## Abstract

Here's a small doc aimed towards beginners and intermediates, written in an attempt to clear up some of the confusion around terms like observables, async, and Rx (read as "reactive") programming. It was originally intended to be included in more extensive documentation, but time was short and the scope seemed ever-increasing. Hopefully it will prove helpful for some.

~ Aahz



## The Observer Pattern

At the heart of Reactive Programming is the Observer pattern. Things that can be observed are Observables, and things that can observe those Observables are Observers. Simple, right?

This pattern simply suggests a loose coupling system whereby an Observable object allows for notification events for "subscribers", also known as Observers. Whenever the Observable object changes, this subscription notifies the Observer of the event, allowing the Observer to react in any way it chooses. This is very similar to a mailing list, where users will subscribe to a website and receive emails when the site wishes to notify users of changes and new/recent events. There is a noticeable inversion compared to the normal method of constantly "polling" an object or property to check if it has changed, instead allowing for the object itself to send out a notification event to anyone listening for that specific change. The Observer pattern is also explained somewhat humorously as *The Hollywood Principle*, as in "Don't call us, we'll call you."

The specific notification events are generally classified into 3 categories:

- **OnNext:** This is the most common event, typically happening every time a new value occurs on an Observable property, or every time an Observable command is executed.
- **OnError:** This rarely happens, but can also be subscribed to in most cases, where an error notification event is sent out.
- **OnCompleted:** This is the final event before disposing itself, when an Observable is signaling that there will be no further updates.

Disposing is a necessary part of the subscription process, and deals with freeing up allocations and removing event listeners. Every subscription results in an IDisposable, which is very important when defining when your subscription is no longer necessary and can be disposed.

## UniRx Overview

UniRx is a re-implementation of Microsoft's .NET Reactive extensions in Unity, designed by neuecc (Yoshifumi Kawai) to be more cross-platform and Unity-friendly. Reactive extensions are used on Observables in order to implement logic in a more declarative way. This Rx-style of programming leads to extremely flexible and powerful code, that can be easily defined, more legible, and more responsive.

*"Okay, my eyes are starting to glaze over... Get to the point!"*

Well, using what we know of the The Observer Pattern, we understand that we can easily subscribe to be notified of "change" events. Reactive programming is where you not only define ***that* you want to be notified** of a change, but also define ***how* you want to react** to such notifications. So let's quickly get to an example while that sinks in:

> The following section, while not technically inaccurate, is written specifically to ease an almost complete beginner into important Rx thought processes, and provide a solid foundation to build upon. The underlying complexity and design patterns involved are slightly more complicated. The idea here is to introduce the intent of Rx and build a working knowledge of common ways to utilize its power within uFrame and the general game development environment as quickly as possible.

On a PlayerHUDView.cs

```
public UILabel HealthLabel;  // Assigned in Inspector
public UILabel HealthLabel2; // Assigned in Inspector
public override void Bind() {
    base.Bind()
    this.Player.HealthProperty.Subscribe(health =>
        HealthLabel.text = health.ToString()).DisposeWith(this);
    this.Player.HealthProperty.Subscribe(HealthChanged).DisposeWith(this);

    // Note: These subscriptions are provided for sake of example, recommended way is
    //    with BindProperty or AddBinding, although all of these methods of binding/
    //    subscribing are essentially exactly the same.
    // this.BindProperty(Player.HealthProperty, HealthChanged);
}
// Receives new health values whenever the HealthProperty Subscription is "pushed" a
//   new value
private void HealthChanged(int health) {
    HealthLabel2.text = health.ToString();
}
```

Both of the above subscriptions tie into the same Player.Health property, both of them receive the same new integer value notifications, and both of them update their individual UILabels according to this new value. They do exactly the same thing, just with two separate UILabels in order to display example use of both lambdas and Actions for defining how we want to **react** to this subscription.

## A Little Further into the Rx Concept

Many explanations of reactive programming will also introduce the idea of async streams, which can be a troublesome entry point for beginners. Breaking things down, the concept of asynchronous is simply that things are not happening at a steady time interval, and communication or events may happen intermittently.

Coming back to Rx with the idea of async streams, this is again referring to the fact that instead of polling for property changes every frame, Observable subscriptions will only notify you when a change occurs. This notification may happen in the first second of the subscription, but not again for the next 5 seconds, and then it may not change for entire minutes. Since you are only receiving notifications of changes, these notifications are happening at irregular intervals instead of a steady stream of "pushed" values.

Therefore compared to the normal, steady, synchronous stream of regular game Update events, these subscriptions offer a parallel asynchronous stream of *pushed* values and events. Time for a quick hypothetical example before moving on:

Imagine a Player with a Health property, the standard go-to example. Let's say we subscribe to that, so now we are receiving HealthChanged events. We may receive an initial notification at 0 seconds, that the Player's Health has been initialized to 100. For whatever reason, let's say the player ran head-first into an enemy and lost 10 health at 5 seconds into the game. Because of our subscription, we will now be sent a HealthChanged notification with the new value of 90. Another 7 seconds later, the player hits spikes and loses 5 health, and yet another 12 seconds after that the player somehow manages to trip and fall on his own sword, taking another 30 points of damage. Let's take a second and try to look at that stream of notifications as a collection of values:

| Time | Player.Health |
|---|---|
| **0 s** | 100 |
| **5 s** | 90 |
| **12 s** | 85 |
| **24 s** | 55 |

A player's health is always one specific value at any particular point in time. Although it is not an actual collection, in the Reactive context of a data sequence, Player.Health could almost be thought of as a List<int> however, where our subscription is continuing to Add values to the collection every time a new value is "pushed" to the subscription. So if this were a collection of Player.Health values, the list would contain the values 100, 90, 85, 55, and so on. This frame of mind can be incredibly useful in the next section, where we combine this with the filtering and querying power of LINQ.

## Reactive Rx Extensions

Let's get straight to the point, where we blend everything to produce ultimate flexibility and phenomenal cosmic power! The idea is to combine the Observer pattern with Reactive subscriptions, and add in the familiarity of LINQ queries. If you're not already familiar with it, there are tons of examples and answered questions on LINQ (**L**anguage **IN**tegrated **Q**uery) you can explore online to learn more.

LINQ is mainly used to query collections, and works with Lists, arrays, and generic IEnumerables. The greatest part of UniRx integration is quite possibly the mimicking of LINQ functionality with Reactive extensions, and once you are familiar with LINQ, much of Rx will seem very natural. With the previous example of imagining a Player's observable Health as a collection of values over time, let's use some reactive power:

On a PlayerHUDView.cs

```
public override void Bind() {
    base.Bind();
    // Let's subscribe to be notified ONLY when Health is divisible by 10
    // (using the modulus operator, when dividing by 10 leaves a remainder of 0
    this.Player.HealthProperty.Where(health => health % 10 == 0)
        .Subscribe(HealthDivisibleByTen)
        .DisposeWith;
}
public void HealthDivisibleByTen(int health) {
    Debug.Log(string.Format("Health was divisible by 10: {0}", health));
    // Given the previous Time vs Player.Health table above, this HealthDivisibleByTen
    //    subscription will be notified of the 100 and 90 values, but NOT the 85 or 55
}
```

The above example demonstrates the .Where LINQ filter query being used on our example theoretical "collection" of pushed values, thanks to the magic of UniRx's Reactive extensions. So because of that simple filter query, our subscription is only being notified of values that are divisible by 10. The .Where filter can be used for many things, and simply needs to be provided with a comparison based on values it can receive, where that comparison results in either true or false. There are many such reactive extensions and other powerful features provided by UniRx, and many are explored in the sub-topic: Common Uses and Examples.

### ⚠ Avoiding Rube Goldberg-ian design of seemingly endless propagations

While you inherently have access to Rx anywhere you can see an observable, it is of the utmost importance to limit the reach of your impact to implementing only what an element should logically take care of itself. Defining an element's behaviour can become complex, and while this is not a problem itself, it can easily become a problem when the element begins affecting other elements, parents, or children. Designing a system whereby elements are referencing a parent two or more levels above themselves to precipitate events, can quickly create troublesome headaches when debugging and tracking down any logic problems that may arise. Wherever possible, limit the impact and reach of an element to itself, for sanity's sake.

## Common Uses and Examples

On Views, there are several convenience observables generated for you and available for easy subscription, including:

- UpdateAsObservable, TransformChangedObservable
- PositionAsObservable, LocalPositionAsObservable
- RotationAsObservable, LocalRotationAsObservable
- ScaleAsObservable

Here are a few examples to get you started:

On a View

```
this.UpdateAsObservable().Subscribe(_ =>{ Debug.Log("This is called every frame.")})
    .DisposeWith(this);
this.UpdateAsObservable().Where(_ => Input.GetMouseButtonDown(0))
    .Subscribe(_ =>{ Debug.Log("This is called when the mouse button is down.")})
    .DisposeWith(this);
this.UpdateAsObservable().Where(_ => Input.GetMouseButtonDown(0))
    .Throttle(TimeSpan.FromSeconds(1))
    .Subscribe(_ =>{ Debug.Log("This is called when clicked, but only once per second.")})
    .DisposeWith(this);
```

## Timers, Intervals, and Observable Coroutines

When desiring specific functionality that occurs at a specific time or interval of time, UniRx has implemented several convenient schedulers to which you can create and subscribe.

### Timers

Timers are used almost like alarm clocks.  Basically you set a time in the future when the Timer should finish, and once that time passes, it sends out a single notification to whatever handler you have subscribed to it, and then it disposes itself.  Example usage:

Example: Observable.Timer

```
public void SomeMagicalFunction() {
    Observable.Timer(TimeSpan.FromSeconds(5)).Subscribe(_ => Debug.Log(
        "This happens after 5 seconds."));
    Observable.Timer(TimeSpan.FromSeconds(10)).Subscribe(_ =>
        DoSomethingLater());
}
public void DoSomethingLater() {
    Debug.Log("This happens after 10 seconds.");
}
```

## Intervals

Intervals are used when you need something to happen every N seconds, and *do not automatically dispose themselves*. You must always specify when this observable should be disposed, otherwise it will continue indefinitely. Example usage:

Example: Observable.Interval

```
// Player is standing in fire and taking 10 damage every 1 second
Observable.Interval(TimeSpan.FromSeconds(1))
    .Subscribe(_ => ExecuteCommand(player.TakeDamage, 10))
    .DisposeWhenChanged(player.IsStandingInFire);

// Somewhat contrived example demonstrating both Timer and Interval
// Deals initial 10 damage, and then further Damage over Time for 4 ticks
ExecuteCommand(player.TakeDamage, 10);
var DoTDamage = Observable.Interval(TimeSpan.FromSeconds(1))
    .Subscribe(_ => ExecuteCommand(player.TakeDamage, 2));
Observable.Timer(TimeSpan.FromSeconds(5)).Subscribe(_ =>
    DoTDamage.Dispose());

// Time-based functionality without coroutines in Controller to trigger waves
if(state is Wave) {
    // Wait for allotted amount of time
    Observable.Interval(TimeSpan.FromSeconds(wavesFPSGame.SpawnWaitSeconds))
        .Where(_ => wavesFPSGame.WavesState is Wave)
        .Take(wavesFPSGame.KillsToNextWave)
        .Subscribe(_ => SpawnEnemy());
}
```

## Observable Coroutines

Normally with Unity, to do any kind of asynchronous logic that happens outside of the game's Update loop, you would typically use IEnumerators and Coroutines. Unity Coroutines can be unwieldy to use and have several problems however, chiefly the inability to use return values and lack of error handling. Let's start with a short snippet from the RTS example:

GameCoreController.cs

```
// A command to trigger start of the game
public override void BeginGame(GameCoreViewModel gameCore) {
    gameCore.State = CoreGameState.GeneratingMap;

    // Turn a GenerateMap IEnumerator, located in the CoreMapController, into
    //    an Observable
    Observable.FromCoroutine(() => CoreMapController.GenerateMap(gameCore.Map)
        .Subscribe(_ => /* unused OnNext */ }, DoneGenerating);
    // There are several signatures for FromCoroutine, here we're using a
    //   subscription that gives Subscribe(OnNext, OnComplete)
}

private void DoneGenerating() {
    // Observable.FromCoroutine(InitializePlayers)
    //  .Subscribe(_ => { /* unused OnNext */ }, () =>
    //        GameCore.State = CoreGameState.Playing);
    // Below is an alternate method of doing the same as the above ^
    InitializePlayers.AsObservable()
        .Subscribe(_ => { /* unused OnNext */ }, () =>
            GameCore.State = CoreGameState.Playing);
}
private IEnumerator InitializePlayers() {
    // Do initializing players logic based on menu settings, AI player count, etc.
}
```

The above logic is all triggered from a single BeginGame command, which sets off a series of chained observable IEnumerators.

1. The **BeginGame** command immediately sets the game's state to *GeneratingMap*, and then creates an Observable sequence using the IEnumerator **GenerateMap**, located in the CoreMapController.
2. Because this is an observable, we subscribe to it and tell it to let us know when it is *complete*, at which point it will call **DoneGenerating**.
3. **DoneGenerating** is a simple function separated out for readability. It creates another Observable sequence using the IEnumerator **InitializePlayers**, which would handle creating AI players, spawning units, etc.
4. When this Observable is complete, it simply changes the game's state to *Playing*, at which point the game actually begins.

Now that we have some familiarity with how to implement similar functionality as Unity's StartCoroutine when using IEnumerators, let's briefly look at one of the ways to utilize return values from these Observable IEnumerators.

Example: Observable.FromCoroutineValue<int>

```
public override void Awake() {
    base.Awake();

    // Simply output the IEnumerators yielded values to the Debug console
    Observable.FromCoroutineValue<int>(DoStuff)
        .Subscribe(i => Debug.Log(i), () => Debug.Log("Done."));
}
private IEnumerator<int> DoStuff() {
    yield return 1;
    yield return 2;
    yield return 42;
    yield return 7;
}
/* Results in Debug.Log messages to the console, reading:
1
2
42
7
Done.
*/
```

More Examples can be found on the UniRx GitHub.