R3

The new future of dotnet/reactive and UniRx, which support many platforms including Unity, Godot, Avalonia, WPF, WinForms, Stride, LogicLooper, MAUI, MonoGame.

I have over 10 years of experience with Rx, experience in implementing a custom Rx runtime (UniRx) for game engine, and experience in implementing an asynchronous runtime (UniTask) for game engine. Based on those experiences, I came to believe that there is a need to implement a new Reactive Extensions for .NET, one that reflects modern C# and returns to the core values of Rx.

- Stopping the pipeline at OnError is a billion-dollar mistake.
- IScheduler is the root of poor performance.
- Frame-based operations, a missing feature in Rx, are especially important in game engines.
- Single asynchronous operations should be entirely left to async/await.
- Synchronous APIs should not be implemented.
- Query syntax is a bad notation except for SQL.
- The Necessity of a subscription list to prevent subscription leaks (similar to a Parallel Debugger)
- Backpressure should be left to IAsyncEnumerable and Channels.
- For distributed processing and queries, there are GraphQL, Kubernetes, Orleans, Akka.NET, gRPC, MagicOnion.

In other words, LINQ is not for EveryThing, and we believe that the essence of Rx lies in the processing of in-memory messaging (LINQ to Events), which will be our focus. We are not concerned with communication processes like Reactive Streams.

To address the shortcomings of dotnet/reactive, we have made changes to the core interfaces. In recent years, Rx-like frameworks optimized for language features, such as Kotlin Flow and Swift Combine, have been standardized. C# has also evolved significantly, now at C# 12, and we believe there is a need for an Rx that aligns with the latest C#.

Improving performance was also a theme in the reimplementation. For example, this is the result of the terrible performance of IScheudler and the performance difference caused by its removal.

Method	Mean	Error	Gen0	Allocated
R3	17.27 us	NA	-	56 B
DotnetReactive_Immediate	559.99 us	NA	42.9688	720927 B
DotnetReactive_CurrentThread	780.70 us	NA	38.0859	640456 B

`Observable.Range(1, 10000).Subscribe()`

You can also see interesting results in allocations with the addition and deletion to Subject.

Method	Mean	Error	Gen0	Gen1	Gen2	Allocated
R3	7.923 ms	NA	-	-	-	988.59 KB
DotnetReactive	38.670 ms	NA	47000.0000	10000.0000	1000.0000	782500.73 KB

`x10000 subject.Subscribe() -> x10000 subscription.Dispose()`

This is because dotnet/reactive has adopted ImmutableArray (or its equivalent) for Subject, which results in the allocation of a new array every time one is added or removed. Depending on the design of the application, a large number of subscriptions can occur (we have seen this especially in the complexity of games), which can be a critical issue. In R3, we have devised a way to achieve high performance while avoiding ImmutableArray.

Core Interface

This library is distributed via NuGet, supporting .NET Standard 2.0, .NET Standard 2.1, .NET 6(.NET 7) and .NET 8 or above.

> PM> Install-Package R3

Some platforms(WPF, Avalonia, Unity, Godot) requires additional step to install. Please see Platform Supports section in below.

R3 code is almostly same as standard Rx. Make the Observable via factory methods(Timer, Interval, FromEvent, Subject, etc...) and chain operator via LINQ methods. Therefore, your knowledge about Rx and documentation on Rx can be almost directly applied. If you are new to Rx, the ReactiveX website and Introduction to Rx.NET would be useful resources for reference.

```
using R3;

var subscription = Observable.Interval(TimeSpan.FromSeconds(1))
    .Select((_, i) => i)
```

```
    .Where(x => x % 2 == 0)
    .Subscribe(x => Console.WriteLine($"Interval:{x}"));

var cts = new CancellationTokenSource();
_ = Task.Run(() => { Console.ReadLine(); cts.Cancel(); });

await Observable.Timer(TimeSpan.FromSeconds(1), TimeSpan.FromSeconds(3))
    .TakeUntil(cts.Token)
    .ForEachAsync(x => Console.WriteLine($"Timer"));

subscription.Dispose();
```

The surface API remains the same as normal Rx, but the interfaces used internally are different and are not `IObservable<T>/IObserver<T>`.

`IObservable<T>` being the dual of `IEnumerable<T>` is a beautiful definition, but it was not very practical in use.

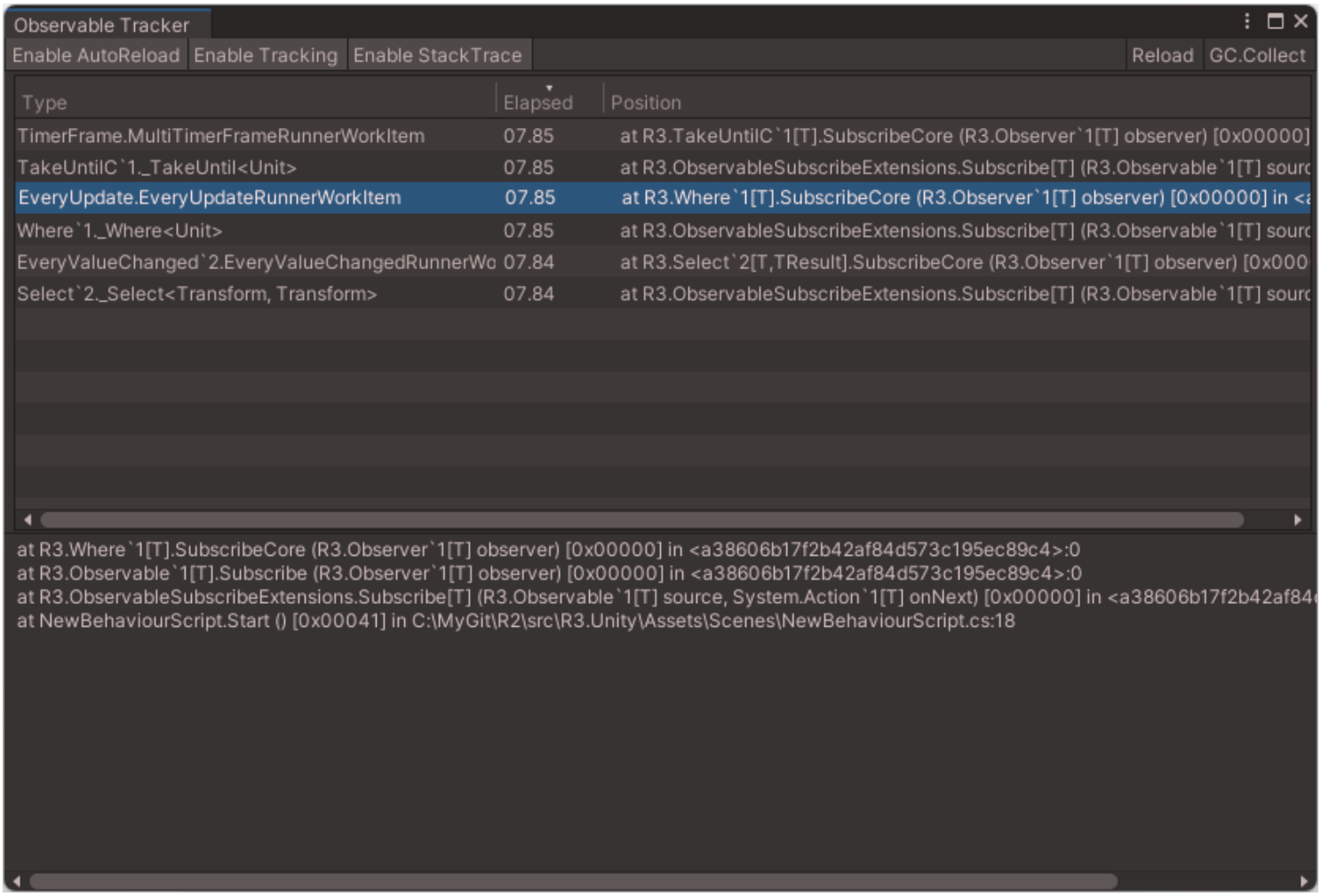
```
public abstract class Observable<T>
{
    public IDisposable Subscribe(Observer<T> observer);
}

public abstract class Observer<T> : IDisposable
{
    public void OnNext(T value);
    public void OnErrorResume(Exception error);
    public void OnCompleted(Result result); // Result is () | Exception
}
```

The biggest difference is that in normal Rx, when an exception occurs in the pipeline, it flows to `OnError` and the subscription is unsubscribed, but in R3, it flows to `OnErrorResume` and the subscription is not unsubscribed.

I consider the automatic unsubscription by OnError to be a bad design for event handling. It's very difficult and risky to resolve it within an operator like Retry, and it also led to poor performance (there are many questions and complex answers about stopping and resubscribing all over the world). Also, converting OnErrorResume to OnError(OnCompleted(Result.Failure)) is easy and does not degrade performance, but the reverse is impossible. Therefore, the design was changed to not stop by default and give users the choice to stop.

Since the original Rx contract was `OnError | OnCompleted`, it was changed to `OnCompleted(Result result)` to consolidate into one method. Result is a readonly struct with two states: `Failure(Exception) | Success()`.

The reason for changing to an abstract class instead of an interface is that Rx has implicit complex contracts that interfaces do not guarantee. By making it an abstract class, we fully controlled the behavior of Subscribe, OnNext, and Dispose. This made it possible to manage the list of all subscriptions and prevent subscription leaks.



Subscription leaks are a common problem in applications with long lifecycles, such as GUIs or games. Tracking all subscriptions makes it easy to prevent leaks.

Internally, when subscribing, an Observer is always linked to the target Observable and doubles as a Subscription. This ensures that Observers are reliably connected from top to bottom, making tracking certain and clear that they are released on OnCompleted/Dispose. In terms of performance, because the Observer itself always becomes a Subscription, there is no need for unnecessary IDisposable allocations.

TimeProvider instead of IScheduler

In traditional Rx, `IScheduler` was used as an abstraction for time-based processing, but in R3, we have discontinued its use and instead opted for the TimeProvider introduced in .NET 8. For example, the operators are defined as follows:

```
public static Observable<Unit> Interval(TimeSpan period, TimeProvider timeProvider);
public static Observable<T> Delay<T>(this Observable<T> source, TimeSpan dueTime,
TimeProvider timeProvider)
public static Observable<T> Debounce<T>(this Observable<T> source, TimeSpan
timeSpan, TimeProvider timeProvider) // same as Throttle in dotnet/reactive
```

Originally, `IScheduler` had performance issues, and the internal implementation of dotnet/reactive was peppered with code that circumvented these issues using `PeriodicTimer` and `IStopwatch`, leading to unnecessary complexity. These can be better expressed with TimeProvider (`TimeProvider.CreateTimer()`, `TimeProvider.GetTimestamp()`).

While TimeProvider is an abstraction for asynchronous operations, excluding the Fake for testing purposes, `IScheduler` included synchronous schedulers like `ImmediateScheduler` and `CurrentThreadScheduler`. However, these were also meaningless as applying them to time-based operators would cause blocking, and `CurrentThreadScheduler` had poor performance.

Method	Mean	Error	Gen0	Allocated
R3	17.27 us	NA	-	56 B
DotnetReactive_Immediate	559.99 us	NA	42.9688	720927 B
DotnetReactive_CurrentThread	780.70 us	NA	38.0859	640456 B

`Observable.Range(1, 10000).Subscribe()`

In R3, anything that requires synchronous execution (like Range) is treated as Immediate, and everything else is considered asynchronous and handled through TimeProvider.

As for the implementation of TimeProvider, the standard TimeProvider.System using the ThreadPool is the default. For unit testing, FakeTimeProvider (Microsoft.Extensions.TimeProvider.Testing) is available. Additionally, many TimeProvider implementations are provided for different platforms, such as DispatcherTimerProvider for WPF and UpdateTimerProvider for Unity, enhancing ease of use tailored to each platform.

Frame based operations

In GUI applications, there's the message loop, and in game engines, there's the game loop. Platforms that operate based on loops are not uncommon. The idea of executing something after a few seconds or frames fits very well with Rx. Just as time has been abstracted

through TimeProvider, we introduced a layer of abstraction for frames called FrameProvider, and added frame-based operators corresponding to all methods that accept TimeProvider.

```
public static Observable<Unit> IntervalFrame(int periodFrame,
FrameProvider frameProvider);
public static Observable<T> DelayFrame<T>(this Observable<T> source, int frameCount,
FrameProvider frameProvider)
public static Observable<T> DebounceFrame<T>(this Observable<T> source, int
frameCount, FrameProvider frameProvider)
```

The effectiveness of frame-based processing has been proven in Unity's Rx implementation, [neuecc/UniRx](), which is one of the reasons why UniRx has gained strong support.

There are also several operators unique to frame-based processing.

```
// push OnNext every frame.
Observable.EveryUpdate().Subscribe(x => Console.WriteLine(x));

// take value until next frame
_eventSoure.TakeUntil(Obserable.NextFrame()).Subscribe();

// polling value changed
Observable.EveryValueChanged(this, x => x.Width).Subscribe(x => WidthText.Text
= x.ToString());
Observable.EveryValueChanged(this, x => x.Height).Subscribe(x => HeightText.Text
= x.ToString());
```

`EveryValueChanged` could be interesting, as it converts properties without Push-based notifications like `INotifyPropertyChanged`.

`

Subjects(ReactiveProperty)

In R3, there are four types of Subjects: `Subject`, `ReactiveProperty`, `ReplaySubject`, and `ReplayFrameSubject`. `ReactiveProperty` corresponds to what would be a `BehaviorSubject`, but with the added functionality of eliminating duplicate values. Since you can choose to enable or disable duplicate elimination, it effectively becomes a superior alternative to `BehaviorSubject`, leading to the removal of `BehaviorSubject`.

`ReactiveProperty` has equivalents in other frameworks as well, such as Android LiveData and Kotlin StateFlow, particularly effective for data binding in UI contexts. In .NET, there is a library called runceel/ReactiveProperty, which I originally created.

Unlike dotnet/reactive's Subject, all Subjects in R3 (Subject, ReactiveProperty, ReplaySubject, ReplayFrameSubject) are designed to call OnCompleted upon disposal. This is because R3 is designed with a focus on subscription management and unsubscription. By calling OnCompleted, it ensures that all subscriptions are unsubscribed from the Subject, the upstream source of events, by default. If you wish to avoid calling OnCompleted, you can do so by calling `Dispose(false)`.

`ReactiveProperty` is mutable, but it can be converted to a read-only `ReadOnlyReactiveProperty`. Following the guidance for the Android UI Layer, the Kotlin code below is

```
class NewsViewModel(...) : ViewModel() {

    private val _uiState = MutableStateFlow(NewsUiState())
    val uiState: StateFlow<NewsUiState> = _uiState.asStateFlow()
```

```
    ...
}
```

can be adapted to the following R3 code.

```
class NewsViewModel
{
    ReactiveProperty<NewsUiState> _uiState = new(new NewsUiState());
    public ReadOnlyReactiveProperty<NewsUiState> UiState => _uiState;
}
```

In R3, we use a combination of a mutable private field and a readonly public property.

By inheriting `ReactiveProperty` and overriding `OnValueChanging` and `OnValueChanged`, you can customize behavior, such as adding validation.

```
// Since the primary constructor sets values to fields before calling base, it is
safe to call OnValueChanging in the base constructor.
public sealed class ClampedReactiveProperty<T>(T initialValue, T min, T max)
    : ReactiveProperty<T>(initialValue) where T : IComparable<T>
{
    private static IComparer<T> Comparer { get; } = Comparer<T>.Default;

    protected override void OnValueChanging(ref T value)
    {
        if (Comparer.Compare(value, min) < 0)
        {
            value = min;
        }
        else if (Comparer.Compare(value, max) > 0)
        {
            value = max;
        }
    }
}

// For regular constructors, please set `callOnValueChangeInBaseConstructor` to
false and manually call it once to correct the value.
public sealed class ClampedReactiveProperty2<T>
    : ReactiveProperty<T> where T : IComparable<T>
{
    private static IComparer<T> Comparer { get; } = Comparer<T>.Default;

    readonly T min, max;
```

```
    // callOnValueChangeInBaseConstructor to avoid OnValueChanging call before min,
max set.
    public ClampedReactiveProperty2(T initialValue, T min, T max)
        : base(initialValue, EqualityComparer<T>.Default,
callOnValueChangeInBaseConstructor: false)
    {
        this.min = min;
        this.max = max;

        // modify currentValue manually
        OnValueChanging(ref GetValueRef());
    }

    protected override void OnValueChanging(ref T value)
    {
        if (Comparer.Compare(value, min) < 0)
        {
            value = min;
        }
        else if (Comparer.Compare(value, max) > 0)
        {
            value = max;
        }
    }
}
```

Additionally, `ReactiveProperty` supports serialization with `System.Text.JsonSerializer` in .NET 6 and above. For earlier versions, you need to implement `ReactivePropertyJsonConverterFactory` under the existing implementation and add it to the Converter.

Disposable

To bundle multiple IDisposables (Subscriptions), it's good to use Disposable's methods. In R3, depending on the performance,

```
Disposable.Combine(IDisposable d1, ..., IDisposable d8);
Disposable.Combine(params IDisposable[]);
Disposable.CreateBuilder();
CompositeDisposable
DisposableBag
```

five types are available for use. In terms of performance advantages, the order is `Combine(d1,...,d8) (>= CreateBuilder) > Combine(IDisposable[]) >= CreateBuilder > DisposableBag > CompositeDisposable`.

When the number of subscriptions is statically determined, Combine offers the best performance. Internally, for less than 8 arguments, it uses fields, and for 9 or more arguments, it uses an array, making Combine especially efficient for 8 arguments or less.

```
public partial class MainWindow : Window
{
    IDisposable disposable;

    public MainWindow()
    {
        var d1 = Observable.IntervalFrame(1).Subscribe();
        var d2 = Observable.IntervalFrame(1).Subscribe();
        var d3 = Observable.IntervalFrame(1).Subscribe();

        disposable = Disposable.Combine(d1, d2, d3);
    }

    protected override void OnClosed(EventArgs e)
    {
        disposable.Dispose();
    }
}
```

If there are many subscriptions and it's cumbersome to hold each one in a variable, `CreateBuilder` can be used instead. At build time, it combines according to the number of items added to it. Since the Builder itself is a struct, there are no allocations.

```
public partial class MainWindow : Window
{
    IDisposable disposable;

    public MainWindow()
    {
        var d = Disposable.CreateBuilder();
        Observable.IntervalFrame(1).Subscribe().AddTo(ref d);
        Observable.IntervalFrame(1).Subscribe().AddTo(ref d);
        Observable.IntervalFrame(1).Subscribe().AddTo(ref d);

        disposable = d.Build();
    }
```

```
    protected override void OnClosed(EventArgs e)
    {
        disposable.Dispose();
    }
}
```

For dynamically added items, using `DisposableBag` is advisable. This is an add-only struct with only `Add/Clear/Dispose` methods. It can be used relatively quickly and with low allocation by holding it in a class field and passing it around by reference. However, it is not thread-safe.

```
public partial class MainWindow : Window
{
    DisposableBag disposable; // DisposableBag is struct, no need new and don't copy

    public MainWindow()
    {
        Observable.IntervalFrame(1).Subscribe().AddTo(ref disposable);
        Observable.IntervalFrame(1).Subscribe().AddTo(ref disposable);
        Observable.IntervalFrame(1).Subscribe().AddTo(ref disposable);
    }

    void OnClick()
    {
        Observable.IntervalFrame(1).Subscribe().AddTo(ref disposable);
    }

    protected override void OnClosed(EventArgs e)
    {
        disposable.Dispose();
    }
}
```

`CompositeDisposable` is a class that also supports `Remove` and is thread-safe. It is the most feature-rich, but comparatively, it has the lowest performance.

```
public partial class MainWindow : Window
{
    CompositeDisposable disposable = new CompositeDisposable();

    public MainWindow()
    {
        Observable.IntervalFrame(1).Subscribe().AddTo(disposable);
```

```
        Observable.IntervalFrame(1).Subscribe().AddTo(disposable);
        Observable.IntervalFrame(1).Subscribe().AddTo(disposable);
    }

    void OnClick()
    {
        Observable.IntervalFrame(1).Subscribe().AddTo(disposable);
    }

    protected override void OnClosed(EventArgs e)
    {
        disposable.Dispose();
    }
}
```

Additionally, there are other utilities for Disposables as follows.

```
Disposable.Create(Action);
Disposable.Dispose(...);
SingleAssignmentDisposable
SingleAssignmentDisposableCore // struct
SerialDisposable
SerialDisposableCore// struct
```

Subscription Management

Managing subscriptions is one of the most crucial aspects of Rx, and inadequate management can lead to memory leaks. There are two patterns for unsubscribing in Rx. One is by disposing of the IDisposable (Subscription) returned by Subscribe. The other is by receiving OnCompleted.

In R3, to enhance subscription cancellation on both fronts, it's now possible to bundle subscriptions using a variety of Disposable classes for Subscriptions, and for OnCompleted, the upstream side of events (such as Subject or Factory) has been made capable of emitting OnCompleted. Especially, Factories that receive a TimeProvider or FrameProvider can now take a CancellationToken.

```
public static Observable<Unit> Interval(TimeSpan period, TimeProvider timeProvider,
CancellationToken cancellationToken)
public static Observable<Unit> EveryUpdate(FrameProvider frameProvider,
CancellationToken cancellationToken)
```

When cancelled, OnCompleted is sent, and all subscriptions are unsubscribed.

ObservableTracker

R3 incorporates a system called ObservableTracker. When activated, it allows you to view all subscription statuses.

```
ObservableTracker.EnableTracking = true; // default is false
ObservableTracker.EnableStackTrace = true;

using var d = Observable.Interval(TimeSpan.FromSeconds(1))
    .Where(x => true)
    .Take(10000)
    .Subscribe();

// check subscription
ObservableTracker.ForEachActiveTask(x =>
{
    Console.WriteLine(x);
});
```

```
TrackingState { TrackingId = 1, FormattedType = Timer._Timer, AddTime = 2024/01/09
4:11:39, StackTrace =... }
TrackingState { TrackingId = 2, FormattedType = Where`1._Where<Unit>, AddTime =
2024/01/09 4:11:39, StackTrace =... }
TrackingState { TrackingId = 3, FormattedType = Take`1._Take<Unit>, AddTime =
2024/01/09 4:11:39, StackTrace =... }
```

Besides directly calling `ForEachActiveTask`, making it more accessible through a GUI can make it easier to check for subscription leaks. Currently, there is an integrated GUI for Unity, and there are plans to provide a screen using Blazor for other platforms.

ObservableSystem, UnhandledExceptionHandler

For time-based operators that do not specify a TimeProvider or FrameProvider, the default Provider of ObservableSystem is used. This is settable, so if there is a platform-specific Provider (for example, DispatcherTimeProvider in WPF), you can swap it out to create a more user-friendly environment.

```
public static class ObservableSystem
{
    public static TimeProvider DefaultTimeProvider { get; set; }
= TimeProvider.System;
    public static FrameProvider DefaultFrameProvider { get; set; } =
new NotSupportedFrameProvider();
```

```
    static Action<Exception> unhandledException = DefaultUnhandledExceptionHandler;

    // Prevent +=, use Set and Get method.
    public static void RegisterUnhandledExceptionHandler(Action<Exception>
unhandledExceptionHandler)
    {
        unhandledException = unhandledExceptionHandler;
    }

    public static Action<Exception> GetUnhandledExceptionHandler()
    {
        return unhandledException;
    }

    static void DefaultUnhandledExceptionHandler(Exception exception)
    {
        Console.WriteLine("R3 UnhandleException: " + exception.ToString());
    }
}
```

In CUI environments, by default, the FrameProvider will throw an exception. If you want to use FrameProvider in a CUI environment, you can set either `NewThreadSleepFrameProvider`, which sleeps in a new thread for a specified number of seconds, or `TimerFrameProvider`, which executes every specified number of seconds.

UnhandledExceptionHandler

When an exception passes through OnErrorResume and is not ultimately handled by Subscribe, the UnhandledExceptionHandler of ObservableSystem is called. This can be set with `RegisterUnhandledExceptionHandler`. By default, it writes to `Console.WriteLine`, but it may need to be changed to use `ILogger` or something else as required.

Result Handling

The `Result` received by OnCompleted has a field `Exception?`, where it's null in case of success and contains the Exception in case of failure.

```
// Typical processing code example
void OnCompleted(Result result)
{
    if (result.IsFailure)
    {
        // do failure
        _ = result.Exception;
```

```
    }
    else // result.IsSuccess
    {
        // do success
    }
}
```

To generate a `Result`, in addition to using `Result.Success` and `Result.Failure(exception)`, Observer has OnCompleted() and OnCompleted(exception) as shortcuts for Success and Failure, respectively.

```
observer.OnCompleted(Result.Success);
observer.OnCompleted(Result.Failure(exception));

observer.OnCompleted(); // same as Result.Success
observer.OnCompleted(exception); // same as Result.Failure(exception)
```

Unit Testing

For unit testing, you can use FakeTimeProvider of Microsoft.Extensions.TimeProvider.Testing.

Additionally, in R3, there is a collection called LiveList, which allows you to obtain subscription statuses as a list. Combining these two features can be very useful for unit testing.

```
var fakeTime = new FakeTimeProvider();

var list = Observable.Timer(TimeSpan.FromSeconds(5), fakeTime).ToLiveList();

fakeTime.Advance(TimeSpan.FromSeconds(4));
list.AssertIsNotCompleted();

fakeTime.Advance(TimeSpan.FromSeconds(1));
list.AssertIsCompleted();
list.AssertEqual([Unit.Default]);
```

For FrameProvider, a `FakeFrameProvider` is provided as standard, and it can be used in the same way as `FakeTimeProvider`.

```
var cts = new CancellationTokenSource();
var frameProvider = new FakeFrameProvider();
```

```
var list = Observable.EveryUpdate(frameProvider, cts.Token)
    .Select(_ => frameProvider.GetFrameCount())
    .ToLiveList();

list.AssertEqual([]); // list.Should().Equal(expected);

frameProvider.Advance();
list.AssertEqual([0]);

frameProvider.Advance(3);
list.AssertEqual([0, 1, 2, 3]);

cts.Cancel();
list.AssertIsCompleted(); // list.IsCompleted.Should().BeTrue();

frameProvider.Advance();
list.AssertEqual([0, 1, 2, 3]);
list.AssertIsCompleted();
```

Interoperability with `IObservable<T>`

`Observable<T>` is not `IObservable<T>`. You can convert both by these methods.

- `public static Observable<T> ToObservable<T>(this IObservable<T> source)`
- `public static IObservable<T> AsSystemObservable<T>(this Observable<T> source)`

Interoperability with `async/await`

R3 has special integration with `async/await`. First, all methods that return a single asynchronous operation have now become ***Async methods, returning `Task<T>`.

Furthermore, you can specify special behaviors when asynchronous methods are provided to Where/Select/Subscribe.

Name	**ReturnType**
SelectAwait(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask<TResult>>` selector, `AwaitOperations` awaitOperations = AwaitOperation.Sequential, `Boolean` configureAwait = True)	`Observable<TResult>`
WhereAwait(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask<Boolean>>` predicate, `AwaitOperations`	`Observable<T>`

Name	ReturnType
awaitOperations = AwaitOperation.Sequential, Boolean configureAwait = True)	
SubscribeAwait(this Observable<T> source, Func<T, CancellationToken, ValueTask> onNextAsync, AwaitOperations awaitOperations = AwaitOperation.Sequential, Boolean configureAwait = True)	IDisposable
SubscribeAwait(this Observable<T> source, Func<T, CancellationToken, ValueTask> onNextAsync, Action<Result> onCompleted, AwaitOperations awaitOperations = AwaitOperation.Sequential, Boolean configureAwait = True)	IDisposable
SubscribeAwait(this Observable<T> source, Func<T, CancellationToken, ValueTask> onNextAsync, Action<Exception> onErrorResume, Action<Result> onCompleted, AwaitOperations awaitOperations = AwaitOperation.Sequential, Boolean configureAwait = True)	IDisposable

```
public enum AwaitOperation
{
    /// <summary>All values are queued, and the next value waits for the completion of the asynchronous method.</summary>
    Sequential,
    /// <summary>Drop new value when async operation is running.</summary>
    Drop,
    /// <summary>If the previous asynchronous method is running, it is cancelled and the next asynchronous method is executed.</summary>
    Switch,
    /// <summary>All values are sent immediately to the asynchronous method.</summary>
    Parallel,
    /// <summary>All values are sent immediately to the asynchronous method, but the results are queued and passed to the next operator in order.</summary>
    SequentialParallel,
    /// <summary>Only the latest value is queued, and the next value waits for the completion of the asynchronous method.</summary>
    Latest,
}
```

```
// for example...
// Drop enables prevention of execution by multiple clicks
button.OnClickAsObservable()
    .SelectAwait(async (_, ct) =>
    {
        var req = await
UnityWebRequest.Get("https://google.com/").SendWebRequest().WithCancellation(ct);
        return req.downloadHandler.text;
    }, AwaitOperation.Drop)
    .SubscribeToText(text);
```

Additionally, the following time-related filtering methods can also accept asynchronous methods.

Name	ReturnType
Debounce(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask>` throttleDurationSelector, `Boolean` configureAwait = true)	`Observable<T>`
ThrottleFirst(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask>` sampler, `Boolean` configureAwait = true)	`Observable<T>`
ThrottleLast(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask>` sampler, `Boolean` configureAwait = true)	`Observable<T>`

Concurrency Policy

The composition of operators is thread-safe, and it is expected that the values flowing through OnNext are on a single thread. In other words, if OnNext is issued on multiple threads, the operators may behave unexpectedly. This is the same as with dotnet/reactive.

ObservableCollections

As a special collection for monitoring changes in collections and handling them in R3, the ObservableCollections's `ObservableCollections.R3` package is available.

It has `ObservableList<T>`, `ObservableDictionary<TKey, TValue>`, `ObservableHashSet<T>`, `ObservableQueue<T>`, `ObservableStack<T>`, `ObservableRingBuffer<T>`, `ObservableFixedSizeRingBuffer<T>` and these observe methods.

```
Observable<CollectionAddEvent<T>> IObservableCollection<T>.ObserveAdd()
Observable<CollectionRemoveEvent<T>> IObservableCollection<T>.ObserveRemove()
Observable<CollectionReplaceEvent<T>> IObservableCollection<T>.ObserveReplace()
```

```
Observable<CollectionMoveEvent<T>> IObservableCollection<T>.ObserveMove()
Observable<CollectionResetEvent<T>> IObservableCollection<T>.ObserveReset()
```

XAML Platforms(`BindableReactiveProperty<T>`)

For XAML based application platforms, R3 provides `BindableReactiveProperty<T>` that can bind observable property to view like Android LiveData and Kotlin StateFlow. It implements INotifyPropertyChanged and INotifyDataErrorInfo.

Simple usage, expose `BindableReactiveProperty<T>` via `new` or `ToBindableReactiveProperty`.

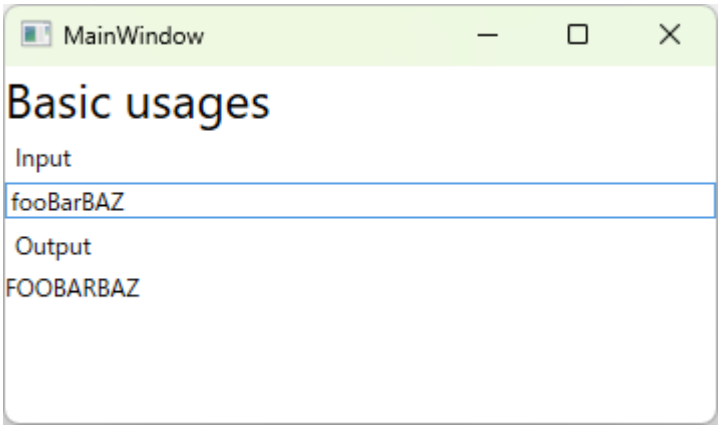
Here is the simple In and Out BindableReactiveProperty ViewModel, Xaml and code-behind. In xaml, `.Value` to bind property.

```
public class BasicUsagesViewModel : IDisposable
{
    public BindableReactiveProperty<string> Input { get; }
    public BindableReactiveProperty<string> Output { get; }

    public BasicUsagesViewModel()
    {
        Input = new BindableReactiveProperty<string>("");
        Output = Input.Select(x => x.ToUpper()).ToBindableReactiveProperty("");
    }

    public void Dispose()
    {
        Disposable.Dispose(Input, Output);
    }
}
```

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Window.DataContext>
        <local:BasicUsagesViewModel />
    </Window.DataContext>
    <StackPanel>
        <TextBlock Text="Basic usages" FontSize="24" />
```

```
        <Label Content="Input" />
        <TextBox Text="{Binding Input.Value, UpdateSourceTrigger=PropertyChanged}"
/>

        <Label Content="Output" />
        <TextBlock Text="{Binding Output.Value}" />
    </StackPanel>
</Window>
```

```
namespace WpfApp1;

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    protected override void OnClosed(EventArgs e)
    {
        (this.DataContext as IDisposable)?.Dispose();
    }
}
```



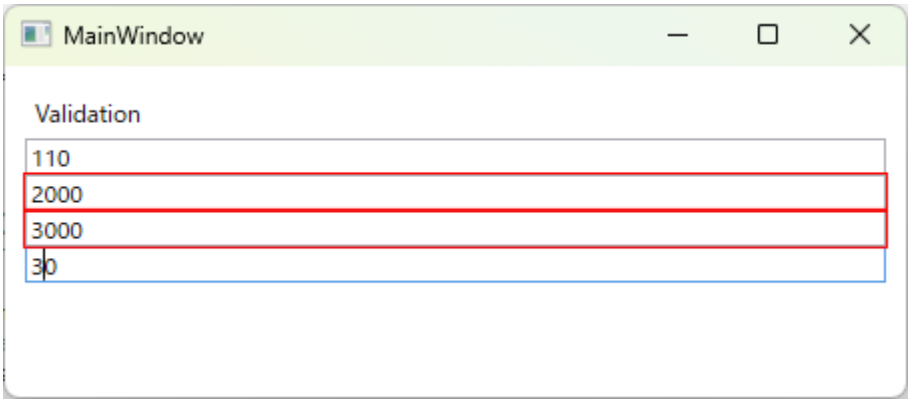
BindableReactiveProperty also supports validation via DataAnnotation or custom logic. If you want to use DataAnnotation attribute, require to call `EnableValidation<T>()` in field initializer or `EnableValidation(Expression selfSelector)` in constructor.

```
public class ValidationViewModel : IDisposable
{
    // Pattern 1. use EnableValidation<T> to enable DataAnnotation validation in
field initializer
```

```
    [Range(0.0, 300.0)]
    public BindableReactiveProperty<double> Height { get; } = new
BindableReactiveProperty<double>().EnableValidation<ValidationViewModel>();

    [Range(0.0, 300.0)]
    public BindableReactiveProperty<double> Weight { get; }

    IDisposable customValidation1Subscription;
    public BindableReactiveProperty<double> CustomValidation1 { get; set; }

    public BindableReactiveProperty<double> CustomValidation2 { get; set; }

    public ValidationViewModel()
    {
        // Pattern 2. use EnableValidation(Expression) to enable
DataAnnotation validation
        Weight = new BindableReactiveProperty<double>().EnableValidation(()
=> Weight);

        // Pattern 3. EnableValidation() and call OnErrorResume to set custom
error meessage
        CustomValidation1 = new BindableReactiveProperty<double>
().EnableValidation();
        customValidation1Subscription = CustomValidation1.Subscribe(x =>
        {
            if (0.0 <= x && x <= 300.0) return;

            CustomValidation1.OnErrorResume(new Exception("value is not
in range."));
        });

        // Pattern 4. simplified version of Pattern3, EnableValidation(Func<T,
Exception?>)
        CustomValidation2 = new BindableReactiveProperty<double>
().EnableValidation(x =>
        {
            if (0.0 <= x && x <= 300.0) return null; // null is no validate result
            return new Exception("value is not in range.");
        });
    }

    public void Dispose()
    {
        Disposable.Dispose(Height, Weight, CustomValidation1,
customValidation1Subscription, CustomValidation2);
```

```
    }
}
```

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Window.DataContext>
        <local:ValidationViewModel />
    </Window.DataContext>

    <StackPanel Margin="10">
        <Label Content="Validation" />
        <TextBox Text="{Binding Height.Value, UpdateSourceTrigger=PropertyChanged}"
/>
        <TextBox  Text="{Binding Weight.Value, UpdateSourceTrigger=PropertyChanged}"
/>
        <TextBox  Text="{Binding CustomValidation1.Value,
UpdateSourceTrigger=PropertyChanged}" />
        <TextBox  Text="{Binding CustomValidation2.Value,
UpdateSourceTrigger=PropertyChanged}" />
    </StackPanel>
</Window>
```

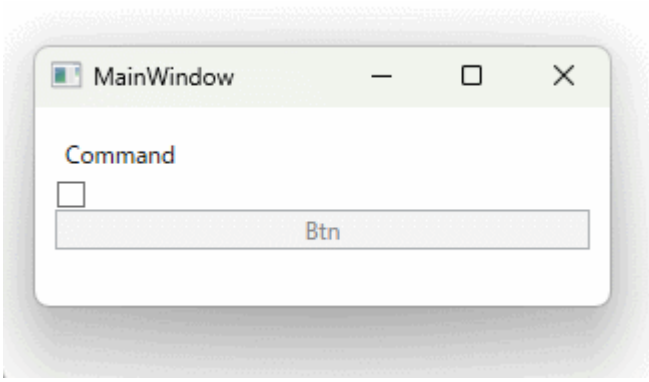


ReactiveCommand

`ReactiveCommand<T>` is observable ICommand implementation. It can create from `Observable<bool> canExecuteSource`.

```
public class CommandViewModel : IDisposable
{
    public BindableReactiveProperty<bool> OnCheck { get; } // bind to CheckBox
    public ReactiveCommand<Unit> ShowMessageBox { get; }   // bind to Button

    public CommandViewModel()
    {
        OnCheck = new BindableReactiveProperty<bool>();
        ShowMessageBox = OnCheck.ToReactiveCommand(_ =>
        {
            MessageBox.Show("clicked");
        });
    }

    public void Dispose()
    {
        Disposable.Combine(OnCheck, ShowMessageBox);
    }
}
```

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Window.DataContext>
        <local:CommandViewModel />
    </Window.DataContext>
    <StackPanel Margin="10">
        <Label Content="Command" />
        <CheckBox IsChecked="{Binding OnCheck.Value}" />
        <Button Content="Btn" Command="{Binding ShowMessageBox}" />
    </StackPanel>
</Window>
```



Platform Supports

Even without adding specific platform support, it is possible to use only the core library. However, Rx becomes more user-friendly by replacing the standard `TimeProvider` and `FrameProvider` with those optimized for each platform. For example, while the standard `TimeProvider` is thread-based, using a UI thread-based `TimeProvider` for each platform can eliminate the need for dispatch through `ObserveOn`, enhancing usability. Additionally, since message loops differ across platforms, the use of individual `FrameProvider` is essential.

Although standard support is provided for the following platforms, by implementing `TimeProvider` and `FrameProvider`, it is possible to support any environment, including in-house game engine or other frameworks.

- WPF
- Avalonia
- MAUI
- WinForms
- Unity
- Godot
- Stride
- MonoGame
- LogicLooper

Add support planning LogicLooper.

WPF

> PM> Install-Package R3Extensions.WPF

R3Extensions.WPF package has two providers.

- WpfDispatcherTimerProvider
- WpfRenderingFrameProvider

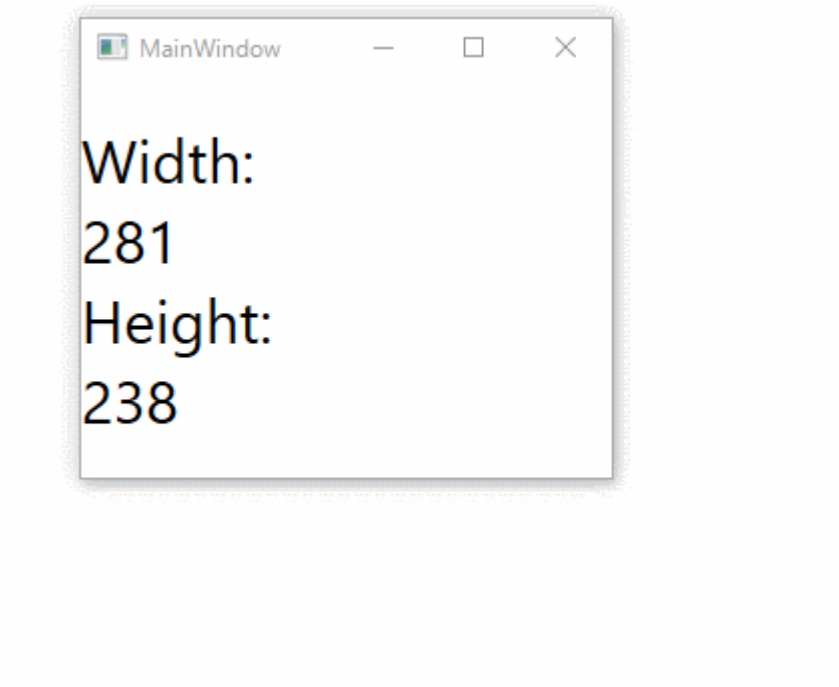
Calling `WpfProviderInitializer.SetDefaultObservableSystem()` at startup will replace `ObservableSystem.DefaultTimeProvider` and `ObservableSystem.DefaultFrameProvider` with the aforementioned providers.

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        // You need to set UnhandledExceptionHandler
        WpfProviderInitializer.SetDefaultObservableSystem(ex => Trace.WriteLine($"R3
UnhandledException:{ex}"));
    }
}
```

As a result, time based operations are replaced with `DispatcherTimer`, allowing you to reflect time based operations on the UI without having to use `ObserveOn`.

`WpfRenderingFrameProvider` is a frame-based loop system synchronized with the `CompositionTarget.Rendering` event. This allows for writing code that, for example, reads and reflects changes in values that do not implement `INotifyPropertyChanged`.

```
public partial class MainWindow : Window
{
    IDisposable disposable;

    public MainWindow()
    {
        InitializeComponent();

        var d1 = Observable.EveryValueChanged(this, x => x.Width).Subscribe(x =>
WidthText.Text = x.ToString());
        var d2 = Observable.EveryValueChanged(this, x => x.Height).Subscribe(x =>
HeightText.Text = x.ToString());

        disposable = Disposable.Combine(d1, d2);
    }

    protected override void OnClosed(EventArgs e)
    {
        disposable.Dispose();
    }
}
```



In addition to the above, the following `ObserveOn`/`SubscribeOn` methods have been added.

- ObserveOnDispatcher
- ObserveOnCurrentDispatcher
- SubscribeOnDispatcher
- SubscribeOnCurrentDispatcher

ViewModel binding support, see `BindableReactiveProperty<T>` section.

Avalonia

> PM> Install-Package R3Extensions.Avalonia

R3Extensions.Avalonia package has these providers.

- AvaloniaDispatcherTimerProvider
- AvaloniaDispatcherFrameProvider
- AvaloniaRenderingFrameProvider

Calling `AvaloniaProviderInitializer.SetDefaultObservableSystem()` at startup will replace `ObservableSystem.DefaultTimeProvider` and `ObservableSystem.DefaultFrameProvider` with `AvaloniaDispatcherTimerProvider` and `AvaloniaDispatcherFrameProvider`.

Additionally, calling `UseR3()` in `ApplicationBuilder` sets the default providers, making it a recommended approach.

```
public static AppBuilder BuildAvaloniaApp()
    => AppBuilder.Configure<App>()
        .UsePlatformDetect()
        .WithInterFont()
        .LogToTrace()
        .UseR3(); // add this line
```

As a result, time based operations are replaced with `DispatcherTimer`, allowing you to reflect time based operations on the UI without having to use `ObserveOn`.

In the case of methods without arguments, integrate the following method into `ObservableSystem.RegisterUnhandledExceptionHandler`. Please customize this as necessary.

```
ex => Logger.Sink?.Log(LogEventLevel.Error, "R3", null, "R3 Unhandled Exception
{0}", ex);
```

`AvaloniaDispatcherFrameProvider` calculates a frame by polling with `DispatcherTimer`. By default, it updates at 60fps.

Using `AvaloniaRenderingFrameProvider` is more performant however it needs `TopLevel`.

```
public partial class MainWindow : Window
{
    AvaloniaRenderingFrameProvider frameProvider;

    public MainWindow()
    {
        InitializeComponent();

        // initialize RenderingFrameProvider
        var topLevel = TopLevel.GetTopLevel(this);
        this.frameProvider = new AvaloniaRenderingFrameProvider(topLevel!);
    }

    protected override void OnLoaded(RoutedEventArgs e)
    {
        // pass frameProvider
        Observable.EveryValueChanged(this, x => x.Width, frameProvider)
            .Subscribe(x => textBlock.Text = x.ToString());
    }

    protected override void OnClosed(EventArgs e)
    {
```

```
            frameProvider.Dispose();
        }
    }
```

In addition to the above, the following `ObserveOn`/`SubscribeOn` methods have been added.

- ObserveOnDispatcher
- ObserveOnUIThreadDispatcher
- SubscribeOnDispatcher
- SubscribeOnUIThreadDispatcher

MAUI

> PM> Install-Package R3Extensions.Maui

R3Extensions.Maui package has these providers.

- MauiDispatcherTimerProvider
- MauiTickerFrameProvider

And ViewModel binding is supported, see `BindableReactiveProperty<T>` section.

Calling `UseR3()` in `MauiAppBuilder` sets the default providers.

```
public static MauiApp CreateMauiApp()
{
    var builder = MauiApp.CreateBuilder();
    builder
        .UseMauiApp<App>()
        .ConfigureFonts(fonts =>
        {
            fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
        })
        .UseR3(); // add this line

    return builder.Build();
}
```

`UseR3()` configures the following.

- Time based operations are replaced with `IDispatcher`, allowing you to reflect time based operations on the UI without having to use `ObserveOn`.

- Frame based operations are replaced with `Ticker`.
- `ObservableSystem.RegisterUnhandledExceptionHandler` is set to `R3MauiDefaultExceptionHandler`:

    ```
    public class R3MauiDefaultExceptionHandler(IServiceProvider serviceProvider)
    : IR3MauiExceptionHandler
    {
        public void HandleException(Exception ex)
        {
            System.Diagnostics.Trace.TraceError("R3 Unhandled Exception
    {0}", ex);

            var logger =
    serviceProvider.GetService<ILogger<R3MauiDefaultExceptionHandler>>();
            logger?.LogError(ex, "R3 Unhandled Exception");
        }
    }
    ```

If you want to customize the ExceptionHandler, there are two ways.

One is to pass a callback to `UseR3e

```
builder.UseR3(ex => Console.WriteLine($"R3 UnhandledException:{ex}"));
```

The second is to create an implementation of the `IR3MAuiExceptionHandler` interface and DI it. Since MAUI is a DI-based framework, this method will make it easier to access the various functions in the DI container.

```
builder.Services.AddSingleton<IR3MauiExceptionHandler, YourCustomExceptionHandler>
();
```

WinForms

> PM> Install-Package R3Extensions.WinForms

R3Extensions.WinForms package has these providers.

- WinFormsFrameProvider
- WinFormsTimerProvider

Calling `WinFormsProviderInitializer.SetDefaultObservableSystem()` at startup(Program.Main) will replace `ObservableSystem.DefaultTimeProvider` and

`ObservableSystem.DefaultFrameProvider` with `WinFormsFrameProvider` and `WinFormsTimerProvider`.

```
using R3.WinForms;

internal static class Program
{
    [STAThread]
    static void Main()
    {
        ApplicationConfiguration.Initialize();

        var form = new Form1();

        // add this line
        WinFormsProviderInitializer.SetDefaultObservableSystem(ex =>
Trace.WriteLine($"R3 UnhandledException:{ex}"), form);

        Application.Run(form);
    }
}
```

`SetDefaultObservableSystem` takes ISynchronizeInvoke (such as Form or Control). This makes the Timer operate on the thread to which it belongs.

FrameProvider is executed as one frame using the hook of MessageFilter.

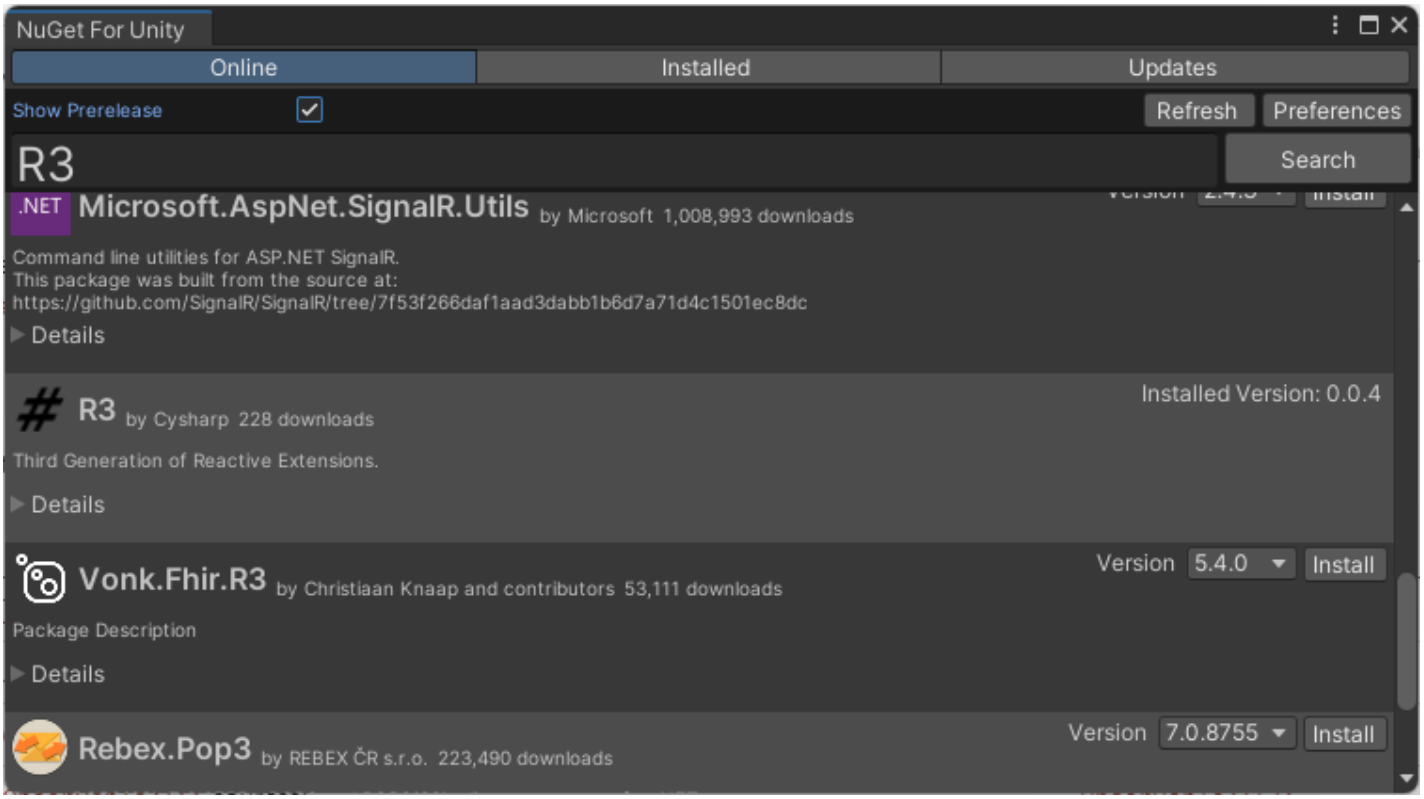
Unity

The minimum Unity support for R3 is **Unity 2021.3**.

There are two installation steps required to use it in Unity.

1. Install `R3` from NuGet using NuGetForUnity

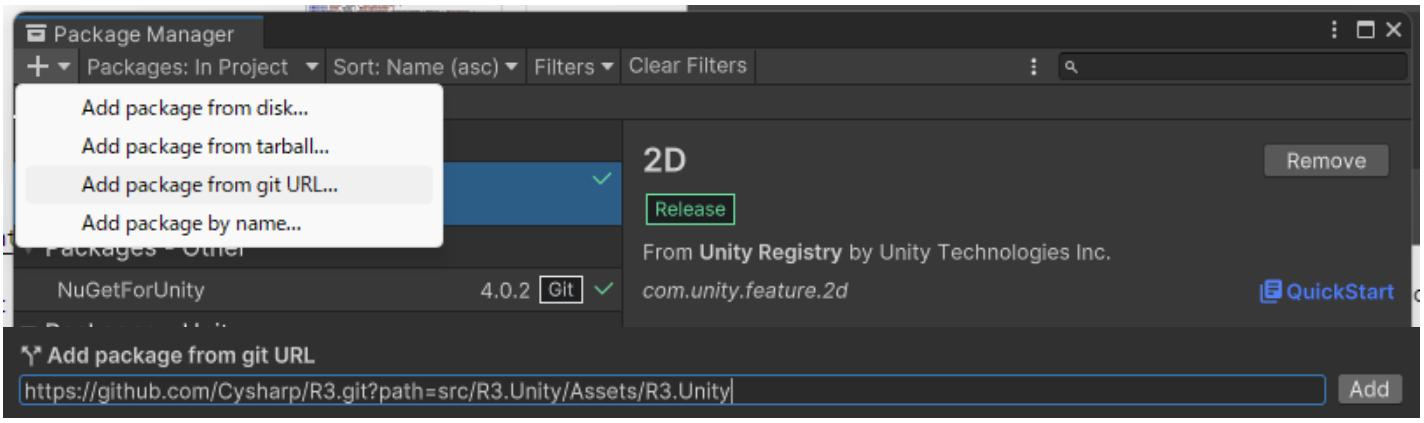
- Open Window from NuGet -> Manage NuGet Packages, Search "R3" and Press Install.



- If you encount version conflicts error, please disable version validation in Player Settings(Edit -> Project Settings -> Player -> Scroll down and expand "Other Settings" than uncheck "Assembly Version Validation" under the "Configuration" section).

2. Install the `R3.Unity` package by referencing the git URL

- `https://github.com/Cysharp/R3.git?path=src/R3.Unity/Assets/R3.Unity`



R3 uses the ..* release tag, so you can specify a version like #1.0.0. For example: `https://github.com/Cysharp/R3.git?path=src/R3.Unity/Assets/R3.Unity#1.0.0`

Unity's TimeProvider and FrameProvider is PlayerLoop based. Additionally, there are variations of TimeProvider that correspond to the TimeScale.

```
UnityTimeProvider.Initialization
UnityTimeProvider.EarlyUpdate
UnityTimeProvider.FixedUpdate
UnityTimeProvider.PreUpdate
UnityTimeProvider.Update
UnityTimeProvider.PreLateUpdate
UnityTimeProvider.PostLateUpdate
UnityTimeProvider.TimeUpdate

UnityTimeProvider.InitializationIgnoreTimeScale
UnityTimeProvider.EarlyUpdateIgnoreTimeScale
UnityTimeProvider.FixedUpdateIgnoreTimeScale
UnityTimeProvider.PreUpdateIgnoreTimeScale
UnityTimeProvider.UpdateIgnoreTimeScale
UnityTimeProvider.PreLateUpdateIgnoreTimeScale
UnityTimeProvider.PostLateUpdateIgnoreTimeScale
UnityTimeProvider.TimeUpdateIgnoreTimeScale

UnityTimeProvider.InitializationRealtime
UnityTimeProvider.EarlyUpdateRealtime
UnityTimeProvider.FixedUpdateRealtime
UnityTimeProvider.PreUpdateRealtime
UnityTimeProvider.UpdateRealtime
UnityTimeProvider.PreLateUpdateRealtime
UnityTimeProvider.PostLateUpdateRealtime
UnityTimeProvider.TimeUpdateRealtime


UnityFrameProvider.Initialization
UnityFrameProvider.EarlyUpdate
UnityFrameProvider.FixedUpdate
UnityFrameProvider.PreUpdate
UnityFrameProvider.Update
UnityFrameProvider.PreLateUpdate
UnityFrameProvider.PostLateUpdate
UnityFrameProvider.TimeUpdate
```

You can write it like this using these:

```
// ignore-timescale based interval
Observable.Interval(TimeSpan.FromSeconds(5),
UnityTimeProvider.UpdateIgnoreTimeScale);

// fixed-update loop
```

```
Observable.EveryUpdate(UnityFrameProvider.FixedUpdate);

// observe PostLateUpdate
Observable.Return(42).ObserveOn(UnityFrameProvider.PostLateUpdate);
```

In the case of Unity, `UnityTimeProvider.Update` and `UnityFrameProvider.Update` are automatically set at startup by default.

```
public static class UnityProviderInitializer
{
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.AfterAssembliesLoaded)]
    public static void SetDefaultObservableSystem()
    {
        SetDefaultObservableSystem(static ex => UnityEngine.Debug.LogException(ex));
    }

    public static void SetDefaultObservableSystem(Action<Exception>
unhandledExceptionHandler)
    {

ObservableSystem.RegisterUnhandledExceptionHandler(unhandledExceptionHandler);
        ObservableSystem.DefaultTimeProvider = UnityTimeProvider.Update;
        ObservableSystem.DefaultFrameProvider = UnityFrameProvider.Update;
    }
}
```

A method has been added to convert from UnityEvent to AsObservable. If a CancellationToken is passed, it allows the event source to call for event unsubscription by issuing OnCompleted when Cancel is invoked. For example, if you pass `MonoBehaviour.destroyCancellationToken`, it will be reliably unsubscribed in conjunction with the GameObject's lifecycle.

```
public static Observable<Unit> AsObservable(this UnityEngine.Events.UnityEvent
unityEvent, CancellationToken cancellationToken = default)
public static Observable<T> AsObservable<T>(this UnityEngine.Events.UnityEvent<T>
unityEvent, CancellationToken cancellationToken = default)
public static Observable<(T0 Arg0, T1 Arg1)> AsObservable<T0, T1>(this
UnityEngine.Events.UnityEvent<T0, T1> unityEvent, CancellationToken
cancellationToken = default)
public static Observable<(T0 Arg0, T1 Arg1, T2 Arg2)> AsObservable<T0, T1, T2>(this
UnityEngine.Events.UnityEvent<T0, T1, T2> unityEvent, CancellationToken
cancellationToken = default)
public static Observable<(T0 Arg0, T1 Arg1, T2 Arg2, T3 Arg3)> AsObservable<T0, T1,
```

```
T2, T3>(this UnityEngine.Events.UnityEvent<T0, T1, T2, T3> unityEvent,
CancellationToken cancellationToken = default)
```

Additionally, with extension methods for uGUI, uGUI events can be easily converted to Observables. OnValueChangedAsObservable starts the subscription by first emitting the latest value at the time of subscription. Andalso when the associated component is destroyed, it emits an OnCompleted event to ensure the subscription is reliably cancelled.

```
public static IDisposable SubscribeToText(this Observable<string> source, Text text)
public static IDisposable SubscribeToText<T>(this Observable<T> source, Text text)
public static IDisposable SubscribeToText<T>(this Observable<T> source, Text text,
Func<T, string> selector)
public static IDisposable SubscribeToInteractable(this Observable<bool> source,
Selectable selectable)
public static Observable<Unit> OnClickAsObservable(this Button button)
public static Observable<bool> OnValueChangedAsObservable(this Toggle toggle)
public static Observable<float> OnValueChangedAsObservable(this Scrollbar scrollbar)
public static Observable<Vector2> OnValueChangedAsObservable(this
ScrollRect scrollRect)
public static Observable<float> OnValueChangedAsObservable(this Slider slider)
public static Observable<string> OnEndEditAsObservable(this InputField inputField)
public static Observable<string> OnValueChangedAsObservable(this
InputField inputField)
public static Observable<int> OnValueChangedAsObservable(this Dropdown dropdown)
```

In addition to the above, the following `ObserveOn`/`SubscribeOn` methods have been added.

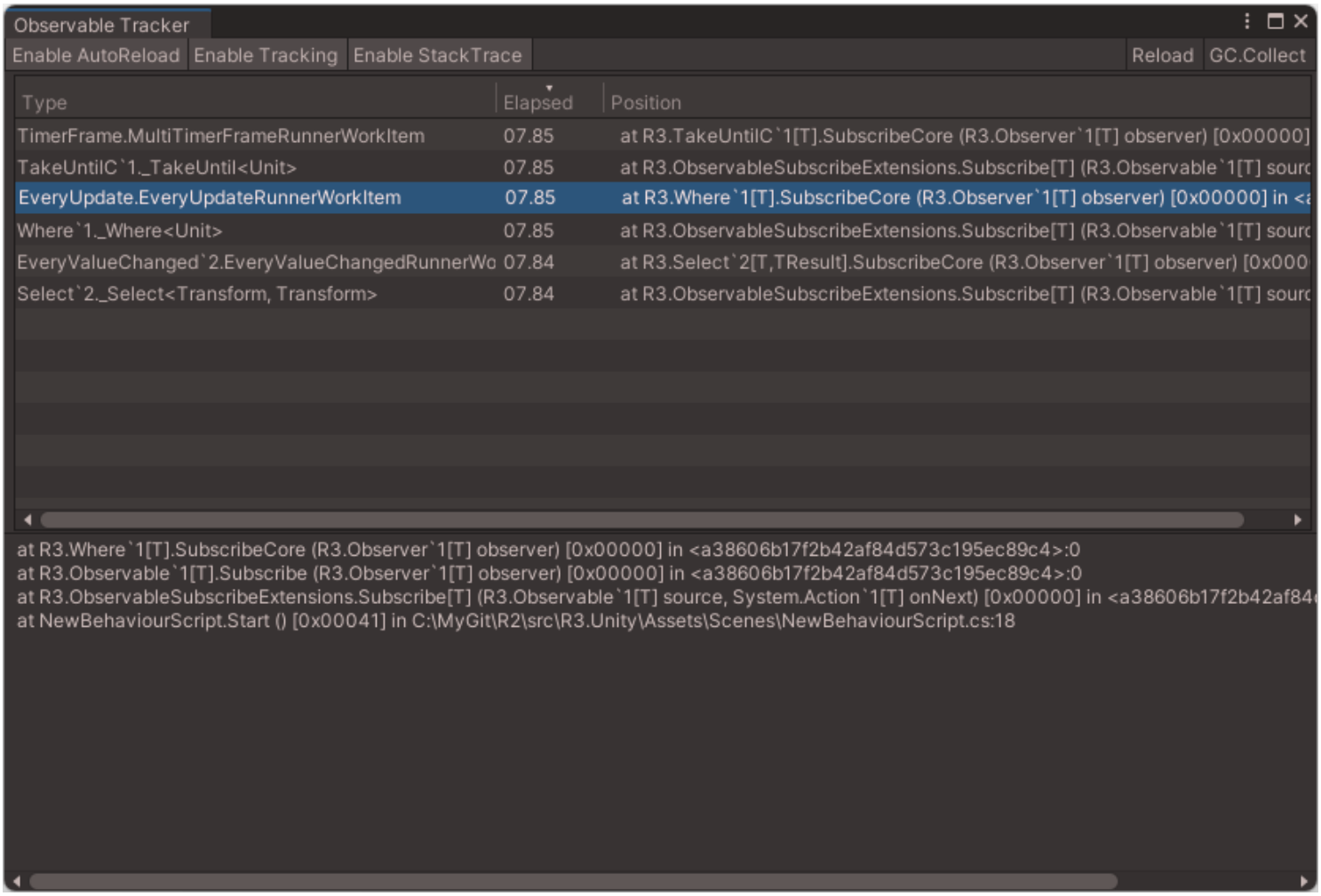
- ObserveOnMainThread
- SubscribeOnMainThread

When using `AddTo(Component / GameObject)` in Unity, it attaches a special component called ObservableDestroyTrigger if gameObject is not active yet, which monitors for destruction. Unity has a characteristic where components that have never been activated do not fire OnDestroy, and the destroyCancellationToken does not get canceled. ObservableDestroyTrigger is designed to monitor for destruction and reliably issue OnDestroy regardless of the active state. It would be wise to use destroyCancellationToken effectively if needed.

```
// simple pattern
Observable.EveryUpdate().Subscribe().AddTo(this);
Observable.EveryUpdate().Subscribe().AddTo(this);
Observable.EveryUpdate().Subscribe().AddTo(this);
```

```
// better performance
var d = Disposable.CreateBuilder();
Observable.EveryUpdate().Subscribe().AddTo(ref d);
Observable.EveryUpdate().Subscribe().AddTo(ref d);
Observable.EveryUpdate().Subscribe().AddTo(ref d);
d.RegisterTo(this.destroyCancellationToken); // Build and Register
```

You open tracker window in `Window -> Observable Tracker`. It enables watch `ObservableTracker` list in editor window.



- Enable AutoReload(Toggle) - Reload automatically.
- Reload - Reload view.
- GC.Collect - Invoke GC.Collect.
- Enable Tracking(Toggle) - Start to track subscription. Performance impact: low.
- Enable StackTrace(Toggle) - Capture StackTrace when observable is subscribed. Performance impact: high.

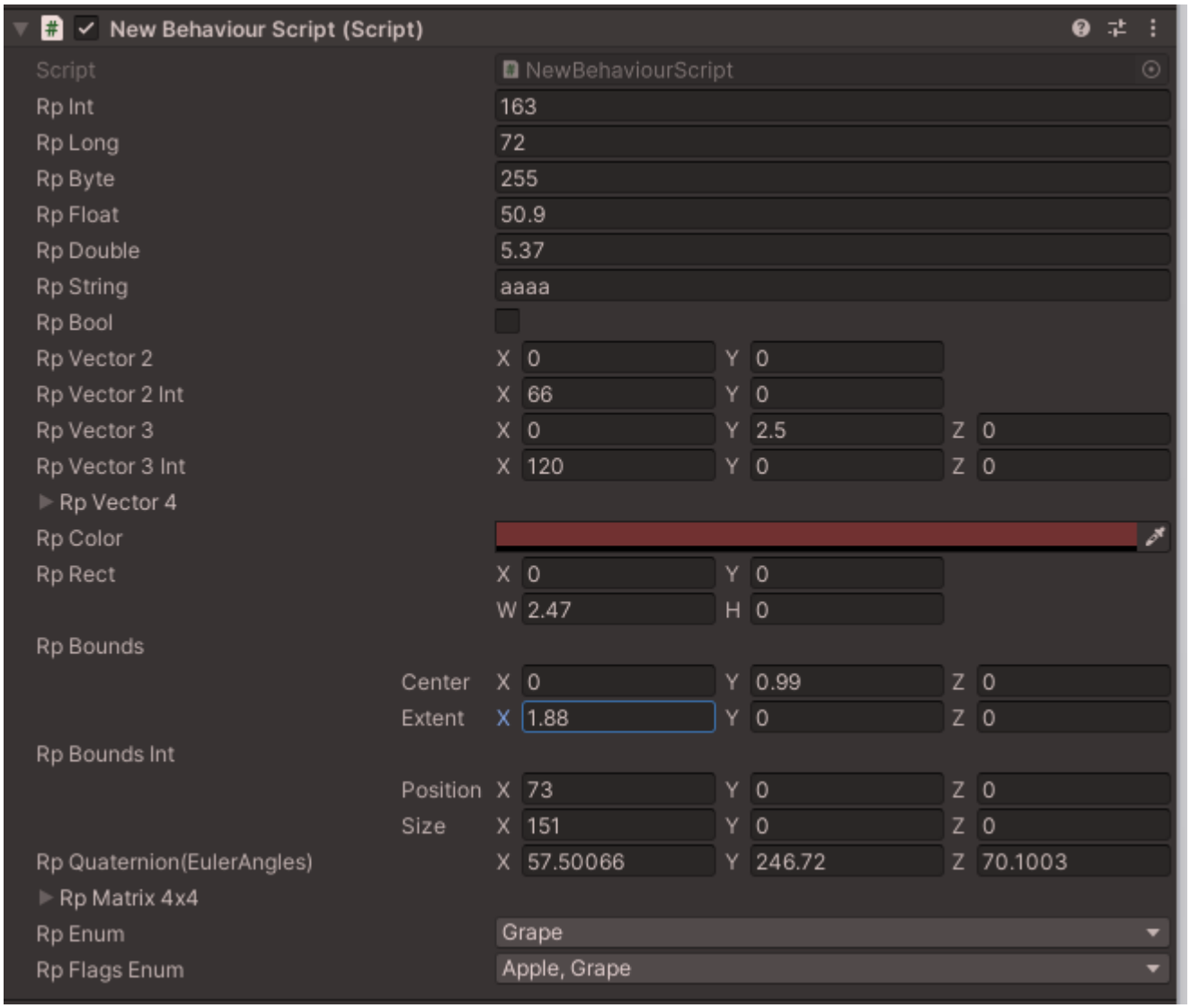
Observable Tracker is intended for debugging use only as enabling tracking and capturing stacktraces is useful but has a heavy performance impact. Recommended usage is to

enable both tracking and stacktraces to find subscription leaks and to disable them both when done.

`SerializableReactiveProperty<T>`

`ReactiveProperty<T>` can not use on `[SerializeField]`. However you can use `SerializableReactiveProperty<T>` instead.

```
public class NewBehaviourScript : MonoBehaviour
{
    public SerializableReactiveProperty<int> rpInt;
    public SerializableReactiveProperty<long> rpLong;
    public SerializableReactiveProperty<byte> rpByte;
    public SerializableReactiveProperty<float> rpFloat;
    public SerializableReactiveProperty<double> rpDouble;
    public SerializableReactiveProperty<string> rpString;
    public SerializableReactiveProperty<bool> rpBool;
    public SerializableReactiveProperty<Vector2> rpVector2;
    public SerializableReactiveProperty<Vector2Int> rpVector2Int;
    public SerializableReactiveProperty<Vector3> rpVector3;
    public SerializableReactiveProperty<Vector3Int> rpVector3Int;
    public SerializableReactiveProperty<Vector4> rpVector4;
    public SerializableReactiveProperty<Color> rpColor;
    public SerializableReactiveProperty<Rect> rpRect;
    public SerializableReactiveProperty<Bounds> rpBounds;
    public SerializableReactiveProperty<BoundsInt> rpBoundsInt;
    public SerializableReactiveProperty<Quaternion> rpQuaternion;
    public SerializableReactiveProperty<Matrix4x4> rpMatrix4x4;
    public SerializableReactiveProperty<FruitEnum> rpEnum;
    public SerializableReactiveProperty<FruitFlagsEnum> rpFlagsEnum;
}
```



Triggers

R3 can handle MonoBehaviour messages with R3.Triggers:

These can also be handled more easily by directly subscribing to observables returned by extension methods on Component/GameObject. These methods inject ObservableTrigger automaticaly.

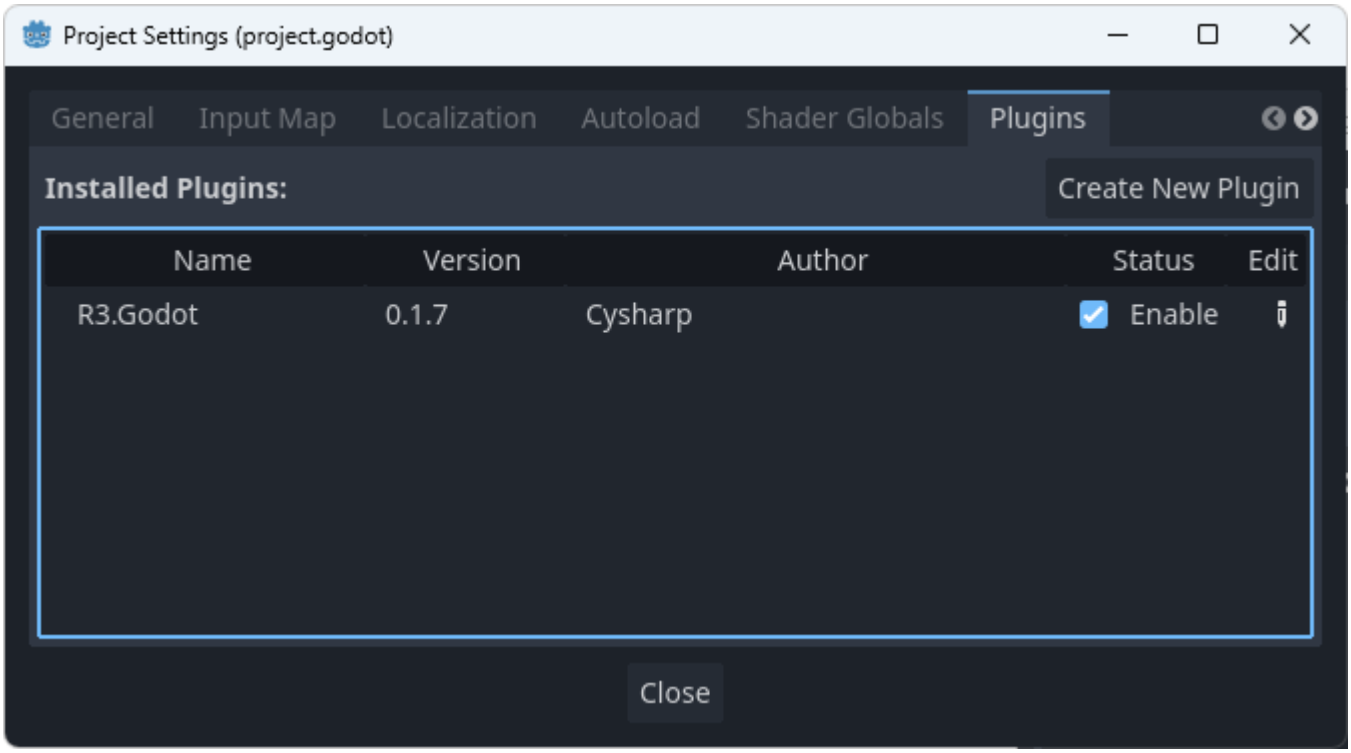
```
using R3;
using R3.Triggers;

// when using R3.Triggers, Component or GameObject has [MonoBehaviour
Messages]AsObservable extension methods.
this.OnCollisionEnterAsObservable()
    .Subscribe(x =>
    {
        Debug.Log("collision enter");
    });
```

Godot

Godot support is for Godot 4.x.

There are some installation steps required to use it in Godot.

1. Install `R3` from NuGet.
2. Download(or clone git submodule) the repository and move the `src/R3.Godot/addons/R3.Godot` directory to your project.
3. Enable the `R3.Godot` plugin from the plugins menu.



Godot support has these TimeProvider and FrameProvider.

```
GodotTimeProvider.Process
GodotTimeProvider.PhysicsProcess


GodotFrameProvider.Process
GodotFrameProvider.PhysicsProcess
```

autoloaded `FrameProviderDispatcher` set `GodotTimeProvider.Process` and `GodotFrameProvider.Process` as default providers. Additionally, UnhandledException is written to `GD.PrintErr`.

This is the minimal sample to use R3.Godot.

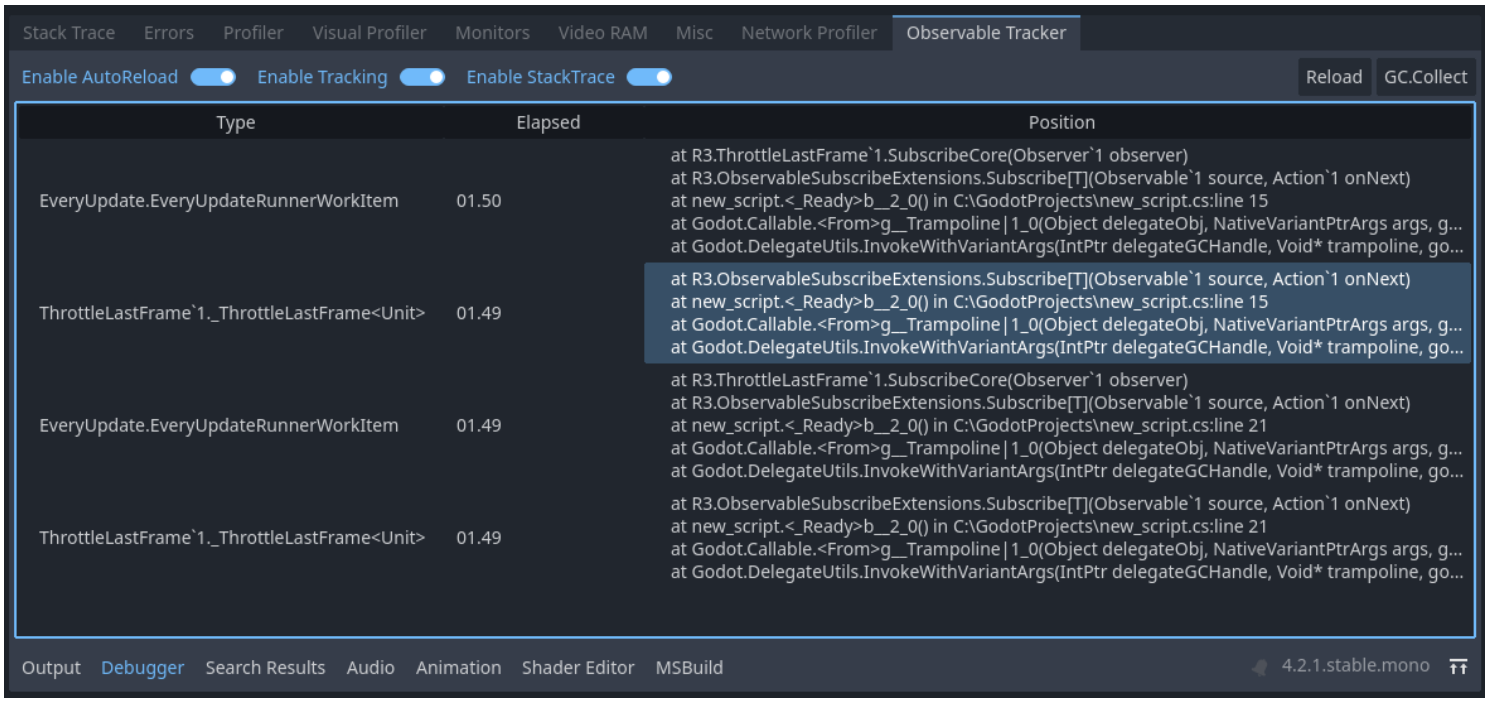
```
using Godot;
using R3;
using System;

public partial class Node2D : Godot.Node2D
{
    IDisposable subscription;

    public override void _Ready()
    {
        subscription = Observable.EveryUpdate()
            .ThrottleLastFrame(10)
            .Subscribe(x =>
            {
                GD.Print($"Observable.EveryUpdate:
{GodotFrameProvider.Process.GetFrameCount()}");
            });
    }

    protected override void Dispose(bool disposing)
    {
        subscription?.Dispose();
    }
}
```

For the UI event observe/subscribe extension are also available.

```
public static IDisposable SubscribeToLabel(this Observable<string> source,
Label label)
public static IDisposable SubscribeToLabel<T>(this Observable<T> source,
Label label)
public static IDisposable SubscribeToLabel<T>(this Observable<T> source, Label
label, Func<T, string> selector)
public static Observable<Unit> OnPressedAsObservable(this BaseButton button,
CancellationToken cancellationToken = default)
public static Observable<bool> OnToggledAsObservable(this BaseButton button,
CancellationToken cancellationToken = default)
public static Observable<double> OnValueChangedAsObservable(this Godot.Range range,
CancellationToken cancellationToken = default)
public static Observable<string> OnTextSubmittedAsObservable(this LineEdit lineEdit,
CancellationToken cancellationToken = default)
public static Observable<string> OnTextChangedAsObservable(this LineEdit lineEdit,
CancellationToken cancellationToken = default)
public static Observable<Unit> OnTextChangedAsObservable(this TextEdit textEdit,
CancellationToken cancellationToken = default)
```

```
public static Observable<long> OnItemSelectedAsObservable(this OptionButton
optionButton, CancellationToken cancellationToken = default)
```

You can watch subscription status in `Debugger -> ObservableTracker` view.



Stride

R3 extensions for Stride game engine.

> PM> Install-Package R3Extensions.Stride

Usage

1. Reference R3.Stride
2. add empty Entity by Stride editor
3. add "R3/StrideFrameProviderComponent"
4. set Stride Frame Provider Component's priority to lower than other scripts which use R3 API

R3Extensions.Stride provides these providers.

- StrideTimeProvider
- StrideFrameProvider

For the UI event observe/subscribe extension are also available.

```
public static Observable<(object? sender, PropertyChangedArgs<MouseOverState> arg)>
MouseOverStateChangedAsObservable(this UIElement element, CancellationToken token
= default)
public static Observable<(object? sender, TouchEventArgs)>
PreviewTouchDownAsObservable(this UIElement element, CancellationToken token
= default)
public static Observable<(object? sender, TouchEventArgs)>
PreviewTouchMoveAsObservable(this UIElement element, CancellationToken token
= default)
public static Observable<(object? sender, TouchEventArgs)>
PreviewTouchUpAsObservable(this UIElement element, CancellationToken token
= default)
public static Observable<(object? sender, TouchEventArgs)>
TouchDownAsObservable(this UIElement element, CancellationToken token = default)
public static Observable<(object? sender, TouchEventArgs)>
TouchMoveAsObservable(this UIElement element, CancellationToken token = default)
public static Observable<(object? sender, TouchEventArgs)> TouchUpAsObservable(this
UIElement element, CancellationToken token = default)
public static Observable<(object? sender, TouchEventArgs)>
TouchEnterAsObservable(this UIElement element, CancellationToken token = default)
public static Observable<(object? sender, TouchEventArgs)>
TouchLeaveAsObservable(this UIElement element, CancellationToken token = default)
public static Observable<(object? sender, RoutedEventArgs arg)>
ClickAsObservable(this ButtonBase btn, CancellationToken token = default)
public static Observable<(object? sender, RoutedEventArgs arg)>
ValueChangedAsObservable(this Slider slider, CancellationToken token = default)
public static Observable<(object? sender, RoutedEventArgs arg)>
TextChangedAsObservable(this EditText editText, CancellationToken token = default)
public static Observable<(object? sender, RoutedEventArgs arg)>
CheckedAsObservable(this ToggleButton toggleButton, CancellationToken token
= default)
public static Observable<(object? sender, RoutedEventArgs arg)>
IndeterminateAsObservable(this ToggleButton button, CancellationToken token
= default)
public static Observable<(object? sender, RoutedEventArgs arg)>
UncheckedAsObservable(this ToggleButton toggleButton, CancellationToken token
= default)
public static Observable<(object? sender, RoutedEventArgs arg)>
OutsideClickAsObservable(this ModalElement modalElement, CancellationToken token
= default)
```

And event extensions.

```
public static Observable<(object? sender, TrackingCollectionChangedEventArgs arg)>
CollectionChangedAsObservable(this ITrackingCollectionChanged hashset,
```

```
CancellationToken token = default)
public static Observable<(object? sender, FastTrackingCollectionChangedEventArgs
arg)> CollectionChangedAsObservable<T>(this FastTrackingCollection<T> collection,
CancellationToken token = default)
public static Observable<T> AsObservable<T>(this EventKey<T> eventKey,
CancellationToken token = default)
public static Observable<Unit> AsObservable(this EventKey eventKey,
CancellationToken token = default)
```

MonoGame

R3 extensions for MonoGame game engine.

> PM> Install-Package R3Extensions.MonoGame

Set up as follows:

1. Reference R3.MonoGame
2. Add an instance of `ObservableSystemComponent` to your Game class.

```
public class Game1 : Game
{
    public Game1()
    {
        var observableSystemComponent = new ObservableSystemComponent(this);
        Components.Add(observableSystemComponent);
    }
}
```

ObservableSystemComponent configure the following:

- Setup TimeProvider and FrameProvider.
  - Time based operations are replaced with `Game.Update(GameTime)`.
  - Frame based operations are replaced with `Game.Update(GameTime)`.
- Set UnhandledExceptionHandler. By default, the unhandled exception handler simply flows to System.Diagnostics.Trace.
  - If you want to change this, do the following:
    - `new ObservableSystemComponent(this, ex => Console.WriteLine($"R3 UnhandledException: {ex}");`

R3Extensions.MonoGame provides these providers.

- MonoGameTimeProvider
- MonoGameFrameProvider

And provides these custom operators.

```
// Observe the current GameTime value.
public static Observable<GameTime> GameTime(this Observable<Unit> source)

// observe the current GameTime and the value of the source observable.
public static Observable<(GameTime GameTime, T Item)> GameTime<T>(this
Observable<T> source)
```

LogicLooper

R3 extensions for LogicLooper

> PM> Install-Package R3Extensions.LogicLooper

That supports two special providers.

- LogicLooperFrameProvider
- LogicLooperTimerProvider

Operator Reference

The standard operators in ReactiveX follow the behavior described in the Reactive X Operator documentation.

Methods that accept a Scheduler will take a `TimeProvider`. Additionally, methods that receive a `TimeProvider` have an added method called `***Frame` that accepts a `FrameProvider`.

For default time based operations that do not take a provider, `ObservableSystem.DefaultTimeProvider` is used, and for frame based operations without provider, `ObservableSystem.DefaultFrameProvider` is used.

Factory

Factory methods are defined as static methods in the static class `Observable`.

Name(Parameter)	**ReturnType**
Amb(params `Observable<T>[]` sources)	`Observable<T>`
Amb(`IEnumerable<Observable<T>>` sources)	`Observable<T>`

Name(Parameter)	ReturnType
CombineLatest(params `Observable<T>[]` sources)	`Observable<T[]>`
CombineLatest(`IEnumerable<Observable<T>>` sources)	`Observable<T[]>`
Concat(params `Observable<T>[]` sources)	`Observable<T>`
Concat(`IEnumerable<Observable<T>>` sources)	`Observable<T>`
Concat(this `Observable<Observable<T>>` sources)	`Observable<T>`
Create(`Func<Observer<T>, IDisposable>` subscribe, `Boolean` rawObserver = false)	`Observable<T>`
Create(`TState` state, `Func<Observer<T>, TState, IDisposable>` subscribe, `Boolean` rawObserver = false)	`Observable<T>`
Create(`Func<Observer<T>, CancellationToken, ValueTask>` subscribe, `Boolean` rawObserver = false)	`Observable<T>`
Create(`TState` state, `Func<Observer<T>, TState, CancellationToken, ValueTask>` subscribe, `Boolean` rawObserver = false)	`Observable<T>`
CreateFrom(`Func<CancellationToken, IAsyncEnumerable<T>>` factory)	`Observable<T>`
CreateFrom(`TState` state, `Func<CancellationToken, TState, IAsyncEnumerable<T>>` factory)	`Observable<T>`
Defer(`Func<Observable<T>>` observableFactory)	`Observable<T>`
Empty()	`Observable<T>`
Empty(`TimeProvider` timeProvider)	`Observable<T>`
Empty(`TimeSpan` dueTime, `TimeProvider` timeProvider)	`Observable<T>`
EveryUpdate()	`Observable<Unit>`
EveryUpdate(`CancellationToken` cancellationToken)	`Observable<Unit>`
EveryUpdate(`FrameProvider` frameProvider)	`Observable<Unit>`

Name(Parameter)	ReturnType
EveryUpdate(`FrameProvider` frameProvider, `CancellationToken` cancellationToken)	`Observable<Unit>`
EveryValueChanged(`TSource` source, `Func<TSource, TProperty>` propertySelector, `CancellationToken` cancellationToken = default)	`Observable<TProperty>`
EveryValueChanged(`TSource` source, `Func<TSource, TProperty>` propertySelector, `FrameProvider` frameProvider, `CancellationToken` cancellationToken = default)	`Observable<TProperty>`
EveryValueChanged(`TSource` source, `Func<TSource, TProperty>` propertySelector, `EqualityComparer<TProperty>` equalityComparer, `CancellationToken` cancellationToken = default)	`Observable<TProperty>`
EveryValueChanged(`TSource` source, `Func<TSource, TProperty>` propertySelector, `FrameProvider` frameProvider, `EqualityComparer<TProperty>` equalityComparer, `CancellationToken` cancellationToken = default)	`Observable<TProperty>`
FromAsync(`Func<CancellationToken, ValueTask>` asyncFactory, `Boolean` configureAwait = true)	`Observable<Unit>`
FromAsync(`Func<CancellationToken, ValueTask<T>>` asyncFactory, `Boolean` configureAwait = true)	`Observable<T>`
FromEvent(`Action<Action>` addHandler, `Action<Action>` removeHandler, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
FromEvent(`Action<Action<T>>` addHandler, `Action<Action<T>>` removeHandler, `CancellationToken` cancellationToken = default)	`Observable<T>`
FromEvent(`Func<Action, TDelegate>` conversion, `Action<TDelegate>` addHandler, `Action<TDelegate>` removeHandler, `CancellationToken` cancellationToken = default)	`Observable<Unit>`

Name(Parameter)	ReturnType
FromEvent(`Func<Action<T>, TDelegate>` conversion, `Action<TDelegate>` addHandler, `Action<TDelegate>` removeHandler, `CancellationToken` cancellationToken = default)	`Observable<T>`
FromEventHandler(`Action<EventHandler>` addHandler, `Action<EventHandler>` removeHandler, `CancellationToken` cancellationToken = default)	`Observable<ValueTuple<Object, EventArgs>>`
FromEventHandler(`Action<EventHandler<TEventArgs>>` addHandler, `Action<EventHandler<TEventArgs>>` removeHandler, `CancellationToken` cancellationToken = default)	`Observable<ValueTuple<Object, TEventArgs>>`
Interval(`TimeSpan` period, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Interval(`TimeSpan` period, `TimeProvider` timeProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
IntervalFrame(`Int32` periodFrame, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
IntervalFrame(`Int32` periodFrame, `FrameProvider` frameProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Merge(params `Observable<T>[]` sources)	`Observable<T>`
Merge(`IEnumerable<Observable<T>>` sources)	`Observable<T>`
Merge(this `Observable<Observable<T>>` sources)	`Observable<T>`
Never()	`Observable<T>`
NextFrame(`CancellationToken` cancellationToken = default)	`Observable<Unit>`
NextFrame(`FrameProvider` frameProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Range(`Int32` start, `Int32` count)	`Observable<Int32>`

Name(Parameter)	ReturnType
Range(`Int32` start, `Int32` count, `CancellationToken` cancellationToken)	`Observable<Int32>`
Repeat(`T` value, `Int32` count)	`Observable<T>`
Repeat(`T` value, `Int32` count, `CancellationToken` cancellationToken)	`Observable<T>`
Return(`T` value)	`Observable<T>`
Return(`T` value, `TimeProvider` timeProvider, `CancellationToken` cancellationToken = default)	`Observable<T>`
Return(`T` value, `TimeSpan` dueTime, `TimeProvider` timeProvider, `CancellationToken` cancellationToken = default)	`Observable<T>`
Return(`Unit` value)	`Observable<Unit>`
Return(`Boolean` value)	`Observable<Boolean>`
Return(`Int32` value)	`Observable<Int32>`
ReturnFrame(`T` value, `CancellationToken` cancellationToken = default)	`Observable<T>`
ReturnFrame(`T` value, `FrameProvider` frameProvider, `CancellationToken` cancellationToken = default)	`Observable<T>`
ReturnFrame(`T` value, `Int32` dueTimeFrame, `CancellationToken` cancellationToken = default)	`Observable<T>`
ReturnFrame(`T` value, `Int32` dueTimeFrame, `FrameProvider` frameProvider, `CancellationToken` cancellationToken = default)	`Observable<T>`
ReturnOnCompleted(`Result` result)	`Observable<T>`
ReturnOnCompleted(`Result` result, `TimeProvider` timeProvider)	`Observable<T>`
ReturnOnCompleted(`Result` result, `TimeSpan` dueTime,	`Observable<T>`

Name(Parameter)	ReturnType
`TimeProvider` timeProvider)	
ReturnUnit()	`Observable<Unit>`
Throw(`Exception` exception)	`Observable<T>`
Throw(`Exception` exception, `TimeProvider` timeProvider)	`Observable<T>`
Throw(`Exception` exception, `TimeSpan` dueTime, `TimeProvider` timeProvider)	`Observable<T>`
Timer(`TimeSpan` dueTime, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Timer(`DateTimeOffset` dueTime, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Timer(`TimeSpan` dueTime, `TimeSpan` period, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Timer(`DateTimeOffset` dueTime, `TimeSpan` period, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Timer(`TimeSpan` dueTime, `TimeProvider` timeProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Timer(`DateTimeOffset` dueTime, `TimeProvider` timeProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Timer(`TimeSpan` dueTime, `TimeSpan` period, `TimeProvider` timeProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Timer(`DateTimeOffset` dueTime, `TimeSpan` period, `TimeProvider` timeProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
TimerFrame(`Int32` dueTimeFrame, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
TimerFrame(`Int32` dueTimeFrame, `Int32` periodFrame,	`Observable<Unit>`

Name(Parameter)	ReturnType
`CancellationToken` cancellationToken = default)	
TimerFrame(`Int32` dueTimeFrame, `FrameProvider` frameProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
TimerFrame(`Int32` dueTimeFrame, `Int32` periodFrame, `FrameProvider` frameProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
ToObservable(this `Task` task, `Boolean` configureAwait = true)	`Observable<Unit>`
ToObservable(this `Task<T>` task, `Boolean` configureAwait = true)	`Observable<T>`
ToObservable(this `ValueTask` task, `Boolean` configureAwait = true)	`Observable<Unit>`
ToObservable(this `ValueTask<T>` task, `Boolean` configureAwait = true)	`Observable<T>`
ToObservable(this `IEnumerable<T>` source, `CancellationToken` cancellationToken = default)	`Observable<T>`
ToObservable(this `IAsyncEnumerable<T>` source)	`Observable<T>`
ToObservable(this `IObservable<T>` source)	`Observable<T>`
Yield(`CancellationToken` cancellationToken = default)	`Observable<Unit>`
Yield(`TimeProvider` timeProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
YieldFrame(`CancellationToken` cancellationToken = default)	`Observable<Unit>`
YieldFrame(`FrameProvider` frameProvider, `CancellationToken` cancellationToken = default)	`Observable<Unit>`
Zip(params `Observable<T>[]` sources)	`Observable<T[]>`
Zip(`IEnumerable<Observable<T>>` sources)	`Observable<T[]>`

Name(Parameter)	ReturnType
ZipLatest(params `Observable<T>[]` sources)	`Observable<T[]>`
ZipLatest(`IEnumerable<Observable<T>>` sources)	`Observable<T[]>`

Methods that accept a `CancellationToken` will emit `OnCompleted` when a Cancel is issued. This allows you to unsubscribe all subscriptions from the event source.

`Range`, `Repeat`, `Return/Empty/Throw` (which do not take a `TimeProvider`) issue values immediately. This means that even if disposed of midway, the emission of values cannot be stopped. For example,

```
Observable.Range(0, int.MaxValue)
    .Do(onNext: x => Console.WriteLine($"Do:{x}"))
    .Take(10)
    .Subscribe(x => Console.WriteLine($"Subscribe:{x}"));
```

In this case, since the disposal of `Take(10)` is conveyed after the emission of `Range`, the stream does not stop. In dotnet/reactive, this could be avoided by specifying `CurrentThreadScheduler`, but it was not adopted in R3 due to a significant performance decrease.

If you want to avoid such cases, you can stop the Range by conveying a cancellation command through a `CancellationToken`.

```
var cts = new CancellationTokenSource();

Observable.Range(0, int.MaxValue, cts.Token)
    .Do(onNext: x => Console.WriteLine($"Do:{x}"))
    .Take(10)
    .DoCancelOnCompleted(cts)
    .Subscribe(x => Console.WriteLine($"Subscribe:{x}"));
```

Among our custom frame-based methods, `EveryUpdate` emits values every frame. `Yield` and `NextFrame` are similar, but `Yield` emits on the first frame loop after subscribing, while `NextFrame` delays emission to the next frame if it's in the same frame as the `FrameProvider.GetFrameCount()` value obtained at the time of subscription. `EveryValueChanged` compares values every frame and notifies when there is a change.

Operator

Operator methods are defined as extension methods to `Observable<T>` in the static class `ObservableExtensions`.

Name(Parameter)	ReturnType
AggregateAsync(this `Observable<T>` source, `Func<T, T, T>` func, `CancellationToken` cancellationToken = default)	`Task<T>`
AggregateAsync(this `Observable<T>` source, `TResult` seed, `Func<TResult, T, TResult>` func, `CancellationToken` cancellationToken = default)	`Task<TResult>`
AggregateAsync(this `Observable<T>` source, `TAccumulate` seed, `Func<TAccumulate, T, TAccumulate>` func, `Func<TAccumulate, TResult>` resultSelector, `CancellationToken` cancellationToken = default)	`Task<TResult>`
AggregateByAsync(this `Observable<TSource>` source, `Func<TSource, TKey>` keySelector, `TAccumulate` seed, `Func<TAccumulate, TSource, TAccumulate>` func, `IEqualityComparer<TKey>` keyComparer = default, `CancellationToken` cancellationToken = default)	`Task<IEnumerable<KeyValuePair<TKey, TAccumulate>>>`
AggregateByAsync(this `Observable<TSource>` source, `Func<TSource, TKey>` keySelector, `Func<TKey, TAccumulate>` seedSelector, `Func<TAccumulate, TSource, TAccumulate>` func, `IEqualityComparer<TKey>` keyComparer = default, `CancellationToken` cancellationToken = default)	`Task<IEnumerable<KeyValuePair<TKey, TAccumulate>>>`
AllAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `CancellationToken` cancellationToken = default)	`Task<Boolean>`
Amb(this `Observable<T>` source, `Observable<T>` second)	`Observable<T>`
AnyAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<Boolean>`
AnyAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `CancellationToken`	`Task<Boolean>`

Name(Parameter)	ReturnType
cancellationToken = default)	
Append(this `Observable<T>` source, `T` value)	`Observable<T>`
AsObservable(this `Observable<T>` source)	`Observable<T>`
AsSystemObservable(this `Observable<T>` source)	`IObservable<T>`
AsUnitObservable(this `Observable<T>` source)	`Observable<Unit>`
AverageAsync(this `Observable<Int32>` source, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<T>` source, `Func<T, Int32>` selector, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<Int64>` source, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<T>` source, `Func<T, Int64>` selector, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<Single>` source, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<T>` source, `Func<T, Single>` selector, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<Double>` source, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<T>` source, `Func<T, Double>` selector, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<Decimal>` source, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<T>` source, `Func<T, Decimal>` selector, `CancellationToken`	`Task<Double>`

Name(Parameter)	ReturnType
cancellationToken = default)	
AverageAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<Double>`
AverageAsync(this `Observable<TSource>` source, `Func<TSource, TResult>` selector, `CancellationToken` cancellationToken = default)	`Task<Double>`
Cast(this `Observable<T>` source)	`Observable<TResult>`
Catch(this `Observable<T>` source, `Observable<T>` second)	`Observable<T>`
Catch(this `Observable<T>` source, `Func<TException, Observable<T>>` errorHandler)	`Observable<T>`
Chunk(this `Observable<T>` source, `Int32` count)	`Observable<T[]>`
Chunk(this `Observable<T>` source, `TimeSpan` timeSpan)	`Observable<T[]>`
Chunk(this `Observable<T>` source, `TimeSpan` timeSpan, `TimeProvider` timeProvider)	`Observable<T[]>`
Chunk(this `Observable<T>` source, `TimeSpan` timeSpan, `Int32` count)	`Observable<T[]>`
Chunk(this `Observable<T>` source, `TimeSpan` timeSpan, `Int32` count, `TimeProvider` timeProvider)	`Observable<T[]>`
Chunk(this `Observable<TSource>` source, `Observable<TWindowBoundary>` windowBoundaries)	`Observable<TSource[]>`
ChunkFrame(this `Observable<T>` source)	`Observable<T[]>`
ChunkFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T[]>`
ChunkFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T[]>`
ChunkFrame(this `Observable<T>` source, `Int32` frameCount, `Int32` count)	`Observable<T[]>`

Name(Parameter)	ReturnType
ChunkFrame(this `Observable<T>` source, `Int32` frameCount, `Int32` count, `FrameProvider` frameProvider)	`Observable<T[]>`
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Func<T1, T2, TResult>` resultSelector)	`Observable<TResult>`
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Func<T1, T2, T3, TResult>` resultSelector)	`Observable<TResult>`
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Func<T1, T2, T3, T4, TResult>` resultSelector)	`Observable<TResult>`
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Func<T1, T2, T3, T4, T5, TResult>` resultSelector)	`Observable<TResult>`
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Func<T1, T2, T3, T4, T5, T6, TResult>` resultSelector)	`Observable<TResult>`
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Func<T1, T2, T3, T4, T5, T6, T7, TResult>` resultSelector)	`Observable<TResult>`
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7,	`Observable<TResult>`

Name(Parameter)	ReturnType
Observable<T8> source8, Func<T1, T2, T3, T4, T5, T6, T7, T8, TResult> resultSelector)	
CombineLatest(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Observable<T6> source6, Observable<T7> source7, Observable<T8> source8, Observable<T9> source9, Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, TResult> resultSelector)	Observable<TResult>
CombineLatest(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Observable<T6> source6, Observable<T7> source7, Observable<T8> source8, Observable<T9> source9, Observable<T10> source10, Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, TResult> resultSelector)	Observable<TResult>
CombineLatest(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Observable<T6> source6, Observable<T7> source7, Observable<T8> source8, Observable<T9> source9, Observable<T10> source10, Observable<T11> source11, Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, TResult> resultSelector)	Observable<TResult>
CombineLatest(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Observable<T6> source6, Observable<T7> source7, Observable<T8> source8, Observable<T9> source9, Observable<T10> source10, Observable<T11> source11, Observable<T12> source12, Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, TResult> resultSelector)	Observable<TResult>
CombineLatest(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3,	Observable<TResult>

Name(Parameter)	ReturnType
`Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, TResult>` resultSelector)	
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Observable<T14>` source14, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, TResult>` resultSelector)	`Observable<TResult>`
CombineLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Observable<T14>` source14, `Observable<T15>` source15, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, TResult>` resultSelector)	`Observable<TResult>`
Concat(this `Observable<T>` source, `Observable<T>` second)	`Observable<T>`
ContainsAsync(this `Observable<T>` source, `T` value, `CancellationToken` cancellationToken = default)	`Task<Boolean>`
ContainsAsync(this `Observable<T>` source, `T` value, `IEqualityComparer<T>` equalityComparer, `CancellationToken` cancellationToken = default)	`Task<Boolean>`

Name(Parameter)	ReturnType
CountAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<Int32>`
CountAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `CancellationToken` cancellationToken = default)	`Task<Int32>`
Debounce(this `Observable<T>` source, `TimeSpan` timeSpan)	`Observable<T>`
Debounce(this `Observable<T>` source, `TimeSpan` timeSpan, `TimeProvider` timeProvider)	`Observable<T>`
Debounce(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask>` throttleDurationSelector, `Boolean` configureAwait = true)	`Observable<T>`
DebounceFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`
DebounceFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
DefaultIfEmpty(this `Observable<T>` source)	`Observable<T>`
DefaultIfEmpty(this `Observable<T>` source, `T` defaultValue)	`Observable<T>`
Delay(this `Observable<T>` source, `TimeSpan` dueTime)	`Observable<T>`
Delay(this `Observable<T>` source, `TimeSpan` dueTime, `TimeProvider` timeProvider)	`Observable<T>`
DelayFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`
DelayFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
DelaySubscription(this `Observable<T>` source, `TimeSpan` dueTime)	`Observable<T>`

Name(Parameter)	ReturnType
DelaySubscription(this `Observable<T>` source, `TimeSpan` dueTime, `TimeProvider` timeProvider)	`Observable<T>`
DelaySubscriptionFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`
DelaySubscriptionFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
Dematerialize(this `Observable<Notification<T>>` source)	`Observable<T>`
Distinct(this `Observable<T>` source)	`Observable<T>`
Distinct(this `Observable<T>` source, `IEqualityComparer<T>` comparer)	`Observable<T>`
DistinctBy(this `Observable<TSource>` source, `Func<TSource, TKey>` keySelector)	`Observable<TSource>`
DistinctBy(this `Observable<TSource>` source, `Func<TSource, TKey>` keySelector, `IEqualityComparer<TKey>` comparer)	`Observable<TSource>`
DistinctUntilChanged(this `Observable<T>` source)	`Observable<T>`
DistinctUntilChanged(this `Observable<T>` source, `IEqualityComparer<T>` comparer)	`Observable<T>`
DistinctUntilChangedBy(this `Observable<T>` source, `Func<T, TKey>` keySelector)	`Observable<T>`
DistinctUntilChangedBy(this `Observable<T>` source, `Func<T, TKey>` keySelector, `IEqualityComparer<TKey>` comparer)	`Observable<T>`
Do(this `Observable<T>` source, `Action<T>` onNext = default, `Action<Exception>` onErrorResume = default, `Action<Result>` onCompleted = default, `Action` onDispose = default, `Action` onSubscribe = default)	`Observable<T>`

Name(Parameter)	ReturnType
Do(this `Observable<T>` source, `TState` state, `Action<T, TState>` onNext = default, `Action<Exception, TState>` onErrorResume = default, `Action<Result, TState>` onCompleted = default, `Action<TState>` onDispose = default, `Action<TState>` onSubscribe = default)	`Observable<T>`
DoCancelOnCompleted(this `Observable<T>` source, `CancellationTokenSource` cancellationTokenSource)	`Observable<T>`
ElementAtAsync(this `Observable<T>` source, `Int32` index, `CancellationToken` cancellationToken = default)	`Task<T>`
ElementAtAsync(this `Observable<T>` source, `Index` index, `CancellationToken` cancellationToken = default)	`Task<T>`
ElementAtOrDefaultAsync(this `Observable<T>` source, `Int32` index, `T` defaultValue = default, `CancellationToken` cancellationToken = default)	`Task<T>`
ElementAtOrDefaultAsync(this `Observable<T>` source, `Index` index, `T` defaultValue = default, `CancellationToken` cancellationToken = default)	`Task<T>`
FirstAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<T>`
FirstAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `CancellationToken` cancellationToken = default)	`Task<T>`
FirstOrDefaultAsync(this `Observable<T>` source, `T` defaultValue = default, `CancellationToken` cancellationToken = default)	`Task<T>`
FirstOrDefaultAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `T` defaultValue = default, `CancellationToken` cancellationToken = default)	`Task<T>`
ForEachAsync(this `Observable<T>` source, `Action<T>` action, `CancellationToken` cancellationToken = default)	`Task`

Name(Parameter)	ReturnType
ForEachAsync(this `Observable<T>` source, `Action<T, Int32>` action, `CancellationToken` cancellationToken = default)	`Task`
IgnoreElements(this `Observable<T>` source)	`Observable<T>`
IgnoreElements(this `Observable<T>` source, `Action<T>` doOnNext)	`Observable<T>`
IgnoreOnErrorResume(this `Observable<T>` source)	`Observable<T>`
IgnoreOnErrorResume(this `Observable<T>` source, `Action<Exception>` doOnErrorResume)	`Observable<T>`
Index(this `Observable<Unit>` source)	`Observable<Int32>`
Index(this `Observable<T>` source)	`Observable<ValueTuple<Int32, T>>`
IsEmptyAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<Boolean>`
LastAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<T>`
LastAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `CancellationToken` cancellationToken = default)	`Task<T>`
LastOrDefaultAsync(this `Observable<T>` source, `T` defaultValue = default, `CancellationToken` cancellationToken = default)	`Task<T>`
LastOrDefaultAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `T` defaultValue = default, `CancellationToken` cancellationToken = default)	`Task<T>`
LongCountAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<Int64>`
LongCountAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `CancellationToken` cancellationToken = default)	`Task<Int64>`

Name(Parameter)	ReturnType
Materialize(this `Observable<T>` source)	`Observable<Notification<T>>`
MaxAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<T>`
MaxAsync(this `Observable<T>` source, `IComparer<T>` comparer, `CancellationToken` cancellationToken = default)	`Task<T>`
MaxAsync(this `Observable<TSource>` source, `Func<TSource, TResult>` selector, `CancellationToken` cancellationToken = default)	`Task<TResult>`
MaxAsync(this `Observable<TSource>` source, `Func<TSource, TResult>` selector, `IComparer<TResult>` comparer, `CancellationToken` cancellationToken = default)	`Task<TResult>`
MaxByAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `CancellationToken` cancellationToken = default)	`Task<T>`
MaxByAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `IComparer<TKey>` comparer, `CancellationToken` cancellationToken = default)	`Task<T>`
Merge(this `Observable<T>` source, `Observable<T>` second)	`Observable<T>`
MinAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<T>`
MinAsync(this `Observable<T>` source, `IComparer<T>` comparer, `CancellationToken` cancellationToken = default)	`Task<T>`
MinAsync(this `Observable<TSource>` source, `Func<TSource, TResult>` selector, `CancellationToken` cancellationToken = default)	`Task<TResult>`

Name(Parameter)	ReturnType
MinAsync(this `Observable<TSource>` source, `Func<TSource, TResult>` selector, `IComparer<TResult>` comparer, `CancellationToken` cancellationToken = default)	`Task<TResult>`
MinByAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `CancellationToken` cancellationToken = default)	`Task<T>`
MinByAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `IComparer<TKey>` comparer, `CancellationToken` cancellationToken = default)	`Task<T>`
MinMaxAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<ValueTuple<T, T>>`
MinMaxAsync(this `Observable<T>` source, `IComparer<T>` comparer, `CancellationToken` cancellationToken = default)	`Task<ValueTuple<T, T>>`
MinMaxAsync(this `Observable<TSource>` source, `Func<TSource, TResult>` selector, `CancellationToken` cancellationToken = default)	`Task<ValueTuple<TResult, TResult>>`
MinMaxAsync(this `Observable<TSource>` source, `Func<TSource, TResult>` selector, `IComparer<TResult>` comparer, `CancellationToken` cancellationToken = default)	`Task<ValueTuple<TResult, TResult>>`
Multicast(this `Observable<T>` source, `ISubject<T>` subject)	`ConnectableObservable<T>`
ObserveOn(this `Observable<T>` source, `SynchronizationContext` synchronizationContext)	`Observable<T>`
ObserveOn(this `Observable<T>` source, `TimeProvider` timeProvider)	`Observable<T>`
ObserveOn(this `Observable<T>` source, `FrameProvider` frameProvider)	`Observable<T>`

Name(Parameter)	ReturnType
ObserveOnCurrentSynchronizationContext(this `Observable<T>` source)	`Observable<T>`
ObserveOnThreadPool(this `Observable<T>` source)	`Observable<T>`
OfType(this `Observable<T>` source)	`Observable<TResult>`
OnErrorResumeAsFailure(this `Observable<T>` source)	`Observable<T>`
Pairwise(this `Observable<T>` source)	`Observable<ValueTuple<T, T>>`
Prepend(this `Observable<T>` source, `T` value)	`Observable<T>`
Publish(this `Observable<T>` source)	`ConnectableObservable<T>`
Publish(this `Observable<T>` source, `T` initialValue)	`ConnectableObservable<T>`
RefCount(this `ConnectableObservable<T>` source)	`Observable<T>`
Replay(this `Observable<T>` source)	`ConnectableObservable<T>`
Replay(this `Observable<T>` source, `Int32` bufferSize)	`ConnectableObservable<T>`
Replay(this `Observable<T>` source, `TimeSpan` window)	`ConnectableObservable<T>`
Replay(this `Observable<T>` source, `TimeSpan` window, `TimeProvider` timeProvider)	`ConnectableObservable<T>`
Replay(this `Observable<T>` source, `Int32` bufferSize, `TimeSpan` window)	`ConnectableObservable<T>`
Replay(this `Observable<T>` source, `Int32` bufferSize, `TimeSpan` window, `TimeProvider` timeProvider)	`ConnectableObservable<T>`
ReplayFrame(this `Observable<T>` source, `Int32` window)	`ConnectableObservable<T>`
ReplayFrame(this `Observable<T>` source, `Int32` window, `FrameProvider` frameProvider)	`ConnectableObservable<T>`
ReplayFrame(this `Observable<T>` source, `Int32` bufferSize, `Int32` window)	`ConnectableObservable<T>`

Name(Parameter)	ReturnType
ReplayFrame(this `Observable<T>` source, `Int32` bufferSize, `Int32` window, `FrameProvider` frameProvider)	`ConnectableObservable<T>`
Scan(this `Observable<TSource>` source, `Func<TSource, TSource, TSource>` accumulator)	`Observable<TSource>`
Scan(this `Observable<TSource>` source, `TAccumulate` seed, `Func<TAccumulate, TSource, TAccumulate>` accumulator)	`Observable<TAccumulate>`
Select(this `Observable<T>` source, `Func<T, TResult>` selector)	`Observable<TResult>`
Select(this `Observable<T>` source, `Func<T, Int32, TResult>` selector)	`Observable<TResult>`
Select(this `Observable<T>` source, `TState` state, `Func<T, TState, TResult>` selector)	`Observable<TResult>`
Select(this `Observable<T>` source, `TState` state, `Func<T, Int32, TState, TResult>` selector)	`Observable<TResult>`
SelectAwait(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask<TResult>>` selector, `AwaitOperation` awaitOperations = AwaitOperation.Sequential, `Boolean` configureAwait = true, `Int32` maxConcurrent = -1)	`Observable<TResult>`
SelectMany(this `Observable<TSource>` source, `Func<TSource, Observable<TResult>>` selector)	`Observable<TResult>`
SelectMany(this `Observable<TSource>` source, `Func<TSource, Observable<TCollection>>` collectionSelector, `Func<TSource, TCollection, TResult>` resultSelector)	`Observable<TResult>`
SelectMany(this `Observable<TSource>` source, `Func<TSource, Int32, Observable<TResult>>` selector)	`Observable<TResult>`

Name(Parameter)	ReturnType
SelectMany(this `Observable<TSource>` source, `Func<TSource, Int32, Observable<TCollection>>` collectionSelector, `Func<TSource, Int32, TCollection, Int32, TResult>` resultSelector)	`Observable<TResult>`
SequenceEqualAsync(this `Observable<T>` source, `Observable<T>` second, `CancellationToken` cancellationToken = default)	`Task<Boolean>`
SequenceEqualAsync(this `Observable<T>` source, `Observable<T>` second, `IEqualityComparer<T>` equalityComparer, `CancellationToken` cancellationToken = default)	`Task<Boolean>`
Share(this `Observable<T>` source)	`Observable<T>`
SingleAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<T>`
SingleAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `CancellationToken` cancellationToken = default)	`Task<T>`
SingleOrDefaultAsync(this `Observable<T>` source, `T` defaultValue = default, `CancellationToken` cancellationToken = default)	`Task<T>`
SingleOrDefaultAsync(this `Observable<T>` source, `Func<T, Boolean>` predicate, `T` defaultValue = default, `CancellationToken` cancellationToken = default)	`Task<T>`
Skip(this `Observable<T>` source, `Int32` count)	`Observable<T>`
Skip(this `Observable<T>` source, `TimeSpan` duration)	`Observable<T>`
Skip(this `Observable<T>` source, `TimeSpan` duration, `TimeProvider` timeProvider)	`Observable<T>`
SkipFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`

Name(Parameter)	ReturnType
SkipFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
SkipLast(this `Observable<T>` source, `Int32` count)	`Observable<T>`
SkipLast(this `Observable<T>` source, `TimeSpan` duration)	`Observable<T>`
SkipLast(this `Observable<T>` source, `TimeSpan` duration, `TimeProvider` timeProvider)	`Observable<T>`
SkipLastFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`
SkipLastFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
SkipUntil(this `Observable<T>` source, `Observable<TOther>` other)	`Observable<T>`
SkipUntil(this `Observable<T>` source, `CancellationToken` cancellationToken)	`Observable<T>`
SkipUntil(this `Observable<T>` source, `Task` task)	`Observable<T>`
SkipWhile(this `Observable<T>` source, `Func<T, Boolean>` predicate)	`Observable<T>`
SkipWhile(this `Observable<T>` source, `Func<T, Int32, Boolean>` predicate)	`Observable<T>`
SubscribeAwait(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask>` onNextAsync, `AwaitOperation` awaitOperations = AwaitOperation.Sequential, `Boolean` configureAwait = true, `Int32` maxConcurrent = -1)	`IDisposable`
SubscribeAwait(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask>` onNextAsync, `Action<Result>` onCompleted, `AwaitOperation` awaitOperations = AwaitOperation.Sequential,	`IDisposable`

Name(Parameter)	ReturnType
Boolean configureAwait = true, Int32 maxConcurrent = -1)	
SubscribeAwait(this Observable<T> source, Func<T, CancellationToken, ValueTask> onNextAsync, Action<Exception> onErrorResume, Action<Result> onCompleted, AwaitOperation awaitOperations = AwaitOperation.Sequential, Boolean configureAwait = true, Int32 maxConcurrent = -1)	IDisposable
SubscribeOn(this Observable<T> source, SynchronizationContext synchronizationContext)	Observable<T>
SubscribeOn(this Observable<T> source, TimeProvider timeProvider)	Observable<T>
SubscribeOn(this Observable<T> source, FrameProvider frameProvider)	Observable<T>
SubscribeOnCurrentSynchronizationContext(this Observable<T> source)	Observable<T>
SubscribeOnThreadPool(this Observable<T> source)	Observable<T>
SumAsync(this Observable<Int32> source, CancellationToken cancellationToken = default)	Task<Int32>
SumAsync(this Observable<TSource> source, Func<TSource, Int32> selector, CancellationToken cancellationToken = default)	Task<Int32>
SumAsync(this Observable<Int64> source, CancellationToken cancellationToken = default)	Task<Int64>
SumAsync(this Observable<TSource> source, Func<TSource, Int64> selector, CancellationToken cancellationToken = default)	Task<Int64>
SumAsync(this Observable<Single> source, CancellationToken cancellationToken = default)	Task<Single>

Name(Parameter)	ReturnType
SumAsync(this `Observable<TSource>` source, `Func<TSource, Single>` selector, `CancellationToken` cancellationToken = default)	`Task<Single>`
SumAsync(this `Observable<Double>` source, `CancellationToken` cancellationToken = default)	`Task<Double>`
SumAsync(this `Observable<TSource>` source, `Func<TSource, Double>` selector, `CancellationToken` cancellationToken = default)	`Task<Double>`
SumAsync(this `Observable<Decimal>` source, `CancellationToken` cancellationToken = default)	`Task<Decimal>`
SumAsync(this `Observable<TSource>` source, `Func<TSource, Decimal>` selector, `CancellationToken` cancellationToken = default)	`Task<Decimal>`
SumAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<T>`
SumAsync(this `Observable<TSource>` source, `Func<TSource, TResult>` selector, `CancellationToken` cancellationToken = default)	`Task<TResult>`
Switch(this `Observable<Observable<T>>` sources)	`Observable<T>`
Synchronize(this `Observable<T>` source)	`Observable<T>`
Synchronize(this `Observable<T>` source, `Object` gate)	`Observable<T>`
Take(this `Observable<T>` source, `Int32` count)	`Observable<T>`
Take(this `Observable<T>` source, `TimeSpan` duration)	`Observable<T>`
Take(this `Observable<T>` source, `TimeSpan` duration, `TimeProvider` timeProvider)	`Observable<T>`
TakeFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`

Name(Parameter)	ReturnType
TakeFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
TakeLast(this `Observable<T>` source, `Int32` count)	`Observable<T>`
TakeLast(this `Observable<T>` source, `TimeSpan` duration)	`Observable<T>`
TakeLast(this `Observable<T>` source, `TimeSpan` duration, `TimeProvider` timeProvider)	`Observable<T>`
TakeLastFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`
TakeLastFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
TakeUntil(this `Observable<T>` source, `Observable<TOther>` other)	`Observable<T>`
TakeUntil(this `Observable<T>` source, `CancellationToken` cancellationToken)	`Observable<T>`
TakeUntil(this `Observable<T>` source, `Task` task)	`Observable<T>`
TakeWhile(this `Observable<T>` source, `Func<T, Boolean>` predicate)	`Observable<T>`
TakeWhile(this `Observable<T>` source, `Func<T, Int32, Boolean>` predicate)	`Observable<T>`
ThrottleFirst(this `Observable<T>` source, `TimeSpan` timeSpan)	`Observable<T>`
ThrottleFirst(this `Observable<T>` source, `TimeSpan` timeSpan, `TimeProvider` timeProvider)	`Observable<T>`
ThrottleFirst(this `Observable<T>` source, `Observable<TSample>` sampler)	`Observable<T>`
ThrottleFirst(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask>` sampler, `Boolean`	`Observable<T>`

Name(Parameter)	ReturnType
configureAwait = true)	
ThrottleFirstFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`
ThrottleFirstFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
ThrottleLast(this `Observable<T>` source, `TimeSpan` timeSpan)	`Observable<T>`
ThrottleLast(this `Observable<T>` source, `TimeSpan` timeSpan, `TimeProvider` timeProvider)	`Observable<T>`
ThrottleLast(this `Observable<T>` source, `Observable<TSample>` sampler)	`Observable<T>`
ThrottleLast(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask>` sampler, `Boolean` configureAwait = true)	`Observable<T>`
ThrottleLastFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`
ThrottleLastFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
Timeout(this `Observable<T>` source, `TimeSpan` dueTime)	`Observable<T>`
Timeout(this `Observable<T>` source, `TimeSpan` dueTime, `TimeProvider` timeProvider)	`Observable<T>`
TimeoutFrame(this `Observable<T>` source, `Int32` frameCount)	`Observable<T>`
TimeoutFrame(this `Observable<T>` source, `Int32` frameCount, `FrameProvider` frameProvider)	`Observable<T>`
ToArrayAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<T[]>`

Name(Parameter)	ReturnType
ToAsyncEnumerable(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`IAsyncEnumerable<T>`
ToDictionaryAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `CancellationToken` cancellationToken = default)	`Task<Dictionary<TKey, T>>`
ToDictionaryAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `IEqualityComparer<TKey>` keyComparer, `CancellationToken` cancellationToken = default)	`Task<Dictionary<TKey, T>>`
ToDictionaryAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `Func<T, TElement>` elementSelector, `CancellationToken` cancellationToken = default)	`Task<Dictionary<TKey, TElement>>`
ToDictionaryAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `Func<T, TElement>` elementSelector, `IEqualityComparer<TKey>` keyComparer, `CancellationToken` cancellationToken = default)	`Task<Dictionary<TKey, TElement>>`
ToHashSetAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<HashSet<T>>`
ToHashSetAsync(this `Observable<T>` source, `IEqualityComparer<T>` comparer, `CancellationToken` cancellationToken = default)	`Task<HashSet<T>>`
ToListAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task<List<T>>`
ToLiveList(this `Observable<T>` source)	`LiveList<T>`
ToLiveList(this `Observable<T>` source, `Int32` bufferSize)	`LiveList<T>`
ToLookupAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `CancellationToken` cancellationToken = default)	`Task<ILookup<TKey, T>>`

Name(Parameter)	ReturnType
ToLookupAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `IEqualityComparer<TKey>` keyComparer, `CancellationToken` cancellationToken = default)	`Task<ILookup<TKey, T>>`
ToLookupAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `Func<T, TElement>` elementSelector, `CancellationToken` cancellationToken = default)	`Task<ILookup<TKey, TElement>>`
ToLookupAsync(this `Observable<T>` source, `Func<T, TKey>` keySelector, `Func<T, TElement>` elementSelector, `IEqualityComparer<TKey>` keyComparer, `CancellationToken` cancellationToken = default)	`Task<ILookup<TKey, TElement>>`
Trampoline(this `Observable<T>` source)	`Observable<T>`
WaitAsync(this `Observable<T>` source, `CancellationToken` cancellationToken = default)	`Task`
Where(this `Observable<T>` source, `Func<T, Boolean>` predicate)	`Observable<T>`
Where(this `Observable<T>` source, `Func<T, Int32, Boolean>` predicate)	`Observable<T>`
Where(this `Observable<T>` source, `TState` state, `Func<T, TState, Boolean>` predicate)	`Observable<T>`
Where(this `Observable<T>` source, `TState` state, `Func<T, Int32, TState, Boolean>` predicate)	`Observable<T>`
WhereAwait(this `Observable<T>` source, `Func<T, CancellationToken, ValueTask<Boolean>>` predicate, `AwaitOperation` awaitOperations = AwaitOperation.Sequential, `Boolean` configureAwait = true, `Int32` maxConcurrent = -1)	`Observable<T>`
WithLatestFrom(this `Observable<TFirst>` first, `Observable<TSecond>` second, `Func<TFirst, TSecond,`	`Observable<TResult>`

Name(Parameter)	ReturnType
TResult> resultSelector)	
Zip(this Observable<T1> source1, Observable<T2> source2, Func<T1, T2, TResult> resultSelector)	Observable<TResult>
Zip(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Func<T1, T2, T3, TResult> resultSelector)	Observable<TResult>
Zip(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Func<T1, T2, T3, T4, TResult> resultSelector)	Observable<TResult>
Zip(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Func<T1, T2, T3, T4, T5, TResult> resultSelector)	Observable<TResult>
Zip(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Observable<T6> source6, Func<T1, T2, T3, T4, T5, T6, TResult> resultSelector)	Observable<TResult>
Zip(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Observable<T6> source6, Observable<T7> source7, Func<T1, T2, T3, T4, T5, T6, T7, TResult> resultSelector)	Observable<TResult>
Zip(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Observable<T6> source6, Observable<T7> source7, Observable<T8> source8, Func<T1, T2, T3, T4, T5, T6, T7, T8, TResult> resultSelector)	Observable<TResult>
Zip(this Observable<T1> source1, Observable<T2> source2, Observable<T3> source3, Observable<T4> source4, Observable<T5> source5, Observable<T6>	Observable<TResult>

Name(Parameter)	ReturnType
source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, TResult>` resultSelector)	
Zip(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, TResult>` resultSelector)	`Observable<TResult>`
Zip(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, TResult>` resultSelector)	`Observable<TResult>`
Zip(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, TResult>` resultSelector)	`Observable<TResult>`
Zip(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, TResult>` resultSelector)	`Observable<TResult>`

Name(Parameter)	ReturnType
Zip(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Observable<T14>` source14, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, TResult>` resultSelector)	`Observable<TResult>`
Zip(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Observable<T14>` source14, `Observable<T15>` source15, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Func<T1, T2, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Func<T1, T2, T3, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Func<T1, T2, T3, T4, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Func<T1, T2, T3, T4, T5, TResult>` resultSelector)	`Observable<TResult>`

Name(Parameter)	ReturnType
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Func<T1, T2, T3, T4, T5, T6, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Func<T1, T2, T3, T4, T5, T6, T7, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Func<T1, T2, T3, T4, T5, T6, T7, T8, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7,	`Observable<TResult>`

Name(Parameter)	ReturnType
`Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, TResult>` resultSelector)	
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7, `Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Observable<T14>` source14, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, TResult>` resultSelector)	`Observable<TResult>`
ZipLatest(this `Observable<T1>` source1, `Observable<T2>` source2, `Observable<T3>` source3, `Observable<T4>` source4, `Observable<T5>` source5, `Observable<T6>` source6, `Observable<T7>` source7,	`Observable<TResult>`

Name(Parameter)	ReturnType
`Observable<T8>` source8, `Observable<T9>` source9, `Observable<T10>` source10, `Observable<T11>` source11, `Observable<T12>` source12, `Observable<T13>` source13, `Observable<T14>` source14, `Observable<T15>` source15, `Func<T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, TResult>` resultSelector)	

In dotnet/reactive, methods that return a single `IObservable<T>` (such as `First`) are all provided only as `***Async`, returning `Task<T>`. Additionally, to align with the naming of Enumerable, `Buffer` has been changed to `Chunk`.

`Throttle` has been changed to `Debounce`, and `Sample` has been changed to `ThrottleLast`. Originally in dotnet/reactive, there were only `Throttle` and `Sample`. But `Sample` needs both first and last, and many Rx libraries defined it as `ThrottleFirst`, the behavior of `ThrottleFirst` is similar to `Sample` (which is `ThrottleLast`), whereas `Throttle` has a completely different behavior. Therefore, `Throttle` was changed to the more commonly used `Debounce`, and `Sample` was changed to `ThrottleLast` for symmetry with `ThrottleFirst`. Additionally, I am opposed to keeping `Sample` as an alias for `ThrottleLast`. As a result of such methods being maintained, other libraries often receive questions like "What is the difference between `ThrottleLast` and `Sample`?"

Class/Method name changes from dotnet/reactive and neuecc/UniRx

- `Buffer` -> `Chunk`
- `BatchFrame` -> `ChunkFrame`
- `Throttle` -> `Debounce`
- `ThrottleFrame` -> `DebounceFrame`
- `Sample` -> `ThrottleLast`
- `SampleFrame` -> `ThrottleLastFrame`
- `ObserveEveryValueChanged(this T value)` -> `Observable.EveryValueChanged(T value)`
- `DistinctUntilChanged(selector)` -> `DistinctUntilChangedBy`
- `Finally` -> `Do(onDisposed:)`
- `Do***` -> `Do(on***:)`
- `BehaviorSubject` -> `ReactiveProperty`
- `AsyncSubject<T>` -> `TaskCompletionSource<T>`
- `StableCompositeDisposable` -> `Disposable.Combine`
- `IScheduler` -> `TimeProvider`
- `ReactiveCollection` / `ReactiveDictionary` -> ObservableCollections.R3

- Return single value methods -> `***Async`
- `ObjectPool` in UniRx -> use UniTask and make yourself
- MessageBroker in UniRx -> MessagePipe
- Logger in UniRx -> ZLogger

Similar to `IObservable<T>`, if you want to stop the stream when an `OnErrorResume` occurs, you connect `OnErrorResumeAsFailure` in the method chain.

License

This library is under the MIT License.

Namespace R3

Namespaces

R3.Triggers

Classes

MonoBehaviourExtensions

ObserveOnExtensions

PlayerLoopHelper

R3LoopRunners

SerializableReactiveProperty<T>

UnityEventExtensions

UnityFrameProvider

UnityProviderInitializer

UnityTimeProvider

Structs

R3LoopRunners.EarlyUpdate

R3LoopRunners.FixedUpdate

R3LoopRunners.Initialization

R3LoopRunners.PostLateUpdate

R3LoopRunners.PreLateUpdate

R3LoopRunners.PreUpdate

R3LoopRunners.TimeUpdate

R3LoopRunners.Update

Enums

PlayerLoopTiming