

# Rx Design Guidelines

---



## 1. Table of Contents

1. Table of Contents .....	2
2. Introduction .....	4
3. When to use Rx .....	5
3.1. Use Rx for orchestrating asynchronous and event-based computations.....	5
3.2. Use Rx to deal with asynchronous sequences of data.....	6
4. The Rx contract .....	8
4.1. Assume the Rx Grammar .....	8
4.2. Assume observer instances are called in a serialized fashion .....	8
4.3. Assume resources are cleaned up after an <i>OnError</i> or <i>OnCompleted</i> message.....	9
4.4. Assume a best effort to stop all outstanding work on Unsubscribe.....	10
5. Using Rx.....	11
5.1. Consider drawing a Marble-diagram .....	11
5.2. Consider passing multiple arguments to Subscribe.....	11
5.3. Consider using LINQ query expression syntax .....	12
5.4. Consider passing a specific scheduler to concurrency introducing operators .....	12
5.5. Call the <i>ObserveOn</i> operator as late and in as few places as possible .....	13
5.6. Consider limiting buffers.....	13
5.7. Make side-effects explicit using the <i>Do</i> operator .....	14
5.8. Use the Synchronize operator only to “fix” custom IObservable implementations.....	14
5.9. Assume messages can come through until unsubscribe has completed .....	15
5.10. Use the <i>Publish</i> operator to share side-effects.....	15
6. Operator implementations .....	17
6.1. Implement new operators by composing existing operators.....	17
6.2. Implement custom operators using Observable.Create(WithDisposable).....	17
6.3. Implement operators for existing observable sequences as generic extension methods. ....	18
6.4. Protect calls to user code from within an operator.....	19
6.5. Subscribe implementations should not throw .....	20
6.6. <i>OnError</i> messages should have abort semantics .....	21
6.7. Serialize calls to <i>IObserver</i> methods within observable sequence implementations .....	22
6.8. Avoid serializing operators.....	24
6.9. Parameterize concurrency by providing a scheduler argument.....	25

6.10.	Provide a default scheduler .....	25
6.11.	The scheduler should be the last argument to the operator.....	26
6.12.	Avoid introducing concurrency .....	27
6.13.	Hand out all disposables instances created inside the operator to consumers .....	28
6.14.	Operators should not block .....	30
6.15.	Avoid deep stacks caused by recursion in operators.....	31
6.16.	Argument validation should occur outside <i>Observable.Create(WithDisposable)</i> .....	32
6.17.	Unsubscription should be idempotent .....	33
6.18.	Unsubscription should not throw .....	33
6.19.	Custom <i>IObservable</i> implementations should follow the Rx contract .....	33
6.20.	Operator implementations should follow guidelines for Rx usage .....	34

## 2. Introduction

This document describes guidelines that aid in developing applications and libraries that use the Reactive Extensions library (<http://go.microsoft.com/fwlink/?LinkId=179929>).

The guidelines listed in this document have evolved over time by the Rx team during the development of the Rx library.

As Rx continues to evolve, these guidelines will continue to evolve with it. Make sure you have the latest version of this document. Updates are announced on the Rx forum:

<http://go.microsoft.com/fwlink/?LinkId=201727>

All information described in this document is merely a set of guidelines to aid development. These guidelines do not constitute an absolute truth. They are patterns that the team found helpful; not rules that should be followed blindly. There are situations where certain guidelines do not apply. The team has tried to list known situations where this is the case. It is up to each individual developer to decide if a certain guideline makes sense in each specific situation.

The guidelines in this document are listed in no particular order. There is neither total nor partial ordering in these guidelines.

Please contact us through the Rx forum: <http://go.microsoft.com/fwlink/?LinkId=201727> for feedback on the guidelines, as well as questions on whether certain guidelines are applicable in specific situations.

### 3. When to use Rx

#### 3.1. Use Rx for orchestrating asynchronous and event-based computations

Code that deals with more than one event or asynchronous computation gets complicated quickly as it needs to build a state-machine to deal with ordering issues. Next to this, the code needs to deal with successful and failure termination of each separate computation. This leads to code that doesn't follow normal control-flow, is hard to understand and hard to maintain.

Rx makes these computations first-class citizens. This provides a model that allows for readable and composable APIs to deal with these asynchronous computations.

##### Sample

```
var scheduler = new ControlScheduler(this);
var keyDown = Observable.FromEvent<KeyEventHandler, KeyEventArgs>(
    d => d.Invoke, h => textBox.KeyUp += h, h => textBox.KeyUp -= h);

var dictionarySuggest = keyDown
    .Select(_ => textBox1.Text)
    .Where(text => !string.IsNullOrEmpty(text))
    .DistinctUntilChanged()
    .Throttle(TimeSpan.FromMilliseconds(250), scheduler)
    .SelectMany(
        text => AsyncLookupInDictionary(text)
            .TakeUntil(keyDown));

dictionarySuggest.Subscribe(
    results =>
        listView1.Items.AddRange(results.Select(
            result=>new ListViewItem(result)).ToArray()),
    error => LogError(error));
```

This sample models a common UI paradigm of receiving completion suggestions while the user is typing input.

Rx creates an observable sequence that models an existing *KeyUp* event (the original WinForms code did not have to be modified).

It then places several filters and projections on top of the event to make the event only fire if a unique value has come through. (The *KeyUp* event fires for every key stroke, so also if the user presses left or right arrow, moving the cursor but not changing the input text).

Next it makes sure the event only gets fired after 250 milliseconds of activity by using the Throttle operator. (If the user is still typing characters, this saves a potentially expensive lookup that will be ignored immediately). A scheduler is passed to ensure the 250 milliseconds delay is issued on the UI thread.

In traditionally written programs, this throttling would introduce separate callbacks through a timer. This timer could potentially throw exceptions (certain timers have a maximum amount of operations in flight).

Once the user input has been filtered down it is time to perform the dictionary lookup. As this is usually an expensive operation (e.g. a request to a server on the other side of the world), this operation is itself asynchronous as well.

The *SelectMany* operator allows for easy combining of multiple asynchronous operations. It doesn't only combine success values; it also tracks any exceptions that happen in each individual operation.

In traditionally written programs, this would introduce separate callbacks and a place for exceptions occurring.

If the user types a new character while the dictionary operation is still in progress, we do not want to see the results of that operation anymore. The user has typed more characters leading to a more specific word, seeing old results would be confusing.

The *TakeUntil(keyDown)* operation makes sure that the dictionary operation is ignored once a new keyDown has been detected.

Finally we subscribe to the resulting observable sequence. Only at this time our execution plan will be used. We pass two functions to the *Subscribe* call:

1. Receives the result from our computation.
2. Receives exceptions in case of a failure occurring anywhere along the computation.

### When to ignore this guideline

If the application/library in question has very few asynchronous/event-based operations or has very few places where these operations need to be composed, the cost of depending on Rx (redistributing the library as well as the learning curve) might outweigh the cost of manually coding these operations.

## 3.2. Use Rx to deal with asynchronous sequences of data

Several other libraries exist to aid asynchronous operations on the .NET platform. Even though these libraries are powerful, they usually work best on operations that return a single message. They usually do not support operations that produce multiple messages over the lifetime of the operation.

Rx follows the following grammar: *OnNext\** (*OnCompleted|OnError*)? (see chapter 0). This allows for multiple messages to come in over time. This makes Rx suitable for both operations that produce a single message, as well as operations that produce multiple messages.

## Sample

```
//open a 4GB file for asynchronous reading in blocks of 64K
var inFile = new FileStream(@"d:\temp\4GBfile.txt",
    FileMode.Open, FileAccess.Read, FileShare.Read,
    2 << 15, true);

//open a file for asynchronous writing in blocks of 64K
var outFile = new FileStream(@"d:\temp\Encrypted.txt",
    FileMode.OpenOrCreate, FileAccess.Write, FileShare.None,
    2 << 15, true);

inFile.AsyncRead(2 << 15)
    .Select(Encrypt)
    .WriteToStream(outFile)
    .Subscribe(
        _ => Console.WriteLine("Successfully encrypted the file."),
        error => Console.WriteLine(
            "An error occurred while encrypting the file: {0}",
            error.Message));
```

In this sample, a 4 GB file is read in its entirety, encrypted and saved out to another file.

Reading the whole file into memory, encrypting it and writing out the whole file would be an expensive operation.

Instead, we rely on the fact that Rx can produce many messages.

We read the file asynchronously in blocks of 64K. This produces an observable sequence of byte arrays. We then encrypt each block separately (for this sample we assume the encryption operation can work on separate parts of the file). Once the block is encrypted, it is immediately sent further down the pipeline to be saved to the other file. The *WriteToStream* operation is an asynchronous operation that can process multiple messages.

### When to ignore this guideline

If the application/library in question has very few operations with multiple messages, the cost of depending on Rx (redistributing the library as well as the learning curve) might outweigh the cost of manually coding these operations.

## 4. The Rx contract

The interfaces *IObservable<T>* and *IObserver<T>* only specify the arguments and return types their methods. The Rx library makes more assumptions about these two interfaces than is expressed in the .NET type system. These assumptions form a contract that should be followed by all producers and consumers of Rx types. This contract ensures it is easy to reason about and prove the correctness of operators and user code.

### 4.1. Assume the Rx Grammar

Messages sent to instances of the *IObserver* interface follow the following grammar:

`OnNext* (OnCompleted | OnError)?`

This grammar allows observable sequences to send any amount (0 or more) of *OnNext* messages to the subscribed observer instance, optionally followed by a single success (*OnCompleted*) or failure (*OnError*) message.

The single message indicating that an observable sequence has finished ensures that consumers of the observable sequence can deterministically establish that it is safe to perform cleanup operations.

A single failure further ensures that abort semantics can be maintained for operators that work on multiple observable sequences (see paragraph 6.6).

#### Sample

```
var count = 0;
xs.Subscribe(v =>
{
    count++;
},
e=> Console.WriteLine(e.Message),
()=>Console.WriteLine("OnNext has been called {0} times.", count)
);
```

In this sample we safely assume that the total amount of calls to the *OnNext* method won't change once the *OnCompleted* method is called as the observable sequence follows the Rx grammar.

#### When to ignore this guideline

Ignore this guideline only when working with a non-conforming implementation of the *IObservable* interface. It is possible to make the observable sequence conforming by calling the *Synchronize* operator on the instance.

### 4.2. Assume observer instances are called in a serialized fashion

As Rx uses a push model and .NET supports multithreading, it is possible for different messages to arrive different execution contexts at the same time. If consumers of observable sequences would have to deal with this in every place, their code would need to perform a lot of housekeeping to avoid common



concurrency problems. Code written in this fashion would be harder to maintain and potentially suffer from performance issues.

As not all observable sequences are prone to having messages from different execution contexts, only the operators that produce such observable sequences are required to perform serialization (see paragraph 6.7). Consumers of observables can safely assume that messages arrive in a serialized fashion.

### Sample

```
var count = 0;
xs.Subscribe(v =>
{
    count++;
    Console.WriteLine("OnNext has been called {0} times.", count);
});
```

In this sample, no locking or interlocking is required to read and write to count as only one call to OnNext can be in-flight at any time.

### When to ignore this guideline

If you have to consume a custom implementation of an observable sequence that doesn't follow the Rx contract for serialization, use the *Synchronize* operator to ensure you can still follow this guideline.

## 4.3. Assume resources are cleaned up after an *OnError* or *OnCompleted* message

Paragraph 4.1 states that no more messages should arrive after an *OnError* or *OnCompleted* message. This makes it possible to cleanup any resource used by the subscription the moment an *OnError* or *OnCompleted* arrives. Cleaning up resources immediately will make sure that any side-effect occurs in a predictable fashion. It also makes sure that the runtime can reclaim these resources.

### Sample

```
Observable.Using(
    () => new FileStream(@"d:\temp\test.txt", FileMode.Create),
    fs => Observable.Range(0, 10000)
        .Select(v => Encoding.ASCII.GetBytes(v.ToString()))
        .WriteToStream(fs))
    .Subscribe();
```

In this sample the *Using* operator creates a resource that will be disposed upon unsubscription. The Rx contract for cleanup ensures that unsubscription will be called automatically once an *OnError* or *OnCompleted* message is sent.

### When to ignore this guideline

There are currently no known cases where to ignore this guideline.

#### 4.4. Assume a best effort to stop all outstanding work on Unsubscribe

When unsubscribe is called on an observable subscription, the observable sequence will make a best effort attempt to stop all outstanding work. This means that any queued work that has not been started will not start.

Any work that is already in progress might still complete as it is not always safe to abort work that is in progress. Results from this work will not be signaled to any previously subscribed observer instances.

##### Sample 1

```
Observable.Timer(TimeSpan.FromSeconds(2)).Subscribe(...).Dispose()
```

In this sample subscribing to the observable sequence generated by *Timer* will queue an action on the *ThreadPool* scheduler to send out an *OnNext* message in 2 seconds. The subscription then gets canceled immediately. As the scheduled action has not started yet, it will be removed from the scheduler.

##### Sample 2

```
Observable.Start(()=>
{
    Thread.Sleep(TimeSpan.FromSeconds(2));
    return 5;
})
.Subscribe(...).Dispose();
```

In this sample the *Start* operator will immediately schedule the execution of the lambda provided as its argument. The subscription registers the observer instance as a listener to this execution. As the lambda is already running once the subscription is disposed, it will keep running and its return value is ignored.

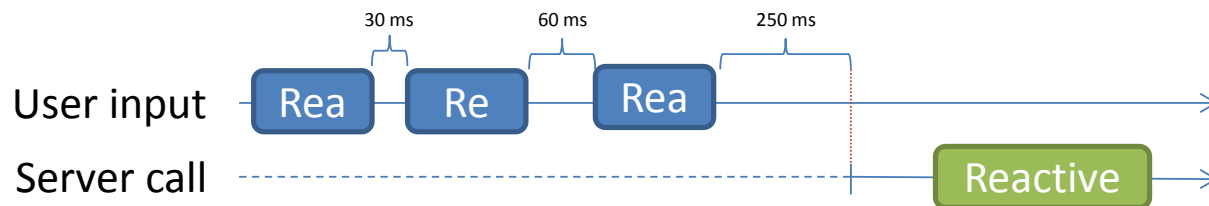
## 5. Using Rx

### 5.1. Consider drawing a Marble-diagram

Draw a marble-diagram of the observable sequence you want to create. By drawing the diagram, you will get a clearer picture on what operator(s) to use.

A marble-diagram is a diagram that shows event occurring over time. A marble diagram contains both input and output sequences(s).

#### Sample



By drawing the diagram we can see that we will need some kind of delay after the user input, before firing of another asynchronous call. The delay in this sample maps to the *Throttle* operator. To create another observable sequence from an observable sequence we will use the *SelectMany* operator. This will lead to the following code:

```
var dictionarySuggest =userInput
    .Throttle(TimeSpan.FromMilliseconds(250))
    .SelectMany(input => serverCall(input));
```

#### When to ignore this guideline

This guideline can be ignored if you feel comfortable enough with the observable sequence you want to write. However, even the Rx team members will still grab the whiteboard to draw a marble-diagram once in a while.

### 5.2. Consider passing multiple arguments to Subscribe

For convenience, Rx provides extensions to the *Subscribe* method that takes delegates instead of an *IObserver* argument. These overloads make subscribing a lot easier as C# and VB do not support anonymous inner-classes.

The *IObserver* interface would require implementing all three methods (*OnNext*, *OnError* & *OnCompleted*). The extensions to the *Subscribe* method allow developers to use defaults chosen for each of these methods.

E.g. when calling the *Subscribe* method that only has an *onNext* argument, the *OnError* behavior will be to rethrow the exception on the thread that the message comes out from the observable sequence. The *OnCompleted* behavior in this case is to do nothing.

In many situations, it is important to deal with the exception (either recover or abort the application gracefully).

Often it is also important to know that the observable sequence has completed successfully. For example, the application notifies the user that the operation has completed.

Because of this, it is best to provide all 3 arguments to the subscribe function.

#### When to ignore this guideline

- When the observable sequence is guaranteed not to complete, e.g. an event such as *KeyUp*.
- When the observable sequence is guaranteed not to have *OnError* messages (e.g. an event, a materialized observable sequence etc...).
- When the default behavior is the desirable behavior.

### 5.3. Consider using LINQ query expression syntax

Rx implements the query expression pattern as described in the C# 3.0 specification. Because of this, it is possible to use the LINQ query expression syntax to write queries over observable sequences.

#### Sample

Consider the following query:

```
var r = xs.SelectMany(x => ys, (x,y) => x + y);
```

This query can be written as:

```
var r1 = from x in xs
         from y in ys
         select x + y;
```

#### When to ignore this guideline

Consider ignoring this guideline if you need to use many operators in your queries that are not supported in the query expression syntax. This might negate the readability argument.

### 5.4. Consider passing a specific scheduler to concurrency introducing operators

Rather than using the *ObserveOn* operator to change the execution context on which the observable sequence produces messages, it is better to create concurrency in the right place to begin with. As operators parameterize introduction of concurrency by providing a scheduler argument overload, passing the right scheduler will lead to fewer places where the *ObserveOn* operator has to be used.

#### Sample

```
var keyup = Observable.FromEvent<KeyEventArgs>(textBox, "KeyUp");
var throttled = keyup.Throttle(TimeSpan.FromSeconds(1),
    Scheduler.Dispatcher);
```

In this sample, callbacks from the *KeyUp* event arrive on the UI thread. The default overload of the *Throttle* operator would place *OnNext* messages on the *ThreadPool* (as it uses the *ThreadPool* timer to time the throttling). By providing the *Scheduler.Dispatcher* instance to the *Throttle* operator, all messages from this observable sequence will originate on the UI thread.

#### When to ignore this guideline

When combining several events that originate on different execution contexts, use guideline 5.5 to put all messages on a specific execution context as late as possible.

### 5.5. Call the *ObserveOn* operator as late and in as few places as possible

By using the *ObserveOn* operator, an action is scheduled for each message that comes through the original observable sequence. This potentially changes timing information as well as puts additional stress on the system. Placing this operator later in the query will reduce both concerns.

#### Sample

```
var result =  
    (from x in xs.Throttle(TimeSpan.FromSeconds(1))  
     from y in ys.TakeUntil(zs).Sample(TimeSpan.FromMilliseconds(250))  
     select x + y)  
     .Merge(ws)  
     .Where(x => x.Length < 10)  
     .ObserveOn(Scheduler.Dispatcher);
```

This sample combines many observable sequences running on many different execution contexts. The query filters out a lot of messages. Placing the *ObserveOn* operator earlier in the query would do extra work on messages that would be filtered out anyway. Calling the *ObserveOn* operator at the end of the query will create the least performance impact.

#### When to ignore this guideline

Ignore this guideline if your use of the observable sequence is not bound to a specific execution context. In that case do not use the *ObserveOn* operator.

### 5.6. Consider limiting buffers

Rx comes with several operators and classes that create buffers over observable sequences, e.g. the *Replay* operator. As these buffers work on any observable sequence, the size of these buffers will depend on the observable sequence it is operating on. If the buffer is unbounded, this can lead to memory pressure. Many buffering operators provide policies to limit the buffer, either in time or size. Providing this limit will address memory pressure issues.

### Sample

```
var result = xs.Replay(10000, TimeSpan.FromHours(1));
```

In this sample, the *Replay* operator creates a buffer. We have limited that buffer to contain at most 10000 messages and keep these messages around for a maximum of 1 hour.

#### When to ignore this guideline

When the amount of messages created by the observable sequence that populates the buffer is small or when the buffer size is limited.

## 5.7. Make side-effects explicit using the *Do* operator

As many Rx operators take delegates as arguments, it is possible to pass any valid user code in these arguments. This code can change global state (e.g. change global variables, write to disk etc...).

The composition in Rx runs through each operator for each subscription (with the exception of the sharing operators, such as *Publish*). This will make every side-effect occur for each subscription.

If this behavior is the desired behavior, it is best to make this explicit by putting the side-effecting code in a *Do* operator.

### Sample

```
var result = xs.Where(x=>x.Failed).Do(x=>Log(x)).Subscribe(...);
```

In this sample, messages are filtered for failure. The messages are logged before handing them out to the code subscribed to this observable sequence. The logging is a side-effect (e.g. placing the messages in the computer's event log) and is explicitly done via a call to the *Do* operator.

#### When to ignore this guideline

Ignore this guideline when the side effect requires data from an operator that is not available to the *Do* operator.

## 5.8. Use the *Synchronize* operator only to “fix” custom *IObservable* implementations

Observable sequences that are created by the Rx operators already follow the Rx contract for grammar (see paragraph 4.1) and serialization (see paragraph 4.2). There is no need to use the *Synchronize* operator on these observable sequences. Only use the *Synchronize* operator on observable sequences that were created by other sources and do not follow the Rx contract for synchronization (see paragraph 4.2).

### Sample

```
var result = from x in xs.Synchronize()  
             from y in ys  
             where x > y  
             select y;
```

In this sample only the observable sequence created by another source that doesn't follow the Rx contract for synchronization is synchronized. All other operations are already synchronized and do not require the *Synchronize* operator.

### When to ignore this guideline

There are currently no known cases where to ignore this guideline.

## 5.9. Assume messages can come through until unsubscribe has completed

As Rx uses a push model, messages can be sent from different execution contexts. Messages can be in flight while calling unsubscribe. These messages can still come through while the call to unsubscribe is in progress. After control has returned, no more messages will arrive. The unsubscription process can still be in progress on a different context.

### When to ignore this guideline

Once the *OnCompleted* or *OnError* method has been received, the Rx grammar guarantees that the subscription can be considered to be finished.

## 5.10. Use the *Publish* operator to share side-effects

As many observable sequences are cold (see [cold vs. hot on Channel 9](#)), each subscription will have a separate set of side-effects. Certain situations require that these side-effects occur only once. The *Publish* operator provides a mechanism to share subscriptions by broadcasting a single subscription to multiple subscribers.

There are several overloads of the *Publish* operator. The most convenient overloads are the ones that provide a function with a wrapped observable sequence argument that shares the side-effects.

## Sample

```
var xs = Observable.CreateWithDisposable<string>(observer =>
{
    Console.WriteLine("Side effect");
    observer.OnNext("Hi");
    observer.OnCompleted();
    return Disposable.Empty;
});
xs.Publish(sharedXs =>
{
    sharedXs.Subscribe(Console.WriteLine);
    sharedXs.Subscribe(Console.WriteLine);
    return sharedXs;
}).Run();
```

In this sample, *xs* is an observable sequence that has side-effects (writing to the console). Normally each separate subscription will trigger these side-effects. The *Publish* operator uses a single subscription to *xs* for all subscribers to *sharedXs*.

### When to ignore this guideline

Only use the *Publish* operator to share side-effects when sharing is required. In most situations you can create separate subscriptions without any problems: either the subscriptions do not have side-effects or the side effects can execute multiple times without any issues.



## 6. Operator implementations

### 6.1. Implement new operators by composing existing operators.

Many operations can be composed from existing operators. This will lead to smaller, easier to maintain code. The Rx team has put a lot of effort in dealing with all corner cases in the base operators. By reusing these operators you'll get all that work for free in your operator.

#### Sample

```
public static IObservable<TResult> SelectMany<TSource, TResult>(
    this IObservable<TSource> source,
    Func<TSource, IObservable<TResult>> selector)
{
    return source.Select(selector).Merge();
}
```

In this sample, the *SelectMany* operator uses two existing operators: *Select* and *Merge*. The *Select* operator already deals with any issues around the selector function throwing an exception. The *Merge* operator already deals with concurrency issues of multiple observable sequences firing at the same time.

#### When to ignore this guideline

- No appropriate set of base operators is available to implement this operator.
- Performance analysis proves that the implementation using existing operators has performance issues.

### 6.2. Implement custom operators using *Observable.Create(WithDisposable)*

When it is not possible to follow guideline 6.1, use the *Observable.Create(WithDisposable)* method to create an observable sequence as it provides several protections make the observable sequence follow the Rx contract (see chapter 0):

- When the observable sequence has finished (either by firing *OnError* or *OnCompleted*), any subscription will automatically be unsubscribed.
- Any subscribed observer instance will only see a single *OnError* or *OnCompleted* message. No more messages are sent through. This ensures the Rx grammar of *OnNext\** (*OnError|OnCompleted*)?

### Sample

```
public static IObservable<TResult> Select<TSource, TResult>(
    this IObservable<TSource> source, Func<TSource, TResult> selector)
{
    return Observable.CreateWithDisposable<TResult>(
        observer => source.Subscribe(
            x =>
            {
                TResult result;
                try
                {
                    result = selector(x);
                }
                catch (Exception exception)
                {
                    observer.OnError(exception);
                    return;
                }
                observer.OnNext(result);
            },
            observer.OnError,
            observer.OnCompleted));
}
```

In this sample, `Select` uses the *Observable.CreateWithDisposable* operator to return a new instance of the *IObservable* interface. This ensures that no matter the implementation of the source observable sequence, the output observable sequence follows the Rx contract (see chapter 0). It also ensures that the lifetime of subscriptions is as short as possible.

#### When to ignore this guideline

- The operator needs to return an observable sequence that doesn't follow the Rx contract. This should usually be avoided (except when writing tests to see how code behaves when the contract is broken).
- The object returned needs to implement more than the *IObservable* interface (e.g. *ISubject*, or a custom class).

### 6.3. Implement operators for existing observable sequences as generic extension methods.

An operator becomes more powerful if it can be applied in many cases. If an operator is implemented as an extension method, it is visible in Intellisense on any existing observable sequence. If the operator is made generic, it can be applied regardless of the data inside the observable sequence.

## Sample

```
public static IObservable<TResult> Select<TSource, TResult>(
    this IObservable<TSource> source, Func<TSource, TResult> selector)
{
    return Observable.CreateWithDisposable<TResult>(
        observer => source.Subscribe(
            x =>
            {
                TResult result;
                try
                {
                    result = selector(x);
                }
                catch (Exception exception)
                {
                    observer.OnError(exception);
                    return;
                }
                observer.OnNext(result);
            },
            observer.OnError,
            observer.OnCompleted));
}
```

In this sample, `Select` is defined as an extension method. Because of this, the operator is visible to any observable sequence. The work that this operator does is applicable to any observable sequence so it can be defined using generics.

### When to ignore this guideline

- The operator does not work on a source observable sequence.
- The operator works on a specific kind of data and cannot be made generic.

## 6.4. Protect calls to user code from within an operator

When user code is called from within an operator, this is potentially happening outside of the execution context of the call to the operator (asynchronously). Any exception that happens here will cause the program to terminate unexpectedly. Instead it should be fed through to the subscribed observer instance so that the exception can be dealt with by the subscribers.

Common kinds of user code that should be protected:

- Selector functions passed in to the operator.
- Comparers passed into the operator.
- Calls to dictionaries, lists and hashsets that use a user-provided comparer.

**Note:** calls to *IScheduler* implementations are not considered for this guideline. The reason for this is that only a small set of issues would be caught as most schedulers deal with asynchronous calls. Instead, protect the arguments passed to schedulers inside each scheduler implementation.

### Sample

```
public static IObservable Select(
    this IObservable source, Func selector)
{
    return Observable.CreateWithDisposable(
        observer => source.Subscribe(
            x =>
            {
                TResult result;
                try
                {
                    result = selector(x);
                }
                catch (Exception exception)
                {
                    observer.OnError(exception);
                    return;
                }
                observer.OnNext(result);
            },
            observer.OnError,
            observer.OnCompleted));
}
```

This sample invokes a selector function which is user code. It catches any exception resulting from this call and transfers the exception to the subscribed observer instance through the *OnError* call.

### When to ignore this guideline

Ignore this guideline for calls to user code that are made before creating the observable sequence (outside of the *Observable.Create(WithDisposable)* call). These calls are on the current execution context and are allowed to follow normal control flow.

**Note:** do not protect calls to *Subscribe*, *Dispose*, *OnNext*, *OnError* and *OnCompleted* methods. These calls are on the edge of the monad. Calling the *OnError* method from these places will lead to unexpected behavior.

## 6.5. Subscribe implementations should not throw

As multiple observable sequences are composed, subscribe to a specific observable sequence might not happen at the time the user calls *Subscribe* (e.g. Within the *Concat* operator, the second observable sequence argument to *Concat* will only be subscribed to once the first observable sequence has completed). Throwing an exception would bring down the program. Instead exceptions in subscribe should be tunneled to the *OnError* method.

### Sample

```
public IObservable<byte[]> ReadSocket(Socket socket)
{
    return Observable.CreateWithDisposable<byte[]>(observer =>
    {
        if (!socket.Connected)
        {
            observer.OnError(new InvalidOperationException(
                "the socket is no longer connected"));
            return Disposable.Empty;
        }
        ...
    });
}
```

In this sample, an error condition is detected in the subscribe method implementation. An error is raised by calling the *OnError* method instead of throwing the exception. This allows for proper handling of the exception if Subscribe is called outside of the execution context of the original call to Subscribe by the user.

### When to ignore this guideline

When a catastrophic error occurs that should bring down the whole program anyway.

## 6.6. *OnError* messages should have abort semantics

As normal control flow in .NET uses abort semantics for exceptions (the stack is unwound, current code path is interrupted), Rx mimics this behavior. To ensure this behavior, no messages should be sent out by an operator once one of its sources has an error message or an exception is thrown within the operator.

### Sample

```
public static IObservable<byte[]> MinimumBuffer(
    this IObservable<byte[]> source, int bufferSize)
{
    return Observable.CreateWithDisposable<byte[]>(
        observer =>
        {
            var data = new List<byte>();

            return source.Subscribe(value =>
            {
                data.AddRange(value);

                if (data.Count > bufferSize)
                {
                    observer.OnNext(data.ToArray());
                }
            });
        });
}
```

```
        data.Clear();
    }
},
observer.OnError,
() =>
{
    if (data.Count > 0)
        observer.OnNext(data.ToArray());

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

In this sample, a buffering operator will abandon the observable sequence as soon as the subscription to source encounters an error. The current buffer is not sent to any subscribers, maintain abort semantics.

#### When to ignore this guideline

There are currently no known cases where to ignore this guideline.

### 6.7. Serialize calls to *IObserver* methods within observable sequence implementations

Rx is a composable API, many operators can play together. If all operators had to deal with concurrency the individual operators would become very complex. Next to this, concurrency is best controlled at the place it first occurs. Finally, Consuming the Rx API would become harder if each usage of Rx would have to deal with concurrency.

#### Sample

```
public static IObservable<TResult> ZipEx<TLeft, TRight, TResult>(
    this IObservable<TLeft> left,
    IObservable<TRight> right,
    Func<TLeft, TRight, TResult> resultSelector)
{
    return Observable.CreateWithDisposable<TResult>(observer =>
    {
        var group = new CompositeDisposable();
        var gate = new object();

        var leftQ = new Queue<TLeft>();
        var rightQ = new Queue<TRight>();

        group.Add(left.Subscribe(
            value =>
            {
                lock(gate)
                {
                    if (rightQ.Count > 0)
```

```
        {
            var rightValue = rightQ.Dequeue();
            var result = default(TResult);
            try
            {
                result = resultSelector(value, rightValue);
            }
            catch (Exception e)
            {
                observer.OnError(e);
                return;
            }
            observer.OnNext(result);
        }
        else
        {
            leftQ.Enqueue(value);
        }
    }
},
observer.OnError,
observer.OnCompleted));

group.Add(right.Subscribe(
    value =>
    {
        lock (gate)
        {
            if (leftQ.Count > 0)
            {
                var leftValue = leftQ.Dequeue();
                var result = default(TResult);
                try
                {
                    result = resultSelector(leftValue, value);
                }
                catch (Exception e)
                {
                    observer.OnError(e);
                    return;
                }
                observer.OnNext(result);
            }
            else
            {
                rightQ.Enqueue(value);
            }
        }
    }
},
observer.OnError,
```

```
        observer.OnCompleted());  
  
        return group;  
    });  
}
```

In this sample, two sequences are zipped together, as messages from left and right can occur simultaneously, the operator needs to ensure that it orders the messages. Next to this it needs to use a lock to ensure the operator's internal state (the two queues) doesn't get corrupted.

### When to ignore this guideline

- The operator works on a single source observable sequence.
- The operator does not introduce concurrency.
- Other constraints guarantee no concurrency is in play.

**NOTE:** If a source observable sequence breaks the Rx contract (see chapter 0), a developer can fix the observable sequence before passing it to an operator by calling the *Synchronize* operator.

## 6.8. Avoid serializing operators

As all Rx operators are bound to guideline 6.7, operators can safely assume that their inputs are serialized. Adding too much synchronization would clutter the code and can lead to performance degradation.

If an observable sequence is not following the Rx contract (see chapter 0), it is up to the developer writing the end-user application to fix the observable sequence by calling the *Synchronize* operator at the first place the developer gets a hold of the observable sequence. This way the scope of additional synchronization is limited to where it is needed.

### Sample

```
public static IObservable Select(  
    this IObservable source, Func selector)  
{  
    return Observable.CreateWithDisposable(  
        observer => source.Subscribe(  
            x =>  
            {  
                TResult result;  
                try  
                {  
                    result = selector(x);  
                }  
                catch (Exception exception)  
                {  
                    observer.OnError(exception);  
                    return;  
                }  
            }  
        )  
    )  
}
```



```
        }
        observer.OnNext(result);
    },
    observer.OnError,
    observer.OnCompleted));
}
```

In this sample, *Select* assumes that the source observable sequence is following the serialization guideline 6.7 and requires no additional locking.

#### When to ignore this guideline

There are currently no known cases where to ignore this guideline.

### 6.9. Parameterize concurrency by providing a scheduler argument.

As there are many different notions of concurrency, and no scenario fits all, it is best to parameterize the concurrency an operator introduces. The notion of parameterizing concurrency in Rx is abstracted through the *IScheduler* interface.

#### Sample

```
public static IObservable<TValue> Return<TValue>(TValue value,
    IScheduler scheduler)
{
    return Observable.CreateWithDisposable<TValue>(
        observer =>
            scheduler.Schedule(() =>
            {
                observer.OnNext(value);
                observer.OnCompleted();
            }));
}
```

In this sample, the *Return* operator parameterizes the level of concurrency the operator has by providing a scheduler argument. It then uses this scheduler to schedule the firing of the *OnNext* and *OnCompleted* messages.

#### When to ignore this guideline

- The operator is not in control of creating the concurrency (e.g. in an operator that converts an event into an observable sequence, the source event is in control of firing the messages, not the operator).
- The operator is in control, but needs to use a specific scheduler for introducing concurrency.

### 6.10. Provide a default scheduler

In most cases there is a good default that can be chosen for an operator that has parameterized concurrency through guideline 0. This will make the code that uses this operator more succinct.

**Note:** Follow guideline 6.12 when choosing the default scheduler, using the immediate scheduler where possible, only choosing a scheduler with more concurrency when needed.

### Sample

```
public static IObservable<TValue> Return<TValue>(TValue value)
{
    return Return(value, Scheduler.Immediate);
}
```

In this sample, we provide an overload to the *Return* operator that takes no scheduler. The implementation forwards to the other overload and uses the immediate scheduler.

### When to ignore this guideline

Ignore this guideline when no good default can be chosen.

## 6.11. The scheduler should be the last argument to the operator

Adding the scheduler as the last argument makes usage of the operator fluent in Intellisense. As Guideline 6.10 ensures an overload with a default scheduler, adding or omitting a scheduler becomes easy.

### Sample

```
public static IObservable<TValue> Return<TValue>(TValue value)
{
    return Return(value, Scheduler.Immediate);
}

public static IObservable<TValue> Return<TValue>(TValue value,
IScheduler scheduler)
{
    return Observable.CreateWithDisposable<TValue>(
        observer =>
            scheduler.Schedule(() =>
            {
                observer.OnNext(value);
                observer.OnCompleted();
            }));
}
```

In this sample the *Return* operator has two overloads, one without a scheduler argument where a default is picked and one with a scheduler argument. As the scheduler argument is the last argument, adding or omitting the argument is clearly visible in Intellisense without having to change the order of the arguments.

### When to ignore this guideline

C# and VB support *params* syntax. With this syntax, the *params* argument has to be the last argument. Make the scheduler the final to last argument in this case.

## 6.12. Avoid introducing concurrency

By adding concurrency, we change the timeliness of an observable sequence. Messages will be scheduled to arrive later. The time it takes to deliver a message is data itself, by adding concurrency we skew that data.

This guideline includes not transferring control to a different context such as the UI context.

### Sample 1

```
public static IObservable<TResult> Select<TSource, TResult>(
    this IObservable<TSource> source, Func<TSource, TResult> selector)
{
    return Observable.CreateWithDisposable<TResult>(
        observer => source.Subscribe(
            x =>
            {
                TResult result;
                try
                {
                    result = selector(x);
                }
                catch (Exception exception)
                {
                    observer.OnError(exception);
                    return;
                }
                observer.OnNext(result);
            },
            observer.OnError,
            observer.OnCompleted));
}
```

In this sample, the select operator does not use a scheduler to send out the *OnNext* message. Instead it uses the source observable sequence call to *OnNext* to process the message, hence staying in the same time-window.

### Sample 2

```
public static IObservable<TValue> Return<TValue>(TValue value)
{
    return Return(value, Scheduler.Immediate);
}
```

In this case, the default scheduler for the *Return* operator is the immediate scheduler. This scheduler does not introduce concurrency.

### When to ignore this guideline

Ignore this guideline in situations where introduction of concurrency is an essential part of what the operator does.

**NOTE:** When we use the *Immediate* scheduler or call the observer directly from within the call to *Subscribe*, we make the *Subscribe* call blocking. Any expensive computation in this situation would indicate a candidate for introducing concurrency.

## 6.13. Hand out all disposables instances created inside the operator to consumers

Disposable instances control lifetime of subscriptions as well as cancelation of scheduled actions. Rx gives users an opportunity to unsubscribe from a subscription to the observable sequence using disposable instances.

After a subscription has ended, no more messages are allowed through. At this point, leaving any state alive inside the observable sequence is inefficient and can lead to unexpected semantics.

To aid composition of multiple disposable instances, Rx provides a set of classes implementing *IDisposable* in the *System.Disposables* namespace such as:

Name	Description
CompositeDisposable	Composes and disposes a group of disposable instances together.
MutableDisposable	A place holder for changing instances of disposable instances. Once new disposable instance is placed, the old disposable instance is disposed.
BooleanDisposable	Maintains state on whether disposing has occurred.
CancellationDisposable	Wraps the <i>CancellationToken</i> pattern into the disposable pattern.
ContextDisposable	Disposes an underlying disposable instance in the specified <i>SynchronizationContext</i> instance.
ScheduledDisposable	Uses a scheduler to dispose an underlying disposable instance.

### Sample

```
public static IObservable<TResult> ZipEx<TLeft, TRight, TResult>(
    this IObservable<TLeft> left,
    IObservable<TRight> right,
    Func<TLeft, TRight, TResult> resultSelector)
```

```
{
    return Observable.CreateWithDisposable<TResult>(observer =>
    {
        var group = new CompositeDisposable();
        var gate = new object();

        var leftQ = new Queue<TLeft>();
        var rightQ = new Queue<TRight>();

        group.Add(left.Subscribe(
            value =>
            {
                lock(gate)
                {
                    if (rightQ.Count > 0)
                    {
                        var rightValue = rightQ.Dequeue();
                        var result = default(TResult);
                        try
                        {
                            result = resultSelector(value, rightValue);
                        }
                        catch(Exception e)
                        {
                            observer.OnError(e);
                            return;
                        }
                        observer.OnNext(result);
                    }
                    else
                    {
                        leftQ.Enqueue(value);
                    }
                }
            },
            observer.OnError,
            observer.OnCompleted));

        group.Add(right.Subscribe(
            value =>
            {
                lock (gate)
                {
                    if (leftQ.Count > 0)
                    {
                        var leftValue = leftQ.Dequeue();
                        var result = default(TResult);
                        try
                        {
                            result = resultSelector(leftValue, value);
                        }
                    }
                }
            }
        ));
    });
}
```

```
        }
        catch (Exception e)
        {
            observer.OnError(e);
            return;
        }
        observer.OnNext(result);
    }
    else
    {
        rightQ.Enqueue(value);
    }
}
},
observer.OnError,
observer.OnCompleted));

return group;

});
}
```

In this sample, the operator groups all disposable instances controlling the various subscriptions together and returns the group as the result of subscribing to the outer observable sequence. When a user of this operator subscribes to the resulting observable sequence, he/she will get back a disposable instance that controls subscription to all underlying observable sequences.

#### When to ignore this guideline

There are currently no known instances where this guideline should be ignored.

### 6.14. Operators should not block

Rx is a library for composing asynchronous and event-based programs using observable collections.

By making an operator blocking we lose these asynchronous characteristics. We also potentially lose composability (e.g. by returning a value typed as *T* instead of *IObservable<T>*).

#### Sample

```
public static IObservable<int> Sum(this IObservable<int> source)
{
    return source.Aggregate(0, (prev, curr) => checked(prev + curr));
}
```

In this sample, the *Sum* operator has a return type of *IObservable<int>* instead of *int*. By doing this, the operator does not block. It also allows the result value to be used in further composition.

If the developer using the operator wants to escape the observable world, he or she can use one of the provided *First\**, *Last\** or *Single\** operators.

### When to ignore this guideline

There are currently no known instances where this guideline should be ignored.

## 6.15. Avoid deep stacks caused by recursion in operators

As code inside Rx operators can be called from different execution context in many different scenarios, it is nearly impossible to establish how deep the stack is before the code is called. If the operator itself has a deep stack (e.g. because of recursion), the operator could trigger a stack overflow quicker than one might expect.

There are two recommended ways to avoid this issue:

- Use the recursive *Schedule* extension method on the *IScheduler* interface
- Implement an infinite looping *IEnumerable<IObservable<T>>* using the yield iterator pattern, convert it to an observable sequence using the *Concat* operator.

### Sample 1

```
public static IObservable<TSource> Repeat<TSource>(
    TSource value, IScheduler scheduler)
{
    return Observable.CreateWithDisposable<TSource>(
        observer =>
            scheduler.Schedule(self =>
            {
                observer.OnNext(value);
                self();
            }));
}
```

In this sample, the recursive *Schedule* extension method is used to allow the scheduler to schedule the next iteration of the recursive function. Schedulers such as the current thread scheduler do not rely on stack semantics. Using such a scheduler with this pattern will avoid stack overflow issues.

### Sample 2

```
public static IObservable<TSource> Repeat<TSource>(TSource value)
{
    return RepeatHelper(value).Concat();
}

private static IEnumerable<IObservable<TSource>>
    RepeatHelper<TSource>(TSource value)
{
    while(true)
```

```
        yield return Observable.Return(value);
    }
```

The yield iterator pattern ensures that the stack depth does not increase drastically. By returning an infinite *IEnumerable<IObservable<TSource>>* instance the *Concat* operator can build an infinite observable sequence.

#### When to ignore this guideline

There are currently no known instances where this guideline should be ignored.

### 6.16. Argument validation should occur outside

#### *Observable.Create(WithDisposable)*

As guideline 6.5 specifies that the *Observable.Create(WithDisposable)* operator should not throw, any argument validation that potentially throws exceptions should be done outside the *Observable.Create(WithDisposable)* operator.

#### Sample

```
public static IObservable<TResult> Select<TSource, TResult>(
    this IObservable<TSource> source, Func<TSource, TResult> selector)
{
    if (source == null)
        throw new ArgumentNullException("source");
    if (selector == null)
        throw new ArgumentNullException("selector");

    return new Observable.Create<TResult>(observer => source.Subscribe(
        x =>
        {
            TResult result;
            try
            {
                result = selector(x);
            }
            catch (Exception exception)
            {
                observer.OnError(exception);
                return;
            }
            observer.OnNext(result);
        },
        observer.OnError,
        observer.OnCompleted));
}
```



In this sample, the arguments are checked for null values before the *Observable.Create* operator is called.

#### When to ignore this guideline

Ignore this guideline if some aspect of the argument cannot be checked until the subscription is active.

### 6.17. Unsubscription should be idempotent

The observable *Subscribe* method returns an *IDisposable* instance that can be used to clean up the subscription. The *IDisposable* instance doesn't give any information about what the state of the subscription is. As consumers do not know the state of the subscription, calling the *Dispose* method multiple times should be allowed. Only the first call the side-effect of cleaning up the subscription should occur.

#### Sample

```
var subscription = xs.Subscribe(Console.WriteLine);
subscription.Dispose();
subscription.Dispose();
```

In this sample, the subscription is disposed twice, the first time the subscription will be cleaned up and the second call will be a no-op.

#### When to ignore this guideline

There are currently no known cases where to ignore this guideline.

### 6.18. Unsubscription should not throw

As the Rx's composition makes that subscriptions are chained, so are unsubscriptions. Because of this, any operator can call an unsubscription at any time. Because of this, just throwing an exception will lead to the application crashing unexpectedly. As the observer instance is already unsubscribed, it cannot be used for receiving the exception either. Because of this, exceptions in unsubscriptions should be avoided.

#### When to ignore this guideline

There are currently no known cases where to ignore this guideline.

### 6.19. Custom *IObservable* implementations should follow the Rx contract

When it is not possible to follow guideline 6.2, the custom implementation of the *IObservable* interface should still follow the Rx contract (see chapter 0) in order to get the right behavior from the Rx operators.

#### When to ignore this guideline

Only ignore this guideline when writing observable sequences that need to break the contract on purpose (e.g. for testing).

## **6.20. Operator implementations should follow guidelines for Rx usage**

As Rx is a composable API, operator implementations often use other operators for their implementation (see paragraph 6.1). Rx usage guidelines as described in chapter 0 should be strongly considered when implementing these operators.

### **When to ignore this guideline**

As described in the introduction, only follow a guideline if it makes sense in that specific situation.