



Unite '17
Seoul





차세대 프로그래밍 패러다임을
유니티에서 사용해보자~!

Reactive Programming in Unity



박민근 = 알콜코더

유니티 마스터





알콜코더 박민근 (@agebreak)

게임 클라이언트 프로그래머

14~16 ShiftUp - <데스티니 차일드>

13~14 NHN NEXT 게임 전공 교수

12~13 Cookie Soft - <에스퍼>

10~12 네오위즈 게임즈 - <야구의 신>

06~10 NTL-inc/반다이남코 <드래곤볼 온라인>

05~06 엔씨소프트 - <PlayNC 빌링 서버>

03~05 액토즈소프트 - <라제스카>

유니티 마스터

‘신입 게임 개발자의 서울 상경기’

(<http://agebreak.blog.me>)

‘초중급 게임 개발자 스터디 (데브루키)’

(cafe.naver.com/devrookie)



들어가기에 앞서

이 발표 자료는 아래의 PT를 기반으로 번역, 제작 되었습니다

未来のプログラミング技術をUnityで
～UniRx～

@toRisouP

2015/03/20



Unite '17
Seoul

<http://www.slideshare.net/torisoup/unity-unirx>



슬라이드 중에 등장하는 샘플

<http://torisoup.net/unirx-examples/>

Reactive Programming에 대해서는 이 PT를 참고



SharedNothing Excel flatMap libevent Verilog
map Reactor Scalable Observable
Pig callback lambda
Actor
DataFlow libev Declarative Concurrent Monad
EventDriven React Responsive
zip Akka IOCP Amdahl
Functional Reactive Programming
Asynchronous Resilient Nonblocking
Transparency HigherOrder libuv ELM
epoll Immutability Backpressure
kqueue

<http://www.slideshare.net/jongwookkim/ndc14-rx-functional-reactive-programming>



그런데...





**【마우스의 더블 클릭 판정】
구현 하실 수 있나요?**


마우스의 더블 클릭 판정

어떻게 구현할까?

- 최후에 클릭 했을 때부터 일정시간 이내라면 더블 클릭?
- 클릭 횟수 변수와 타이머 변수를 필드에 정의?
- Update() 내에 판정 처리를 구현?




귀찮다



이것을 UniRx를 사용하면
단지 몇줄로 끝낼 수 있습니다



```
var clickStream = UpdateAsObservable()  
    .Where(_ => Input.GetMouseButtonDown(0));  
  
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```



```
var clickStream = UpdateAsObservable()  
    .Where(_ => Input.GetMouseButtonDown(0));  
  
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```

단지 이것뿐!





이번 발표의 목표

실제의 사용예를 통해서

UniRx의 대단함과 편리함을

전하고 싶다



대상

LINQ는 사용할 수 있는 레벨

프로그래밍 초보자에게는 조금 어려울지도

1. UniRx란?
2. UniRx가 무엇이 편리한가?
3. 스트림을 사용하는 메리트와 예
4. 자주 사용하는 오퍼레이터 설명
5. Unity에서의 실용 사례 4가지
6. 정리



UniRx란?

- Reactive Extensions for Unity
- MIT라이선스 공개
- AssetStore, 또는 github에서 다운로드 가능 (무료)

UniRx - Reactive Extensions for Unity

By [neuecc](#), Free

Reactive Extensions for Unity that allows LINQ to asynchronous and LINQ to multithreading and LINQ to events and more. Welcome to the reactive game architecture! This library is free and opensource on GitHub. More info, supports and sourcecodes, see <https://github.com/neuecc/UniRx>. This library supports PC/Mac/iOS/Android/WP8/Windows Store App/etc.

Unity Forums support thread is here, ask me any questions - <http://forum.unity3d.com/threads/248535-UniRx-Reactive-Extensions-for-Unity>

[Open Asset Store](#)

Requires Unity 4.5 or higher. Update now.

UniRx
Reactive Extensions for Unity



Unite '17
Seoul

unity

Functional Reactive Programming을 C#에서 구현하기 위한 라이브러리

- LINQ to Events
- 시작은 Microsoft Research가 개발 (.NET)

최근 들어 유행의 조짐이 오고 있다

- 다양한 언어로 이식 되고 있음

RxJS, RxJava, ReactiveCocoa, RxPy, RxLua, UniRx...

- 어떤 언어라도 Rx의 개념은 같다



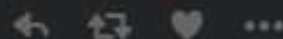
이제 악마랑 놀자!

DESTINY CHILD
For KAKAO

데스티니 차일드에서도 **LuaRx**를 일부 도입



알콜코더 @agebreak 1h
간만의 트윗과 함께.. 정말 간만의
외부 발표.. 이번 유나이티2017에서
UniRx 소개를 발표하게 되었습니
다.. 근데... PT에 내용을 눌러담다보
니 거의 200페이지에 꽉찬내용... 물론
실제 발표때는 시간관계상 중요치
않는 부분들은 패스할듯..



데브캣 나크
@DungeonKim

Replying to @agebreak

알콜코더님 PT로 입문해서 UniRx
잘 쓰고 있습니다. 이 PT도 기대되
네요.

10:47pm · 14 May 2017 · Twitter for Android

.NET용 Rx은 유니티의 Mono에서는 사용 불가

- Mono의 .NET 버전 문제
- 그리고 .NET용 Rx는 무겁고 크다

UniRx

- Unity(C#)에서 사용할 수 있게 만든 가볍고 빠른 유니티 전용 Rx
- Unity 전용으로 사용할 수 있는 Rx 스트림들을 제공
- 일본에서 개발



UniRx는 무엇이 편리한가?



UniRx를 사용 하면

시간의 취급이

굉장히 간단해 집니다

시간이 결정하는 처리의 예

이벤트의 기다림

- 마우스 클릭이나 버튼의 입력 타이밍에 무언가를 처리 한다

비동기 처리


- 다른 스레드에서 통신을 하거나, 데이터를 로드할 때

시간 측정이 판정에 필요한 처리


- 홀드, 더블클릭의 판정

시간 변화하는 값의 감시

- False->True가 되는 순간에 1회만 처리하고 싶을 때



이런 처리를 Rx를 사용하면
상당히 간결하게 작성 가능합니다



우선은 Rx의 기본적인
개념을 실제의 코드를 보면서
설명 하겠습니다

버튼이 클릭되면 화면에 표시

Button을 클릭할 때에 Text에 “Clicked”라고 표시해 보자



클릭되면 화면에 표시하는 스크립트

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine;

namespace UnityEngine
{
    public class ButtonClick : MonoBehaviour
    {
        [SerializeField] private Button button;
        [SerializeField] private Text text;

        private void Start()
        {
            button.onClick
                .AsObservable()
                .Subscribe(_ =>
                {
                    text.text = "Clicked";
                });
        }
    }
}
```

<- 메인 처리

클릭되면 화면에 표시하는 스크립트

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;

namespace UnityEngine
{
    public class ButtonClick : MonoBehaviour
    {
        [SerializeField] private Button button;
        [SerializeField] private Text text;

        private void Start()
        {
            button.onClick
                .AsObservable()
                .Subscribe(_ =>
                {
                    text.text = "Clicked";
                });
        }
    }
}
```

← Unity가 제공하는 클릭 이벤트

클릭되면 화면에 표시하는 스크립트

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.EventSystems;

namespace UnityEngine
{
    public class ButtonClick : MonoBehaviour
    {
        [SerializeField] private Button button;
        [SerializeField] private Text text;

        private void Start()
        {
            button.onClick
                .AsObservable()
                .Subscribe(_ =>
                {
                    text.text = "Clicked";
                });
        }
    }
}
```

<- 이벤트를 스트림으로 변경

클릭되면 화면에 표시하는 스크립트

```
using UnityEngine;
using UnityEngine.UI;
using UnityEngine;

namespace UniRxSamples
{
    public class ButtonClick : MonoBehaviour
    {
        [SerializeField] private Button button;
        [SerializeField] private Text text;

        private void Start()
        {
            button.onClick
                .AsObservable()
                .Subscribe(_ =>
                {
                    text.text = "Clicked";
                });
        }
    }
}
```

<- 스트림의 구독
(최종적으로 무엇을 할것인가를 작성)

[이벤트가 흐르는 파이프] 같은 이미지

- 어렵게 말하자면, [타임라인에 배열되어 있는 이벤트의 시퀀스]
- 분기 되거나 합쳐지는게 가능하다

코드 안에서는 IObservable<T> 로 취급된다

- LINQ에서 IEnumerable<T>에 해당



스트림에 흐르는 이벤트 <메시지>

메시지는 3종류가 있다

- **OnNext**
 - 일반적으로 사용되는 메시지
 - 보통은 이것을 사용한다
- **OnError**
 - 에러 발생시에 예외를 통지하는 메시지
- **OnCompleted**
 - 스트림이 완료되었음을 통지하는 메시지

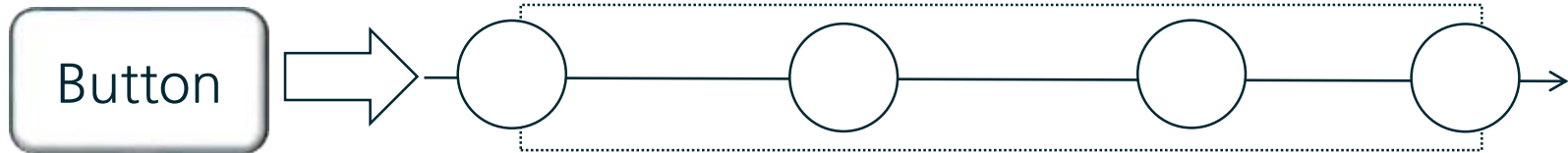
스트림과 버튼 클릭

버튼은

[클릭 때에 이벤트를 스트림에 보낸다]

라고 생각하는 것이 가능하다

버튼이 클릭된 타이밍에
스트림에 메시지를 보내 넣는다 (OnNext)



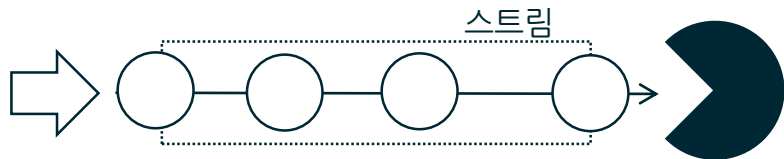
Subscribe (스트림의 구독)

스트림의 말단에서 메시지가 올때 무엇을 할 것인지를 정의 한다

스트림은 Subscribe 된 순간에 생성 된다

- 기본적으로 Subscribe하지 않는 한 스트림은 동작하지 않는다
- Subscribe 타이밍에 의해서 결과가 바뀔 가능성이 있다

OnError, OnComplete가 오면 Subscribe는 종료 된다



Subscribe
스트림을 구독해서
메시지가 올 때에 처리를 한다

Subscribe와 메시지

Subscribe는 오버로드로 여러 개 정의되어 있어서, 용도에 따라 사용하는게 좋다

- OnNext 만
- OnNext & OnCompleted
- OnNext & OnError & OnCompleted

```
ObservableWWW.Get("http://unity-chan.com/")
    .Subscribe(result =>
    {
        //OnNext
        Debug.Log(result);
    }, ex =>
    {
        //OnError
        Debug.LogError(ex);
    }, () =>
    {
        //OnCompleted
        Debug.Log("Done!");
    });
```

전체 흐름

```
using UnityEngine;
using UniRx;
using UnityEngine.UI;

namespace UniRxSamples
{
    public class ButtonClick : MonoBehaviour
    {
        [SerializeField] private Button button;
        [SerializeField] private Text text;
```

```
        private void Start()
        {
            button.onClick
                .AsObservable()
                .Subscribe(_ =>
                {
                    text.text = "Clicked";
                });
        }
    }
}
```

<-

버튼 클릭이벤트를
스트림으로 변경해서
메시지가 도착할 때에
텍스트에 "Clicked"를 표시한다



Subscribe의 타이밍

```
using UnityEngine;
using UniRx;
using UnityEngine.UI;

namespace UniRxSamples
{
    public class ButtonClick : MonoBehaviour
    {
        [SerializeField] private Button button;
        [SerializeField] private Text text;

        private void Start()
        {
            button.onClick
                .AsObservable()
                .Subscribe(_ =>
                {
                    text.text = "Clicked";
                });
        }
    }
}
```


Awake()/Start()에서 Subscribe 해야 한다
Update()에 쓰면 무수한 스트림이 생성 된다.

UniRx에는,


uGUI용의 Observable과 Subscribe가 준비되어 있다

```
button
    .OnClickAsObservable()
    .SubscribeToText(text, _ => "clicked");
```

앞의 내용을 이걸로 간략화 할 수 있다



<스트림>이라는 개념의 메리트



이벤트의
투영, 필터링, 합성
등이 가능하다

예제

Button이 3회 눌리면 Text에 표시한다

버튼이 클릭된 횟수를 카운트 한다?

- 카운트용의 변수를 필드에 정의 한다?

Button

clicked

Button이 3회 눌리면 Text에 표시한다

Buffer(3)을 추가만 하면 됨

- 굳이 필드 변수 추가 필요 없음
- 혹은 Skip(2)로도 똑같은 동작을 한다
 - 여기에서는 이해를 돕기위해 Buffer를 사용했지만, n회후에 동작하는 경우에는 Skip 쪽이 적절하다

```
button
    .OnClickAsObservable()
    .Buffer(3)
    .SubscribeToText(text, _ => "clicked");
```

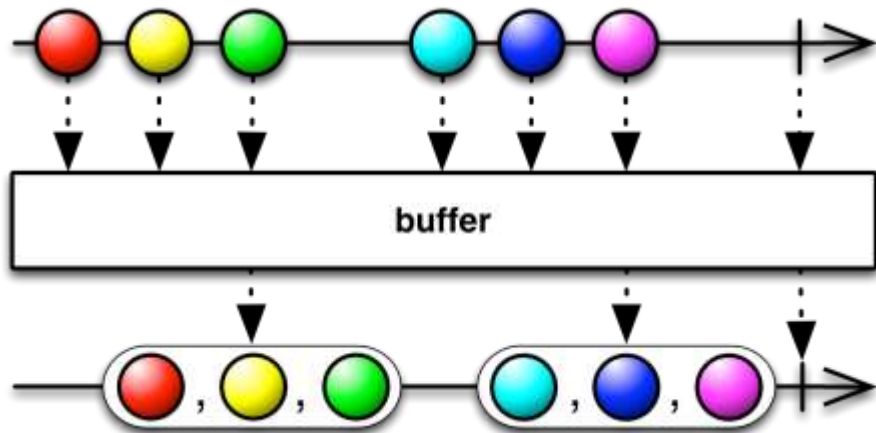


메시지를 모아서 특정 타이밍에 보낸다

- 방출 조건은 여러가지 지정이 가능하다

*n개 모아서 보내기

*다른 스트림에 메시지가 흐르면, 보내기

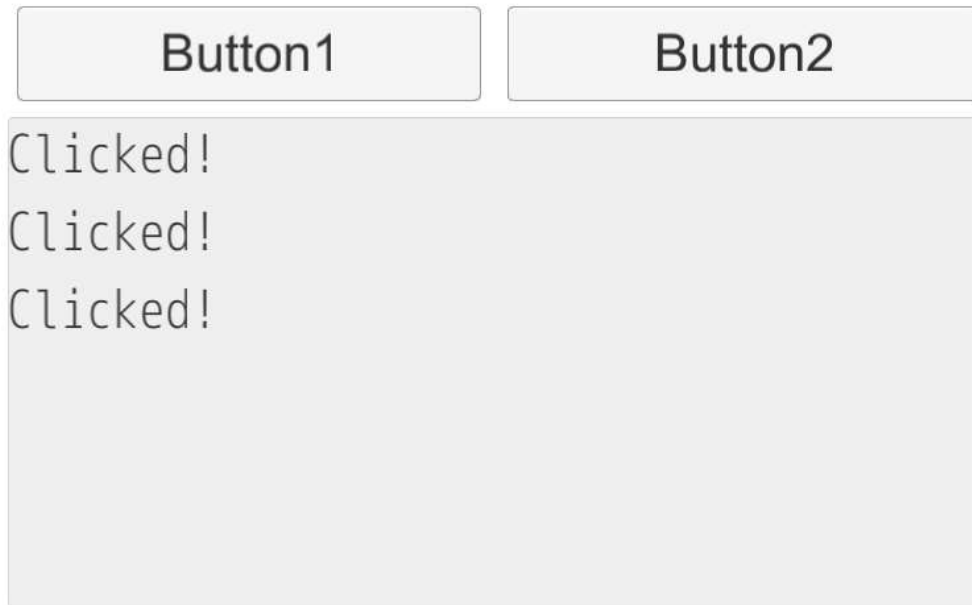


예제 2

Button이 둘다 눌리면 Text에 표시 한다

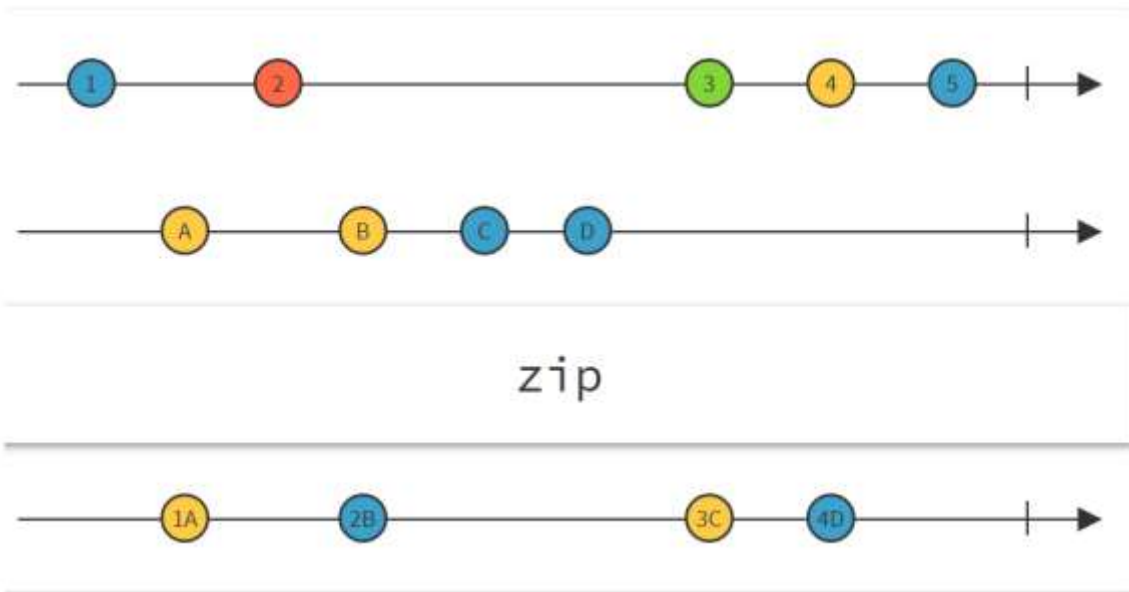
양쪽이 교차로 1회씩 눌릴 때 Text에 표시한다

- 연타하더라도 [1회 눌림]으로 판정한다



여러 개의 스트림의 메시지가 완전히 모일때까지 기다림

- 메시지가 모일때에 하나의 이벤트로 취급해서 보낸다
- 합쳐진 메시지는 임의로 가공해서 출력이 가능하다



Button이 둘다 눌리면 Text에 표시 한다

```
button1.OnClickAsObservable()  
    .Zip(button2.OnClickAsObservable(), (b1, b2) => "Clicked!")  
    .First()  
    .Repeat()  
    .SubscribeToText(text, x => text.text + x + "\n");
```


Button이 둘다 눌리면 Text에 표시 한다

```
button1.OnClickAsObservable()  
    .Zip(button2.OnClickAsObservable(), (b1, b2) => "Clicked!")  
    .First()  
    .Repeat()  
    .SubscribeToText(text, x => text.text + x + "\n");
```


< 1번 동작한 후에 Zip내의 버퍼를 클리어 한다
(뒤에 설명합니다)



Rx을 사용하지 않는 종래의 방법에서-
이벤트를 받은 후에
어떻게 할 것인가를 작성 하였다



Rx에서는
이벤트를 받기 전에
무엇을 하고 싶다는 작성한다



[스트림을 가공해서
자신이 받고 싶은 이벤트만
통지 받으면 좋잖아!]



정리하자면,

Rx는

1. 스트림을 준비해서
2. 스트림을 오퍼레이터로 가공 해서
3. 최후에 Subscribe 한다

라는 개념으로 사용 된다



오퍼레이터

스트림을 조작하는 메소드

스트림에 조작을 가하는 함수

무진장 많음

Select, Where, Skip, SkipUntil, SkipWhile, Take, TakeUntil, TakeWhile, Throttle, Zip, Merge, CombineLatest, Distinct, DistinctUntilChanged, Delay, DelayFrame, First, FirstOrDefault, Last, LastOrDefault, StartWith, Concat, Buffer, Cast, Catch, CatchIgnore, ObserveOn, Do, Sample, Scan, Single, SingleOrDefault, Retry, Repeat, Time, TimeStamp, TimeInterval...



자주 사용 하는 오퍼레이터 소개

조건을 만족하는 메시지만 통과 시키는 오퍼레이터

- 다른 언어에서는 [filter]라고도 한다

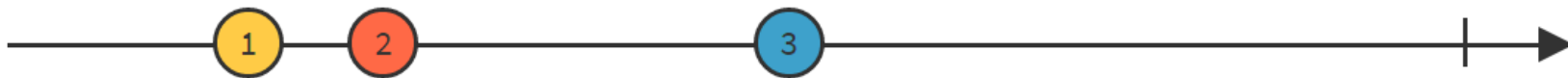


```
filter(x => x > 10)
```

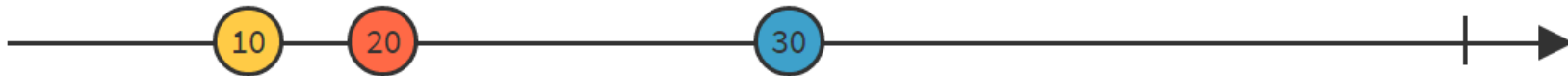


요소의 값을 변경한다

- 다른 언어에서는 [map]이라고 한다

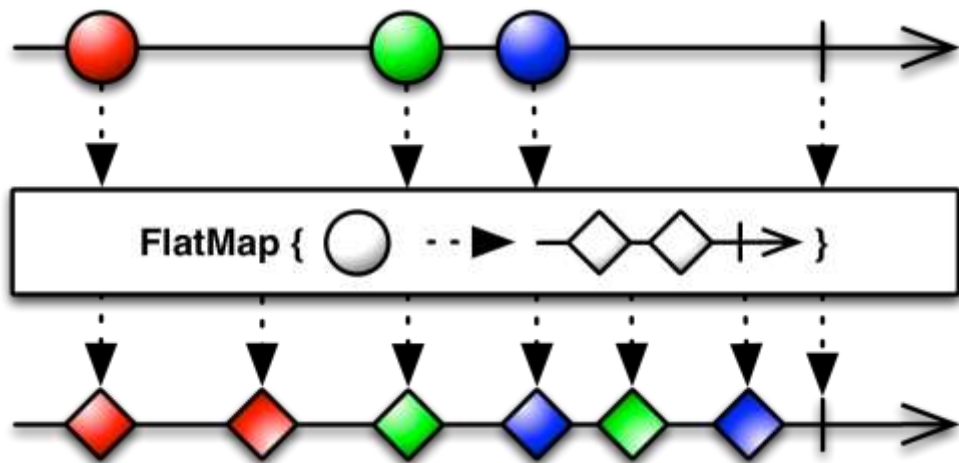


`map(x => 10 * x)`



새로운 스트림을 생성하고, 그 스트림이 흐르는 메시지를 본래의 스트림의 메시지로 취급

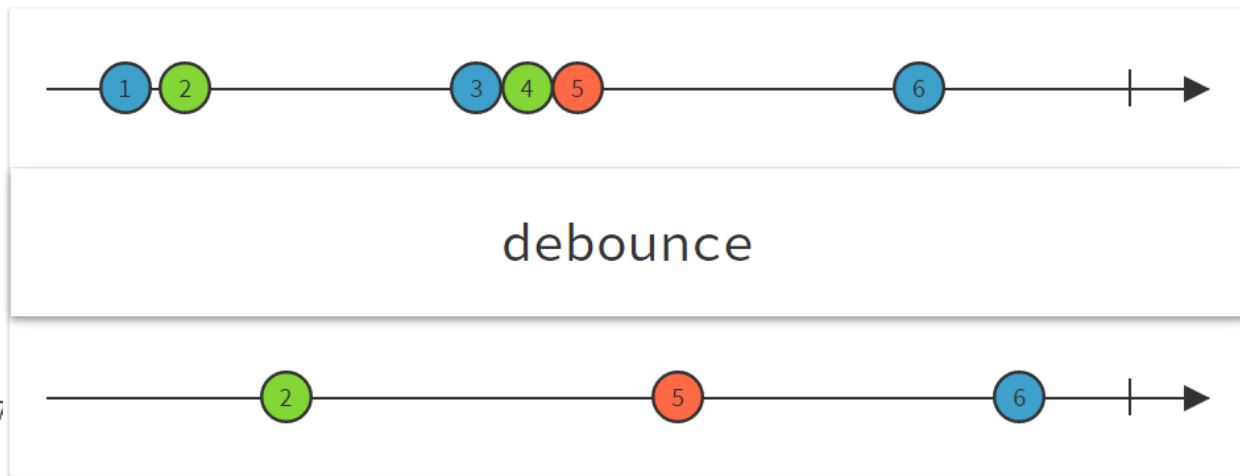
- 스트림을 다른 스트림으로 교체하는 이미지 (정밀히 말하면 다름)
- 다른 언어에서는 [flatMap]이라고도 한다



Throttle/ThrottleFrame

도착한 때에 최후의 메시지를 보낸다

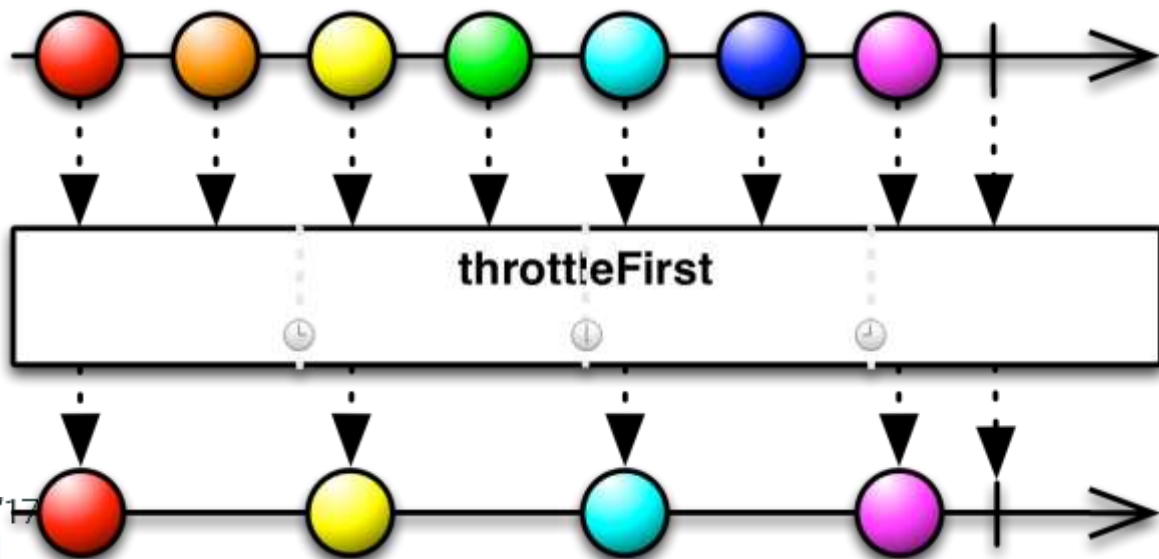
- 메시지가 집중해서 들어 올때에 마지막 이외를 무시 한다
- 다른 언어에서는 [debounce]라고도 한다
- 자주 사용됨



ThrottleFirst/ThrottleFirstFrame

최초에 메시지가 올때부터 일정 시간 무시 한다

- 하나의 메시지가 온때부터 잠시 메시지를 무시 한다
- 대용량으로 들어오는 데이터의 첫번째만 사용하고 싶을 때 유효



Delay/DelayFrame

메시지의 전달을 연기 한다



delay



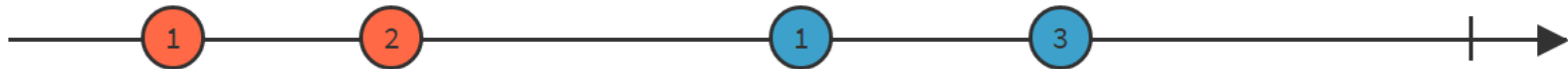
DistinctUntilChanged

메시지가 변화한 순간에만 통지한다

- 같은 값이 연속되는 경우에는 무시한다

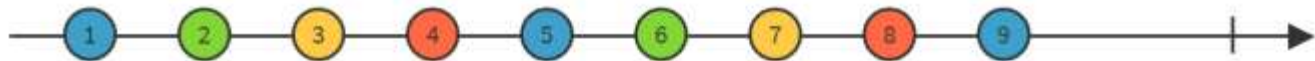


`distinctUntilChanged`



지정한 스트림에 메시지가 올때까지 메시지를 Skip 한다

- 같은 값이 연속되는 경우에는 무시한다

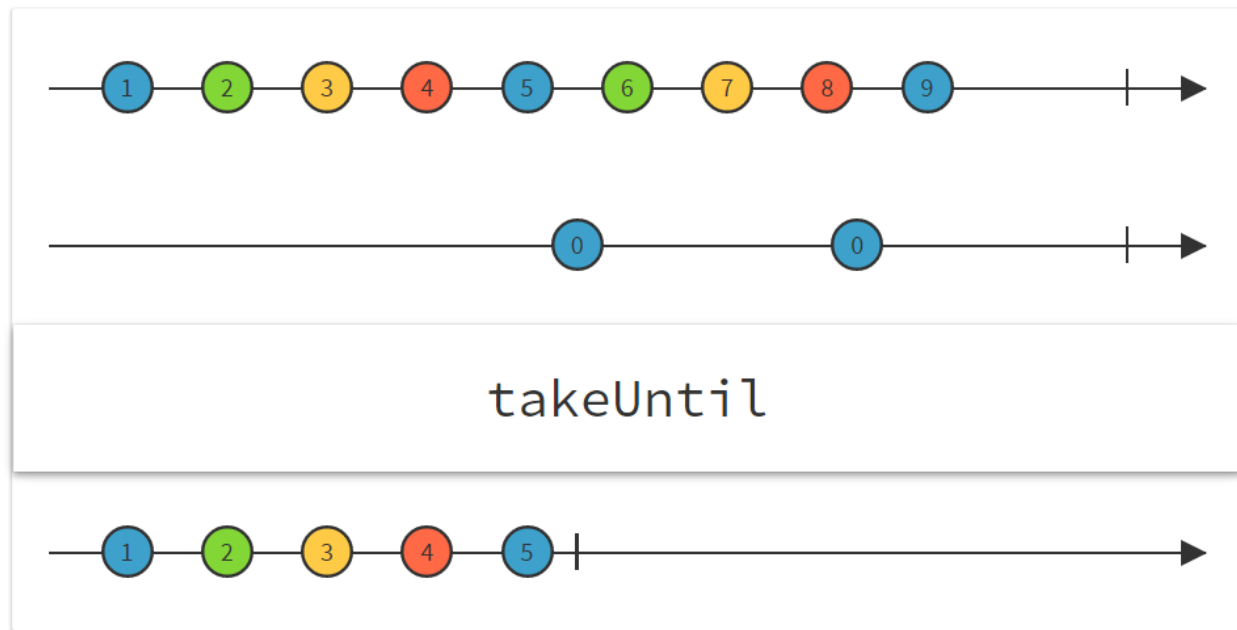


skipUntil



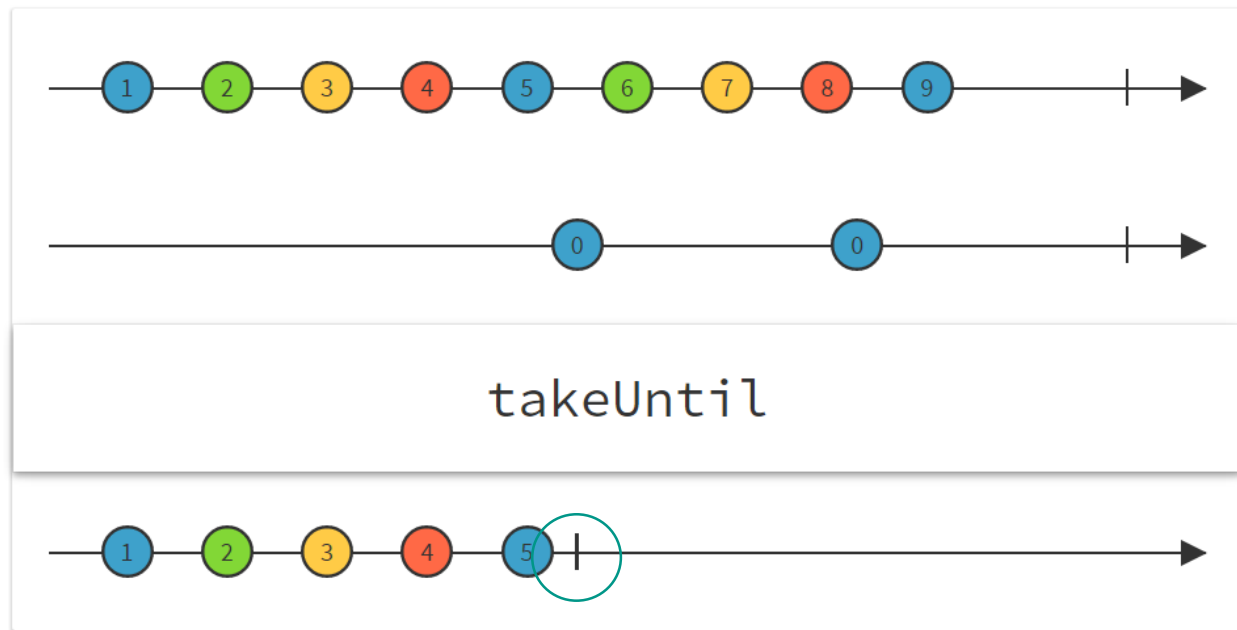
TakeUntil

지정한 스트림에 메시지가 오면, 자신의 스트림에 OnCompleted를 보내서
종료 한다



TakeUntil

지정한 스트림에 메시지가 오면, 자신의 스트림에 OnCompleted를 보내서
종료 한다



Repeat

스트림이 OnCompleted로 종료될 때에 다시 한번 Subscribe를 한다

SkipUntil + TakeUntil + Repeat

자주 사용되는 조합

- 이벤트 A가 올때부터 이벤트 B가 올때까지 처리를 하고 싶을때 사용

SkipUntil + TakeUntil + Repeat의 예제

예) 드래그로 오브젝트를 회전 시키기

- MouseDown이 올 때부터 Mouse Up이 올때까지 처리할 때

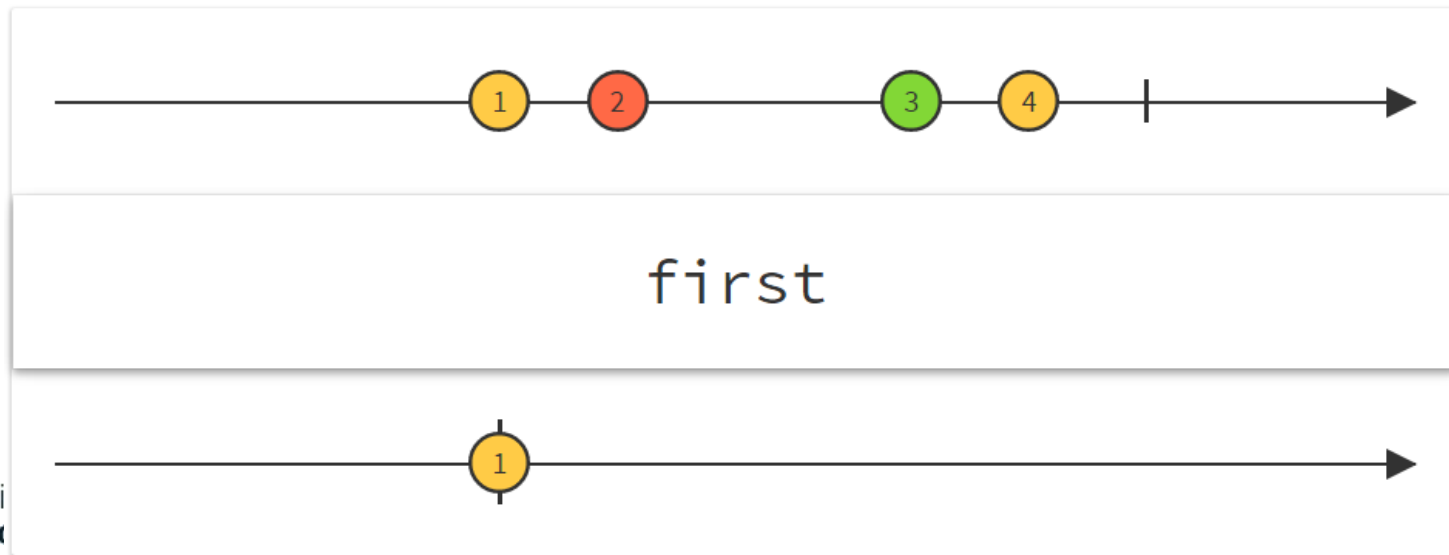
```
//OnMouseDownとOnMouseUpの組み合わせでドラッグ中のみ処理をする
//(<OnMouseDownを使えばいいじゃんって野暮なツツコミは無して...)

UpdateAsObservable() //Update()のタイミングを通知するObservable
.SkipUntil(OnMouseDownAsObservable()) //マウスがクリックされるまでストリームを無視
.Select(_ => //マウスの移動量をストリームに流す
    new Vector2(Input.GetAxis("Mouse X"),
        Input.GetAxis("Mouse Y")))
.TakeUntil(OnMouseUpAsObservable()) //マウスが離されるまで
.Repeat() //TakeUntilでストリームが終了するので再Subscribe
.Subscribe(move =>
{
    //オブジェクトをドラッグするとそのオブジェクトを回転させる
    transform.rotation =
        Quaternion.AngleAxis(move.y*_rotationSpeed*Time.deltaTime, Vector3.right)*
        Quaternion.AngleAxis(-move.x*_rotationSpeed*Time.deltaTime, Vector3.up)*
        transform.rotation;
});
```



스트림에 최초로 받은 메시지만 보낸다

- onNext 직후에 onComplete도 보낸다



앞의 Zip 예제에서 First + Repeat를 사용한 의도

```
button1.OnClickAsObservable()  
    .Zip(button2.OnClickAsObservable(), (b1, b2) => "Clicked!")  
    .First()  
    .Repeat()  
    .SubscribeToText(text, x => text.text + x + "\n");
```

First + Repeat로 1회 동작할때마다 스트림을 다시 만든다
(Zip내의 메시지큐를 리셋하기 위해)



여기까지가 기초

기초 설명만으로 슬라이드 70장 돌파

이제부터 실제 사례를 소개



실제 사용예 5가지

1. 더블 클릭 판정
2. 값의 변화를 감시하기
3. 값의 변화를 가다듬기
4. WWW를 사용하기 쉽게 하기
5. 기존 라이브러리를 스트림으로 변환하기

1. 더블 클릭 판정

더블 클릭 감지 코드

```
var clickStream = UpdateAsObservable()  
    .Where(_ => Input.GetMouseButtonDown(0));  
  
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```

더블 클릭 감지 코드

클릭 스트림을 정의 (# 생성)

```
var clickStream = UpdateAsObservable()  
    .Where(_ => Input.GetMouseButtonDown(0));
```

```
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```

더블 클릭 감지 코드

현재는 UniRx.Trigger를 Using에 추가하고,
this.UpdateAsObservable()로 호출 합니다

```
var clickStream = UpdateAsObservable()  
    .Where(_ => Input.GetMouseButtonDown(0));  
  
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```

클릭 스트림

```
var clickStream = UpdateAsObservable()  
    .Where(_ => Input.GetMouseButtonDown(0));
```

UpdateAsObservable()

Update의 타이밍을 통지



Where()

클릭이 된 프레임만 통과



clickStream

클릭 이벤트의 스트림

자세히보면 2개의 스트림이 있다

```
var clickStream = UpdateAsObservable()  
    .Where(_ => Input.GetMouseButtonDown(0));
```


```
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```

자세히 보면 2개의 스트림이 있다

```
var clickStream = UpdateAsObservable()  
    .Where(_ => Input.GetMouseButtonDown(0));  
  
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```

클릭 스트림을 멈춰둔다

개방조건은 [최후에 클릭된 이후 200밀리초 경과할 때]



```
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```

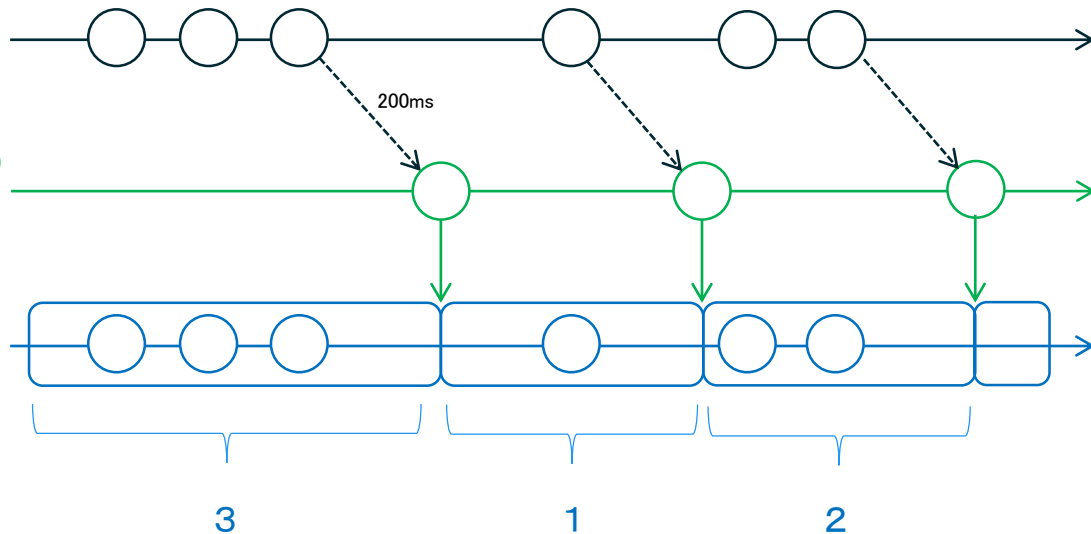

클릭 스트림

```
clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(200)))  
    .Where(x => x.Count >= 2)  
    .SubscribeToText(_text, x =>  
        string.Format("DoubleClick detected!\n Count:{0}", x.Count));
```

clickStream
마우스 클릭의 스트림

Throttle(200ms)
200ms간 이벤트가
발생하지 않을때 통지

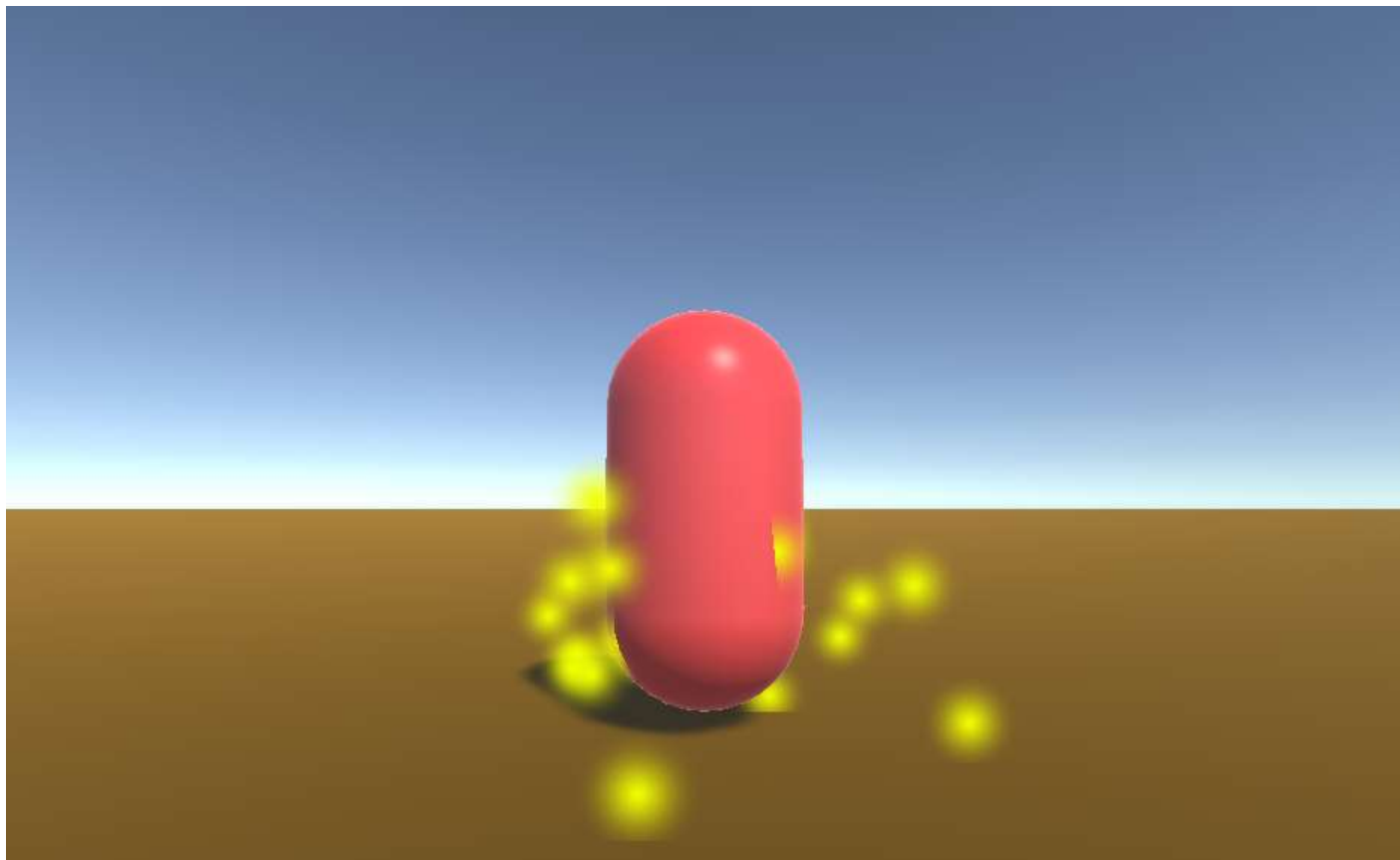
Buffer
이벤트를 모은다
종료조건은 Throttle
이벤트가 올때까지





2. 값의 변화를 감시하기

플레이어가 지면에 떨어지는 순간에 이펙트를 발생



지면에 떨어진 순간의 감지 방법

1. CharacterController.isGrounded를 매프레임 체크
2. 현재 프레임의 값을 필드 변수에 저장
3. 매프레임에 False->True로 변화할 때에 이펙트를 재생한다

지면에 떨어진 순간의 감지 방법

```
public class OnGroundedWithoutUniRx : MonoBehaviour
{
    CharacterController _characterController ;
    ParticleSystem _particleSystem ;
    private bool oldFlag;

    private void Start()
    {
        _characterController = GetComponent<CharacterController>();
        _particleSystem = GetComponentInChildren<ParticleSystem>();

        oldFlag = _characterController.isGrounded;
    }

    private void Update()
    {
        var currentFlag = _characterController.isGrounded;
        if (currentFlag && !oldFlag)
        {
            _particleSystem.Play();
        }
        oldFlag = currentFlag;
    }
}
```



U
S



지면에 떨어진 순간의 감지 방법

```
public class OnGroundedWithoutUniRx : MonoBehaviour
{
    CharacterController _characterController ;
    ParticleSystem _particleSystem ;
    private bool oldFlag;    <- 이전값 보존용만을 위한 필드 변수!!

    private void Start()
    {
        _characterController = GetComponent<CharacterController>();
        _particleSystem = GetComponentInChildren<ParticleSystem>();

        oldFlag = _characterController.isGrounded;
    }

    private void Update()
    {
        var currentFlag = _characterController.isGrounded;
        if (currentFlag && !oldFlag)
        {
            _particleSystem.Play();
        }
        oldFlag = currentFlag;    <- 혹 보면 뭘하는건지 판단하기 어렵다
    }
}
```



U
S



UniRx로 지면 도착 순간을 감지해 보자

```
public class OnGroundedScript : ObservableMonoBehaviour
{
    public override void Start()
    {
        var characterController = GetComponent<CharacterController>();
        var particleSystem = GetComponentInChildren<ParticleSystem>();

        UpdateAsObservable()
            .Select(_ => characterController.isGrounded)
            .DistinctUntilChanged()
            .Where(x => x)
            .Subscribe(_ => particleSystem.Play());
    }
}
```

UniRx로 지면 도착 순간을 감지해 보자

```
public class OnGroundedScript : ObservableMonoBehaviour
{
    public override void Start()
    {
        var characterController = GetComponent<CharacterController>();
        var particleSystem = GetComponentInChildren<ParticleSystem>();

        UpdateAsObservable()
            .Select(_ => characterController.isGrounded)
            .DistinctUntilChanged()
            .Where(x => x)
            .Subscribe(_ => particleSystem.Play());
    }
}
```

이것뿐!
필드 변수 따위 필요 없다!

도작 판정의 이미지 도식

```
UpdateAsObservable()  
    .Select(_ => characterController.isGrounded)  
    .DistinctUntilChanged()  
    .Where(x => x)  
    .Subscribe(_ => particleSystem.Play());
```

UpdateAsObservable()
Update의 타이밍을 통지



Select()
IsGrounded 값으로 교체



DistinctUntilChanged()
값에 변화가 있는 순간만 통과



Where()
값이 True인 경우에만 통과



Subscribe()
isGrounded가 False->True가 된 순간에 통지 된다

ObserveEveryValueChanged

매 프레임 값의 변화를 감시한다면 [ObserveEveryValueChanged]의 쪽이
심플하다

```
// キャラクタが着地した瞬間の検知
```

```
characterController
```

```
.ObserveEveryValueChanged(x => x.isGrounded)
```

```
.Where(x => x)
```

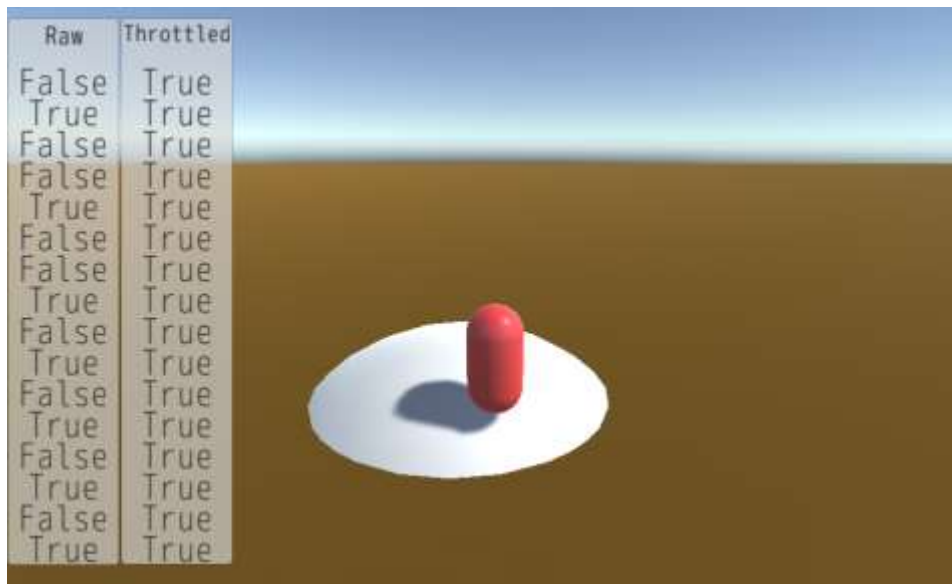
```
.Subscribe(_ => Debug.Log("OnGrounded!"));
```

3. 값의 변화를 다듬기

isGrounded의 변화를 다듬기

isGrounded의 정밀도의 개선

- 곡면을 이동하게 되면 True/False가 격렬하게 변환하다
- 이 값의 변화를 UniRx로 정제해 보자



isGrounded의 변화를 Throttle로 무시한다

- DistinctUntilChanged와 같이 쓰면 OK

UniRx로 isGrounded의 변화를 정돈하

기

```
UpdateAsObservable()
```

```
.Select(_ => playerCharacterController.isGrounded)
```

```
.DistinctUntilChanged()
```

```
.ThrottleFrame(5)
```

```
.Subscribe(x => throttledIsGrounded = x);
```



도식

```
UpdateAsObservable()  
    .Select(_ => playerCharacterController.isGrounded)  
    .DistinctUntilChanged()  
    .ThrottleFrame(5)  
    .Subscribe(x => throttledIsGrounded = x);
```

UpdateAsObservable()
Update의 타이밍을 통지



Select()
IsGrounded의 값으로 교체



DistinctUntilChanged()
값에 변화가 있는 순간만 통과



ThrottleFrame(5)
값이 5프레임간
오지 않는다면 최후의 값을 보냄



Subscribe()
isGrounded가 6프레임 이상
안정된 때에 최후의 값을 통지 한다



4. WWW를 사용하기 쉽게

Unity가 준비한 HTTP 통신용 모듈

- 코루틴으로 사용할 필요가 있다
- 그래서 사용 편의성이 좋지는 않다

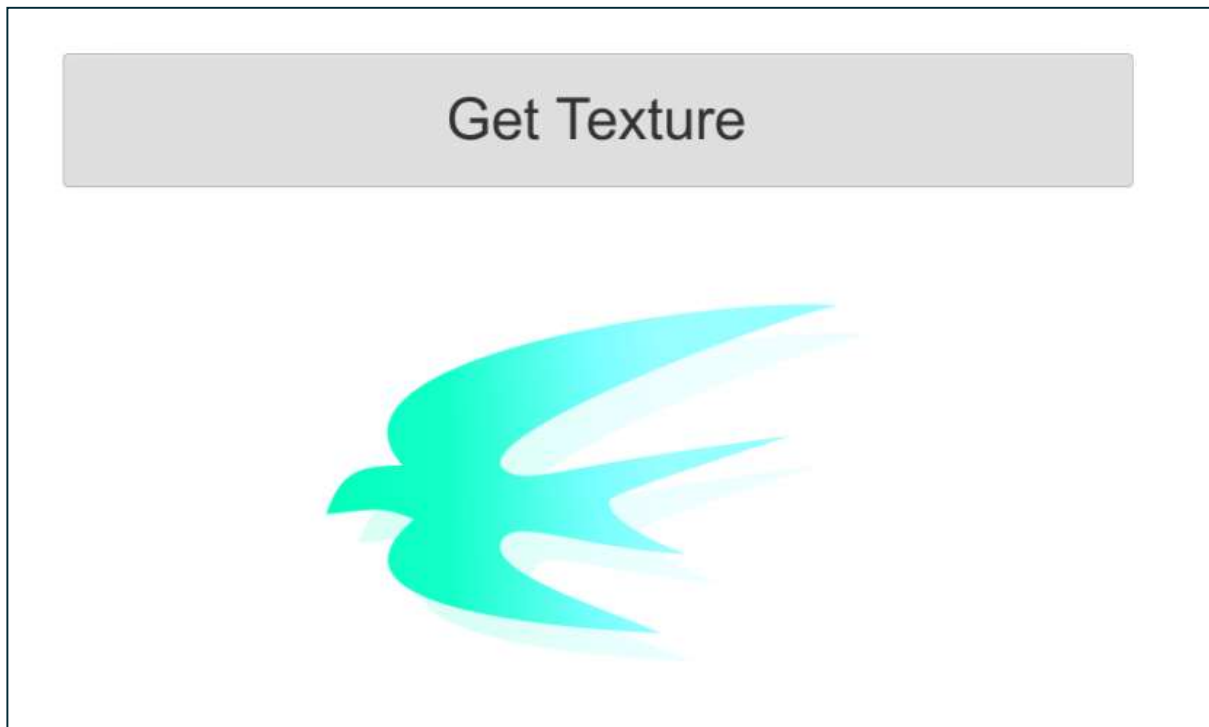
WWW를 Observable로써 취급할 수 있게 한것

- Subscribe된 순간에 통신을 실행한다
- 나중에 알아서 뒤에서 통신한 결과를 스트림에 보내 준다
- 코루틴을 사용하지 않아도 된다

```
ObservableWWW.Get("http://torisoup.net/index.html")  
    .Subscribe(result => Debug.Log(result));
```

예제) 버튼이 눌리면 텍스처를 읽어 온다

버튼이 눌리면 지정한 URL의 텍스처를 다운로드해서 Image에 표시한다



WWW로 텍스처 읽어 오기

```
_button.OnClickAsObservable()  
    .First()  
    .SelectMany(ObservableWWW.GetWWW(resourceURL))  
    .Select(www => Sprite.Create(www.texture, new Rect(0, 0, 400, 400), Vector2.zero))  
    .Subscribe(sprite =>  
    {  
        _image.sprite = sprite;  
        _button.interactable = false;  
    }, Debug.LogError);
```

WWW로 텍스처 읽어 오기

```
_button.OnClickAsObservable()  
    .First() <- 버튼을 연타해도 통신은 1회만 하도록 First를 넣는다  
    .SelectMany(ObservableWWW.GetWWW(resourceURL))  
    .Select(www => Sprite.Create(www.texture, new Rect(0, 0, 400, 400), Vector2.zero))  
    .Subscribe(sprite =>  
    {  
        _image.sprite = sprite;  
        _button.interactable = false;  
    }, Debug.LogError);
```

클릭 스트림을 ObservableWWW의
스트림으로 덮어쓴다

WWW로 텍스처 읽어 오기

```
_button.OnClickAsObservable()  
    .First()  
    .SelectMany(ObservableWWW.GetWWW(resourceURL))  
    .Select(www => Sprite.Create(www.texture, new Rect(0, 0, 400, 400), Vector2.zero))  
    .Subscribe(sprite =>  
    {  
        _image.sprite = sprite;  
        _button.interactable = false;  
    }, Debug.LogError);
```


복잡한 작업도 위에서부터 읽으면 무엇을 하는지 쉽게 이해 된다

1. 버튼이 클릭되면
2. HTTP로 텍스처를 다운로드 해서
3. 그 결과를 Sprite로 변화해서
4. Image로 표시한다

타임 아웃을 추가하기

```
_button.OnClickAsObservable()  
    .First()  
    .SelectMany(ObservableWWW.GetWWW(resourceURL).Timeout(TimeSpan.FromSeconds(3)))  
    .Select(www => Sprite.Create(www.texture, new Rect(0, 0, 400, 400), Vector2.zero))  
    .Subscribe(sprite =>  
    {  
        _image.sprite = sprite;  
        _button.interactable = false;  
    }, Debug.LogError);
```

타임 아웃이 필요하다면 여기에 오퍼레이터를 추가한다



ObservableWWW로 이것저것

동시에 통신해서 모든 데이터가 모이면 처리를 진행한다

```
var parallel = Observable.WhenAll(  
    ObservableWWW.Get("http://google.com/"),  
    ObservableWWW.Get("http://bing.com/"),  
    ObservableWWW.Get("http://unity3d.com/"));  
  
parallel.Subscribe(xs =>  
{  
    Debug.Log(xs[0].Substring(0, 100)); // google  
    Debug.Log(xs[1].Substring(0, 100)); // bing  
    Debug.Log(xs[2].Substring(0, 100)); // unity  
});
```



U
S



ObservableWWW로 이것저것


앞의 통신 결과를 사용해서 다음 통신을 실행한다

- 서버에 [리소스의 URL]을 물어보고,
서버에서 가르쳐준 URL로부터 데이터를 다운로드 한다

```
var resourcePathURL //欲しいファイルへのURLが書いてあるtxtファイル
    = "http://torisoup.net/unirx-examples/resources/resourcepath.txt";

ObservableWWW.Get(resourcePathURL)
    .SelectMany(resourceUrl => ObservableWWW.Get(resourceUrl))
    .Subscribe(Debug.Log);
```

resourcepath.txt에 적혀있는 URL에 액세스하는 코드



5. 기존 라이브러리를 스트림으로 변환 (PhtonCloud 예제)

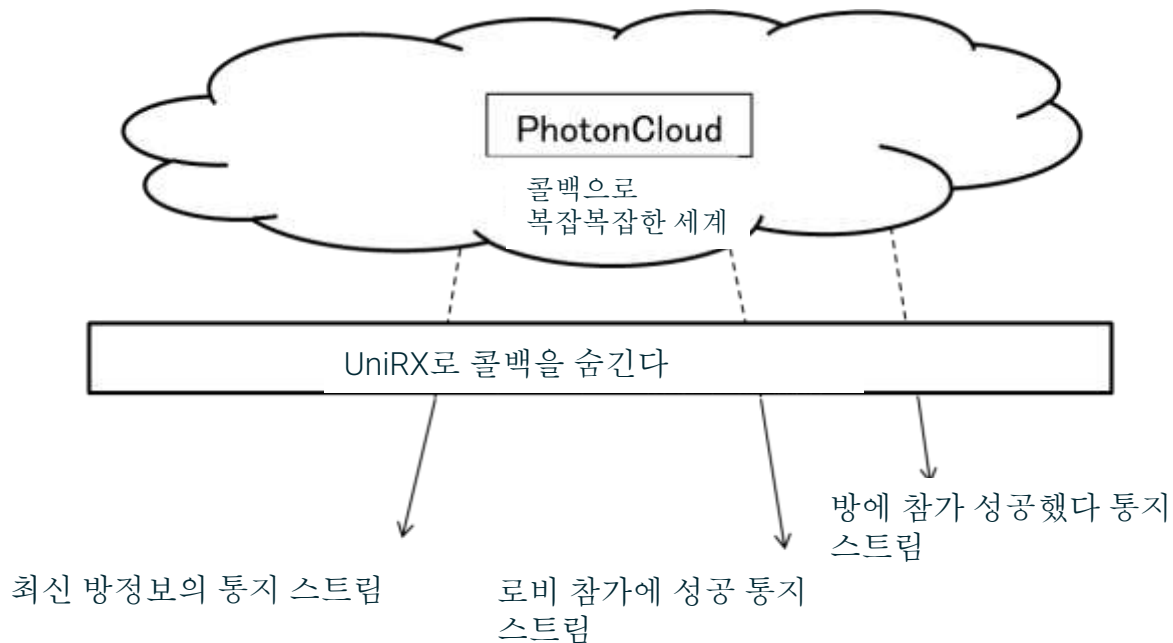
**Unity에서 간단히 네트워크 대전이 구현 가능한 라이브러리
통지가 전부 콜백이어서 미묘하게 사용하기가 나쁘다**

- UniRx에 어떻게든 해보자

UniRx와 조합하게 되면?

PhotonCloud의 콜백이 스트림으로 변환된다

- 복잡한 콜백은 숨기자



콜백으로부터 스트림을 변경하는 메리트

코드가 명시적이 된다

- Photon의 콜백은 SendMessage로 호출된다
- Observable의 제대로 정의해서 사용하면 명시적이 된다

다양한 오퍼레이터에 의한 유연한 제어가 가능해 지게 된다

- 로그인에 실패하면 3초후에 다시 리트라이 시도한다던가
- 유저들로부터 요청이 있을때 처리를 한다던가
- 방정보 리스트가 갱신될때 통지 한다던가

예) 최신 방정보를 통시하는 스트림을 만들 자

OnReceivedRoomListUpdate

- PhotonNetwork.GetRoomList()가 변경될때에 실행 된다
- 이것을 스트림으로 만들어 보자

```
public class PhotonRoomListModel : MonoBehaviour
{
    public RoomInfo[] Rooms { private set; get; }

    private void OnReceivedRoomListUpdate()
    {
        Rooms = PhotonNetwork.GetRoomList();
    }
}
```

즉 이런 형태가 되고 싶다

```
public class RoomsViewer : MonoBehaviour
{
    [SerializeField]
    private PhotonRoomListModel roomModel;

    void Start()
    {
        roomModel
            .CurrentRoomsObservable
            .Subscribe(rooms =>
            {
                //roomsに最新の部屋リストが入っている
                //ここに描画処理を書く
            });
    }
}
```

갱신된 RoomInfo[]가 흐르는 스트림
이것을 Subscribe하게 되면,
방리스트의 갱신이 있을때마다 즉시 알게됨



스트림의 소스를 만드는 법

Observable의 팩토리 메소드를 사용

기존의 이벤트등으로부터 변경한다

Subject<T>계를 사용

ReactiveProperty<T>를 사용

Subscribe가 가능한 변수

Observable로서 Subscribe가 가능하다

값이 쓸때에 OnNext 메시지가 날라간다

```
//초기화
reactiveProperty<int> reactiveInt = new ReactiveProperty<int>(0);

//Observable로서 Subscribe 가능
reactiveInt
    .AsObservable()
    .Subscribe(x => Debug.Log("ValueChanged:" + x.ToString()));

//Get으로 현재 값을 읽어오기 가능
var crrentvalue = reactiveInt.Value;

//Set으로 값의 변경이 가능
// 동시에 그때의 값이 OnNext에 보낸다
reactiveInt.Value = 10;
```

ReactiveProperty로 방정보를 통지하기

```
public class PhotonRoomListModel : MonoBehaviour
{
    public IObservable<RoomInfo[]> CurrentRoomsObservable
    {
        get { return _reactiveRooms.AsObservable(); }
    }

    private ReactiveProperty<RoomInfo[]> _reactiveRooms
        = new ReactiveProperty<RoomInfo[]>(new RoomInfo[0]);

    private void OnReceivedRoomListUpdate()
    {
        _reactiveRooms.Value = PhotonNetwork.GetRoomList();
    }
}
```

ReactiveProperty로 방정보를 통지하기

```
public class PhotonRoomListModel : MonoBehaviour
{
    public IObservable<RoomInfo[]> CurrentRoomsObservable
    {
        get { return _reactiveRooms.AsObservable(); }
    }

    private ReactiveProperty<RoomInfo[]> _reactiveRooms
        = new ReactiveProperty<RoomInfo[]>(new RoomInfo[0]);

    private void OnReceivedRoomListUpdate()
    {
        _reactiveRooms.Value = PhotonNetwork.GetRoomList();
    }
}
```

OnReceiveRoomListUpdate의 타이밍에
_reactiveRooms의 값이 바뀌면, 동시에 스트림에 통지가 날라간다

ReactiveProperty로 방정보를 통지하기

```
public class PhotonRoomListModel : MonoBehaviour
{
    public IObservable<RoomInfo[]> CurrentRoomsObservable
    {
        Observable로써 클래스 외부에 공개
        get { return _reactiveRooms.AsObservable(); }
    }

    private ReactiveProperty<RoomInfo[]> _reactiveRooms
        = new ReactiveProperty<RoomInfo[]>(new RoomInfo[0]);

    private void OnReceivedRoomListUpdate()
    {
        _reactiveRooms.Value = PhotonNetwork.GetRoomList();
    }
}
```

수신측 (아까와 같은 슬라이드)


```
public class RoomsViewer : MonoBehaviour
{
    [SerializeField]
    private PhotonRoomListModel roomModel;

    void Start()
    {
        roomModel
            .CurrentRoomsObservable
            .Subscribe(rooms =>
            {
                //roomsに最新の部屋リストが入っている
                //ここに描画処理を書く
            });
    }
}
```





UniRx에서는
아직도 더 많은 기능 들이 있습니다



**더 많은 내용은
UniRx, Rx 관련 자료들을
검색해 보세요**

UniRx는 편리하니까 사용해 보자!!!

- [시간]을 상당히 간단히 취급할 수 있게 된다
- GUI 관련 구현도 간단히 쓸 수 있다
- 게임 로직에 적용하는 것도 가능하다

UniRx는 편리하지만, 어려운 면도 있다

- 학습 코스트가 높고 개념적으로도 어렵다
- 도입하는 경우에는 프로그램의 설계부터 다시 생각할 필요가 생긴다
 - * 진가를 발휘하기 위해서는 설계 근본에서 UniRx를 포함시켜야 함
 - * [편리한 라이브러리]가 아닌 [언어 확장]이라고 생각할 필요가 있다

추가 내용

1. UniRx의 사용예

- Update()를 없애기
- 컴포넌트를 스트림으로 연결하기
- uGUI와 조합해서 사용하기
- 코루틴과 조합하기

2. 마지막으로

추가 내용

1. UniRx의 사용예

- **Update()를 없애기**
- 컴포넌트를 스트림으로 연결하기
- uGUI와 조합해서 사용하기
- 코루틴과 조합하기

2. 마지막으로

Update를 없애기

**Update()를 Observable로 변환해서
Awake()/Start()에 모아서 작성하기**

Observable화 하지 않은 구현

```
private void Update()
{
    //移動可能状態なら移動とジャンプ処理ができる
    if (canPlayerMove)
    {
        //移動処理
        var inputVector =
            (new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical")));

        //スティックがしきい値を超えるだけ倒されていたら移動する
        if (inputVector.magnitude > 0.5f)
        {
            Move(inputVector.normalized);
        }

        //接地中にジャンプボタンが押されたらジャンプする
        if (isOnGrounded && Input.GetButtonDown("Jump"))
        {
            Jump();
        }
    }

    //頻弾があるなら攻撃できる
    if (ammoCount > 0)
    {
        if (Input.GetButtonDown("Attack"))
        {
            Attack();
        }
    }
}
```

하고 싶은 것을 순차적으로 작성해야 해서, 흐름이 따라가기 힘들다

비교 Observable

```
private void Start()
```

```
{
```

```
//이동 가능할때에 이동키가 일정 이상 입력 받으면 이동
```

```
this.UpdateAsObservable()  
    .Where(_ => canPlayerMove)  
    .Select(_ => new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical")))  
    .Where(input => input.magnitude > 0.5f)  
    .Subscribe(Move);
```

```
//이동 가능하고, 지면에 있을때에 점프 버튼이 눌리면 점프
```

```
this.UpdateAsObservable()  
    .Where(_ => canPlayerMove && isOnGrounded && Input.GetButtonDown("Jump"))  
    .Subscribe(_ => Jump());
```

```
// 총알이 있는 경우 공격 버튼이 눌리면 공격
```

```
this.UpdateAsObservable()  
    .Where(_ => ammoCount > 0 && Input.GetButtonDown("Attack"))  
    .Subscribe(_ => Attack());
```

```
}
```

작업을 병렬로 작성할 수 있어서 읽기가 쉽다

Update를 없애기의 메리트

작업별로 병렬로 늘어서 작성하는것이 가능

- 작업별로 스코프가 명시적이게 된다
- 기능 추가, 제거, 변경이 용이해지게 된다
- 코드가 선언적이 되어서, 처리의 의도가 알기 쉽게 된다

Rx의 오퍼레이터가 로직 구현에 사용 가능

- 복잡한 로직을 오퍼레이터의 조합으로 구현 가능하다

Observable로 변경하는 3가지 방법

UpdateAsObservable

- 지정한 gameObject에 연동된 Observable가 만들어진다.
- gameObject의 Destroy때에 **OnCompleted**가 발행된다

Observable.EveryUpdate

- gameObject로부터 **독립된** Observable이 만들어 진다
- MonoBehaviour에 관계 없는 곳에서도 사용 가능

ObserveEveryValueChanged

- Observable.EveryUpdate의 파생 버전
- 값의 변화를 **매프레임 감시**하는 Observable이 생성된다

Observable.EveryUpdate의 주의점

Destroy때에 OnCompleted가 발생되지 않는다

- UpdateAsObservable과 같은 느낌으로 사용하면 함정에 빠진다

```
[SerializeField] private Text text;  
private void Start()  
{  
    //座標をUIに描画  
    Observable.EveryUpdate()  
        .Select(_ => transform.position)  
        .SubscribeToText(text);  
}
```

이 gameObject가 파괴되면

Null이 되어 예외가 발생한다

수명 관리의 간단한 방법

AddTo

- 특정 gameObject가 파괴되면 자동 Dispose 되게 해준다
- OnCompleted가 발행되는것은 아니다

```
[SerializeField] private Text text;  
private void Start()  
{  
    //座標をUIに描画  
    Observable.EveryUpdate()  
        .Select(_ => transform.position)  
        .SubscribeToText(text)  
        .AddTo(this.gameObject);  
}
```

**AddTo로 넘겨진 gameObject가
Destroy되면 같이 Dispose 된다**

목차

1. UniRx의 사용예

- Update()를 없애기
- **컴포넌트를 스트림으로 연결하기**
- uGUI와 조합해서 사용하기
- 코루틴과 조합하기

2. 마지막으로

스트림으로 연결

컴포넌트를 스트림으로 연결하는 것으로,
Observer패턴한 설계로 만들 수 있다

- 전체가 이벤트 기반이 되게 할 수 있다
- 더불어 Rx는 Observer패턴 그 자체

타이머 예제

타이머의 카운트를 화면에 표시 한다

- UniRx를 사용하지 않고 구현
- UniRx의 스트림으로 구현



타이머 예제

타이머의 카운트를 화면에 표시 한다

- UniRx를 사용하지 않고 구현
- UniRx의 스트림으로 구현

UniRx를 사용하지 않고 구현

```
public class TimerDisplayComponent : MonoBehaviour
{
    [SerializeField]
    private TimerComponent _timerComponent;
    private Text _timerText;

    void Start()
    {
        _timerText = GetComponent<Text>();
    }

    void Update()
    {
        //現在のカウントを取得
        var currentTimeText = _timerComponent.CurrentTime.ToString();

        //カウントが更新されていたら描画を書き換える
        if (currentTimeText != _timerText.text)
        {
            _timerText.text = currentTimeText;
        }
    }
}
```

UniRx를 사용하지 않고 구현

```
public class TimerDisplayComponent : MonoBehaviour
```

```
{
```

```
    [SerializeField]
```

```
    private TimerComponent _timerComponent;
```

```
    private Text _timerText;
```

```
    void Start()
```

```
    {
```

```
        _timerText = GetComponent<Text>();
```

```
    }
```

```
    void Update()
```

```
    {
```

```
        //現在のカウンタを取得
```

```
        var currentTimeText = _timerComponent.CurrentTime.ToString();
```

```
        //カウンタが更新されていたら描画を書き換える
```

```
        if (currentTimeText != _timerText.text)
```

```
        {
```

```
            _timerText.text = currentTimeText;
```

```
        }
```

```
    }
```

```
}
```

매 프레임,

값이 변경되었는지를 확인 한다

(매 프레임 값을 체크해야만 한다)

타이머 예제

타이머의 카운트를 화면에 표시 한다

- UniRx를 사용하지 않고 구현
- UniRx의 스트림으로 구현

우선 타이머측을 스트림으로 변경

```
public class TimerComponent : MonoBehaviour
{
    private readonly ReactiveProperty<int> _timerReactiveProperty = new IntReactiveProperty(30);

    /// <summary>
    /// 現在のカウント
    /// </summary>
    public ReadOnlyReactiveProperty<int> CurrentTime
    {
        get { return _timerReactiveProperty.ToReadOnlyReactiveProperty(); }
    }

    private void Start()
    {
        //1초마다 timerReactiveProperty 값을 마이너스 한다
        Observable.Timer(TimeSpan.FromSeconds(1))
            .Subscribe(_ => _timerReactiveProperty.Value--)
            .AddTo(gameObject); // 게임오브젝트 파괴시에 자동 정지 시킨다
    }
}
```

우선 타이머측을 스트림으로 변경

```
public class TimerComponent : MonoBehaviour
{
```

```
    private readonly ReactiveProperty<int> _timerReactiveProperty = new IntReactiveProperty(30);
```

```
    /// <summary>
```

```
    /// 現在のカウント
```

```
    /// </summary>
```

```
    public ReadOnlyReactiveProperty<int> CurrentTime
```

```
    {
```

```
        get { return _timerReactiveProperty.ToReadOnlyReactiveProperty(); }
```

```
    }
```

CurrentTime을 ReactiveProperty로 공개한다

```
    private void Start()
```

```
    {
```

```
        //1초마다 timerReactiveProperty 값을 마이너스 한다
```

```
        Observable.Timer(TimeSpan.FromSeconds(1))
```

```
            .Subscribe(_ => _timerReactiveProperty.Value--)
```

```
            .AddTo(gameObject); // 게임오브젝트 파괴시에 자동 정지 시킨다
```

```
    }
```

```
}
```

(값이 변경되면, onNext로 그 값이 통지 된다)

타이머를 사용하는 측의 구현

```
public class TimerDisplayComponent : MonoBehaviour
{
    [SerializeField]
    private TimerComponent _timerComponent;
    private Text _timerText;

    void Start()
    {
        _timerComponent.CurrentTime
            .SubscribeToText(_timerText);
    }
}
```

타이머를 사용하는 측의 구현

```
public class TimerDisplayComponent : MonoBehaviour
{
    [SerializeField]
    private TimerComponent _timerComponent;
    private Text _timerText;
```

```
void Start()
{
    _timerComponent.CurrentTime
        .SubscribeToText(_timerText);
}
```

타이머에서 값 갱신 통지가 오면,
그 타이밍에 화면을 변경하는것 뿐

```
}
```

스트림으로 연결하는 메리트

Observer 패턴이 간단히 구현 가능하다

- 변화를 폴링(Polling)하는 구현이 사라진다
- 필요한 타이밍에 필요한 처리를 하는 방식으로 작성하는 것이 좋다

기존의 이벤트 통지구조보다 간단

- C#의 Event는 준비단계가 귀찮아서 쓰고 싶지 않다
- Unity의 SendMessage는 쓰고 싶지 않다
- Rx라면 Observable를 준비하면 OK! 간단!

목차

1. UniRx의 사용예

- Update()를 없애기
- 컴포넌트를 스트림으로 연결하기
- **uGUI와 조합해서 사용하기**
- 코루틴과 조합하기

2. 마지막으로

UniRx와 uGUI를 조합하기

uGUI에서 사용할 수 있는 Model-View-00 패턴

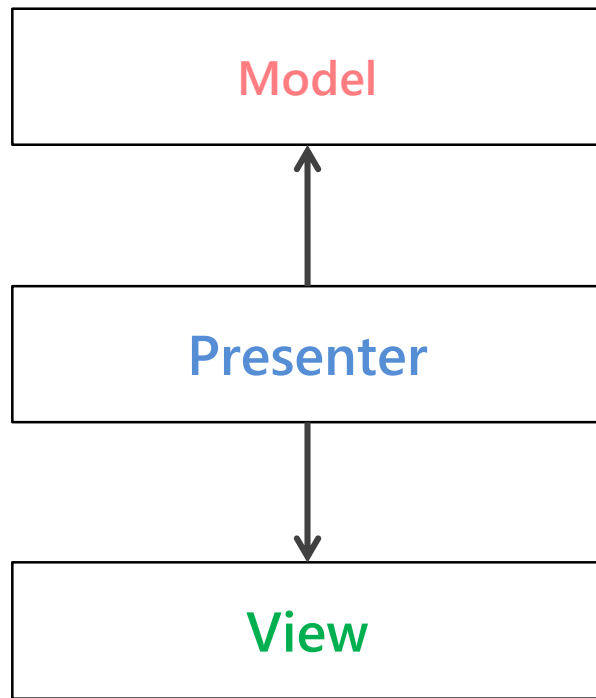
- uGUI에 유용한 MVO패턴이 지금까지 존재하지 않았다
 - * MVC 패턴은 원래 사람에 따라서 개념이 각각 너무 다름
 - * MVVM은 데이터바인딩이 없기 때문에 사용할 수 없다
- Observable과 ReactiveProperty를 조합하게 되면, uGUI 관련을 깔끔하게 작성가능

Model-View-(Reactive)Presenter

MV(R)P 패턴

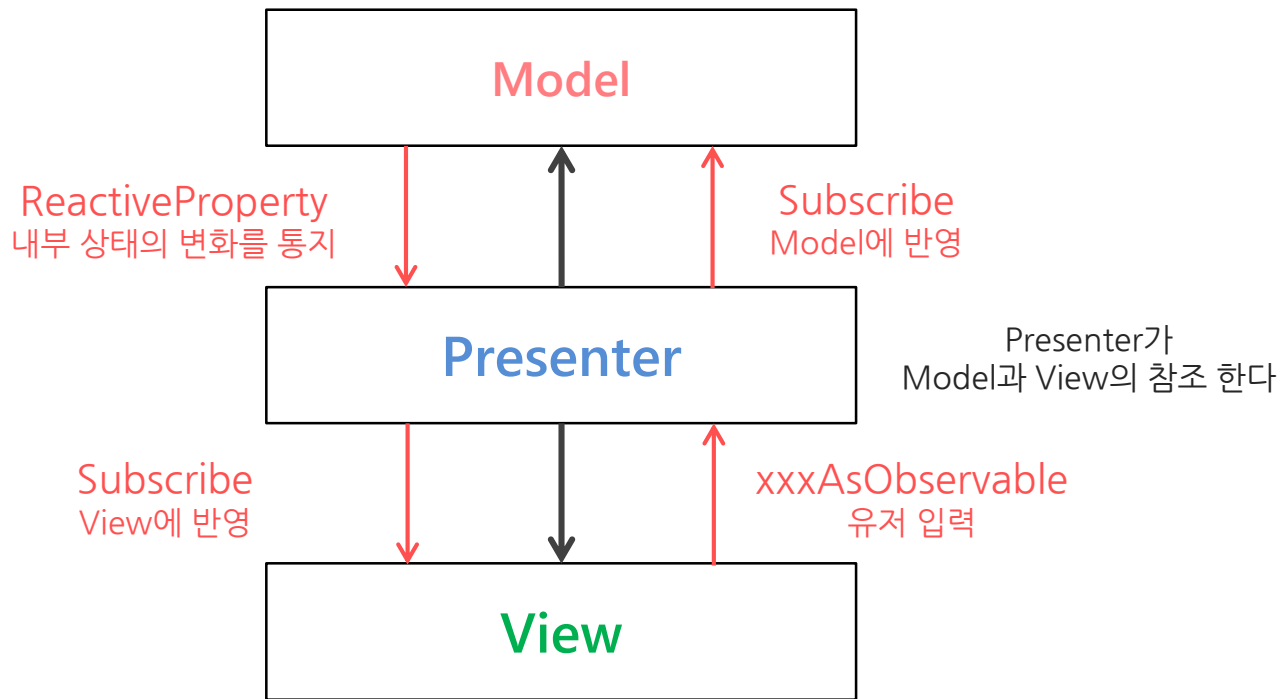
- Model - View - Presenter 패턴 + UniRx
- 3개의 레이어를 Observable로 심리스하게 연결

MV(R)P의 구조



Presenter가
Model과 View를 참조한다

MV(R)P의 구조



MV(R)P패턴 만드는 법

1. **Model**에 ReactiveProperty를 가지게 한다
2. **Presenter**을 만든다
3. **Presenter**에 **Model**과 **View**를 등록 한다
4. **Presenter**내에서 **View**의 **Observable**과 **Model**의 **ReactiveProperty**를
각각 Subscribe해서 연결한다

사용예) 미쿠미쿠 마우스

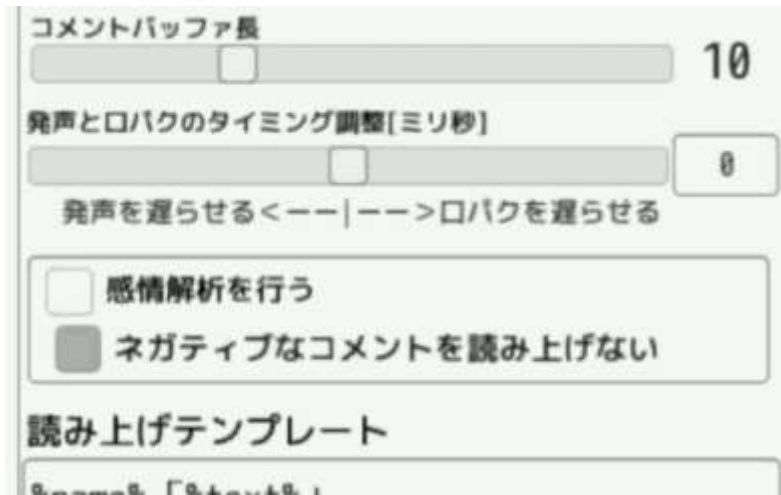
MMD 모델에 니코생방송의 코멘트를 읽게 하는 툴
UI를 MV(R)P 패턴으로 구현하였다



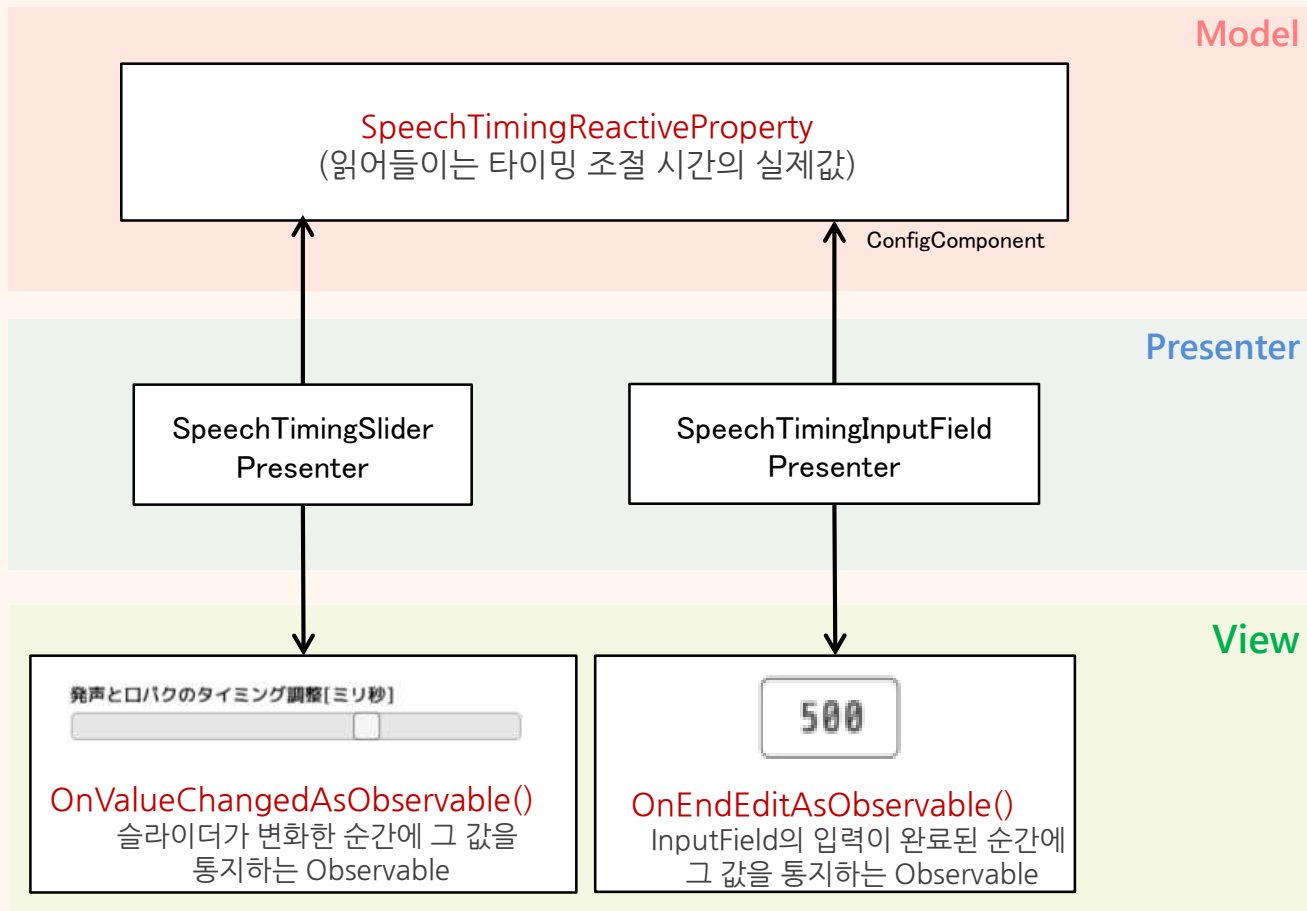
MV(R)P 구현예제 소개

코멘트를 읽어들이는 타이밍의 조절 슬라이더

- Slider와 InputField가 연동한다
- 입력된 값은 ConfigComponent가 유지 한다



컴포넌트간의 관계도



Model의 구현

```
/// <summary>
/// 設定を管理するコンポーネント
/// </summary>
public class ConfigComponent : SingletonMonoBehaviour<ConfigComponent>
{
    /// <summary>
    /// 읽어들이는 타이밍(ms)
    /// </summary>
    public ReactiveProperty<int> SpeechTimingReactiveProperty
        = new IntReactiveProperty(0);

    //以下他のReactivePropertyの定義等が続く
}
```

ReactiveProperty를 외부에 공개

Presenter의 구현 (Slider 측)

```
using UnityEngine;
using UniRx;
using UnityEngine.UI;

public class SpeechTimingSliderPresenter : MonoBehaviour
{
    void Start()
    {
        var slider = GetComponent<Slider>();
        var config = ConfigComponent.Instance;

        //Model->Slider 반영
        config.SpeechTimingReactiveProperty
            .Subscribe(x => slider.value = x / 10);

        //Slider->Model 반영
        slider
            .OnValueChangedAsObservable()
            .DistinctUntilChanged()
            .Subscribe(x => config.SpeechTimingReactiveProperty.Value = (int)(x * 10));
    }
}
```

Model → View

View → Model

Presenter의 구현 (Input 측)

```
using System;
using UnityEngine;
using UniRx;
using UnityEngine.UI;

public class SpeechTimingInputField : MonoBehaviour
{
    public void Start()
    {
        var config = ConfigComponent.Instance;
        var inputField = GetComponent<InputField>();

        //View → Model
        inputField.OnEndEditAsObservable()
            .Select(x => Int32.Parse(x))
            .Select(x => Mathf.Clamp(x, -1500, 1500))
            .Subscribe(x => config.SpeechTiming = x);

        //Model→View
        config
            .SpeechTimingObservable
            .Select(x => x.ToString())
            .Subscribe(x => inputField.text = x);
    }
}
```

View → Model

Model → View

uGUI와 조합하기 정리

MV(R)P 패턴으로 uGUI 관련 설계가 편해진다

- uGUI를 사용할 경우에 아마도 현시점에서의 최적 해결법
- 프로그래머에게는 다루기 쉽지만,
비프로그래머에게는 다루기 어려워질 가능성이 있으니 주의

Presenter의 베스트한 작성 방법에 대해서는 아직 모색중

- Presenter에 어떻게 Model과 View를 등록할 것인가
- Presenter을 하나로 모을 것인가, 분할해서 만들 것인가
- 동적으로 Presenter를 생성하는 경우에는 어떻게 할 것인가

목차

1. UniRx의 사용예

- Update()를 없애기
- 컴포넌트를 스트림으로 연결하기
- uGUI와 조합해서 사용하기
- **코루틴과 조합하기**

2. 마지막으로

코루틴과 조합하기

Unity의 코루틴과 Obervable은 **상호교환가능**

- 코루틴과 조합함으로서 Rx의 약점을 보완하는 것이 가능하다

코루틴 <-> Observable

코루틴 -> Observable

- **Observable.FromCoroutine**

Observable -> 코루틴

- **StartAsCoroutine**

코루틴 <-> Observable

코루틴 -> Observable

- `Observable.FromCoroutine`

Observable -> 코루틴

- `StartAsCoroutine`

코루틴->Observable의 메리트

복잡한 스트림을 간단하게 작성 가능

- 팩토리 메소드나 오퍼레이터 체인 만으로는 작성할 수 없는 복잡한 로직의 스트림을 작성하는 것이 가능하다

연속적인 처리를 캡슐화 가능하다

- 복잡한 처리를 코루틴에 숨겨서 외부에서는 Observable로서 선언적으로 취급하는 것이 가능하다

예제) 코루틴으로부터 Observable을 만들기

플레이어의 생사에 연동된 타이머

- 플레이어가 살아있을 때만 카운트다운
- 플레이어가 사망했을 때는 타이머를 정지한다
- 팩토리메소드나 오퍼레이터 체인으로 만드는것은 복잡하다

타이머의 정의

```
private void Start()
{
    //타이머의 Subscribe
    CountdownCoroutine(observer => CountdownCoroutine(observer, 30, player))
        .Subscribe(count => Debug.Log(count));
}
```

```
/// <summary>
```

//플레이어가 살아있을때만 카운트다운하는 타이머
// 플레이어가 죽어있는 때는 카운트는 정지한다

```
IEnumerator CountdownCoroutine(IObserver<int> observer,int startTime,Player player)
{
    var currentTime = startTime;
    while (currentTime > 0)
    {
        if (player.IsAlive)
        {
            observer.OnNext(currentTime--);
        }
        yield return new WaitForSeconds(1);
    }
    observer.OnNext(0);
    observer.OnCompleted();
}
```

코루틴에서 변환 한다

```
private void Start()
{
    //타이머의 Subscribe
    CountdownCoroutine(observer => CountdownCoroutine(observer, 30, player))
        .Subscribe(count => Debug.Log(count));
}
```

```
/// <summary>
```

//플레이어가 살아있을때만 카운트다운하는 타이머
// 플레이어가 죽어있는 때는 카운트는 정지한다

```
IEnumerator CountdownCoroutine(IObserver<int> observer,int startTime,Player player)
{
    var currentTime = startTime;
    while (currentTime > 0)
    {
        if (player.IsAlive)
        {
            observer.OnNext(currentTime--);
        }
        yield return new WaitForSeconds(1);
    }
    observer.OnNext(0);
    observer.OnCompleted();
}
```

코루틴안에서 OnNext

```
private void Start()
{
    //타이머의 Subscribe
    CountdownCoroutine(observer => CountdownCoroutine(observer, 30, player))
        .Subscribe(count => Debug.Log(count));
}
```

/// <summary>

//플레이어가 살아있을때만 카운트다운하는 타이머
// 플레이어가 죽어있는 때는 카운트는 정지한다

```
IEnumerator CountdownCoroutine(IObserver<int> observer,int startTime,Player player)
{
    var currentTime = startTime;
    while (currentTime > 0)
    {
        if (player.IsAlive)
        {
            observer.OnNext(currentTime--);
        }
        yield return new WaitForSeconds(1);
    }
    observer.OnNext(0);
    observer.OnCompleted();
}
```

코루틴 <-> Observable

코루틴 -> Observable

- Observable.FromCoroutine

Observable -> 코루틴

- StartAsCoroutine

Observable을 코루틴으로 변경한다

StartAsCoroutine

- **OnCompleted**가 발생되기 전에 yield return null을 계속 한다
- 완료시에 최후의 OnNext값을 하나 출력 한다

```
var v = default(int);  
yield return Observable.Range(1, 10).StartAsCoroutine(x => v = x);  
Debug.Log(v); //10
```

비동기 처리를 동기처리처럼 쓸 수 있게 된다

- Task의 await와 비슷한 일을 Unity 코루틴으로 실현 가능

구현예

텍스트를 Web으로부터 다운로드

버튼으로 텍스트를 넘기면서 표시한다



구현한 코루틴

```
private void Start()
{
    StartCoroutine(ShowTextCoroutine());
}

private IEnumerator ShowTextCoroutine()
{
    _text.text = "다운로드開始...\n";

    IEnumerator<string> textIterator = null;

    //텍스트를 다운로드해서 변환한다
    yield return ObservableWWW.Get(url)
        .Select(result => TextParser(url))
        .StartAsCoroutine(iterator => textIterator = iterator);

    _text.text += "다운로드完了...\n\n";

    while (textIterator.MoveNext())
    {
        _text.text += textIterator.Current;

        //버튼 클릭이 올때까지 대기 한다
        yield return _nextButton.OnClickAsObservable().FirstOrDefault().StartAsCoroutine();
    }
}
```

텍스트의 다운로드를 기다림

```
private void Start()
{
    StartCoroutine(ShowTextCoroutine());
}

private IEnumerator ShowTextCoroutine()
{
    _text.text = "다운로드開始...\n";

    IEnumerator<string> textIterator = null;

    //텍스트를 다운로드해서 변환한다
    yield return ObservableWWW.Get(url)
        .Select(result => TextParser(url))
        .StartAsCoroutine(iterator => textIterator = iterator);

    _text.text += "다운로드完了...\n\n";

    while (textIterator.MoveNext())
    {
        _text.text += textIterator.Current;

        //버튼 클릭이 올때까지 대기 한다
        yield return _nextButton.OnClickAsObservable().FirstOrDefault().StartAsCoroutine();
    }
}
```


버튼이 눌릴때까지 기다림

```
private void Start()
{
    StartCoroutine(ShowTextCoroutine());
}

private IEnumerator ShowTextCoroutine()
{
    _text.text = "다운로드開始...\n";

    IEnumerator<string> textIterator = null;

    //텍스트를 다운로드해서 변환한다
    yield return ObservableWWW.Get(url)
        .Select(result => TextParser(url))
        .StartAsCoroutine(iterator => textIterator = iterator);

    _text.text += "다운로드完了...\n\n";

    while (textIterator.MoveNext())
    {
        _text.text += textIterator.Current;

        //버튼 클릭이 올때까지 대기 한다
        yield return _nextButton.OnClickAsObservable().FirstOrDefault().StartAsCoroutine();
    }
}
```

코루틴과 조합하기 정리

Unity의 코루틴과 Obervable은 **상호교환가능**

- 복잡한 스트림을 코루틴을 사용해서 구현 가능
- 코루틴 안에서 이벤트를 기다리는 것이 가능
- 비동기처리를 **동기처리처럼** 작성하는 것이 가능하게 된다

목차

1. UniRx의 사용예

- Update()를 없애기
- 컴포넌트를 스트림으로 연결하기
- uGUI와 조합해서 사용하기
- 코루틴과 조합하기

2. 마지막으로

UniRx를 쓸때의 마음가짐

[모든것은 스트림] 이라는 세계관을 가지자

- 스트림화 가능한 곳을 찾아서 스트림으로 바꾸면 편리해진다
- 다만 과용하는 것은 금물

Rx는 만능이 아니라는 것을 알자

- 하고 싶은 것이 선언적으로 잘되지 않는 부분도 당연히 있다
- 그런 경우에는 코루틴과 조합해서 사용하면 유용하다

설계를 의식하자

- 무조건 쓰고보자라는 설계로 UniRx를 사용하는것은 상당히 위험
- Observer패턴의 장단점을 활용하자



감사합니다



스트림의 소스를 만드는것

- Subject, ReplySubject, BehaviorSubject, AsyncSubject로 여러 종류 있다
- 외부에 공개할 때에는 반드시 AsObservable을 거쳐서 공개한다
- * 외부에서 직접 onNext가 호출되는 상태로 하지 않음

```
//newでストリームが作れる
Subject<int> stream = new Subject<int>();

//IObservableを実装しているのでそのままオペレータが連結できる
//ただし外部に公開するときはAsObservable()を挟むべき
stream
    .Where(x => x > 10)
    .Subscribe(x => Debug.Log(x));

//OnNextでメッセージを流す
stream.OnNext(1);
stream.OnNext(20);
stream.OnNext(30);

//OnCompleteで終了
stream.OnCompleted();
```

보충) Subject를 멈추는 법

Dispose를 호출하면 Subscribe가 중지 된다

- 스트림의 소스가 해제되면 자동적으로 Dispose 된다
- 스트림이 완료상태가 되어도 Dispose 된다
- static한 스트림을 만들 경우에는 수동 Dispose가 필요

```
//Subscribeの返り値はIDisposable  
IDisposable disposable = button.onClick  
    .AsObservable()  
    .Subscribe(_ => text.text = "clicked");  
  
//Dispose()を呼び出せばストリームの購読を終了する  
disposable.Dispose();
```

둘다 Update()의 타이밍에 통지되는 Observable

- UpdateAsObservable

~~* ObservableMonoBehavior를 상속받아 사용~~

* Obverseable을 계승하는 방식은 없어졌음

대신에 UniRx.Trigger 네임 스페이스에 준비된 확장 메소드를 사용

* lobservable<Unit>

* 컴포넌트가 Destroy 될때 자동 Dispose

- Observable.EveryUpdate

* 어느 스크립트에서 사용할 수 있다

* lobservable<long> (Subscribe된 때부터의 프레임 수)

* 사용이 끝나면 명시적으로 Dispose 할 필요가 있다

혹은 OnCompleted가 제대로 불리는 스트림으로 한다



보너스



보너스) 코루틴을 Observable로 변환하기

Observable.FromCoroutine를 사용해서 변경 가능

- 코루틴을 실행순서나 실행 조건을 스트림으로 정의 가능

```
private void Start()
{
    Observable.FromCoroutine(CoroutineA)
        .SelectMany(Observable.FromCoroutine(CoroutineB))
        .Subscribe(_ => Debug.Log("CoroutineA & CoroutineB are done.));
}

private IEnumerator CoroutineA()
{
    Debug.Log("CoroutineA start");
    yield return new WaitForSeconds(1);
    Debug.Log("CoroutineA end");
}

private IEnumerator CoroutineB()
{
    Debug.Log("CoroutineB start");
    yield return new WaitForSeconds(1);
    Debug.Log("CoroutineB end");
}
```

FromCoroutine<T>를 사용하면 자유롭게 스트림을 만들 수 있다

```
private void Start()
{
    Observable.FromCoroutine<int>(observer => TimerCoroutine(observer, 10))
        .Subscribe(_ => Debug.Log(_));
}

private IEnumerator TimerCoroutine(IObserver<int> observer, int timeCount)
{
    do
    {
        observer.OnNext(timeCount);
        yield return new WaitForSeconds(1.0f);
    } while (--timeCount > 0);

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

카운트다운 타이머의 예

계속) 카운트다운 타이머를 만들려면...

하지만 `FromCoroutine<T>`로 카운트다운 타이머를 만드는것보다
`Observable.Timer`로 만드는것이 스마트하다

```
private void Start()
{
    //10秒カウントダウン
    createCountDownObservable(10)
        .Subscribe(x => Debug.Log(x), () => Debug.Log("OnComplete"));
}

private IObservable<int> createCountDownObservable(int time)
{
    return Observable.Timer(TimeSpan.FromSeconds(0), TimeSpan.FromSeconds(1))
        .Select(x => (int) (time - x)) //xは起動してからの秒数
        .Take(time + 1);
}
```

처리를 실행하는 스레드를 교체하는 오퍼레이터

- ObserveOn를 사용하는 스레드간에 데이터의 취급을 고려할 필요가 없어진다

```
//メチャクチャ重いファイルを読み込む時にスレッドを分ける
_button.OnClickAsObservable()
    .ObserveOn(Scheduler.ThreadPool) // 여기서부터 스레드 풀에서 실행
    .Select(_ => LoadFile())         // 무지하게 무거운 파일을 로드
    .ObserveOnMainThread()           // 메인 스레드로 복귀
    .Subscribe(data =>
    {
        //ロードしたファイルを使った処理
    });
```

사용예) 텍스트가 입력되면 검색 서제스트를 띄우기

1. InputField에 텍스트를 입력 받을 때
2. 최후에 입력된 때부터 200m초 이상 간극이 생기면
3. GoogleSuggestAPI를 호출해서
4. 그때의 서제스트 결과를 Text 표시한다



사용예) 텍스트가 입력되면 검색 서제스트를 띄우기

```
private readonly string _apiUrlFormat
    = "http://www.google.com/complete/search?hl=ja&output=toolbar&q={0}";

private void Start()
{
    _inputField
        .OnValueChangedAsObservable()
        .Throttle(TimeSpan.FromMilliseconds(200))
        .Where(word => word.Length > 0)
        .SelectMany(word => ObservableWWW.Get(string.Format(_apiUrlFormat, WWW.EscapeURL(word))))
        .Select(xml => XMLResultToStrings(xml)) //通信結果がXMLなのでパースする
        .Where(suggests => suggests.Any())
        .SubscribeToText(_text, suggestResults =>
            suggestResults.Aggregate((s, n) => s + n + System.Environment.NewLine)
        );
}
```

Thank you!