ObserveOn and SubscribeOn - where the work is being done

Ask Question

51

50

Based on reading this question: [What's the difference between SubscribeOn and ObserveOn](https://stackoverflow.com)

`ObserveOn` sets where the code is in the `Subscribe` handler is is executed:

```
stream.Subscribe(_ => { // this code here });
```

The `SubscribeOn` method sets which thread the setup of the stream is done on.

I'm led to understand that if these aren't explicitly set, then the TaskPool is used.

Now my question is, lets say I do something like this:

```
Observable.Interval(new Timespan(0, 0, 1)).Where(t =>
predicate(t)).SelectMany(t =>
lots_of(t)).ObserveOnDispatcher().Subscribe(t =>
some_action(t));
```

Where are the `Where` `predicate` and `SelectMany` `lots_of` being executed, given that `some_action` is being executed on the dispatcher?

c# .net system.reactive

edited May 23 '17 at 11:54
Community ♦
1 1

asked Dec 8 '13 at 9:41
Cheetah
5,633 24 77 139

3 Answers

159

✓

There's a lot of misleading info out there about `SubscribeOn` and `ObserveOn`.

Summary

- **`SubscribeOn`** intercepts calls to the single method of **`IObservable<T>`**, which is `Subscribe`, and calls to `Dispose` on the `IDisposable` handle returned by `Subscribe`.
- **`ObserveOn`** intercepts calls to the methods of **`IObserver<T>`**, which are `OnNext`, `OnCompleted` & `OnError`.
- Both methods cause the respective calls to be made on the specified scheduler.

Analysis & Demonstrations

The statement

> ObserveOn sets where the code in the Subscribe handler is executed:

is more confusing than helpful. What you are referring to as the "Subscribe handler" is really an `OnNext` handler. Remember, the `Subscribe` method of `IObservable` accepts an `IObserver` that has `OnNext`, `OnCompleted` and `OnError` methods, but it is extension methods that provide the convenience overloads that accept lambdas and build an `IObserver` implementation for you.

Let me appropriate the term though; I think of the "Subscribe handler" being the code *in the observable* that is invoked when `Subscribe` is called. In this way, the description above more closely resembles the purpose of `SubscribeOn`.

SubscribeOn

`SubscribeOn` causes the `Subscribe` method of an observable to be executed asynchronously on the specified scheduler or context. You use it when you don't want to call the `Subscribe` method on an observable from whatever thread you are running on - typically because it can be long-running and you don't want to block the calling thread.

When you call `Subscribe`, you are calling an observable that may be part of a long chain of observables. It's only the observable that `SubscribeOn` is applied to that it effects. Now it may be the case that all the observables in the chain will be subscribed to immediately and on the same thread - but it doesn't have to be the case. Think about `Concat` for example - that only subscribes to each successive stream once the preceding stream has finished, and typically this will take place on whatever thread the preceding stream called `OnCompleted` from.

So `SubscribeOn` sits between your call to `Subscribe` and the observable you are subscribing to, intercepting the call and making it asynchronous.

It also affects disposal of subscriptions. `Subscribe` returns an `IDisposable` handle which is used to unsubscribe. `SubscribeOn` ensures calls to `Dispose` are scheduled on the supplied scheduler.

A common point of confusion when trying to understand what `SubscribeOn` does is that the `Subscribe` handler of an observable may well call `OnNext` , `OnCompleted` or `OnError` on this same thread. However, its purpose is not to affect these calls. It's not uncommon for a stream to complete before the `Subscribe` method returns. `Observable.Return` does this, for example. Let's take a look.

If you use the Spy method I wrote, and run the following code:

```
Console.WriteLine("Calling from Thread: " + Thread.Current
var source = Observable.Return(1).Spy("Return");
source.Subscribe();
Console.WriteLine("Subscribe returned");
```

You get this output (thread id may vary of course):

```
Calling from Thread: 1
Return: Observable obtained on Thread: 1
Return: Subscribed to on Thread: 1
Return: OnNext(1) on Thread: 1
Return: OnCompleted() on Thread: 1
Return: Subscription completed.
Subscribe returned
```

You can see that the entire subscription handler ran on the same thread, and finished before returning.

Let's use `SubscribeOn` to run this asynchronously. We will Spy on both the `Return` observable and the `SubscribeOn` observable:

```
Console.WriteLine("Calling from Thread: " + Thread.Current
var source = Observable.Return(1).Spy("Return");
```

```
source.SubscribeOn(Scheduler.Default).Spy("SubscribeOn").S
Console.WriteLine("Subscribe returned");
```

This outputs (line numbers added by me):

```
01 Calling from Thread: 1
02 Return: Observable obtained on Thread: 1
03 SubscribeOn: Observable obtained on Thread: 1
04 SubscribeOn: Subscribed to on Thread: 1
05 SubscribeOn: Subscription completed.
06 Subscribe returned
07 Return: Subscribed to on Thread: 2
08 Return: OnNext(1) on Thread: 2
09 SubscribeOn: OnNext(1) on Thread: 2
10 Return: OnCompleted() on Thread: 2
11 SubscribeOn: OnCompleted() on Thread: 2
12 Return: Subscription completed.
```

01 - The main method is running on thread 1.

02 - the `Return` observable is evaluated on the calling thread. We're just getting the `IObservable` here, nothing is subscribing yet.

03 - the `SubscribeOn` observable is evaluated on the calling thread.

04 - Now finally we call the `Subscribe` method of `SubscribeOn`.

05 - The `Subscribe` method completes asynchronously...

06 - ... and thread 1 returns to the main method. **This is the effect of SubscribeOn in action!**

07 - Meanwhile, `SubscribeOn` scheduled a call on the default scheduler to `Return`. Here it is received on thread 2.

08 - And as `Return` does, it calls `OnNext` on the `Subscribe` thread...

09 - and `SubscribeOn` is just a pass through now.

10,11 - Same for `OnCompleted`

12 - And last of all the `Return` subscription handler is done.

Hopefully that clears up the purpose and effect of `SubscribeOn` !

ObserveOn

If you think of `SubscribeOn` as an interceptor for the `Subscribe` method that passes the call on to a different thread, then `ObserveOn` does the same job, but for the `OnNext` , `OnCompleted` and `OnError` calls.

Recall our original example:

```
Console.WriteLine("Calling from Thread: " + Thread.Current
var source = Observable.Return(1).Spy("Return");
source.Subscribe();
Console.WriteLine("Subscribe returned");
```

Which gave this output:

```
Calling from Thread: 1
Return: Observable obtained on Thread: 1
Return: Subscribed to on Thread: 1
Return: OnNext(1) on Thread: 1
Return: OnCompleted() on Thread: 1
Return: Subscription completed.
Subscribe returned
```

Now lets alter this to use `ObserveOn` :

```
Console.WriteLine("Calling from Thread: " + Thread.Current
var source = Observable.Return(1).Spy("Return");
source.ObserveOn(Scheduler.Default).Spy("ObserveOn").Subsc
Console.WriteLine("Subscribe returned");
```

We get the following output:

```
01 Calling from Thread: 1
02 Return: Observable obtained on Thread: 1
```

```
03 ObserveOn: Observable obtained on Thread: 1
04 ObserveOn: Subscribed to on Thread: 1
05 Return: Subscribed to on Thread: 1
06 Return: OnNext(1) on Thread: 1
07 ObserveOn: OnNext(1) on Thread: 2
08 Return: OnCompleted() on Thread: 1
09 Return: Subscription completed.
10 ObserveOn: Subscription completed.
11 Subscribe returned
12 ObserveOn: OnCompleted() on Thread: 2
```

01 - The main method is running on Thread 1.

02 - As before, the `Return` observable is evaluated on the calling thread. We're just getting the `IObservable` here, nothing is subscribing yet.

03 - The `ObserveOn` observable is evaluated on the calling thread too.

04 - Now we subscribe, again on the calling thread, first to the `ObserveOn` observable...

05 - ... which then passes the call through to the `Return` observable.

06 - Now `Return` calls `OnNext` in its `Subscribe` handler.

07 - **Here is the effect of `ObserveOn`**. We can see that the `OnNext` is scheduled asynchronously on Thread 2.

08 - Meanwhile `Return` calls `OnCompleted` on Thread 1...

09 - And `Return`'s subscription handler completes...

10 - and then so does `ObserveOn`'s subscription handler...

11 - so control is returned to the main method

12 - Meanwhile, `ObserveOn` has shuttled `Return`'s `OnCompleted` call this over to Thread 2. This could have happened at any time during 09-11 because it is running asynchronously. Just so happens it's finally called now.

What are the typical use cases?

You will most often see `SubscribeOn` used in a GUI when you need to `Subscribe` to a long running observable and want to get off the dispatcher thread as soon as possible - maybe because you know it's one of those observables that does all it's work in the subscription handler. Apply it at the end of the observable chain, because this is the first observable called when you subscribe.

You will most often see `ObserveOn` used in a GUI when you want to ensure `OnNext` , `OnCompleted` and `OnError` calls are marshalled back to the dispatcher thread. Apply it at the end of the observable chain to transition back as late as possible.

Hopefully you can see that the answer to your question is that `ObserveOnDispatcher` won't make any difference to the threads that `Where` and `SelectMany` are executed on - it all depends what thread *stream* is calling them from! stream's subscription handler will be invoked on the calling thread, but it's impossible to say where `Where` and `SelectMany` will run without knowing how `stream` is implemented.

Observables with lifetimes that outlive the Subscribe call

Up until now, we've been looking exclusively at `Observable.Return` . `Return` completes its stream within the `Subscribe` handler. That's not atypical, but it's equally common for streams to outlive the `Subscribe` handler. Look at `Observable.Timer` for example:

```
Console.WriteLine("Calling from Thread: " + Thread.Current
var source = Observable.Timer(TimeSpan.FromSeconds(1)).Spy
source.Subscribe();
Console.WriteLine("Subscribe returned");
```

This returns the following:

```
Calling from Thread: 1
Timer: Observable obtained on Thread: 1
Timer: Subscribed to on Thread: 1
Timer: Subscription completed.
Subscribe returned
Timer: OnNext(0) on Thread: 2
Timer: OnCompleted() on Thread: 2
```

You can clearly see the subscription to complete and then `OnNext` and `OnCompleted` being called later on a different thread.

Note that no combination of `SubscribeOn` or `ObserveOn` will have *any effect whatsoever* on which thread or scheduler `Timer` choses to invoke `OnNext` and `OnCompleted` on.

Sure, you can use `SubscribeOn` to determine the `Subscribe` thread:

```
Console.WriteLine("Calling from Thread: " + Thread.Current
var source = Observable.Timer(TimeSpan.FromSeconds(1)).Spy
source.SubscribeOn(NewThreadScheduler.Default).Spy("Subscr
Console.WriteLine("Subscribe returned");
```

(I am deliberately changing to the `NewThreadScheduler` here to prevent confusion in the case of `Timer` happening to get the same thread pool thread as `SubscribeOn`)

Giving:

```
Calling from Thread: 1
Timer: Observable obtained on Thread: 1
SubscribeOn: Observable obtained on Thread: 1
SubscribeOn: Subscribed to on Thread: 1
SubscribeOn: Subscription completed.
Subscribe returned
Timer: Subscribed to on Thread: 2
Timer: Subscription completed.
Timer: OnNext(0) on Thread: 3
SubscribeOn: OnNext(0) on Thread: 3
Timer: OnCompleted() on Thread: 3
SubscribeOn: OnCompleted() on Thread: 3
```

Here you can clearly see the main thread on thread (1) returning after its `Subscribe` calls, but the `Timer` subscription getting its own thread (2), but the `OnNext` and `OnCompleted` calls running on thread (3).

Now for `ObserveOn`, let's change the code to (for those following along in code, use nuget package rx-wpf):

```
var dispatcher = Dispatcher.CurrentDispatcher;
Console.WriteLine("Calling from Thread: " + Thread.Current
var source = Observable.Timer(TimeSpan.FromSeconds(1)).Spy
source.ObserveOnDispatcher().Spy("ObserveOn").Subscribe();
Console.WriteLine("Subscribe returned");
```

This code is a little different. The first line ensures we have a dispatcher, and we also bring in `ObserveOnDispatcher` - this is just like `ObserveOn`, except it specifies we should use the `DispatcherScheduler` *of whatever thread `ObserveOnDispatcher` is evaluated on*.

This code gives the following output:

```
Calling from Thread: 1
Timer: Observable obtained on Thread: 1
ObserveOn: Observable obtained on Thread: 1
ObserveOn: Subscribed to on Thread: 1
Timer: Subscribed to on Thread: 1
Timer: Subscription completed.
ObserveOn: Subscription completed.
Subscribe returned
Timer: OnNext(0) on Thread: 2
ObserveOn: OnNext(0) on Thread: 1
Timer: OnCompleted() on Thread: 2
ObserveOn: OnCompleted() on Thread: 1
```

Note that the dispatcher (and main thread) are thread 1. `Timer` is still calling `OnNext` and `OnCompleted` on the thread of its choosing (2) - but the `ObserveOnDispatcher` is marshalling calls back onto the dispatcher thread, thread (1).

Also note that if we were to block the dispatcher thread (say by a `Thread.Sleep`) you would see that the

`ObserveOnDispatcher` would block (this code works best inside a LINQPad main method):

```
var dispatcher = Dispatcher.CurrentDispatcher;
Console.WriteLine("Calling from Thread: " + Thread.Current
var source = Observable.Timer(TimeSpan.FromSeconds(1)).Spy
source.ObserveOnDispatcher().Spy("ObserveOn").Subscribe();
Console.WriteLine("Subscribe returned");
Console.WriteLine("Blocking the dispatcher");
Thread.Sleep(2000);
Console.WriteLine("Unblocked");
```

And you'll see output like this:

```
Calling from Thread: 1
Timer: Observable obtained on Thread: 1
ObserveOn: Observable obtained on Thread: 1
ObserveOn: Subscribed to on Thread: 1
Timer: Subscribed to on Thread: 1
Timer: Subscription completed.
ObserveOn: Subscription completed.
Subscribe returned
Blocking the dispatcher
Timer: OnNext(0) on Thread: 2
Timer: OnCompleted() on Thread: 2
Unblocked
ObserveOn: OnNext(0) on Thread: 1
ObserveOn: OnCompleted() on Thread: 1
```

With the calls through the `ObserveOnDispatcher` only able to get out once the `Sleep` has run.

Key points

It's useful to keep in mind that Reactive Extensions is essentially a free-threaded library, and tries to be as lazy as possible about what thread it runs on - you have to deliberately interfere with `ObserveOn`, `SubscribeOn` and passing specific schedulers to operators that accept them to change this.

There's nothing a consumer of an observable can do to control what it's doing internally - `ObserveOn` and `SubscribeOn` are decorators that wrap the surface area of observers and observables to marshal calls across threads. Hopefully these examples have made that clear.

edited May 23 '17 at 12:10
Community ♦
1 1

answered Dec 8 '13 at 11:29
James World
22.8k 5 68 98

Right, I think I understand, but just so I am sure, in my example the `Where` would happen on the thread that is calling `onNext`, which will be the thread the observable is created on, unless I specify a different context in the `SubscribeOn` method. The `SubscribeOn` call must come before the `Where` for this to take effect. – Cheetah Dec 8 '13 at 12:49

5 I'll add a tidbit intended to help those attempting to remember all of this. Paraphrasing a portion of what James said: `SubscribeOn` intercepts calls to `Subscribe`, and `ObserverOn` intercepts calls to the `IObserver<T>`. Both methods cause the respective calls to be made on the specified scheduler. – cwharris Dec 9 '13 at 2:59

2 Awesome answer. Rx MVP for this alone. A++++, would +1 again. – Will Jan 29 '14 at 18:05

1 I think it's also very important to note that `SubscribeOn` also schedules calls to `Dispose` on subscriptions, not just to `Subscribe` on observables. – Dave Sexton Dec 2 '14 at 16:52

2 There should be a micropayments feature on S.O. for this kind of answer. Could be a good business plan for a profitable ICO! – Sentinel Jan 25 '18 at 9:08

11

I found James's answer very clear and comprehensive. However, I despite this I still find myself having to explain the differences.

Therefore I created a very simple/stupid example that allows me to graphically demonstrate what schedulers things are being called on. I've created a class `MyScheduler` that executes actions immediately, but will change the console colour.

The text output from the `SubscribeOn` scheduler is output in red and that from `ObserveOn` scheduler is output in blue.

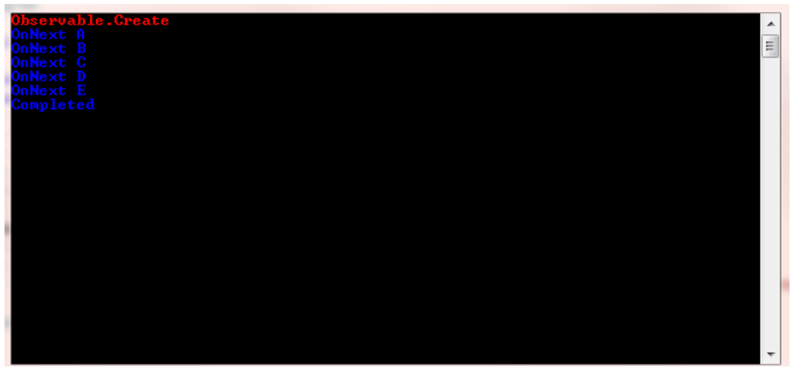
```
using System;
using System.Reactive.Concurrency;
using System.Reactive.Disposables;
using System.Reactive.Linq;

namespace SchedulerExample
{

    class Program
    {
        static void Main(string[] args)
        {
            var mydata = new[] {"A", "B", "C", "D", "E"};
            var observable = Observable.Create<string>(obse
                                        {
                                            Console.Wri
                                            return myda
                                                Subscri
                                        });

            observable.
                SubscribeOn(new MyScheduler(ConsoleColor.Re
                ObserveOn(new MyScheduler(ConsoleColor.Blue
                Subscribe(s => Console.WriteLine("OnNext {0
            Console.ReadKey();
        }
    }
}
```

This outputs:



```
Observable.Create
OnNext A
OnNext B
OnNext C
OnNext D
OnNext E
Completed
```

And for reference MyScheduler (not suitable for real usage):

```
using System;
using System.Reactive.Concurrency;
using System.Reactive.Disposables;

namespace SchedulerExample
{
    class MyScheduler : IScheduler
    {
        private readonly ConsoleColor _colour;

        public MyScheduler(ConsoleColor colour)
        {
            _colour = colour;
        }

        public IDisposable Schedule<TState>(TState state, F
IDisposable> action)
        {
            return Execute(state, action);
        }

        private IDisposable Execute<TState>(TState state, F
IDisposable> action)
        {
            var tmp = Console.ForegroundColor;
            Console.ForegroundColor = _colour;
            action(this, state);
            Console.ForegroundColor = tmp;
            return Disposable.Empty;
        }

        public IDisposable Schedule<TState>(TState state, T
Func<IScheduler, TState, IDisposable> action)
        {
            throw new NotImplementedException();
        }

        public IDisposable Schedule<TState>(TState state, D
Func<IScheduler, TState, IDisposable> action)
        {
            throw new NotImplementedException();
        }

        public DateTimeOffset Now
        {
            get { return DateTime.UtcNow; }
        }
    }
}
```

edited Mar 22 '16 at 16:15
Contango
40.8k 47 187 236

answered Feb 5 '14 at 14:36
Dave Hillier
9,709 7 34 80

1 Make sure you add the NuGet package "Rx-Main". – Contango Mar 22 '16 at 16:14

Love the color example! `return mydata.ToObservable().Subscribe(observer);` gives a `IDisposable` whose `Dispose` method would print in `Red` , too - if one told it to print anything. – nitzel Mar 8 '18 at 15:09

0

I often mistake that `.SubcribeOn` is used to set thread where code inside `.Subscribe` is being executed. But to remember, just think that publish and subscribe must be pair like yin-yang. To set where `Subscribe's code` being executed use `ObserveOn`. To set where `Observable's code` executed used `SubscribeOn`. Or in summary mantra: `where-what`, `Subscribe-Observe`, `Observe-Subscribe`.

answered Feb 19 '16 at 15:47
o0omycomputero0o
1,303 1 15 25

Does it mean `subscribeon` set where `where` `filter` `select` etc are excuted on? – joe Jun 25 '18 at 6:02

@joe: TLDR: use observeon in this case instead of subscribe on; more detail: medium.com/upday-devs/… – o0omycomputero0o Jun 25 '18 at 6:59