

[Reactive extensions](#) یا به صورت خلاصه Rx، کتابخانه‌ی [سورس باز](#) تهیه شده‌ای توسط مایکروسافت است که اگر بخواهیم آن را به ساده‌ترین شکل ممکن تعریف کنیم، معنای Linq to events را می‌دهد و امکان مدیریت تعامل‌های پیچیده‌ی async را به صورت declaratively فراهم می‌کند. هدف آن بسط فضای نام System.Linq و تبدیل نتایج یک کوئری LINQ به یک مجموعه‌ی Observable است؛ به همراه مدیریت مسایل همزمانی آن. این افزونه جزو موفق‌ترین کتابخانه‌های دات نت مایکروسافت در سال‌های اخیر به شما می‌رود؛ تا حدی که معادل‌های بسیاری از آن برای زبان‌های دیگر مانند Java، JavaScript، Python، CPP و غیره نیز تهیه شده‌اند.

استفاده از Rx به همراه یک کوئری LINQ

یک برنامه‌ی کنسول جدید را ایجاد کنید. سپس برای نصب کتابخانه‌ی Rx، دستور ذیل را در کنسول پاورشل [نیوگت](#) اجرا نمایید:

```
PM> Install-Package Rx-Main
```

نصب آن از طریق نیوگت، به صورت خودکار کلیه وابستگی‌های مرتبط با آن را نیز به پروژه‌ی جاری اضافه می‌کند:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Rx-Core" version="2.2.4" targetFramework="net45" />
  <package id="Rx-Interfaces" version="2.2.4" targetFramework="net45" />
  <package id="Rx-Linq" version="2.2.4" targetFramework="net45" />
  <package id="Rx-Main" version="2.2.4" targetFramework="net45" />
  <package id="Rx-PlatformServices" version="2.2.4" targetFramework="net45" />
</packages>
```

سپس متد Main این برنامه را به نحو ذیل تغییر دهید:

```
using System;
using System.Linq;

namespace Rx01
{
    class Program
    {
        static void Main(string[] args)
        {
            var query = Enumerable.Range(1, 5).Select(number => number);
            foreach (var number in query)
            {
                Console.WriteLine(number);
            }
            finished();
        }

        private static void finished()
        {
            Console.WriteLine("Done!");
        }
    }
}
```

در اینجا یک سری عملیات متداول را مشاهده می‌کنید. بازه‌ای از اعداد توسط متد Enumerable.Range ایجاد شده و سپس به کمک یک حلقه، تمام آیتم‌های آن نمایش داده می‌شوند. همچنین در پایان کار نیز یک متد دیگر فراخوانی شده‌است. اکنون اگر بخواهیم همین عملیات را توسط Rx انجام دهیم، به شکل زیر خواهد بود:

```
using System;
```

```
using System.Linq;
using System.Reactive.Linq;

namespace Rx01
{
    class Program
    {
        static void Main(string[] args)
        {
            var query = Enumerable.Range(1, 5).Select(number => number);
            var observableQuery = query.ToObservable();
            observableQuery.Subscribe(onNext: number => Console.WriteLine(number), onCompleted: () =>
finished());
        }

        private static void finished()
        {
            Console.WriteLine("Done!");
        }
    }
}
```

ابتدا نیاز است تا کوثری متداول LINQ را تبدیل به نمونه‌ی Observable آن کرد. اینکار را توسط متد الحاقی ToObservable که در فضای نام System.Reactive.Linq تعریف شده‌است، انجام می‌دهیم. به این ترتیب، هر زمانیکه که عددی به query اضافه می‌شود، با استفاده از متد Subscribe می‌توان تغییرات آن را تحت کنترل قرار داد. برای مثال در اینجا هر بار که عددی در بازه‌ی 1 تا 5 تولید می‌شود، یکبار پارامتر onNext اجرا خواهد شد. برای نمونه در مثال فوق، از نتیجه‌ی آن برای نمایش مقدار دریافتی، استفاده شده‌است. سپس توسط پارامتر اختیاری onCompleted، در پایان کار، یک متد خاص را می‌توان فراخوانی کرد. خروجی برنامه در این حالت نیز به صورت ذیل است:

```
1
2
3
4
5
Done!
```

البته اگر قصد خلاصه نویسی داشته باشیم، سطر آخر متد Main، با سطر ذیل یکی است:

```
observableQuery.Subscribe(Console.WriteLine, finished);
```

در این مثال ساده صرفاً یک Syntax دیگر را نسبت به حلقه‌ی foreach متداول مشاهده کردیم که اندکی فشرده‌تر است. در هر دو حالت نیز عملیات انجام شده در تردجاری صورت گرفته‌اند. اما قابلیت‌ها و ارزش‌های واقعی Rx زمانی آشکار خواهند شد که پردازش موازی و پردازش در تردهای دیگر را در آن فعال کنیم.

الگوی Observer

Rx پیاده سازی کننده‌ی الگوی طراحی شیء‌گرایی به نام [Observer](#) است. برای توضیح آن یک لامپ و سوئیچ برق را در نظر بگیرید. زمانیکه لامپ مشاهده می‌کند سوئیچ برق در حالت روشن قرار گرفته‌است، روشن خواهد شد و برعکس. در اینجا به سوئیچ، subject و به لامپ، observer گفته می‌شود. هر زمان که حالت سوئیچ تغییر می‌کند، از طریق یک callback، وضعیت خود را به observer اعلام خواهد کرد. علت استفاده از callbackها، ارائه راه‌حل‌های عمومی است تا بتواند با انواع و اقسام اشیاء کار کند. به این ترتیب هر بار که شیء observer از نوع متفاوتی تعریف می‌شود (مثلاً بجای لامپ یک خودرو قرار گیرد)، نیازی نخواهد بود تا subject را تغییر داد.

در Rx دو اینترفیس معادل observer و subject تعریف شده‌اند. در اینجا اینترفیس IObservable معادل observer است و اینترفیس IObservable معادل subject می‌باشد:

```
class Subject : IObservable<int>
{
    public IDisposable Subscribe(IObserver<int> observer)
    {
```

```
}
}
```

کار متد Subscribe، اتصال به Observer است و برای این حالت نیاز به کلاسی دارد که اینترفیس IObservable را پیاده سازی کند.

```
class Observer : IObservable<int>
{
    public void OnCompleted()
    {
    }

    public void OnError(Exception error)
    {
    }

    public void OnNext(int value)
    {
    }
}
```

در اینجا OnCompleted زمانی اجرا می‌شود که پردازش مجموعه‌ای از اعداد int پایان یافته باشد. OnError در زمان وقوع استثنایی اجرا می‌شود و OnNext به ازای هر عدد موجود در مجموعه‌ای در حال پردازش، یکبار اجرا می‌شود. البته نیازی به پیاده سازی صریح این اینترفیس نیست و توسط متد توکار Observable.Create می‌توان به همین نتیجه رسید. مجموعه‌های Observable کلید کار با Rx هستند. در مثال قبل ملاحظه کردیم که با استفاده از متد الحاقی ToObservable بر روی یک کوئری LINQ و یا هر نوع IEnumerable ای، می‌توان یک مجموعه‌ی Observable را ایجاد کرد. خروجی کوئری حاصل از آن به صورت خودکار اینترفیس IObservable را پیاده سازی می‌کند که دارای یک متد به نام Subscribe است. در متد Subscribe کاری که به صورت خودکار صورت خواهد گرفت، ایجاد یک حلقه‌ی foreach بر روی مجموعه‌ی مورد آنالیز و سپس فراخوانی متد OnNext کلاس پیاده سازی کننده‌ی IObservable به ازای هر آیتیم موجود در مجموعه است (فراخوانی observer.OnNext). در پایان کار هم فقط return this در اینجا صورت خواهد گرفت. در حین پردازش حلقه، اگر خطایی رخ دهد، متد observer.OnError انجام می‌شود.

در مثال قبل، کوئری LINQ نوشته شده، خروجی از نوع IObservable ندارد. به کمک متد الحاقی ToObservable:

```
public static System.IObservable<TSource> ToObservable<TSource>(
    this System.Collections.Generic.IEnumerable<TSource> source,
    System.Reactive.Concurrency.IScheduler scheduler)
```

به صورت خودکار، IEnumerable حاصل از کوئری LINQ را تبدیل به یک IObservable کرده‌ایم. به این ترتیب اکنون کوئری LINQ ما همانند سوئیچ برق عمل می‌کند و با تغییر آیتیم‌های موجود در آن، مشاهده‌گرهایی که به آن متصل شده‌اند (از طریق فراخوانی متد Subscribe)، امکان دریافت سیگنال‌های تغییر وضعیت آن‌را خواهند داشت. البته استفاده از متد Subscribe به نحوی که در مثال قبل ذکر شد، خلاصه شده‌ی الگوی Observer است. اگر بخواهیم دقیقاً مانند الگو عمل کنیم، چنین شکلی را خواهد داشت:

```
var query = Enumerable.Range(1, 5).Select(number => number);
var observableQuery = query.ToObservable();
var observer = Observer.Create<int>(onNext: number => Console.WriteLine(number));
observableQuery.Subscribe(observer);
```

ابتدا توسط متد ToObservable یک IObservable (سوئیچ) را ایجاد کرده‌ایم. سپس توسط کلاس Observer موجود در فضای نام System.Reactive، یک IObservable (لامپ) را ایجاد کرده‌ایم. کار اتصال سوئیچ به لامپ در متد Subscribe انجام می‌شود. اکنون هر زمانیکه تغییری در وضعیت observableQuery حاصل شود، سیگنالی را به observer ارسال می‌کند. در اینجا callbacks کار مدیریت observer را انجام می‌دهند.

پردازش نتایج یک کوئری LINQ در تدری دیگر توسط Rx

برای اجرای نتایج متد Subscribe در یک ترد جدید، می‌توان پارامتر scheduler متد ToObservable را مقدار دهی کرد:

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

namespace Rx01
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Thread-Id: {0}", Thread.CurrentThread.ManagedThreadId);
            var query = Enumerable.Range(1, 5).Select(number => number);
            var observableQuery = query.ToObservable(scheduler: NewThreadScheduler.Default);
            observableQuery.Subscribe(onNext: number =>
            {
                Console.WriteLine("number: {0}, on Thread-id: {1}", number,
                Thread.CurrentThread.ManagedThreadId);
            }, onCompleted: () => finished());
        }

        private static void finished()
        {
            Console.WriteLine("Done!");
        }
    }
}
```

خروجی این مثال به نحو ذیل است:

```
Thread-Id: 1
number: 1, on Thread-id: 3
number: 2, on Thread-id: 3
number: 3, on Thread-id: 3
number: 4, on Thread-id: 3
number: 5, on Thread-id: 3
Done!
```

پیش از آغاز کار و در متد Main، ترد آی دی ثبت شده مساوی 1 است. سپس هربار که callback متد Subscribe فراخوانی شده‌است، ملاحظه می‌کنید که ترد آی دی آن مساوی عدد 3 است. به این معنا که کلیه نتایج در یک ترد مشخص دیگر پردازش شده‌اند.

NewThreadScheduler.Default در فضای نام System.Reactive.Concurrency واقع شده‌است.

یک نکته

در نگارش‌های آغازین Rx، مقدار scheduler را می‌شد معادل Scheduler.NewThread نیز قرار داد که در نگارش‌های جدید منسوخ شده در نظر گرفته شده و به زودی حذف خواهد شد. معادل‌های جدید آن اکنون NewThreadScheduler.Default، ThreadPoolscheduler.Default و امثال آن هستند.

مدیریت خاتمه‌ی اعمال انجام شده‌ی در تردهای دیگر توسط Rx

یکی از مواردی که حین اجرای نتیجه‌ی callback‌های پردازش شده‌ی در تردهای دیگر نیاز است بدانیم، زمان خاتمه‌ی کار آن‌ها است. برای نمونه در مثال قبل، نمایش Done پس از پایان تمام callbacks انجام شده‌است. فرض کنید، callback پایان عملیات را حذف کرده و متد finished را پس از فراخوانی متد observableQuery.Subscribe قرار دهیم:

```
observableQuery.Subscribe(onNext: number =>
{
    Console.WriteLine("number: {0}, on Thread-id: {1}", number,
    Thread.CurrentThread.ManagedThreadId);
}/*, onCompleted: () => finished()*/);
```

```
finished();
```

اینبار اگر برنامه را اجرا کنیم به خروجی ذیل خواهیم رسید:

```
Thread-Id: 1
number: 1, on Thread-id: 3
Done!
number: 2, on Thread-id: 3
number: 3, on Thread-id: 3
number: 4, on Thread-id: 3
number: 5, on Thread-id: 3
```

این خروجی بدین معنا است که متد `observableQuery.Subscribe` در حین اجرا شدن در تردی دیگر، صبر نخواهد کرد تا عملیات مرتبط با آن خاتمه یابد و سپس سطر بعدی را اجرا کند. بنابراین برای حل این مشکل، تنها کافی است به آن اعلام کنیم که پس از پایان عملیات، `onCompleted` را اجرا کن.

مدیریت استثنای رخ داده در حین پردازش مجموعه‌های واکنشگرا

متد `Subscribe` دارای چندین `overload` است. تا اینجا نمونه‌ای که دارای پارامترهای `onNext` و `onCompleted` بودند را بررسی کردیم. اگر بخواهیم مدیریت استثناءها را نیز در اینجا اضافه کنیم، فقط کافی است از `overload` دیگر آن که دارای پارامتر `onError` است، استفاده نمائیم:

```
observableQuery.Subscribe(
    onNext: number => Console.WriteLine(number),
    onError: exception => Console.WriteLine(exception.Message),
    onCompleted: () => finished());
```

اگر `callback` پارامتر `onError` اجرا شود، دیگر به `onCompleted` نخواهیم رسید. همچنین دیگر `onNext` ایی نیز اجرا نخواهد شد.

مدیریت ترد اجرای نتایج حاصل از Rx در یک برنامه‌ی دسکتاپ WPF یا WinForms

تا اینجا مشاهده کردیم که اجرای `callback`های `observer` در یک ترد دیگر، به سادگی تنظیم پارامتر `scheduler` متد `ToObservable` است. اما در برنامه‌های دسکتاپ برای به روز رسانی عناصر رابط کاربری، حتما باید در تردی قرار داشته باشیم که آن رابط کاربری در آن ایجاد شده است یا به عبارتی در ترد اصلی برنامه؛ در غیر اینصورت برنامه کرش خواهد کرد. مدیریت این مساله نیز در Rx بسیار ساده است. ابتدا نیاز است بسته‌ی [Rx-WPF](#) را نصب کرد:

```
PM> Install-Package Rx-WPF
```

سپس توسط متد `ObserveOn` می‌توان مشخص کرد که نتیجه‌ی عملیات باید بر روی کدام ترد اجرا شود:

```
observableQuery.ObserveOn(DispatcherScheduler.Current).Subscribe(...)
```

روش دیگر آن استفاده از متد `ObserveOnDispatcher` می‌باشد:

```
observableQuery.ObserveOnDispatcher().Subscribe(...)
```

بنابراین مشخص سازی پارامتر `scheduler` متد `ToObservable`، به معنای اجرای `query` آن در یک ترد دیگر و استفاده از متد `ObserveOn`، به معنای مشخص سازی ترد اجرای `callback`های مشاهده‌گر است.

و یا اگر از WinForms استفاده می‌کنید، ابتدا [بسته‌ی Rx خاص آن را](#) نصب کنید:

```
PM> Install-Package Rx-WinForms
```

و سپس ترد اجرای callback ها را `SynchronizationContext.Current` مشخص نمائید:

```
observableQuery.ObserveOn(SynchronizationContext.Current).Subscribe(...)
```

یک نکته

در Rx فرض می‌شود که کوثری شما زمانبر است و callback های مشاهده‌گر سریع عمل می‌کنند. بنابراین هدف از callback های آن، پردازش‌های سنگین نیست. جهت آزمایش این مساله، اینبار query ابتدایی برنامه را به شکل ذیل تغییر دهید که در آن بازگشت زمانبر یک سری داده شبیه سازی شده‌اند.

```
var query = Enumerable.Range(1, 5).Select(number =>
{
    Thread.Sleep(250);
    return number;
});
```

سپس با استفاده از متد `ToObservable`، ترد دیگری را برای اجرای واقعی آن مشخص کنید تا در حین اجرای آن برنامه در حالت هنگ به نظر نرسد و سپس نمایش آن را به کمک متد `ObserveOn`، بر روی ترد اصلی برنامه انجام دهید.

نظرات خوانندگان

نویسنده:

ژوپتر

تاریخ:

۱۱:۳۷ ۱۳۹۳/۰۲/۲۹

به نظرم باید نوع آرگومان اینجا مشخص باشه:

```
var observableQuery = query.ToObservable(scheduler: NewThreadScheduler.Default);
```

```
var observableQuery = query.ToObservable<int>(scheduler: NewThreadScheduler.Default);
```

نویسنده:

وحید نصیری

تاریخ:

۱۱:۵۲ ۱۳۹۳/۰۲/۲۹

زمانیکه از ریشارپر استفاده می‌کنید، این تعیین نوع صریح را به صورت کم رنگ (به معنای کد مرده یا زاید) معرفی می‌کند:

```
var observableQuery = query.ToObservable<int>(scheduler: NewThreadScheduler.Default);
//observableQuery.ObserveOn(Synchronizat
observableQuery/*.ObserveOnDispatcher()*.Type argument specification is redundant
current).Subscribe(
```

علت اینجا است که نوع آرگومان جنریک به صورت خودکار توسط نوع پارامتر ارسالی به متد قابل تشخیص است (در اینجا چون ToObservable یک متد الحاقی است، اولین پارامتر آن، عناصر توالی query هستند که از نوع IEnumerable of int تعریف شدند). برای مطالعه بیشتر مراجعه کنید به [Inference of type arguments part 25.6.4](#) C# specs (ECMA-334)

نویسنده:

ژوپتر

تاریخ:

۱۲:۱۵ ۱۳۹۳/۰۲/۲۹

حق با شماست. متأسفانه نمی‌دانم چرا ابتدا کامپایلر از این خط خطا می‌گرفت و می‌گفت باید نوع آرگومان تعیین شود.

در مطلب « [معرفی Reactive extensions](#) » با نحوه‌ی تبدیل IEnumerableها به نمونه‌های Observable آشنا شدیم. اما سایر حالات چگونه؟ آیا Rx صرفاً محدود است به کار با IEnumerableها؟ در ادامه نگاهی خواهیم داشت به نحوه‌ی تبدیل بسیاری از منابع داده دیگر به توالی‌های Observable قابل استفاده در Rx.

روش‌های متفاوت ایجاد توالی (sequence) در Rx

الف) استفاده از متدهای Factory

Observable.Create (1)

نمونه‌ای از استفاده از آن را در مطلب « [معرفی Reactive extensions](#) » مشاهده کردید.

```
var query = Enumerable.Range(1, 5).Select(number => number);
var observableQuery = query.ToObservable();
var observer = Observer.Create<int>(onNext: number => Console.WriteLine(number));
observableQuery.Subscribe(observer);
```

کار آن، تدارک delegate ایی است که توسط متد Subscribe، به ازای هربار پردازش مقدار موجود در توالی معرفی شده به آن، فراخوانی می‌گردد و هدف اصلی از آن این است که به صورت دستی اینترفیس IObservable را پیاده سازی نکنید (امکان پیاده سازی inline یک اینترفیس توسط Actionها). البته در این مثال فقط delegate مربوط به onNext را ملاحظه می‌کند. توسط سایر overloadهای آن امکان ذکر delegateهای onError/onCompleted نیز وجود دارد.

Observable.Return (2)

برای ایجاد یک خروجی Observable از یک مقدار مشخص، می‌توان از متد جنریک Observable.Return استفاده کرد. برای مثال:

```
var observableValue1 = Observable.Return("Value");
var observableValue2 = Observable.Return(2);
```

در ادامه نحوه‌ی پیاده سازی این متد را توسط Observable.Create مشاهده می‌کنید:

```
public static IObservable<T> Return<T>(T value)
{
    return Observable.Create<T>(o =>
    {
        o.OnNext(value);
        o.OnCompleted();
        return Disposable.Empty;
    });
}
```

البته دو سطر نوشته شده در اصل معادل هستند با سطرهای ذیل؛ که ذکر نوع جنریک آن‌ها ضروری نیست. زیرا به صورت خودکار از نوع آرگومان معرفی شده، تشخیص داده می‌شود:

```
var observableValue1 = Observable.Return<string>("Value");
var observableValue2 = Observable.Return<int>(2);
```

Observable.Empty (3)

برای بازگشت یک توالی خالی که تنها کار اطلاع رسانی onCompleted را انجام می‌دهد.


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

در کدهای ذیل، پیاده سازی این متد را توسط Observable.Create مشاهده می‌کنید:

```
public static IObservable<T> Empty<T>()
{
    return Observable.Create<T>(o =>
    {
        o.OnCompleted();
        return Disposable.Empty;
    });
}
```

Observable.Never (4)

برای بازگشت یک توالی بدون قابلیت اطلاع رسانی و notification

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

این متد به نحو زیر توسط Observable.Create پیاده سازی شده‌است:

```
public static IObservable<T> Never<T>()
{
    return Observable.Create<T>(o =>
    {
        return Disposable.Empty;
    });
}
```

Observable.Throw (5)

برای ایجاد یک توالی که صرفاً کار اطلاع رسانی OnError را توسط استثنای معرفی شده به آن انجام می‌دهد.

```
var throwObservable = Observable.Throw<string>(new Exception());
```

در ادامه نحوه‌ی پیاده سازی این متد را توسط Observable.Create مشاهده می‌کنید:

```
public static IObservable<T> Throws<T>(Exception exception)
{
    return Observable.Create<T>(o =>
    {
        o.OnError(exception);
        return Disposable.Empty;
    });
}
```

Observable.Range توسط (6)

به سادگی می‌توان بازه‌ی Observable ایی را ایجاد کرد:

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

Observable.Generate (7)

اگر بخواهیم عملیات Observable.Range را پیاده سازی کنیم، می‌توان از متد Observable.Generate استفاده کرد:

```
public static IObservable<int> Range(int start, int count)
{
    var max = start + count;
    return Observable.Generate(
        initialState: start,
```

```

        condition: value => value < max,
        iterate: value => value + 1,
        resultSelector: value => value);
    }

```

توسط پارامتر `initialState`، مقدار آغازین را دریافت می‌کند. پارامتر `condition`، مشخص می‌کند که توالی چه زمانی باید خاتمه یابد. در پارامتر `iterate`، مقدار جاری دریافت شده و مقدار بعدی تولید می‌شود. `resultSelector` کار تبدیل و بازگشت مقدار خروجی را به عهده دارد.

Observable.Interval (8)

عموماً از انواع و اقسام تایمرهای موجود در دات نت مانند `System.Timers.Timer`، `System.Threading.Timer` و `System.Windows.Threading.DispatcherTimer` برای ایجاد یک توالی از رخدادها استفاده می‌شود. تمام این‌ها را به سادگی می‌توان توسط متد `Observable.Interval`، که قابل انتقال به تمام پلتفرم‌هایی است که Rx برای آن‌ها تهیه شده‌است، جایگزین کرد:

```

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

```

در اینجا تایمر تهیه شده، هر 450 میلی‌ثانیه یکبار اجرا می‌شود. برای خاتمه‌ی آن باید شیء `interval` را `Dispose` کنید. `Overload` دوم این متد، امکان معرفی `scheduler` و اجرای بر روی تردی دیگر را نیز میسر می‌کند.

Observable.Timer (9)

تفاوت `Observable.Timer` با `Observable.Interval` در مفهوم پارامتر ارسالی به آن‌ها است:

```

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

```

یکی `due time` دارد (مدت زمان صبر کردن تا تولید اولین خروجی) و دیگری `period` (به صورت متوالی تکرار می‌شود). خروجی `Observable.Interval` مثال زده شده به نحو زیر است و خاتمه‌ای ندارد:

0
1
2
3
4
5

اما خروجی `Observable.Timer` به نحو ذیل بوده و پس از یک ثانیه، خاتمه می‌یابد:

0

completed

متد `Observable.Timer` دارای هفت `overload` متفاوت است که توسط آن‌ها `dueTime` (مدت زمان صبر کردن تا تولید اولین خروجی)، `period` (کار `Observable.Timer` را به صورت متوالی در بازه‌ی زمانی مشخص شده تکرار می‌کند) و `scheduler` (تعیین ترد اجرایی عملیات) قابل مقدار دهی هستند.

اگر می‌خواهید `Observable.Timer` بلافاصله شروع به کار کند، مقدار `dueTime` آن‌را مساوی `TimeSpan.Zero` قرار دهید. به این ترتیب یک `Observable.Interval` را به وجود آورده‌اید که بلافاصله شروع به کار کرده است و تا مدت زمان مشخص شده‌ای جهت اجرای اولین `callback` خود صبر نمی‌کند.

ب) تبدیلگرهایی که خروجی `IObservable` ایجاد می‌کنند

برای تبدیل مدل‌های برنامه نویسی Async قدیمی دات نت مانند APM، رخدادها و امثال آن به معادل‌های Rx، متدهای الحاقی خاصی تهیه شده‌اند.

1) تبدیل delegates به معادل Observable

متد Observable.Start، امکان تبدیل یک Func یا Action زمانبر را به یک توالی observable میسر می‌کند. در این حالت به صورت پیش فرض، پردازش عملیات بر روی یکی از تردهای ThreadPool انجام می‌شود.

```
static void StartAction()
{
    var start = Observable.Start(() =>
    {
        Console.WriteLine("Observable.Start");
        for (int i = 0; i < 10; i++)
        {
            Thread.Sleep(100);
            Console.WriteLine(".");
        }
    });
    start.Subscribe(
        onNext: unit => Console.WriteLine("published"),
        onCompleted: () => Console.WriteLine("completed"));
}

static void StartFunc()
{
    var start = Observable.Start(() =>
    {
        Console.WriteLine("Observable.Start");
        for (int i = 0; i < 10; i++)
        {
            Thread.Sleep(100);
            Console.WriteLine(".");
        }
        return "value";
    });
    start.Subscribe(
        onNext: Console.WriteLine,
        onCompleted: () => Console.WriteLine("completed"));
}
```

در اینجا دو مثال از بکارگیری Action و Funcها را توسط Observable.Start مشاهده می‌کنید. زمانیکه از Func استفاده می‌شود، تابع یک خروجی را ارائه داده و سپس توالی خاتمه می‌یابد. اگر از Action استفاده شود، نوع Observable بازگشت داده شده از نوع Unit است که در برنامه نویسی functional معادل void است و هدف از آن مشخص سازی پایان عملیات Action می‌باشد. Unit دارای مقداری نبوده و صرفاً سبب اجرای اطلاع رسانی OnNext می‌شود. تفاوت مهم Observable.Start و Observable.Return در این است که Observable.Start مقدار تابع را به صورت تنبل (lazily) پردازش می‌کند، اما Observable.Return پردازش حریصانه‌ای (eagerly) را به همراه خواهد داشت. به این ترتیب Observable.Start بسیار شبیه به یک Task (پردازش‌های غیرهمزمان) عمل می‌کند. در اینجا شاید این سؤال مطرح شود که استفاده از قابلیت‌های Async سی‌شارپ 5 برای اینگونه کارها مناسب است یا Rx؟ قابلیت‌های Async بیشتر به اعمال مخصوص IO bound مانند کار با شبکه، دریافت فایل از اینترنت، کار با یک بانک اطلاعاتی خارج از مرزهای سیستم، مرتبط می‌شوند؛ اما اعمال CPU bound مانند محاسبات سنگین حاصل از توالی‌های observable را به خوبی می‌توان توسط Rx مدیریت کرد.

2) تبدیل Events به معادل Observable

دات نت از روزهای اول خود به همراه یک event driven programming model بوده‌است. Rx متدهایی را برای دریافت یک رخداد و تبدیل آن به یک توالی Observable ارائه داده‌است. برای نمونه ObservableCollection زیر را در نظر بگیرید

```
var items = new System.Collections.ObjectModel.ObservableCollection<string>
{
    "Item1", "Item2", "Item3"
```

```
};
```

اگر بخواهیم مانند روش‌های متداول، حذف شدن آیتم‌های آن را تحت نظر قرار دهیم، می‌توان نوشت:

```
items.CollectionChanged += (sender, ea) =>
{
    if (ea.Action == NotifyCollectionChangedAction.Remove)
    {
        foreach (var oldItem in ea.OldItems.Cast<string>())
        {
            Console.WriteLine("Removed {0}", oldItem);
        }
    }
};
```

این نوع کدها در WPF زیاد کاربرد دارند. اکنون معادل کدهای فوق با Rx به صورت زیر هستند:

```
var removals =
    Observable.FromEventPattern<NotifyCollectionChangedEventHandler,
        NotifyCollectionChangedEventArgs>
        (
            addHandler: handler => items.CollectionChanged += handler,
            removeHandler: handler => items.CollectionChanged -= handler
        )
        .Where(e => e.EventArgs.Action == NotifyCollectionChangedAction.Remove)
        .SelectMany(c => c.EventArgs.OldItems.Cast<string>());

var disposable = removals.Subscribe(onNext: item => Console.WriteLine("Removed {0}",
    item));
```

با استفاده از متد `Observable.FromEventPattern` می‌توان معادل `Observable` رخداد `CollectionChanged` را تهیه کرد. پارامتر اول جنریک آن، نوع رخداد است و پارامتر اختیاری دوم آن، `EventArgs` این رخداد. همچنین با توجه به قسمت `Where` نوشته شده، در این بین مواردی را که `Action` مساوی حذف شدن را دارا هستند، فیلتر کرده و نهایتاً لیست `Observable` آن‌ها بازگشت داده می‌شوند. اکنون می‌توان با استفاده از متد `Subscribe`، این تغییرات را دریافت کرد. برای مثال با فراخوانی

```
items.Remove("Item1");
```

بلافاصله خروجی `Removed item1` ظاهر می‌شود.

(3) تبدیل Task به معادل Observable

متد `ToObservable` واقع در فضای نام `System.Reactive.Threading.Tasks` را بر روی یک `Task` نیز می‌توان فراخوانی کرد:

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

البته باید دقت داشت استفاده از `Task` دات نت 4.5 که بیشتر جهت پردازش‌های `async` اعمال `I/O-bound` طراحی شده‌است، بر `IObservable` مقدم است. صرفاً اگر نیاز است این `Task` را با سایر `observables` ادغام کنید از متد `ToObservable` برای کار با آن استفاده نمائید.

(4) تبدیل IEnumerable به معادل Observable

با این مورد [تاکنون](#) آشنا شده‌اید. فقط کافی است متد `ToObservable` را بر روی یک `IEnumerable`، جهت تهیه خروجی `Observable` فراخوانی کرد.

(5) تبدیل APM به معادل Observable

APM یا Asynchronous programming model، همان روش کار با متدهای Async با نام‌های BeginXXX و EndXXX است که از نگارش‌های آغازین دات نت به همراه آن بوده‌اند. کار کردن با آن مشکل است و مدیریت آن به همراه پراکندگی‌های بسیاری جهت کار با callbacks آن است. برای تبدیل این نوع روش برنامه نویسی به روش Rx نیز متدهایی پیش بینی شده‌است؛ مانند Observable.FromAsyncPattern.

یک نکته

کتابخانه‌ای به نام Rxx بسیاری از این محصور کننده‌ها را تهیه کرده‌است:

<http://Rxx.codeplex.com>

ابتدا بسته‌ی نیوگت آن را نصب کنید:

```
PM> Install-Package Rxx
```

سپس برای نمونه، برای کار با یک فایل استریم خواهیم داشت:

```
using (new FileStream("file.txt", FileMode.Open)
        .ReadToEndObservable()
        .Subscribe(x => Console.WriteLine(x.Length)))
{
    Console.ReadKey();
}
```

متد ReadToEndObservable یکی از متدهای الحاقی کتابخانه‌ی Rxx است.

پس از [معرفی](#) و مشاهده‌ی نحوه‌ی [ایجاد توالی‌ها در Rx](#)، بهتر است با نمونه‌ای از نحوه‌ی استفاده از آن در یک برنامه‌ی WPF آشنا شویم.

بنابراین ابتدا دو بسته‌ی Rx-Main و Rx-WPF را توسط نیوگت، به یک برنامه‌ی جدید WPF اضافه کنید:

```
PM> Install-Package Rx-Main
PM> Install-Package Rx-WPF
```

فرض کنید قصد داریم محتوای یک فایل حجیم را به نحو ذیل خوانده و توسط Rx نمایش دهیم.

```
private static IEnumerable<string> readFile(string filename)
{
    using (TextReader reader = File.OpenText(filename))
    {
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            Thread.Sleep(100);
            yield return line;
        }
    }
}
```

در اینجا برای ایجاد یک توالی `IEnumerable`، از `yield return` استفاده شده‌است. همچنین `Thread.Sleep` آن جهت بررسی قفل شدن رابط کاربری در حین خواندن فایل به عمد قرار گرفته است. UI برنامه نیز به نحو ذیل است:

```
<Window x:Class="WpfApplicationRxTests.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="450" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Button Grid.Row="0" Name="btnGenerateSequence" Click="btnGenerateSequence_Click">Generate
sequence</Button>
        <ListBox Grid.Row="1" Name="lstNumbers" />
        <Button Grid.Row="2" IsEnabled="False" Name="btnStop" Click="btnStop_Click">Stop</Button>
    </Grid>
</Window>
```

با این کدها

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.IO;
using System.Reactive.Concurrency;
using System.Reactive.Linq;
using System.Threading;
using System.Windows;

namespace WpfApplicationRxTests
{
    public partial class MainWindow
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

```

    }

    private static IEnumerable<string> readFile(string filename)
    {
        using (TextReader reader = File.OpenText(filename))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                Thread.Sleep(100);
                yield return line;
            }
        }
    }

    private IDisposable _subscribe;
    private void btnGenerateSequence_Click(object sender, RoutedEventArgs e)
    {
        btnGenerateSequence.IsEnabled = false;
        btnStop.IsEnabled = true;

        var items = new ObservableCollection<string>();
        lstNumbers.ItemsSource = items;
        _subscribe = readFile("test.txt").ToObservable()
            .SubscribeOn(ThreadPoolScheduler.Instance)
            .ObserveOn(DispatcherScheduler.Current)
            .Finally(finallyAction: () =>
            {
                btnGenerateSequence.IsEnabled = true;
                btnStop.IsEnabled = false;
            })
            .Subscribe(onNext: line =>
            {
                items.Add(line);
            },
            onError: ex => { },
            onCompleted: () =>
            {
                //lstNumbers.ItemsSource = items;
            });
    }

    private void btnStop_Click(object sender, RoutedEventArgs e)
    {
        _subscribe.Dispose();
    }
}

```

توضیحات

حاصل متد `readFile` را که یک توالی معمولی `IEnumerable` را ایجاد می‌کند، توسط فراخوانی متد `ToObservable`، تبدیل به یک خروجی `IObservable` کرده‌ایم تا بتوانیم هربار که سطری از فایل مدنظر خوانده می‌شود، نسبت به آن واکنش نشان دهیم. متد `SubscribeOn` مشخص می‌کند که این توالی `Observable` باید بر روی چه تردی اجرا شود. در اینجا از `ThreadPoolScheduler.Instance` استفاده شده‌است تا در حین خواندن فایل، رابط کاربری در حالت هنگ به نظر نرسد و ترد جاری (ترد اصلی برنامه) به صورت خودکار آزاد گردد. از متد `ObserveOn` با پارامتر `DispatcherScheduler.Current` استفاده کرده‌ایم، تا نتیجه‌ی واکنش‌های به خوانده شدن سطرهای یک فایل مفروض، در ترد اصلی برنامه صورت گیرد. در غیر اینصورت امکان کار کردن با عناصر رابط کاربری در یک ترد دیگر وجود نخواهد داشت و برنامه کرش می‌کند. در قسمت‌های قبل، صرفاً متد `Subscribe` را مشاهده کرده بودید. در اینجا از متد `Finally` نیز استفاده شده‌است. علت اینجا است که اگر در حین خواندن فایل خطایی رخ دهد، قسمت `onError` متد `Subscribe` اجرا شده و دیگر به پارامتر `onCompleted` آن نخواهیم رسید. اما متد `Finally` آن همیشه در پایان عملیات اجرا می‌شود. خروجی حاصل از متد `Subscribe`، از نوع `IDisposable` است. `Rx` به صورت خودکار پس از پردازش آخرین عنصر توالی، این شیء را `Dispose` می‌کند. اینجا است که `callback` متد `Finally` یاد شده فراخوانی خواهد شد. اما اگر در حین خواندن یک فایل طولانی، کاربر علاقمند باشد تا عملیات را متوقف کند، تنها کافی است که به صورت صریح، این شیء را `Dispose` نماید. به همین جهت است که مشاهده می‌کنید، این خروجی به صورت یک فیلد تعریف شده‌است تا در متد `Stop` بتوانیم آن را در صورت نیاز

Dispose کنیم.

مثال فوق را از اینجا نیز می‌توانید دریافت کنید:

[WpfApplicationRxTests.zip](#)

به صورت پیش فرض، Rx هر بار تنها یک مقدار را بررسی می‌کند. اما گاهی از اوقات نیاز است تا در هر بار، بیشتر از یک مقدار دریافت و پردازش شوند. برای این منظور Rx متدهای الحاقی ویژه‌ای را به نام‌های Scan، Buffer و Window تدارک دیده‌است تا بتواند از یک توالی، چندین توالی را تولید کند (توالی توالی‌ها = Sequence of sequences).

متد Scan

فرض کنید قصد دارید تعدادی عدد را با هم جمع بزنید. برای اینکار عموماً عدد اول با عدد دوم جمع زده شده و سپس حاصل آن با عدد سوم جمع زده خواهد شد و به همین ترتیب تا آخر توالی. کار متد Scan نیز دقیقاً به همین نحو است. هر بار که قرار است توالی پردازش شود، حاصل عملیات مرحله‌ی قبل را در اختیار مصرف کننده قرار می‌دهد. در مثال ذیل، قصد داریم حاصل جمع اعداد موجود در آرایه‌ای را بدست بیاوریم:

```
var sequence = new[] { 12, 3, -4, 7 }.ToObservable();
var runningSum = sequence.Scan((accumulator, value) =>
{
    Console.WriteLine("accumulator {0}", accumulator);
    Console.WriteLine("value {0}", value);
    return accumulator + value;
});
runningSum.Subscribe(result => Console.WriteLine("result {0}\n", result));
```

با این خروجی

```
result 12
accumulator 12
value 3
result 15
accumulator 15
value -4
result 11
accumulator 11
value 7
result 18
```

در اولین بار اجرای متد Subscribe، کار مقدار دهی accumulator با اولین عنصر آرایه صورت می‌گیرد. در دفعات بعدی، مقدار این accumulator با عدد جاری جمع زده شده و حاصل این عملیات در تکرار آتی، مجدداً توسط accumulator قابل دسترسی خواهد بود.

یک نکته: اگر علاقمند باشیم که مقدار اولیه‌ی accumulator، اولین عنصر توالی نباشد، می‌توان آن را توسط پارامتر seed متد Scan مقدار دهی کرد:

```
var runningSum = sequence.Scan(seed: 10, accumulator: (accumulator, value) =>
```

متد Buffer

متد بافر، کار تقسیم یک توالی را به توالی‌های کوچکتر، بر اساس زمان، یا تعداد عنصر مشخص شده، انجام می‌دهد. برای مثال در برنامه‌های دسکتاپ شاید نیازی نباشد تا به ازای هر عنصر توالی، یکبار رابط کاربری را به روز کرد. عموماً بهتر است تا تعداد

مشخصی از عناصر یکجا پردازش شده و نتیجه‌ی این پردازش به تدریج نمایش داده شود.

```
var sequence = Enumerable.Range(1, 200)
    .ToObservable()
    .Buffer(count: 10);

sequence.Subscribe(onNext: numbers =>
{
    Console.WriteLine(numbers.Sum());
});
```

در اینجا نحوه‌ی استفاده از متد بافر را به همراه مشخص کردن تعداد اعضای بافر ملاحظه می‌کنید. هربار که `onNext` متد `Subscribe` فراخوانی شود، 10 عنصر از توالی را در اختیار خواهیم داشت (بجای یک عنصر حالت متداول بافر نشده). به این ترتیب می‌توان فشار حجم اطلاعات ورودی با فرکانس بالا را کنترل کرد و در نتیجه از منابع موجود بهتر استفاده نمود. برای مثال اگر می‌خواهید عملیات `bulk insert` را انجام دهید، می‌توان بر اساس یک `batch size` مشخص، گروه گروه اطلاعات را به بانک اطلاعاتی اضافه کرد تا فشار کار کاهش یابد.

همینکار را بر اساس زمان نیز می‌توان انجام داد:

```
var sequence = Enumerable.Range(1, 200)
    .ToObservable()
    .Buffer(TimeSpan.FromSeconds(2));
```

در مثال فوق هر 2 ثانیه یکبار، مجموعه‌ای از عناصر به متد `onNext` ارسال خواهند شد.

متد Window

متد `Window` نیز دقیقاً همان پارامترهای متد بافر را قبول می‌کند. با این تفاوت که هربار، یک توالی `observable` را به متد `onNext` ارسال می‌کند.

نوع `numbers` پارامتر `onNext`، در حین بکارگیری متد بافر در مثال‌های فوق، `IList of int` است. اما اگر متدهای `Buffer` را تبدیل به متد `Window` کنیم، اینبار نوع `numbers`، معادل `IObservable of int` خواهد شد.

```
var sequence = Enumerable.Range(1, 200)
    .ToObservable()
    .Window(TimeSpan.FromSeconds(2));

sequence.Subscribe(onNext: numbers =>
{
    numbers.Subscribe(onNext: number => Console.WriteLine(number));
});
```

چه زمانی باید از Buffer استفاده کرد و چه زمانی از Window؟

در متد بافر، به ازای هر توالی که به پارامتر `onNext` ارسال می‌شود، یکبار وهله‌ی جدیدی از توالی مدنظر در حافظه ایجاد و ارسال خواهد شد. در متد `Window` صرفاً اشاره‌گرهایی به این توالی را در اختیار داریم؛ بنابراین مصرف حافظه‌ی کمتری را شاهد خواهیم بود. متد `Window` بسیار مناسب است برای اعمال `aggregation`. مثلاً اگر نیاز است جمع، میانگین، حداقل و حداکثر عناصر دریافتی محاسبه شوند، بهتر است از متد `Window` استفاده شود که نهایتاً قابلیت استفاده از متدهای الحاقی `Sum` و `Min` و `Max` را به همراه دارد. با این تفاوت که حاصل این‌ها نیز یک `IObservable` است که باید `Subscribe` آن را برای دریافت نتیجه فراخوانی کرد:

```
var sequence = Enumerable.Range(1, 200)
    .ToObservable()
    .Window(10);

sequence.Subscribe(onNext: numbers =>
{
    numbers.Sum().Subscribe(onNext: number => Console.WriteLine(number));
});
```

```
});
```

در این حالت متد Window، برخلاف متد Buffer، توالی numbers را هربار کش نمی‌کند و به این ترتیب می‌توان به مصرف حافظه‌ی کمتری رسید.

کاربردهای دنیای واقعی

در اینجا دو مثال از بکارگیری متد Buffer را جهت پردازش مجموعه‌های عظیمی از اطلاعات و نمایش همزمان آن‌ها در رابط کاربری ملاحظه می‌کنید.

مثال اول: فرض کنید قصد دارید تمام فایل‌های درایو C خود را توسط یک TreeView نمایش دهید. در این حالت نباید رابط کاربری برنامه در حالت هنگ به نظر برسد. همچنین به علت زیاد بودن تعداد فایل‌ها و نمایش همزمان آن‌ها در UI، نباید CPU Usage برنامه تا حدی باشد که در کار سایر برنامه‌ها اختلال ایجاد کند. در این مثال‌ها با استفاده از Rx و متد بافر آن، هربار مثلاً 1000 آیتم را بافر کرده و سپس یکجا در TreeView نمایش می‌دهند. به این ترتیب دو شرط یاد شده محقق می‌شوند.

[The Rx Framework By Example](#)

مثال دوم: خواندن تعداد زیادی رکورد از بانک اطلاعاتی به همراه نمایش همزمان آن‌ها در UI بدون اختلالی در کار سیستم و همچنین هنگ کردن برنامه.

[Using Reactive Extensions for Streaming Data from Database](#)