Ordinarily, Network operations in Unity require the use of WWW and Coroutine. That said, using Coroutine is not good practice for asynchronous operations for the following (and other) reasons:

1. Coroutines can't return any values, since its return type must be IEnumerator.
2. Coroutines can't handle exceptions, because yield return statements cannot be surrounded with a try-catch construction.

This kind of lack of composability causes operations to be close-coupled, which often results in huge monolithic IEnumerators.

Rx cures that kind of "asynchronous blues". Rx is a library for composing asynchronous and event-based programs using observable collections and LINQ-style query operators.

The game loop (every Update, OnCollisionEnter, etc), sensor data (Kinect, Leap Motion, VR Input, etc.) are all types of events. Rx represents events as reactive sequences which are both easily composable and support time-based operations by using LINQ query operators.

Unity is generally single threaded but UniRx facilitates multithreading for joins, cancels, accessing GameObjects, etc.

UniRx helps UI programming with uGUI. All UI events (clicked, valuechanged, etc) can be converted to UniRx event streams.

Unity supports async/await from 2017 with C# upgrades, UniRx family prjects provides more lightweight, more powerful async/await integration with Unity. Please see [Cysharp/UniTask](https://github.com/Cysharp/UniTask).

**Introduction**

Great introduction to Rx article: [The introduction to Reactive Programming you've been missing](https://gist.github.com/staltz/868e7e9bc2a7b8c1f754).

The following code implements the double click detection example from the article in UniRx:

var clickStream = Observable.EveryUpdate()

.Where(\_ => Input.GetMouseButtonDown(0));

clickStream.Buffer(clickStream.Throttle(TimeSpan.FromMilliseconds(250)))

.Where(xs => xs.Count >= 2)

.Subscribe(xs => Debug.Log("DoubleClick Detected! Count:" + xs.Count));

This example demonstrates the following features (in only five lines!):

* The game loop (Update) as an event stream
* Composable event streams
* Merging self stream
* Easy handling of time based operations

**Network operations**

Use ObservableWWW for asynchronous network operations. Its Get/Post functions return subscribable IObservables:

ObservableWWW.Get("http://google.co.jp/")

.Subscribe(

x => Debug.Log(x.Substring(0, 100)), // onSuccess

ex => Debug.LogException(ex)); // onError

Rx is composable and cancelable. You can also query with LINQ expressions:

// composing asynchronous sequence with LINQ query expressions

var query = from google in ObservableWWW.Get("http://google.com/")

from bing in ObservableWWW.Get("http://bing.com/")

from unknown in ObservableWWW.Get(google + bing)

select new { google, bing, unknown };

var cancel = query.Subscribe(x => Debug.Log(x));

// Call Dispose is cancel.

cancel.Dispose();

Use Observable.WhenAll for parallel requests:

// Observable.WhenAll is for parallel asynchronous operation

// (It's like Observable.Zip but specialized for single async operations like Task.WhenAll)

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

});

Progress information is available:

// notifier for progress use ScheduledNotifier or new Progress<float>(/\* action \*/)

var progressNotifier = new ScheduledNotifier<float>();

progressNotifier.Subscribe(x => Debug.Log(x)); // write www.progress

// pass notifier to WWW.Get/Post

ObservableWWW.Get("http://google.com/", progress: progressNotifier).Subscribe();

Error handling:

// If WWW has .error, ObservableWWW throws WWWErrorException to onError pipeline.

// WWWErrorException has RawErrorMessage, HasResponse, StatusCode, ResponseHeaders

ObservableWWW.Get("http://www.google.com/404")

.CatchIgnore((WWWErrorException ex) =>

{

Debug.Log(ex.RawErrorMessage);

if (ex.HasResponse)

{

Debug.Log(ex.StatusCode);

}

foreach (var item in ex.ResponseHeaders)

{

Debug.Log(item.Key + ":" + item.Value);

}

})

.Subscribe();

**Using with IEnumerators (Coroutines)**

IEnumerator (Coroutine) is Unity's primitive asynchronous tool. UniRx integrates coroutines and IObservables. You can write asynchronious code in coroutines, and orchestrate them using UniRx. This is best way to control asynchronous flow.

// two coroutines

IEnumerator AsyncA()

{

Debug.Log("a start");

yield return new WaitForSeconds(1);

Debug.Log("a end");

}

IEnumerator AsyncB()

{

Debug.Log("b start");

yield return new WaitForEndOfFrame();

Debug.Log("b end");

}

// main code

// Observable.FromCoroutine converts IEnumerator to Observable<Unit>.

// You can also use the shorthand, AsyncA().ToObservable()

// after AsyncA completes, run AsyncB as a continuous routine.

// UniRx expands SelectMany(IEnumerator) as SelectMany(IEnumerator.ToObservable())

var cancel = Observable.FromCoroutine(AsyncA)

.SelectMany(AsyncB)

.Subscribe();

// you can stop a coroutine by calling your subscription's Dispose.

cancel.Dispose();

If in Unity 5.3, you can use ToYieldInstruction for Observable to Coroutine.

IEnumerator TestNewCustomYieldInstruction()

{

// wait Rx Observable.

yield return Observable.Timer(TimeSpan.FromSeconds(1)).ToYieldInstruction();

// you can change the scheduler(this is ignore Time.scale)

yield return Observable.Timer(TimeSpan.FromSeconds(1), Scheduler.MainThreadIgnoreTimeScale).ToYieldInstruction();

// get return value from ObservableYieldInstruction

var o = ObservableWWW.Get("http://unity3d.com/").ToYieldInstruction(throwOnError: false);

yield return o;

if (o.HasError) { Debug.Log(o.Error.ToString()); }

if (o.HasResult) { Debug.Log(o.Result); }

// other sample(wait until transform.position.y >= 100)

yield return this.transform.ObserveEveryValueChanged(x => x.position).FirstOrDefault(p => p.y >= 100).ToYieldInstruction();

}

Normally, we have to use callbacks when we require a coroutine to return a value. Observable.FromCoroutine can convert coroutines to cancellable IObservable[T] instead.

// public method

public static IObservable<string> GetWWW(string url)

{

// convert coroutine to IObservable

return Observable.FromCoroutine<string>((observer, cancellationToken) => GetWWWCore(url, observer, cancellationToken));

}

// IObserver is a callback publisher

// Note: IObserver's basic scheme is "OnNext\* (OnError | Oncompleted)?"

static IEnumerator GetWWWCore(string url, IObserver<string> observer, CancellationToken cancellationToken)

{

var www = new UnityEngine.WWW(url);

while (!www.isDone && !cancellationToken.IsCancellationRequested)

{

yield return null;

}

if (cancellationToken.IsCancellationRequested) yield break;

if (www.error != null)

{

observer.OnError(new Exception(www.error));

}

else

{

observer.OnNext(www.text);

observer.OnCompleted(); // IObserver needs OnCompleted after OnNext!

}

}

Here are some more examples. Next is a multiple OnNext pattern.

public static IObservable<float> ToObservable(this UnityEngine.AsyncOperation asyncOperation)

{

if (asyncOperation == null) throw new ArgumentNullException("asyncOperation");

return Observable.FromCoroutine<float>((observer, cancellationToken) => RunAsyncOperation(asyncOperation, observer, cancellationToken));

}

static IEnumerator RunAsyncOperation(UnityEngine.AsyncOperation asyncOperation, IObserver<float> observer, CancellationToken cancellationToken)

{

while (!asyncOperation.isDone && !cancellationToken.IsCancellationRequested)

{

observer.OnNext(asyncOperation.progress);

yield return null;

}

if (!cancellationToken.IsCancellationRequested)

{

observer.OnNext(asyncOperation.progress); // push 100%

observer.OnCompleted();

}

}

// usecase

Application.LoadLevelAsync("testscene")

.ToObservable()

.Do(x => Debug.Log(x)) // output progress

.Last() // last sequence is load completed

.Subscribe();

**Using for MultiThreading**

// Observable.Start is start factory methods on specified scheduler

// default is on ThreadPool

var heavyMethod = Observable.Start(() =>

{

// heavy method...

System.Threading.Thread.Sleep(TimeSpan.FromSeconds(1));

return 10;

});

var heavyMethod2 = Observable.Start(() =>

{

// heavy method...

System.Threading.Thread.Sleep(TimeSpan.FromSeconds(3));

return 10;

});

// Join and await two other thread values

Observable.WhenAll(heavyMethod, heavyMethod2)

.ObserveOnMainThread() // return to main thread

.Subscribe(xs =>

{

// Unity can't touch GameObject from other thread

// but use ObserveOnMainThread, you can touch GameObject naturally.

(GameObject.Find("myGuiText")).guiText.text = xs[0] + ":" + xs[1];

});

**DefaultScheduler**

UniRx's default time based operations (Interval, Timer, Buffer(timeSpan), etc) use Scheduler.MainThread as their scheduler. That means most operators (except for Observable.Start) work on a single thread, so ObserverOn isn't needed and thread safety measures can be ignored. This is differet from the standard RxNet implementation but better suited to the Unity environment.

Scheduler.MainThread runs under Time.timeScale's influence. If you want to ignore the time scale, use Scheduler.MainThreadIgnoreTimeScale instead.

**MonoBehaviour triggers**

UniRx can handle MonoBehaviour events with UniRx.Triggers:

using UniRx;

using UniRx.Triggers; // need UniRx.Triggers namespace

public class MyComponent : MonoBehaviour

{

void Start()

{

// Get the plain object

var cube = GameObject.CreatePrimitive(PrimitiveType.Cube);

// Add ObservableXxxTrigger for handle MonoBehaviour's event as Observable

cube.AddComponent<ObservableUpdateTrigger>()

.UpdateAsObservable()

.SampleFrame(30)

.Subscribe(x => Debug.Log("cube"), () => Debug.Log("destroy"));

// destroy after 3 second:)

GameObject.Destroy(cube, 3f);

}

}

Supported triggers are listed in [UniRx.wiki#UniRx.Triggers](https://github.com/neuecc/UniRx/wiki#unirxtriggers).

These can also be handled more easily by directly subscribing to observables returned by extension methods on Component/GameObject. These methods inject ObservableTrigger automaticaly (except for ObservableEventTrigger and ObservableStateMachineTrigger):

using UniRx;

using UniRx.Triggers; // need UniRx.Triggers namespace for extend gameObejct

public class DragAndDropOnce : MonoBehaviour

{

void Start()

{

// All events can subscribe by \*\*\*AsObservable

this.OnMouseDownAsObservable()

.SelectMany(\_ => this.UpdateAsObservable())

.TakeUntil(this.OnMouseUpAsObservable())

.Select(\_ => Input.mousePosition)

.Subscribe(x => Debug.Log(x));

}

}

Previous versions of UniRx provided ObservableMonoBehaviour. This is a legacy interface that is no longer supported. Please use UniRx.Triggers instead.

**Creating custom triggers**

Converting to Observable is the best way to handle Unity events. If the standard triggers supplied by UniRx are not enough, you can create custom triggers. To demonstrate, here's a LongTap trigger for uGUI:

public class ObservableLongPointerDownTrigger : ObservableTriggerBase, IPointerDownHandler, IPointerUpHandler

{

public float IntervalSecond = 1f;

Subject<Unit> onLongPointerDown;

float? raiseTime;

void Update()

{

if (raiseTime != null && raiseTime <= Time.realtimeSinceStartup)

{

if (onLongPointerDown != null) onLongPointerDown.OnNext(Unit.Default);

raiseTime = null;

}

}

void IPointerDownHandler.OnPointerDown(PointerEventData eventData)

{

raiseTime = Time.realtimeSinceStartup + IntervalSecond;

}

void IPointerUpHandler.OnPointerUp(PointerEventData eventData)

{

raiseTime = null;

}

public IObservable<Unit> OnLongPointerDownAsObservable()

{

return onLongPointerDown ?? (onLongPointerDown = new Subject<Unit>());

}

protected override void RaiseOnCompletedOnDestroy()

{

if (onLongPointerDown != null)

{

onLongPointerDown.OnCompleted();

}

}

}

It can be used as easily as the standard triggers:

var trigger = button.AddComponent<ObservableLongPointerDownTrigger>();

trigger.OnLongPointerDownAsObservable().Subscribe();

**Observable Lifecycle Management**

When is OnCompleted called? Subscription lifecycle management is very important to consider when using UniRx. ObservableTriggers call OnCompleted when the GameObject they are attached to is destroyed. Other static generator methods (Observable.Timer, Observable.EveryUpdate, etc...) do not stop automatically, and their subscriptions should be managed manually.

Rx provides some helper methods, such as IDisposable.AddTo which allows you to dispose of several subscriptions at once:

// CompositeDisposable is similar with List<IDisposable>, manage multiple IDisposable

CompositeDisposable disposables = new CompositeDisposable(); // field

void Start()

{

Observable.EveryUpdate().Subscribe(x => Debug.Log(x)).AddTo(disposables);

}

void OnTriggerEnter(Collider other)

{

// .Clear() => Dispose is called for all inner disposables, and the list is cleared.

// .Dispose() => Dispose is called for all inner disposables, and Dispose is called immediately after additional Adds.

disposables.Clear();

}

If you want to automatically Dispose when a GameObjects is destroyed, use AddTo(GameObject/Component):

void Start()

{

Observable.IntervalFrame(30).Subscribe(x => Debug.Log(x)).AddTo(this);

}

AddTo calls facilitate automatic Dispose. If you needs special OnCompleted handling in the pipeline, however, use TakeWhile, TakeUntil, TakeUntilDestroy and TakeUntilDisable instead:

Observable.IntervalFrame(30).TakeUntilDisable(this)

.Subscribe(x => Debug.Log(x), () => Debug.Log("completed!"));

If you handle events, Repeat is an important but dangerous method. It may cause an infinite loop, so handle with care:

using UniRx;

using UniRx.Triggers;

public class DangerousDragAndDrop : MonoBehaviour

{

void Start()

{

this.gameObject.OnMouseDownAsObservable()

.SelectMany(\_ => this.gameObject.UpdateAsObservable())

.TakeUntil(this.gameObject.OnMouseUpAsObservable())

.Select(\_ => Input.mousePosition)

.Repeat() // dangerous!!! Repeat cause infinite repeat subscribe at GameObject was destroyed.(If in UnityEditor, Editor is freezed)

.Subscribe(x => Debug.Log(x));

}

}

UniRx provides an additional safe Repeat method. RepeatSafe: if contiguous "OnComplete" are called repeat stops. RepeatUntilDestroy(gameObject/component), RepeatUntilDisable(gameObject/component) allows to stop when a target GameObject has been destroyed:

this.gameObject.OnMouseDownAsObservable()

.SelectMany(\_ => this.gameObject.UpdateAsObservable())

.TakeUntil(this.gameObject.OnMouseUpAsObservable())

.Select(\_ => Input.mousePosition)

.RepeatUntilDestroy(this) // safety way

.Subscribe(x => Debug.Log(x));

UniRx gurantees hot observable(FromEvent/Subject/ReactiveProperty/UnityUI.AsObservable..., there are like event) have unhandled exception durability. What is it? If subscribe in subcribe, does not detach event.

button.OnClickAsObservable().Subscribe(\_ =>

{

// If throws error in inner subscribe, but doesn't detached OnClick event.

ObservableWWW.Get("htttp://error/").Subscribe(x =>

{

Debug.Log(x);

});

});

This behaviour is sometimes useful such as user event handling.

All class instances provide an ObserveEveryValueChanged method, which watches for changing values every frame:

// watch position change

this.transform.ObserveEveryValueChanged(x => x.position).Subscribe(x => Debug.Log(x));

It's very useful. If the watch target is a GameObject, it will stop observing when the target is destroyed, and call OnCompleted. If the watch target is a plain C# Object, OnCompleted will be called on GC.

**Converting Unity callbacks to IObservables**

Use Subject (or AsyncSubject for asynchronious operations):

public class LogCallback

{

public string Condition;

public string StackTrace;

public UnityEngine.LogType LogType;

}

public static class LogHelper

{

static Subject<LogCallback> subject;

public static IObservable<LogCallback> LogCallbackAsObservable()

{

if (subject == null)

{

subject = new Subject<LogCallback>();

// Publish to Subject in callback

UnityEngine.Application.RegisterLogCallback((condition, stackTrace, type) =>

{

subject.OnNext(new LogCallback { Condition = condition, StackTrace = stackTrace, LogType = type });

});

}

return subject.AsObservable();

}

}

// method is separatable and composable

LogHelper.LogCallbackAsObservable()

.Where(x => x.LogType == LogType.Warning)

.Subscribe();

LogHelper.LogCallbackAsObservable()

.Where(x => x.LogType == LogType.Error)

.Subscribe();

In Unity5, Application.RegisterLogCallback was removed in favor of Application.logMessageReceived, so we can now simply use Observable.FromEvent.

public static IObservable<LogCallback> LogCallbackAsObservable()

{

return Observable.FromEvent<Application.LogCallback, LogCallback>(

h => (condition, stackTrace, type) => h(new LogCallback { Condition = condition, StackTrace = stackTrace, LogType = type }),

h => Application.logMessageReceived += h, h => Application.logMessageReceived -= h);

}

**Stream Logger**

// using UniRx.Diagnostics;

// logger is threadsafe, define per class with name.

static readonly Logger logger = new Logger("Sample11");

// call once at applicationinit

public static void ApplicationInitialize()

{

// Log as Stream, UniRx.Diagnostics.ObservableLogger.Listener is IObservable<LogEntry>

// You can subscribe and output to any place.

ObservableLogger.Listener.LogToUnityDebug();

// for example, filter only Exception and upload to web.

// (make custom sink(IObserver<EventEntry>) is better to use)

ObservableLogger.Listener

.Where(x => x.LogType == LogType.Exception)

.Subscribe(x =>

{

// ObservableWWW.Post("", null).Subscribe();

});

}

// Debug is write only DebugBuild.

logger.Debug("Debug Message");

// or other logging methods

logger.Log("Message");

logger.Exception(new Exception("test exception"));

**Debugging**

Debug operator in UniRx.Diagnostics namespace helps debugging.

// needs Diagnostics using

using UniRx.Diagnostics;

---

// [DebugDump, Normal]OnSubscribe

// [DebugDump, Normal]OnNext(1)

// [DebugDump, Normal]OnNext(10)

// [DebugDump, Normal]OnCompleted()

{

var subject = new Subject<int>();

subject.Debug("DebugDump, Normal").Subscribe();

subject.OnNext(1);

subject.OnNext(10);

subject.OnCompleted();

}

// [DebugDump, Cancel]OnSubscribe

// [DebugDump, Cancel]OnNext(1)

// [DebugDump, Cancel]OnCancel

{

var subject = new Subject<int>();

var d = subject.Debug("DebugDump, Cancel").Subscribe();

subject.OnNext(1);

d.Dispose();

}

// [DebugDump, Error]OnSubscribe

// [DebugDump, Error]OnNext(1)

// [DebugDump, Error]OnError(System.Exception)

{

var subject = new Subject<int>();

subject.Debug("DebugDump, Error").Subscribe();

subject.OnNext(1);

subject.OnError(new Exception());

}

shows sequence element on OnNext, OnError, OnCompleted, OnCancel, OnSubscribe timing to Debug.Log. It enables only #if DEBUG.

**Unity-specific Extra Gems**

// Unity's singleton UiThread Queue Scheduler

Scheduler.MainThreadScheduler

ObserveOnMainThread()/SubscribeOnMainThread()

// Global StartCoroutine runner

MainThreadDispatcher.StartCoroutine(enumerator)

// convert Coroutine to IObservable

Observable.FromCoroutine((observer, token) => enumerator(observer, token));

// convert IObservable to Coroutine

yield return Observable.Range(1, 10).ToYieldInstruction(); // after Unity 5.3, before can use StartAsCoroutine()

// Lifetime hooks

Observable.EveryApplicationPause();

Observable.EveryApplicationFocus();

Observable.OnceApplicationQuit();

**Framecount-based time operators**

UniRx provides a few framecount-based time operators:

| **Method** |
| --- |
| EveryUpdate |
| EveryFixedUpdate |
| EveryEndOfFrame |
| EveryGameObjectUpdate |
| EveryLateUpdate |
| ObserveOnMainThread |
| NextFrame |
| IntervalFrame |
| TimerFrame |
| DelayFrame |
| SampleFrame |
| ThrottleFrame |
| ThrottleFirstFrame |
| TimeoutFrame |
| DelayFrameSubscription |
| FrameInterval |
| FrameTimeInterval |
| BatchFrame |

For example, delayed invoke once:

Observable.TimerFrame(100).Subscribe(\_ => Debug.Log("after 100 frame"));

Every\* Method's execution order is

EveryGameObjectUpdate(in MainThreadDispatcher's Execution Order) ->

EveryUpdate ->

EveryLateUpdate ->

EveryEndOfFrame

EveryGameObjectUpdate invoke from same frame if caller is called before MainThreadDispatcher.Update(I recommend MainThreadDispatcher called first than others(ScriptExecutionOrder makes -32000)  
EveryLateUpdate, EveryEndOfFrame invoke from same frame.  
EveryUpdate, invoke from next frame.

**MicroCoroutine**

MicroCoroutine is memory efficient and fast coroutine worker. This implemantation is based on [Unity blog's 10000 UPDATE() CALLS](http://blogs.unity3d.com/2015/12/23/1k-update-calls/), avoid managed-unmanaged overhead so gets 10x faster iteration. MicroCoroutine is automaticaly used on Framecount-based time operators and ObserveEveryValueChanged.

If you want to use MicroCoroutine instead of standard unity coroutine, use MainThreadDispatcher.StartUpdateMicroCoroutine or Observable.FromMicroCoroutine.

int counter;

IEnumerator Worker()

{

while(true)

{

counter++;

yield return null;

}

}

void Start()

{

for(var i = 0; i < 10000; i++)

{

// fast, memory efficient

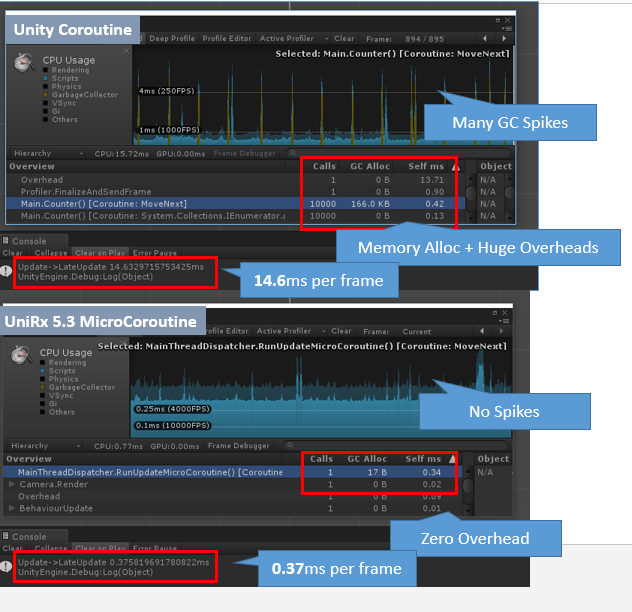
MainThreadDispatcher.StartUpdateMicroCoroutine(Worker());

// slow...

// StartCoroutine(Worker());

}

}

[](https://cloud.githubusercontent.com/assets/46207/15267997/86e9ed5c-1a0c-11e6-8371-14b61a09c72c.png)

MicroCoroutine's limitation, only supports yield return null and update timing is determined start method(StartUpdateMicroCoroutine, StartFixedUpdateMicroCoroutine, StartEndOfFrameMicroCoroutine).

If you combine with other IObservable, you can check completed property like isDone.

IEnumerator MicroCoroutineWithToYieldInstruction()

{

var www = ObservableWWW.Get("http://aaa").ToYieldInstruction();

while (!www.IsDone)

{

yield return null;

}

if (www.HasResult)

{

UnityEngine.Debug.Log(www.Result);

}

}

**uGUI Integration**

UniRx can handle UnityEvents easily. Use UnityEvent.AsObservable to subscribe to events:

public Button MyButton;

// ---

MyButton.onClick.AsObservable().Subscribe(\_ => Debug.Log("clicked"));

Treating Events as Observables enables declarative UI programming.

public Toggle MyToggle;

public InputField MyInput;

public Text MyText;

public Slider MySlider;

// On Start, you can write reactive rules for declaretive/reactive ui programming

void Start()

{

// Toggle, Input etc as Observable (OnValueChangedAsObservable is a helper providing isOn value on subscribe)

// SubscribeToInteractable is an Extension Method, same as .interactable = x)

MyToggle.OnValueChangedAsObservable().SubscribeToInteractable(MyButton);

// Input is displayed after a 1 second delay

MyInput.OnValueChangedAsObservable()

.Where(x => x != null)

.Delay(TimeSpan.FromSeconds(1))

.SubscribeToText(MyText); // SubscribeToText is helper for subscribe to text

// Converting for human readability

MySlider.OnValueChangedAsObservable()

.SubscribeToText(MyText, x => Math.Round(x, 2).ToString());

}

For more on reactive UI programming please consult Sample12, Sample13 and the ReactiveProperty section below.

**ReactiveProperty, ReactiveCollection**

Game data often requires notification. Should we use properties and events (callbacks)? That's often too complex. UniRx provides ReactiveProperty, a lightweight property broker.

// Reactive Notification Model

public class Enemy

{

public ReactiveProperty<long> CurrentHp { get; private set; }

public ReactiveProperty<bool> IsDead { get; private set; }

public Enemy(int initialHp)

{

// Declarative Property

CurrentHp = new ReactiveProperty<long>(initialHp);

IsDead = CurrentHp.Select(x => x <= 0).ToReactiveProperty();

}

}

// ---

// onclick, HP decrement

MyButton.OnClickAsObservable().Subscribe(\_ => enemy.CurrentHp.Value -= 99);

// subscribe from notification model.

enemy.CurrentHp.SubscribeToText(MyText);

enemy.IsDead.Where(isDead => isDead == true)

.Subscribe(\_ =>

{

MyButton.interactable = false;

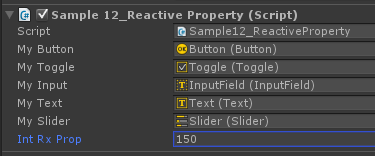
});

You can combine ReactiveProperties, ReactiveCollections and observables returned by UnityEvent.AsObservable. All UI elements are observable.

Generic ReactiveProperties are not serializable or inspecatble in the Unity editor, but UniRx provides specialized subclasses of ReactiveProperty that are. These include classes such as Int/LongReactiveProperty, Float/DoubleReactiveProperty, StringReactiveProperty, BoolReactiveProperty and more (Browse them here: [InspectableReactiveProperty.cs](https://github.com/neuecc/UniRx/blob/master/Assets/Plugins/UniRx/Scripts/UnityEngineBridge/InspectableReactiveProperty.cs)). All are fully editable in the inspector. For custom Enum ReactiveProperty, it's easy to write a custom inspectable ReactiveProperty[T].

If you needs [Multiline] or [Range] attach to ReactiveProperty, you can use MultilineReactivePropertyAttribute and RangeReactivePropertyAttribute instead of Multiline and Range.

The provided derived InpsectableReactiveProperties are displayed in the inspector naturally and notify when their value is changed even when it is changed in the inspector.

[](https://github.com/neuecc/UniRx/blob/master/StoreDocument/RxPropInspector.png)

This functionality is provided by [InspectorDisplayDrawer](https://github.com/neuecc/UniRx/blob/master/Assets/Plugins/UniRx/Scripts/UnityEngineBridge/InspectorDisplayDrawer.cs). You can supply your own custom specialized ReactiveProperties by inheriting from it:

public enum Fruit

{

Apple, Grape

}

[Serializable]

public class FruitReactiveProperty : ReactiveProperty<Fruit>

{

public FruitReactiveProperty()

{

}

public FruitReactiveProperty(Fruit initialValue)

:base(initialValue)

{

}

}

[UnityEditor.CustomPropertyDrawer(typeof(FruitReactiveProperty))]

[UnityEditor.CustomPropertyDrawer(typeof(YourSpecializedReactiveProperty2))] // and others...

public class ExtendInspectorDisplayDrawer : InspectorDisplayDrawer

{

}

If a ReactiveProperty value is only updated within a stream, you can make it read only by using from ReadOnlyReactiveProperty.

public class Person

{

public ReactiveProperty<string> GivenName { get; private set; }

public ReactiveProperty<string> FamilyName { get; private set; }

public ReadOnlyReactiveProperty<string> FullName { get; private set; }

public Person(string givenName, string familyName)

{

GivenName = new ReactiveProperty<string>(givenName);

FamilyName = new ReactiveProperty<string>(familyName);

// If change the givenName or familyName, notify with fullName!

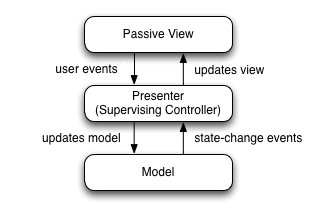
FullName = GivenName.CombineLatest(FamilyName, (x, y) => x + " " + y).ToReadOnlyReactiveProperty();

}

}

**Model-View-(Reactive)Presenter Pattern**

UniRx makes it possible to implement the MVP(MVRP) Pattern.

[](https://github.com/neuecc/UniRx/blob/master/StoreDocument/MVP_Pattern.png)

Why should we use MVP instead of MVVM? Unity doesn't provide a UI binding mechanism and creating a binding layer is too complex and loss and affects performance. Still, Views need updating. Presenters are aware of their view's components and can update them. Although there is no real binding, Observables enables subscription to notification, which can act much like the real thing. This pattern is called a Reactive Presenter:

// Presenter for scene(canvas) root.

public class ReactivePresenter : MonoBehaviour

{

// Presenter is aware of its View (binded in the inspector)

public Button MyButton;

public Toggle MyToggle;

// State-Change-Events from Model by ReactiveProperty

Enemy enemy = new Enemy(1000);

void Start()

{

// Rx supplies user events from Views and Models in a reactive manner

MyButton.OnClickAsObservable().Subscribe(\_ => enemy.CurrentHp.Value -= 99);

MyToggle.OnValueChangedAsObservable().SubscribeToInteractable(MyButton);

// Models notify Presenters via Rx, and Presenters update their views

enemy.CurrentHp.SubscribeToText(MyText);

enemy.IsDead.Where(isDead => isDead == true)

.Subscribe(\_ =>

{

MyToggle.interactable = MyButton.interactable = false;

});

}

}

// The Model. All property notify when their values change

public class Enemy

{

public ReactiveProperty<long> CurrentHp { get; private set; }

public ReactiveProperty<bool> IsDead { get; private set; }

public Enemy(int initialHp)

{

// Declarative Property

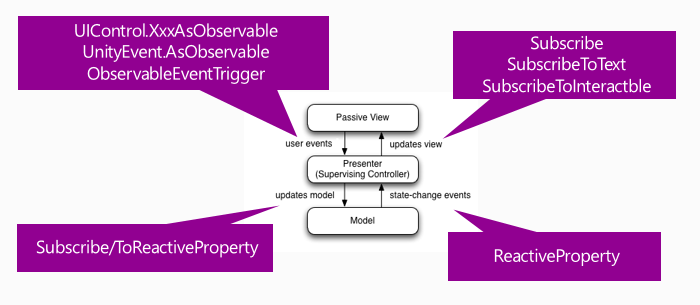
CurrentHp = new ReactiveProperty<long>(initialHp);

IsDead = CurrentHp.Select(x => x <= 0).ToReactiveProperty();

}

}

A View is a scene, that is a Unity hierarchy. Views are associated with Presenters by the Unity Engine on initialize. The XxxAsObservable methods make creating event signals simple, without any overhead. SubscribeToText and SubscribeToInteractable are simple binding-like helpers. These may be simple tools, but they are very powerful. They feel natural in the Unity environment and provide high performance and a clean architecture.

[](https://github.com/neuecc/UniRx/blob/master/StoreDocument/MVRP_Loop.png)

V -> RP -> M -> RP -> V completely connected in a reactive way. UniRx provides all of the adaptor methods and classes, but other MVVM(or MV\*) frameworks can be used instead. UniRx/ReactiveProperty is only simple toolkit.

GUI programming also benefits from ObservableTriggers. ObservableTriggers convert Unity events to Observables, so the MV(R)P pattern can be composed using them. For example, ObservableEventTrigger converts uGUI events to Observable:

var eventTrigger = this.gameObject.AddComponent<ObservableEventTrigger>();

eventTrigger.OnBeginDragAsObservable()

.SelectMany(\_ => eventTrigger.OnDragAsObservable(), (start, current) => UniRx.Tuple.Create(start, current))

.TakeUntil(eventTrigger.OnEndDragAsObservable())

.RepeatUntilDestroy(this)

.Subscribe(x => Debug.Log(x));

**(Obsolete)PresenterBase**

Note: PresenterBase works enough, but too complex.  
You can use simple Initialize method and call parent to child, it works for most scenario.  
So I don't recommend using PresenterBase, sorry.

**ReactiveCommand, AsyncReactiveCommand**

ReactiveCommand abstraction of button command with boolean interactable.

public class Player

{

public ReactiveProperty<int> Hp;

public ReactiveCommand Resurrect;

public Player()

{

Hp = new ReactiveProperty<int>(1000);

// If dead, can not execute.

Resurrect = Hp.Select(x => x <= 0).ToReactiveCommand();

// Execute when clicked

Resurrect.Subscribe(\_ =>

{

Hp.Value = 1000;

});

}

}

public class Presenter : MonoBehaviour

{

public Button resurrectButton;

Player player;

void Start()

{

player = new Player();

// If Hp <= 0, can't press button.

player.Resurrect.BindTo(resurrectButton);

}

}

AsyncReactiveCommand is a variation of ReactiveCommand that CanExecute(in many cases bind to button's interactable) is changed to false until asynchronous execution was finished.

public class Presenter : MonoBehaviour

{

public UnityEngine.UI.Button button;

void Start()

{

var command = new AsyncReactiveCommand();

command.Subscribe(\_ =>

{

// heavy, heavy, heavy method....

return Observable.Timer(TimeSpan.FromSeconds(3)).AsUnitObservable();

});

// after clicked, button shows disable for 3 seconds

command.BindTo(button);

// Note:shortcut extension, bind aync onclick directly

button.BindToOnClick(\_ =>

{

return Observable.Timer(TimeSpan.FromSeconds(3)).AsUnitObservable();

});

}

}

AsyncReactiveCommand has three constructor.

* () - CanExecute is changed to false until async execution finished
* (IObservable<bool> canExecuteSource) - Mixed with empty, CanExecute becomes true when canExecuteSource send to true and does not executing
* (IReactiveProperty<bool> sharedCanExecute) - share execution status between multiple AsyncReactiveCommands, if one AsyncReactiveCommand is executing, other AsyncReactiveCommands(with same sharedCanExecute property) becomes CanExecute false until async execution finished

public class Presenter : MonoBehaviour

{

public UnityEngine.UI.Button button1;

public UnityEngine.UI.Button button2;

void Start()

{

// share canExecute status.

// when clicked button1, button1 and button2 was disabled for 3 seconds.

var sharedCanExecute = new ReactiveProperty<bool>();

button1.BindToOnClick(sharedCanExecute, \_ =>

{

return Observable.Timer(TimeSpan.FromSeconds(3)).AsUnitObservable();

});

button2.BindToOnClick(sharedCanExecute, \_ =>

{

return Observable.Timer(TimeSpan.FromSeconds(3)).AsUnitObservable();

});

}

}

**MessageBroker, AsyncMessageBroker**

MessageBroker is Rx based in-memory pubsub system filtered by type.

public class TestArgs

{

public int Value { get; set; }

}

---

// Subscribe message on global-scope.

MessageBroker.Default.Receive<TestArgs>().Subscribe(x => UnityEngine.Debug.Log(x));

// Publish message

MessageBroker.Default.Publish(new TestArgs { Value = 1000 });

AsyncMessageBroker is variation of MessageBroker, can await Publish call.

AsyncMessageBroker.Default.Subscribe<TestArgs>(x =>

{

// show after 3 seconds.

return Observable.Timer(TimeSpan.FromSeconds(3))

.ForEachAsync(\_ =>

{

UnityEngine.Debug.Log(x);

});

});

AsyncMessageBroker.Default.PublishAsync(new TestArgs { Value = 3000 })

.Subscribe(\_ =>

{

UnityEngine.Debug.Log("called all subscriber completed");

});

**UniRx.Toolkit**

UniRx.Toolkit includes serveral Rx-ish tools. Currently includes ObjectPool and AsyncObjectPool. It can Rent, Return and PreloadAsync for fill pool before rent operation.

// sample class

public class Foobar : MonoBehaviour

{

public IObservable<Unit> ActionAsync()

{

// heavy, heavy, action...

return Observable.Timer(TimeSpan.FromSeconds(3)).AsUnitObservable();

}

}

public class FoobarPool : ObjectPool<Foobar>

{

readonly Foobar prefab;

readonly Transform hierarchyParent;

public FoobarPool(Foobar prefab, Transform hierarchyParent)

{

this.prefab = prefab;

this.hierarchyParent = hierarchyParent;

}

protected override Foobar CreateInstance()

{

var foobar = GameObject.Instantiate<Foobar>(prefab);

foobar.transform.SetParent(hierarchyParent);

return foobar;

}

// You can overload OnBeforeRent, OnBeforeReturn, OnClear for customize action.

// In default, OnBeforeRent = SetActive(true), OnBeforeReturn = SetActive(false)

// protected override void OnBeforeRent(Foobar instance)

// protected override void OnBeforeReturn(Foobar instance)

// protected override void OnClear(Foobar instance)

}

public class Presenter : MonoBehaviour

{

FoobarPool pool = null;

public Foobar prefab;

public Button rentButton;

void Start()

{

pool = new FoobarPool(prefab, this.transform);

rentButton.OnClickAsObservable().Subscribe(\_ =>

{

var foobar = pool.Rent();

foobar.ActionAsync().Subscribe(\_\_ =>

{

// if action completed, return to pool

pool.Return(foobar);

});

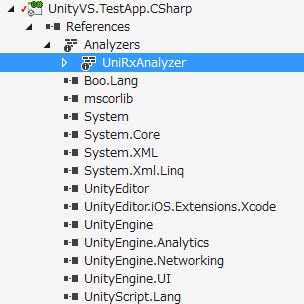
});

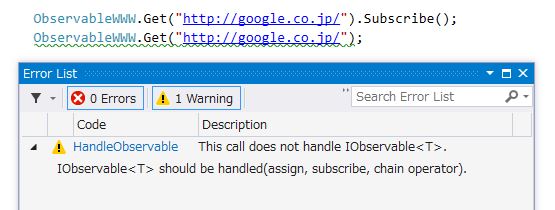
}

}

**Visual Studio Analyzer**

For Visual Studio 2015 users, a custom analyzer, UniRxAnalyzer, is provided. It can, for example, detect when streams aren't subscribed to.

[](https://github.com/neuecc/UniRx/blob/master/StoreDocument/AnalyzerReference.jpg)

[](https://github.com/neuecc/UniRx/blob/master/StoreDocument/VSAnalyzer.jpg)

ObservableWWW doesn't fire until it's subscribed to, so the analyzer warns about incorrect usage. It can be downloaded from NuGet.

* Install-Package [UniRxAnalyzer](http://www.nuget.org/packages/UniRxAnalyzer)

Please submit new analyzer ideas on GitHub Issues!

**Samples**

See [UniRx/Examples](https://github.com/neuecc/UniRx/tree/master/Assets/Plugins/UniRx/Examples)

The samples demonstrate how to do resource management (Sample09\_EventHandling), what is the MainThreadDispatcher, among other things.

**Windows Store/Phone App (NETFX\_CORE)**

Some interfaces, such as UniRx.IObservable<T> and System.IObservable<T>, cause conflicts when submitting to the Windows Store App. Therefore, when using NETFX\_CORE, please refrain from using such constructs as UniRx.IObservable<T> and refer to the UniRx components by their short name, without adding the namespace. This solves the conflicts.

**DLL Separation**

If you want to pre-build UniRx, you can build own dll. clone project and open UniRx.sln, you can see UniRx, it is fullset separated project of UniRx. You should define compile symbol like UNITY;UNITY\_5\_4\_OR\_NEWER;UNITY\_5\_4\_0;UNITY\_5\_4;UNITY\_5; + UNITY\_EDITOR, UNITY\_IPHONE or other platform symbol. We can not provides pre-build binary to release page, asset store because compile symbol is different each other.

**UPM Package**

After Unity 2019.3.4f1, Unity 2020.1a21, that support path query parameter of git package. You can add https://github.com/neuecc/UniRx.git?path=Assets/Plugins/UniRx/Scripts to Package Manager

or add "com.neuecc.unirx": "https://github.com/neuecc/UniRx.git?path=Assets/Plugins/UniRx/Scripts" to Packages/manifest.json.

**Reference**

* [UniRx/wiki](https://github.com/neuecc/UniRx/wiki)

UniRx API documents.

* [ReactiveX](http://reactivex.io/)

The home of ReactiveX. [Introduction](http://reactivex.io/intro.html), [All operators](http://reactivex.io/documentation/operators.html) are illustrated with graphical marble diagrams, there makes easy to understand. And UniRx is official [ReactiveX Languages](http://reactivex.io/languages.html).

* [Introduction to Rx](http://introtorx.com/)

A great online tutorial and eBook.

* [Beginner's Guide to the Reactive Extensions](http://msdn.microsoft.com/en-us/data/gg577611)

Many videos, slides and documents for Rx.NET.

* [The future of programming technology in Unity - UniRx -(JPN)](http://www.slideshare.net/torisoup/unity-unirx)
  + [Korean translation](http://www.slideshare.net/agebreak/160402-unirx)

Intro slide by [@torisoup](https://github.com/torisoup)

* [Reactive Programming, ​Unity 3D and you](http://slides.com/sammegidov/unirx#/)
  + [Repository of UniRxSimpleGame](https://github.com/Xerios/UniRxSimpleGame)

Intro slide and sample game by [@Xerios](https://github.com/Xerios)

* [GDC 2016 Sessions of Adventure Capialist](https://www.youtube.com/watch?v=j3YhG91mPsU&feature=youtu.be&t=9m12s)

How to integrate with PlayFab API

**Help & Contribute**

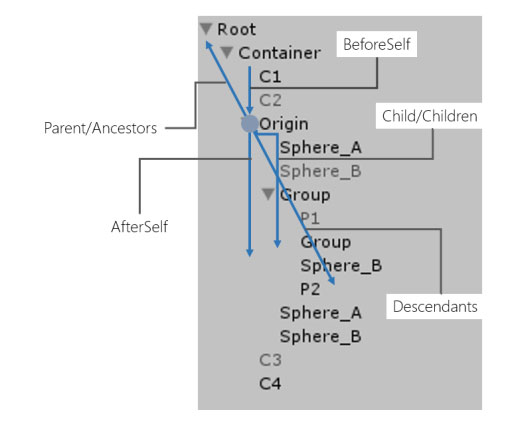
Support thread on the Unity forum. Ask me any question - <http://forum.unity3d.com/threads/248535-UniRx-Reactive-Extensions-for-Unity>

Become a backer, Sponsored, one time donation are welcome, we're using [Open Collective - UniRx](https://opencollective.com/unirx/)

We welcome any contributions, be they bug reports, requests or pull request.  
Please consult and submit your reports or requests on GitHub issues.  
Source code is available in Assets/Plugins/UniRx/Scripts.

**Author's other Unity + LINQ Assets**

[LINQ to GameObject](https://github.com/neuecc/LINQ-to-GameObject-for-Unity/) is a group of GameObject extensions for Unity that allows traversing the hierarchy and appending GameObject to it like LINQ to XML. It's free and opensource on GitHub.

[](https://raw.githubusercontent.com/neuecc/LINQ-to-GameObject-for-Unity/master/Images/axis.jpg)