core`, `System.Reactive.Linq`, and `System.Reactive.PlatformServices`. There are several other packages that are useful in more specialized scenarios, such as if you’re using Windows forms.

If we want to see some tangible results, we need to *subscribe* to the IObservable. We can do this with the Trace method defined earlier:

```
ticks.Trace("ticks");
```

At this point, you'll start to see the following messages appear in the console, one second apart:

```
ticks -> 0
ticks -> 1
ticks -> 2
ticks -> 3
ticks -> 4
...
```

Because this IObservable never completes, you'll have to reset the REPL to stop the noise—sorry!

14.2.2 Using Subject to tell an IObservable when it should signal

Another way to create an IObservable is by instantiating a Subject, which is an IObservable that you can imperatively tell to produce a value that it will in turn push to its observers. For example, the following program turns inputs from the console into values signaled by a Subject.

Listing 14.3 Modeling user inputs as a stream

```
using System.Reactive.Subjects;

public static void Main()
{
    var inputs = new Subject<string>();  <— Creates a Subject

    using (inputs.Trace("inputs"))      <— Subscribes to the Subject
    {
        for (string input; (input = ReadLine()) != "q";)
            inputs.OnNext(input);

        inputs.OnCompleted();           <— Tells the Subject to
                                         signal completion
    }
}
```

Tells the Subject to produce a value, which it will push to its observers

Leaving the using block disposes the subscription.

Every time the user types in some input, the code pushes that value to the Subject by calling its OnNext method. When the user types “q”, the code exits the for loop and calls the Subject’s OnCompleted method, signaling that the stream has ended. Here we’ve subscribed to the stream of inputs using the Trace method defined in 14.1, so we’ll get a diagnostic message printed for each user input.

An interaction with the program looks like this (user inputs in bold):

```
hello
inputs -> hello
world
inputs -> world
q
inputs END
```

14.2.3 Creating IObservables from callback-based subscriptions

If your system subscribes to an external data source, such as a message queue, event broker, or publisher/subscriber, you can model that data source as an IObservable.

For example, Redis can be used as a publisher/subscriber, and the following listing shows how you can use Observable.Create to create an IObservable from the callback-based Subscribe methods that allows you to subscribe to messages published to Redis.

Listing 14.4 Creating an IObservable from messages published to Redis

Create takes an observer, so the given function will only be called when a subscription is being made.

Converts from the callback-based implementation of Subscribe to values produced by the IObservable

```
using StackExchange.Redis;
using System.Reactive.Linq;
```

```
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect("localhost");
```

```
IObservable<RedisValue> RedisNotifications(RedisChannel channel)
```

```
=> Observable.Create<RedisValue>(observer =>
```

```
{
    var sub = redis.GetSubscriber();
```

```
    sub.Subscribe(channel, (_, value) => observer.OnNext(value));
```

```
    return () => sub.Unsubscribe(channel);
```

```
});
```

Returns a function that will be called when the subscription is disposed

The preceding method returns an IObservable that will produce the values received from Redis on the given channel. You could use this as follows:

```
RedisChannel weather = "weather";
```

```
var weatherUpdates = RedisNotifications(weather);
```


```
weatherUpdates.Subscribe(
    onNext: val => WriteLine($"It's {val} out there"));
```

```
redis.GetDatabase(0).Publish(weather, "stormy");
// prints: It's stormy out there
```

Subscribes to the IObservable

Gets an IObservable that signals when messages are published on the "weather" channel

Publishing a value causes weatherUpdates to signal, and the onNext handler is called as a result.

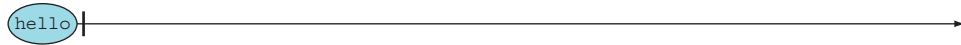
 **AVOID USING Subject** Subject works *imperatively* (you tell a Subject when to fire), and this goes somewhat counter to the *reactive* philosophy of Rx (you specify how to react to certain things when they happen).

For this reason, it's recommended that you avoid Subjects whenever possible, and instead use other methods, such as `Observable.Create`. For example, as an exercise, try to rewrite the code in listing 14.3, using `Observable.Create` to create an `IObservable` of user inputs.

14.2.4 Creating IObservables from simpler structures

I said that `IObservable<T>` is more general than a value `T`, a `Task<T>`, or an `IEnumerable<T>`, so let's see how each of these can be “promoted” to an `IObservable`. This becomes useful if you want to combine one of these less powerful structures with an `IObservable`.

`Return` allows you to lift a single value into an `IObservable` that looks like this:



That is, it immediately produces the value and then completes. Here's an example:

```
IObservable<string> justHello = Observable.Return("hello");
justHello.Trace("justHello");

// prints: justHello -> hello
//          justHello END
```

`Return` takes a value, `T`, and lifts it into an `IObservable<T>`. This is the first container where the `Return` function is actually called `Return`!

Let's see about creating an `IObservable` from a single asynchronous value—a `Task`. Here, we have an `IObservable` that looks like this:



That is, after some time we'll get a single value, immediately followed by the signal for completion. In code, it looks like this:

```
Observable.FromAsync(() => Yahoo.GetRate("USDEUR"))
    .Trace("singleUsdEur");

// prints: singleUsdEur -> 0.92
//          singleUsdEur END
```

Finally, an `IObservable` created from an `IEnumerable` looks like this:



That is, it immediately produces all the values in the `IEnumerable`, and completes:

```

IEnumerable<char> e = new[] { 'a', 'b', 'c' };
IObservable<char> chars = e.ToObservable();
chars.Trace("chars");

// prints: chars -> a
//         chars -> b
//         chars -> c
//         chars END

```

You’ve now seen many—but not all—methods for creating `IObservables`. You may end up creating `IObservables` in other ways; for example, in Windows application development you can turn UI events such as mouse clicks into event streams by using `Observable.FromEvent` and `FromEventPattern`.

Now that you know about creating and subscribing to `IObservable`, let’s move on to the most fascinating area: transforming and combining different streams.

14.3 Transforming and combining data streams

The power of using streams comes from the many ways in which you can combine them and define new streams based on existing ones. Rather than dealing with individual values in a stream (like in most event-driven designs), you deal with the stream as a whole.

Rx offers *a lot* of functions (often called *operators*) to transform and combine `IObservables` in a variety of ways. I’ll discuss the most commonly used ones, and add a few operators of my own. You’ll recognize the typical traits of a functional API: purity and composability.

14.3.1 Stream transformations

You can create new observables by transforming an existing observable in some way. One of the simplest operations is mapping. This is achieved with the `Select` method, which works—as with any other “container”—by applying the given function to each element in the stream, as shown in figure 14.2.

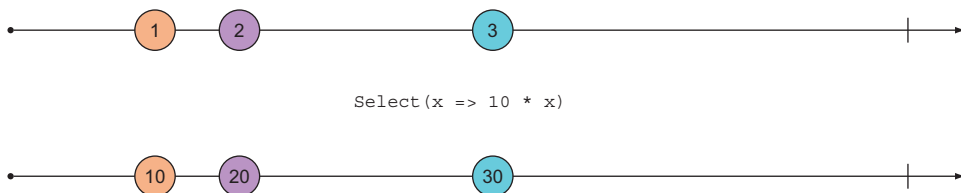


Figure 14.2 `Select` maps a function onto a stream.

Here's some code that creates a timer and then maps a simple function on it:

```
var oneSec = TimeSpan.FromSeconds(1);
var ticks = Observable.Interval(oneSec);

ticks.Select(n => n * 10)
    .Trace("ticksX10");
```

We're attaching an observer on the last line, with the Trace method, so the preceding code will cause the following messages to be printed every second:

```
ticksX10 -> 0
ticksX10 -> 10
ticksX10 -> 20
ticksX10 -> 30
ticksX10 -> 40
...
```

Because Select follows the LINQ query pattern, we can write the same thing using LINQ:

```
from n in ticks select n * 10
```

Using Select, we can rewrite our simple program that checks exchange rates (first introduced in listing 12.1) in terms of observables:

```
public static void Main()
{
    var inputs = new Subject<string>();
    var rates = from pair in inputs
                select Yahoo.GetRate(pair).Result;

    using (inputs.Trace("inputs"))
    using (rates.Trace("rates"))
        for (string input; (input = ReadLine().ToUpper()) != "Q";)
            inputs.OnNext(input);
}
```

**Subscribes to both streams
to produce debug messages**

**The stream of values
entered by the user**

**Maps user inputs to the
corresponding retrieved
values**

Here, inputs represents the stream of currency pairs entered by the user, and in rates we map those pairs to the corresponding values retrieved from the Yahoo API. We're subscribing to both observables with the usual Trace method, so an interaction with this program could be as follows:

```
eurusd
inputs -> EURUSD
rates -> 1.0852
chfusd
inputs -> CHFUSD
rates -> 1.0114
```

Notice, however, that we're calling `Result` to wait for the remote query in `GetRate` to complete. In a real application, we wouldn't want to block a thread, so how could we avoid that?

We saw that a `Task` can easily be promoted to an `IObservable`, so we could generate an `IObservable` of `IObservables`. Sound familiar? `Bind`! We can use `SelectMany` instead of `Select`, which will flatten the result into a single `IObservable`. We can therefore rewrite the definition of the rates stream as follows:

```
var rates = inputs.SelectMany
    (pair => Observable.FromAsync(() => Yahoo.GetRate(pair)));
```

`Observable.FromAsync` promotes the `Task` returned by `GetRate` to an `IObservable`, and `SelectMany` flattens all these `IObservables` into a single `IObservable`.

Because it's always possible to promote a `Task` to an `IObservable`, an overload of `SelectMany` exists that does just that (this is similar to how we overloaded `Bind` to work with an `IEnumerable` and an `Option`-returning function in chapter 4). This means we can avoid explicitly calling `FromAsync` and return a `Task` instead. Furthermore, we can use a LINQ query:

```
var rates =
    from pair in inputs
    from rate in Yahoo.GetRate(pair)
    select rate;
```

The program thus modified will work the same way as before, but without the blocking call to `Result`.

`IObservable` also supports many of the other operations that are supported by `IEnumerable`, such as filtering with `Where`, `Take` (takes the first n values), `Skip`, `First`, and so on.

14.3.2 Combining and partitioning streams

There are also many operators that allow you to combine two streams into a single one. For example, `Concat` produces all the values of one `IObservable`, followed by all the values in another, as shown in figure 14.3.

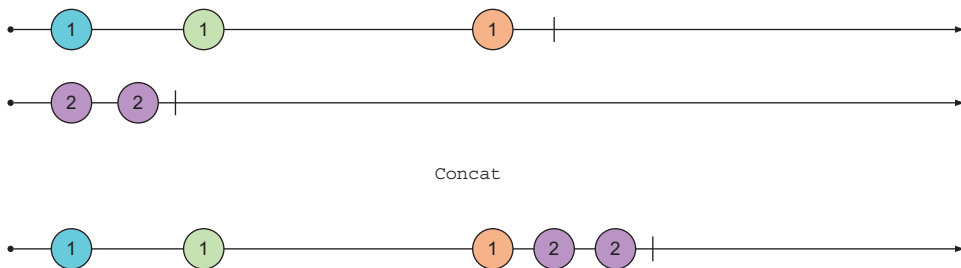


Figure 14.3 `Concat` waits for an `IObservable` to complete and then produces elements from the other `IObservable`.

For instance, in our exchange rate lookup, we have an observable called `rates` with the retrieved rates. If we want an observable of all the messages the program should output to the console, this must include the retrieved rates, but also an initial message prompting the user for some input. We can lift this single message into an `IObservable` with `Return` and then use `Concat` to combine it with the other messages:

```
IObservable<decimal> rates = //...

IObservable<string> outputs = Observable
    .Return("Enter a currency pair like 'EURUSD', or 'q' to quit")
    .Concat(rates.Select(Decimal.ToString));
```

In fact, the need to provide a starting value for an `IObservable` is so common that there's a dedicated function, `StartWith`. The preceding code is equivalent to this:

```
var outputs = rates.Select(Decimal.ToString)
    .StartWith("Enter a currency pair like 'EURUSD', or 'q' to quit");
```

Whereas `Concat` waits for the left `IObservable` to complete before producing values from the right observable, `Merge` combines values from two `IObservables` without delay, as shown in figure 14.4.

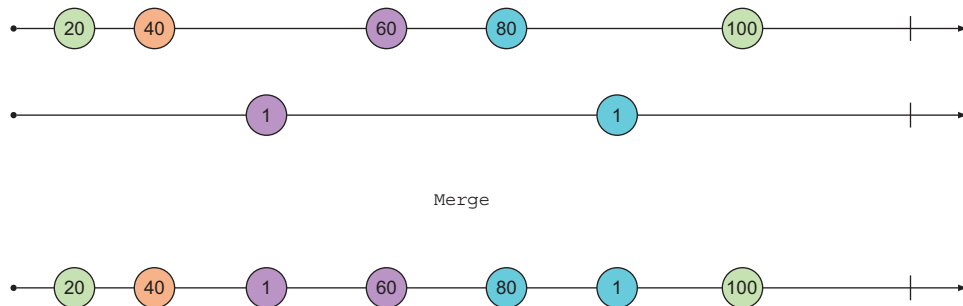


Figure 14.4 Merge merges two `IObservables` into one.

For example, if you have a stream of valid values and one of error messages, you could combine them with `Merge` as follows:

```
IObservable<decimal> rates = //...
IObservable<string> errors = //...

var outputs = rates.Select(Decimal.ToString)
    .Merge(errors);
```

Just as you might want to merge values from different streams, the opposite operation—partitioning a stream according to some criterion—is also often useful. Figure 14.5 illustrates this.

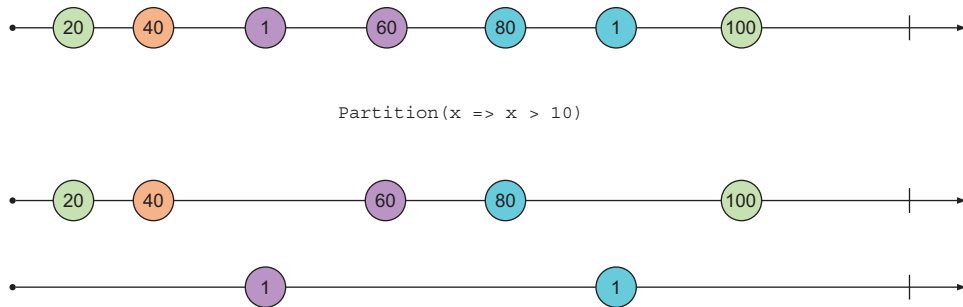


Figure 14.5 Partitioning an IObservable according to a predicate

This is one of many cases in which C# 7 tuple syntax facilitates working with IObservable effectively. Partition is defined as follows:

```
public static (IObservable<T> Passed, IObservable<T> Failed)
    Partition<T>(this IObservable<T> ts, Func<T, bool> predicate)
=> ( Passed: from t in ts where predicate(t) select t
    , Failed: from t in ts where !predicate(t) select t );
```

It can be used in client code like this:

```
var (evens, odds) = ticks.Partition(x => x % 2 == 0);
```

Partitioning an IObservable of values is roughly equivalent to an if when dealing with a single value, so it's useful when you have a stream of values that you want to process differently, depending on some condition. For example, if you have a stream of messages and some criterion for validation, you can partition the stream into two streams of valid and invalid messages, and process them accordingly.

14.3.3 Error handling with IObservable

Error handling when working with IObservable works differently from what you might expect. In most programs, an uncaught exception either causes the whole application to crash, or causes the processing of a single message/request to fail, while subsequent requests work fine. To illustrate how things work differently in Rx, consider this version of our program for looking up exchange rates:

```
public static void Main()
{
    var inputs = new Subject<string>();

    var rates =
        from pair in inputs
        from rate in Yahoo.GetRate(pair)
        select rate;
```

```

var outputs = from r in rates select r.ToString();

using (inputs.Trace("inputs"))
using (rates.Trace("rates"))
using (outputs.Trace("outputs"))
    for (string input; (input = ReadLine().ToUpper()) != "Q";)
        inputs.OnNext(input);
}

```

The program captures three streams, each dependent on another (outputs is defined in terms of rates, and rates is defined in terms of inputs, as shown in figure 14.6), and we're printing diagnostic messages for all of them with Trace.



Figure 14.6 Simple dataflow between three IObservables

Now look what happens if you break the program by passing an invalid currency pair:

```

eurusd
inputs -> EURUSD
rates -> 1.0852
outputs -> 1.0852
chfUSD
inputs -> CHFUSD
rates -> 1.0114
outputs -> 1.0114
xxx
inputs -> XXX
rates ERROR: Input string was not in a correct format.
outputs ERROR: Input string was not in a correct format.
chfUSD
inputs -> CHFUSD
eurusd
inputs -> EURUSD

```

What this shows is that once rates errors, it never signals again (as specified in the IObservable contract). As a result, everything downstream is also “dead.” But IObservables upstream of the failed one are fine: inputs is still signaling, as would any other IObservables defined in terms of inputs.

To prevent your system from going into such a state, where a “branch” of the dataflow dies, while the remaining graph keeps functioning, you can use the techniques you learned for functional error handling.

To do this, you can use a helper function I’ve defined in the `LaYumba.Functional` library, which allows you to safely apply a Task-returning function to each element in a stream. The result will be a pair of streams: a stream of successfully computed values, and a stream of exceptions.

Listing 14.5 Safely performing a Task and returning two streams

Converts each Task<R> to a Task<Exceptional<R>> to get a stream of Exceptionals

```
public static (IObservable<R> Completed, IObservable<Exception> Faulted)
    Safely<T, R>(this IObservable<T> ts, Func<T, Task<R>> f)
    => ts
        .SelectMany(t => f(t).Map(
            Faulted: ex => ex,
            Completed: r => Exceptional(r)))
        .Partition();

static (IObservable<T> Successes, IObservable<Exception> Exceptions)
    Partition<T>(this IObservable<Exceptional<T>> excTs)
{
    bool IsSuccess(Exceptional<T> ex)
        => ex.Match(_ => false, _ => true);

    T ValueOrDefault(Exceptional<T> ex)
        => ex.Match(exc => default(T), t => t);

    Exception ExceptionOrDefault(Exceptional<T> ex)
        => ex.Match(exc => exc, _ => default(Exception));

    return (
        Successes: excTs
            .Where(IsSuccess)
            .Select(ValueOrDefault),
        Exceptions: excTs
            .Where(e => !IsSuccess(e))
            .Select(ExceptionOrDefault)
    );
}
```

Partitions a stream of Exceptionals into successfully computed values and exceptions

For each T in the given stream, we apply the Task-returning function f. We then use the binary overload of Map defined in chapter 13 to convert each resulting Task<R> to a Task<Exceptional<R>>. This is where we gain safety: instead of an inner value R that will throw an exception when it's accessed, we have an Exceptional<R> in the appropriate state. SelectMany flattens away the Tasks in the stream and returns a stream of Exceptionals. We can then partition this in successes and exceptions.

With this in place, we can refactor our program to handle errors more gracefully:

```
var (rates, errors) = inputs.Safely(Yahoo.GetRate);
```

14.3.4 Putting it all together

Let's showcase the various techniques you've learned in this section by refactoring the exchange rates lookup program to safely handle errors, and without the debug information.

Listing 14.6 The program refactored to safely handle errors

```

public static void Main()
{
    var inputs = new Subject<string>();

    var (rates, errors) = inputs.Safely(Yahoo.GetRate);

    var outputs = rates
        .Select(Decimal.ToString)
        .Merge(errors.Select(ex => ex.Message))
        .StartWith("Enter a currency pair like 'EURUSD', or 'q' to quit");

    using (outputs.Subscribe(WriteLine))
        for (string input; (input = ReadLine().ToUpper()) != "Q";)
            inputs.OnNext(input);
}

```

The dataflow diagram in figure 14.7 shows the various IObservables involved, and how they depend on one another.

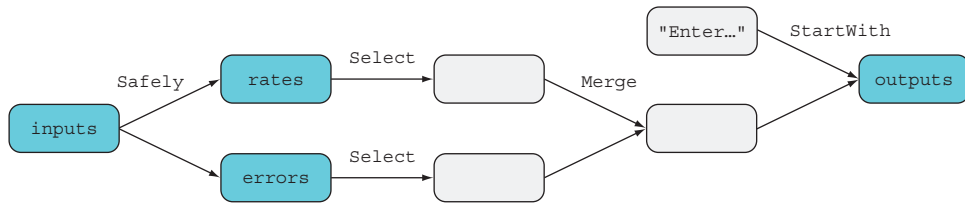


Figure 14.7 Dataflow with a separate branch for handling errors

Notice how Safely allows us to create two branches, each of which can be processed independently until a uniform representation for both cases is obtained and they can be merged. Also take note of the three parts of a program that uses IObservables:

- 1 *Set up the data sources*—In our case: inputs, which requires a Subject; and the single value “Enter...”
- 2 *Process the data*—This is where functions like Select, Merge, and so on are used.
- 3 *Consume the results*—Observers consume the most downstream IObservables; in this case: outputs.

14.4 Implementing logic that spans multiple events

So far I’ve mostly aimed at familiarizing you with IObservables and the many operators that can be used with them. For this, I’ve used familiar examples like the exchange rates lookup. After all, given that you can promote any value `T`, `Task<T>`, or `IEnumerable<T>` to an `IObservable<T>`, you could pretty much write all of your code in terms of IObservables! But should you?

The answer, of course, is “probably not.” The area in which IObservable and Rx really shine is when you can use them to write stateful programs without any explicit

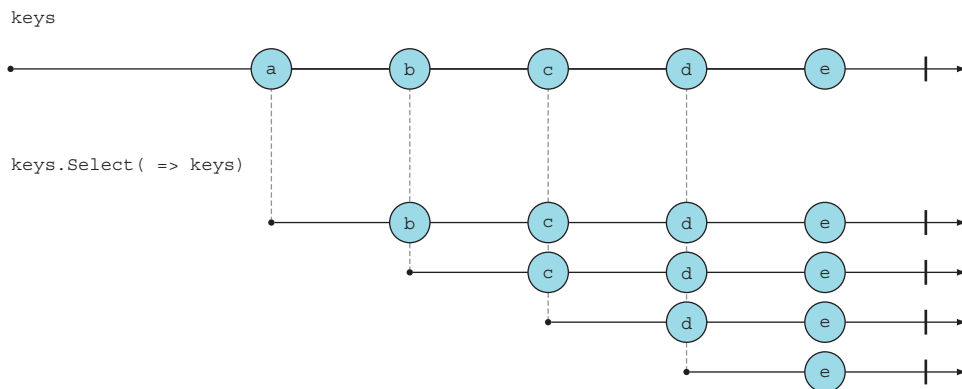
state manipulation. By “stateful programs,” I mean programs in which events aren’t treated independently; past events influence how new events are treated. In this section, you’ll see a few such examples.

14.4.1 Detecting sequences of pressed keys

At some point, you’ve probably written an event handler that listens to a user’s keypresses and performs some actions based on what key and key modifiers were pressed. A callback-based approach is satisfactory for many cases, but what if you want to listen to a specific *sequence* of keypresses? For example, say you want to implement some behavior when the user presses the combination Alt-K-B.

In this case, pressing Alt-B should lead to different behavior, based on whether it was shortly preceded by the leading Alt-K, so keypresses can’t be treated independently. If you have a callback-based mechanism that deals with single keypressed events, you effectively need to set in motion a state machine when the user presses Alt-K, and then wait for the possible Alt-B that will follow, reverting to the previous state if no Alt-B is received in time. It’s actually pretty complicated!

With `IObservable`, this can be solved much more elegantly. Let’s assume that we have a stream of keypress events, `keys`. We’re looking for two events—Alt-K and Alt-B—that happen on that same stream in quick succession. In order to do this, we need to explore how to combine a stream with itself. Consider the following diagram:



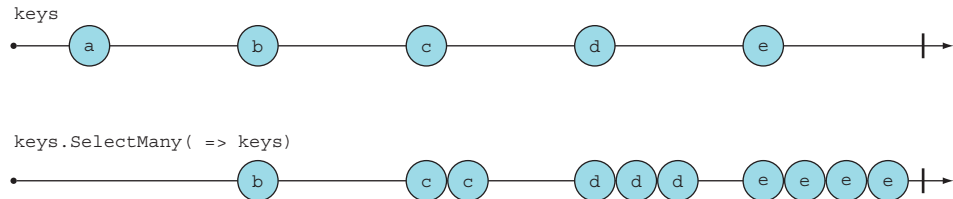
It’s important to understand this diagram. The expression `keys.Select(_ => keys)` yields a new `IObservable` that maps each value produced by `keys` to `keys` itself. So when `keys` produces its first value, `a`, this new `IObservable` produces an `IObservable` that has *all following values* in `keys`. When `keys` produces its second value, `b`, the new `IObservable` produces another `IObservable` that has all the values that follow `b`, and so on.³

³ Imagine what `keys.Select(_ => keys)` would look like if `keys` were an `IEnumerable`: for each value, you’d be taking the whole `IEnumerable`, so in the end you’d have an `IEnumerable` containing n replicas of `keys` (n being the length of `keys`). With `IObservable`, the behavior is different because of the element of time, so when you say “give me `keys`,” what you really get is “all values `keys` will produce in the future.”

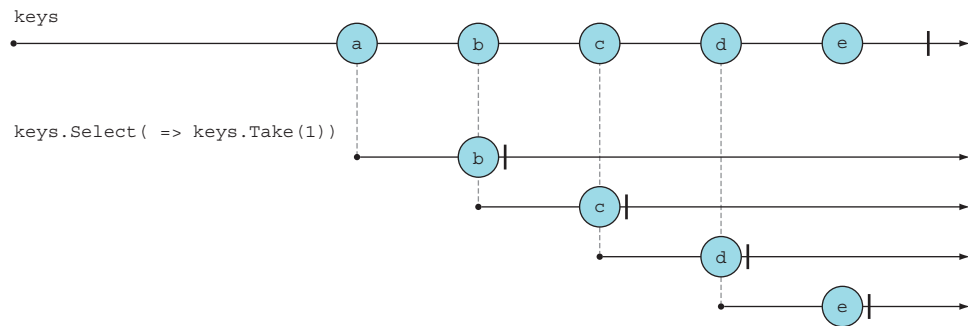
Looking at the types can also help clarify this:

```
keys : IObservable<KeyInfo>
_ => keys : KeyInfo → IObservable<KeyInfo>
keys.Select(_ => keys) : IObservable<IObservable<KeyInfo>>
```

If we use `SelectMany` instead, all these values are flattened into a single stream:



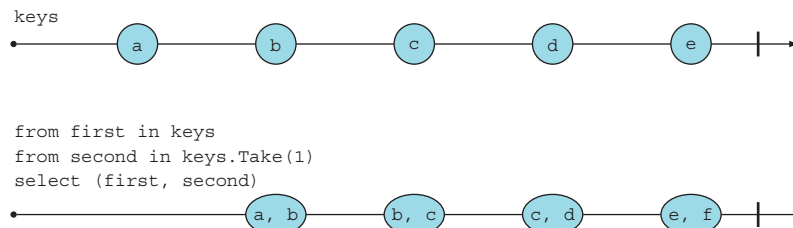
Of course, if we're looking for *two* consecutive keypresses, we don't need *all* values that follow an item, but just the next one. So instead of mapping each value to the whole `IObservable`, let's reduce it to the first item with `Take`:



We're getting close. Now, let's make the following changes:

- Instead of ignoring the current value, pair it with the following value.
- Use `SelectMany` to obtain a single `IObservable`.
- Use LINQ syntax.

The resulting expression pairs each value in an `IObservable` with the previously emitted value:



This is a pretty useful function in its own right, and I'll call it `PairWithPrevious`. We'll use it later.

But for this particular scenario, we only want pairs to be created if they're sufficiently close in time. This can be achieved easily: in addition to taking only the next value with `Take(1)`, we only take values within a timespan, using an overload of `Take` that takes a `TimeSpan`. The solution is shown in the following listing.

Listing 14.7 Detecting when the user presses the Alt-K-B key sequence

```
IObservable<ConsoleKeyInfo> keys = //...
var halfSec = TimeSpan.FromMilliseconds(500);

var keysAlt = keys
    .Where(key => key.Modifiers.HasFlag(ConsoleModifiers.Alt));

var twoKeyCombis =
    from first in keysAlt
    from second in keysAlt.Take(halfSec).Take(1)
    select (First: first, Second: second);

var altKB =
    from pair in twoKeyCombis
    where pair.First.Key == ConsoleKey.K
        && pair.Second.Key == ConsoleKey.B
    select Unit();
```

For any keypress, pairs it with the next keypress that occurs within a half-second

As you can see, the solution is simple and elegant, and you can apply this approach to recognize more complex patterns within sequences of events—all without explicitly keeping track of state and introducing side effects!

You've probably also realized that coming up with such a solution isn't necessarily easy. It takes a while to get familiar with `IObservable` and its many operators, and develop an understanding of how to use them.

14.4.2 Reacting to multiple event sources

Imagine we have a bank account denominated in euros, and we'd like to keep track of its value in US dollars. Both changes in balance and changes in the exchange rate cause the dollar balance to change. To react to changes from different streams, we could use `CombineLatest`, which takes the latest values from two observables, whenever one of them signals, as shown in figure 14.8.

Its usage would be as follows:

```
IObservable<decimal> balance = //...
IObservable<decimal> eurUsdRate = //...

var balanceInUsd = balance.CombineLatest(eurUsdRate
    , (bal, rate) => bal * rate);
```

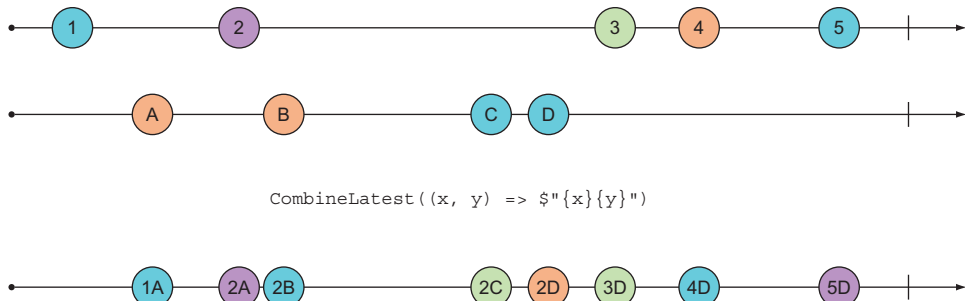


Figure 14.8 CombineLatest signals whenever one of two IObservables signals.

This works, but it doesn't take into account the fact that the exchange rate is much more volatile than the account balance. In fact, if exchange rates come from the FX market, there may well be dozens or hundreds of tiny movements every second! Surely this level of detail isn't required for a private client who wants to keep an eye on their finances. Reacting to each change in exchange rate would flood the client with unwanted notifications.

This is an example of an IObservable producing too much data (see the sidebar on backpressure). For this, we can use Sample, an operator that takes an IObservable that acts as a data source, and another IObservable that signals *when* values should be produced. Sample is illustrated in figure 14.9.

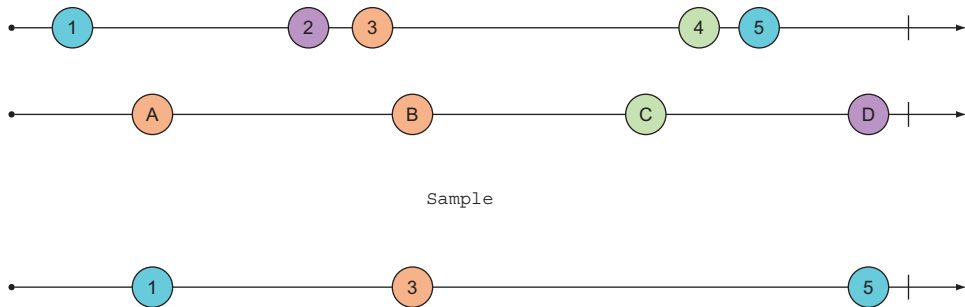


Figure 14.9 Sample produces the values from a “source” stream whenever a “sampler” stream signals.

In this scenario, we can create an IObservable that signals at 10 minute intervals, and use it to sample the stream of exchange rates.

Listing 14.8 Sampling a value from an IObservable every 10 minutes

```
IObservable<decimal> balance = //...
IObservable<decimal> eurUsdRate = //...

var tenMins = Timestamp.FromMinutes(10);
```



```
var sampler = Observable.Interval(tenMins);  
var eurUsdSampled = eurUsdRate.Sample(sampler);  
  
var balanceInUsd = balance.CombineLatest(eurUsdSampled  
    , (bal, rate) => bal * rate);
```

Both `CombineLatest` and `Sample` are cases in which our logic spans multiple events, and Rx allows us to do so without explicitly keeping any state.

Backpressure: when an `IObservable` produces data too quickly

When you iterate over the items in an `IEnumerable`, you're "pulling" or requesting items, so you can process them at your own pace. With `IObservable`, items are "pushed" to you (the consuming code). If an `IObservable` produces values more rapidly than they can be consumed by the subscribed observers, this can cause excessive *backpressure*, causing strain to your system.

To ease backpressure, Rx provides several operators:

- `Throttle`
- `Sample`
- `Buffer`
- `Window`
- `Debounce`

Each has a different behavior and several overloads, so we won't discuss them in detail. The point is that with these operators, you can easily and declaratively implement logic like, "I want to consume items in batches of 10 at a time," or "if a cluster of values come in quick succession, I only want to consume the last one." Implementing such logic in a callback-based solution, where each value is received independently, would require you to manually keep some state.

14.4.3 Notifying when an account becomes overdrawn

For a final, more business-oriented example, imagine that, in the context of the BOC application, we consume a stream of all transactions that affect bank accounts, and we want to send clients a notification if their account's balance becomes negative.

An account's balance is the sum of all the transactions that have affected it, so at any point, given a list of past `Transactions` for an account, you could compute its current balance using `Aggregate`. There is an `Aggregate` function for `IObservable`; it waits for an `IObservable` to complete, and aggregates all the values it produces into a single value.

But this isn't what we need: we don't want to wait for the sequence to complete, but to know the balance with every `Transaction` received. For this, we can use `Scan` (see figure 14.10), which is similar to `Aggregate` but aggregates all previous values with every new value that is produced.

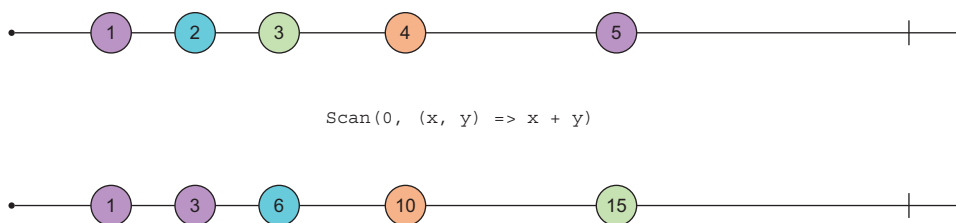


Figure 14.10 Scan aggregates all values produced so far.

As a result, we can effectively use `Scan` to keep state. Given an `IObservable` of `Transactions` affecting a bank account, we can use `Scan` to add up the amounts of all past transactions as they happen, obtaining an `IObservable` that signals with the new balance whenever the account balance changes:

```
IObservable<Transaction> transactions = //... decimal initialBalance = 0;

IObservable<decimal> balance = transactions.Scan(initialBalance
    , (bal, trans) => bal + trans.Amount);
```

Now that we have a stream of values representing an account's current balance, we need to single out what changes in balance cause the account to “dip into the red,” going from positive to negative.

For this, we need to look at changes in the balance, and we can do this with `PairWithPrevious`, which signals the current value, together with the previously emitted value. We've discussed this before, but here it is again for reference:

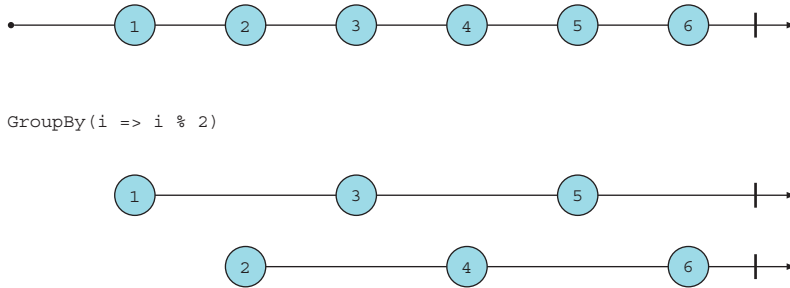
```
// ----1-----2-----3-----4----->
//
//           PairWithPrevious
//
// ----- (1,2) ---- (2,3) ---- (3,4) -->
//
public static IObservable<T Previous, T Current>
    PairWithPrevious<T>(this IObservable<T> source)
    => from first in source
        from second in source.Take(1)
        select (Previous: first, Current: second);
```

This is one of many examples of custom operations that can be defined in terms of existing operations. It's also an example of how you can use ASCII marble diagrams to document your code.

We can now use this to signal when an account dips into the red as follows:

```
IObservable<Unit> dipsIntoTheRed =
    from bal in balance.PairWithPrevious()
    where bal.Previous >= 0
        && bal.Current < 0
    select Unit();
```

Now let's make things a bit closer to the real world. If your system receives a stream of transactions, this will probably include transactions for all accounts. Therefore, we must group them by account ID in order to correctly compute the balance. `GroupBy` works for `IObservable` similarly to how it does for `IEnumerable`, but it returns a stream of streams.



Let's rewrite the code, assuming an initial stream of transactions for all accounts.

Listing 14.9 Signalling whenever an account becomes overdrawn

```

IObservable<Transaction> transactions = //...
Groups by account ID |> IObservable<Guid> dipsIntoRed = transactions
    .GroupBy(t => t.AccountId)
    .Select(DipsIntoTheRed)
    .MergeAll();
Flattens back into a single observable |> static IObservable<Guid> DipsIntoTheRed
    (IGroupedObservable<Guid, Transaction> transactions)
    {
        Guid accountId = transactions.Key;
        decimal initialBalance = 0;

        var balance = transactions.Scan(initialBalance
            , (bal, trans) => bal + trans.Amount);

        return from bal in balance.PairWithPrevious()
            where bal.Previous >= 0
                && bal.Current < 0
            select accountId;

        public static IObservable<T> MergeAll<T>
            (this IObservable<IObservable<T>> source)
            => source.SelectMany(x => x);
    }

```

Includes transactions from all accounts

Applies the transformation to each grouped observable

Signals the ID of the offending account

Now we're starting with a stream of `Transactions` for all accounts, and we end up with a stream of `Guids` that will signal whenever an account dips into the red, with the `Guid` identifying the offending account. Notice how this program is effectively keeping

track of the balances of all accounts, without the need for us to do any explicit state manipulation.

14.5 When should you use IObservable?

In this chapter, you've seen how you can use IObservable to represent data streams, and Rx to create and manipulate IObservables. There are many details and features of Rx that we haven't discussed at all,⁴ but we've still covered enough ground for you to start using IObservables and to further explore the features of Rx as needed.

As you've seen, having an abstraction that captures a data stream enables you to detect patterns and specify logic that spans across multiple events, within the same stream or across different streams. This is where I'd recommend using IObservable. The corollary is that, if your events can be handled independently, then you probably shouldn't use IObservables, because using them will probably reduce the readability of your code.

A very important thing to keep in mind is that because OnNext has no return value, an IObservable can only push data downstream, and never receives any data back. Hence, IObservables are best combined into *one-directional data flows*. For instance, if you read events from a queue and write some data into a DB as a result, IObservable can be a good fit. Likewise if you have a server that communicates with web clients via WebSockets, where messages are exchanged between client and server in a fire-and-forget fashion. On the other hand, IObservables are not well-suited to a request-response model such as HTTP. You could model the received requests as a stream and compute a stream of responses, but you'd then have no easy way to tie these responses back to the original requests.

Finally, if you have complex synchronization patterns that can't be captured with the operators in Rx, and you need more fine-grained control over how messages are sequenced and processed, you may find the building blocks in the System.DataFlow namespace (based on in-memory queues) more appropriate.

Summary

- IObservable<T> represents a *stream* of Ts: a sequence of values in time.
- An IObservable produces messages according to the grammar `OnNext* (OnCompleted|OnError)?`.
- Writing a program with IObservables involves three steps:
 - Create IObservables using the methods in `System.Reactive.Linq.Observable`.
 - Transform and combine IObservables using the operators in Rx, or other operators you may define.
 - Subscribe to and consume the values produced by the IObservable.
 - Associate an observer to an IObservable with `Subscribe`.

⁴ To give you an idea of what was not covered, there are many more operators along with important implementation details of Rx: schedulers (which determine how calls to observers are dispatched), *hot* vs. *cold* observables (not all observables are lazy), and Subjects with different behaviors, for example.

- Remove an observer by disposing of the subscription returned by `Subscribe`.
- Separate side effects (in observers) from logic (in stream transformations).
- When deciding on whether to use `IObservable`, consider the following:
 - `IObservable` allows you to specify logic that spans multiple events.
 - `IObservable` is good for modeling unidirectional data flows, not request-response.

Functional Programming in C#

Enrico Buonanno



Functional programming changes the way you think about code. For C# developers, FP techniques can greatly improve state management, concurrency, event handling, and long-term code maintenance. And C# offers the flexibility that allows you to benefit fully from the application of functional techniques. This book gives you the awesome power of a new perspective.

Functional Programming in C# teaches you to apply functional thinking to real-world problems using the C# language. You'll start by learning the principles of functional programming and the language features that allow you to program functionally. As you explore the many practical examples, you'll learn the power of function composition, data flow programming, immutable data structures, and monadic composition with LINQ.

What's inside

- Write readable, team-friendly code
- Master async and data streams
- Radically improve error handling
- Event sourcing and other FP patterns

Written for proficient C# programmers with no prior FP experience.

Enrico Buonanno studied computer science at Columbia University and has 15 years of experience as a developer, architect, and trainer.

"Functional programming can make your head explode. This book stitches it back together."

—Daniel Marbach, Particular Software

"A top-ten technical book that turned me on to functional programming. The author does a fantastic job of organizing the content in a clear and concise manner—with humor."

—Alex Basile, Bloomberg

"Mind-bending. If you are an experienced C# developer with lots of questions about good code practice and general architecture, this book lifts you to another level."

—Aurélien Gounot, SNCF

"Best way to start learning FP using C# and getting in touch with future versions, C# 6 and C# 7."

—Gonzalo Barba López, MoneyMate

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/functional-programming-in-c-sharp

 **MANNING** US \$49.99 / Can \$65.99 [including eBook]

