

SAMPLE CHAPTER

# Rx.NET

## IN ACTION

Tamir Dresher

FOREWORD BY Erik Meijer



MANNING



*Rx.NET in Action*  
by Tamir Dresher

**Sample Chapter 7**

Copyright 2017 Manning Publications

# *brief contents*

---

## **PART 1 GETTING STARTED WITH REACTIVE EXTENSIONS ..... 1**

- 1 ■ Reactive programming 3
- 2 ■ Hello, Rx 27
- 3 ■ Functional thinking in C# 54

## **PART 2 CORE IDEAS ..... 87**

- 4 ■ Creating observable sequences 89
- 5 ■ Creating observables from .NET asynchronous types 115
- 6 ■ Controlling the observer-observable relationship 135
- 7 ■ Controlling the observable temperature 157
- 8 ■ Working with basic query operators 184
- 9 ■ Partitioning and combining observables 205
- 10 ■ Working with Rx concurrency and synchronization 231
- 11 ■ Error handling and recovery 259

# 7

## *Controlling the observable temperature*

---

### ***This chapter covers***

- Creating publishers with subjects
- Working with hot and cold observables
- Moving from hot to cold and vice versa
- Controlling the hot observable lifetime

The abstraction provided by observables hides from the observers the knowledge of how the underlying source makes the emissions. Depending on the way the observable is implemented, the same emissions (the object instance) might be shared between the various observers, or alternatively, each observer might get different instances. The observable might be implemented so that each observer receives the entire sequence, or instead receives part of the sequence, depending on when it subscribed.

Say an observable emits sound waves. As an observer, you don't know whether the sound is coming from a live concert, or played from an album that was started the moment the observer subscribed. During a concert, all the listeners (the observers) share the same tunes. But when played from an album, the tunes are played to each listener independently, and the full sequence of songs can be consumed no matter when the observer subscribed.

The term *observable temperature* refers to the state of the observable at the moment of its subscription. This state describes the time an observable begins and stops its emissions and whether the emissions are shared between observers. A *hot observable* is in an active state, like a singer performing live or an observable that emits the mouse's current position. In contrast, a *cold observable* is in a passive state, like an album waiting to be played or an observable that pushes the elements in a loop when an observer subscribes.

To control and change the observable temperature—for example, when you want to make sure all observers observe the same items, or when you want to “record” notifications to replay them later—you need to use one of the Rx building blocks—the `Subject`, a type that's both an observable and an observer. `Subject` acts as a hub that allows multicasting notifications. You can also use `Subject` to create a `PubSub` inside your application. At the end of the chapter, you'll know how to identify and control the shareability of your observable so that the results of your queries will always be predictable.

## 7.1 Multicasting with subjects

A type that implements the `IObservable<T>` interface and `IObserver<M>` interface is called a *subject*. This type acts as both an observer and an observable, as shown in figure 7.1. It allows you to create an object that becomes a hub, which is able to intercept notifications it receives as an observer and push them to its observers. This, for example, can be used inside a shopping-cart class to notify various observers (such as the relevant UI component) about items added or removed from the cart. The cart exposes `Subject` as an observable, and the cart `Add` and `Remove` methods call the subject's `OnNext` method to notify about the change.

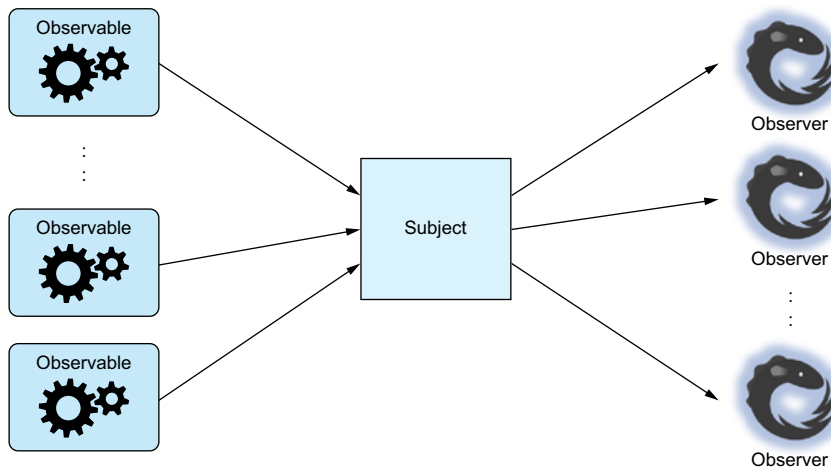


Figure 7.1 A subject is a type that's both an observable and an observer. It allows multicasting the notifications emitted by the sources to the observers.

The following listing provides the definition of the `ISubject` interface that resides in the `System.Reactive.Subjects` namespace.

**Listing 7.1 The `ISubject` interface**

```
interface ISubject<in TSource, out TResult> : IObservable<TSource>,
                                           IObservable<TResult>
{
}
```

The `Subject` type represents a PubSub (publisher-subscriber) pattern: the subject consumes notifications on one side (or is triggered by a notification) and emits notifications on the other side. This lets you create types that add special logic (transformations, caching, buffering, and so on) within the notifications received before they're published, or allows multicasting from one source to multiple destinations.

When `TSource` and `TResult` generic parameters are of the same type, you can use the simpler version of the `ISubject` interface.

**Listing 7.2 `ISubject` interface with `Source` and `Result` types that are the same**

```
interface ISubject<T> : ISubject<T, T>
{
}
```

Rx provides these subject implementations:

- `Subject<T>`—Broadcasts every observed notification to all observers.
- `AsyncSubject<T>`—Represents an asynchronous operation that emits its value upon completion.
- `ReplaySubject<T>`—Broadcasts notifications for current and future observers.
- `BehaviorSubject<T>`—Broadcasts notifications and saves the latest value for future observers. When created, it's initialized with a value that emits until changed.

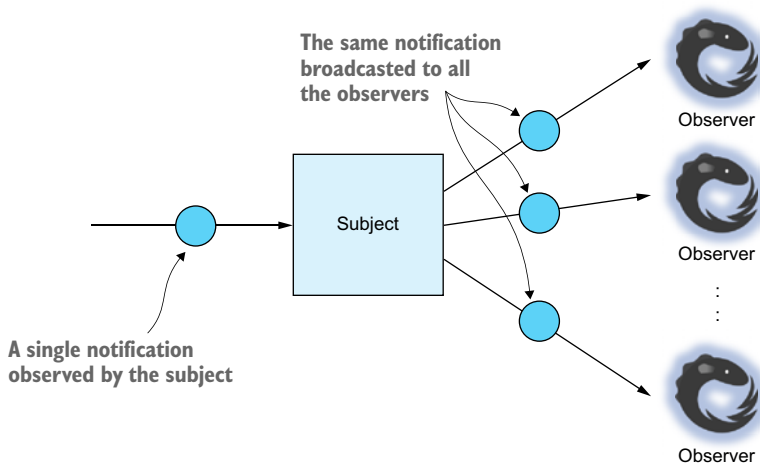
In all the standard implementations of subjects inside the Rx library, the observers receive the notifications sequentially, in the order that they subscribed.

**Why is it called a subject?**

In chapter 1, I mentioned that Rx drew its inspiration from the original GoF observer design pattern. In this pattern, the *subject* is observed by the observers and can be externally triggered to raise the notifications. The Rx `Subject` plays the same role as the *subject* in the observer pattern, therefore its name.

### 7.1.1 Simple broadcasting with Subject<T>

The simplest subject implementation is `Subject<T>`, which serves as a simple broadcaster, as shown in figure 7.2. This type adds no behavior around the received notification. Each observed notification is broadcast to the observers without any additional processing. This is why it makes `Subject<T>` a good fit for a backing field to an observable that's exposed by your class. All you need to do is tell it to push notifications from various methods in the class (such as the shopping cart that needs to notify parts of the application that it has changed).



**Figure 7.2** `Subject<T>` is a broadcaster. Each notification it observes is broadcast to its observers.

Because `Subject<T>` is an observer, it exposes the `OnNext`, `OnCompleted`, and `OnError` methods, so when they're called, the same methods are called on all the observers. You can manually signal a subject to emit notifications by calling its exposed methods.

This example uses a subject to publish two notifications to two observers and then completes:

```
using System.Reactive.Subjects;

Subject<int> sbj = new Subject<int>();

sbj.SubscribeConsole("First");
sbj.SubscribeConsole("Second");

sbj.OnNext(1);
sbj.OnNext(2);
sbj.OnCompleted();
```

**Creates a subject of integers**

**Subscribes two observers**

**Emits two notifications**

**Notifies subscribed observers about the end of the observable sequence**

Running this example displays the following output:

```
First - OnNext(1)
Second - OnNext(1)
First - OnNext(2)
Second - OnNext(2)
First - OnCompleted()
Second - OnCompleted()
```

Each time you call the `OnNext` or `OnCompleted` methods on the subject, the observers receive the notification in the order in which they subscribe.

#### MULTIPLE SOURCE, BUT ONE COMPLETION

One misunderstanding I see when working with `Subject<T>` is that although there can be many source observables, only one completion will occur and be passed to the observers. Subjects conform to the observable-observer protocol mandate that after completion, no more notifications are emitted.

Consider this example: a subject subscribes to two observables representing two chat rooms, each emitting messages as they're received from participants. Each observable emits five notifications but at different rates—every 1 second and every 2 seconds. The desired behavior is that the observer subscribing to the subject will receive the messages from both chat rooms and, if one chat room completes (all the participants leave), the messages from the other chat room will continue to be observed. But, confusingly, the real behavior is that the observer will receive the values emitted only until either observable completes; the rest of the notifications from the other observable won't pass through, as shown in figure 7.3.

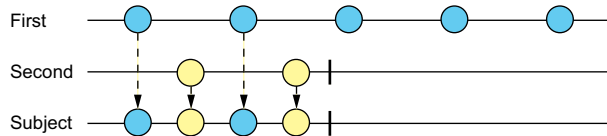


Figure 7.3 The subject can subscribe to multiple sources, but when any of the sources completes (the second in this figure), so does the subject.

#### Listing 7.3 Subscribing the subject to multiple observables

```
Subject<string> sbj = new Subject<string>();
```

```
Observable.Interval(TimeSpan.FromSeconds(1))
    .Select(x => "First: " + x)
    .Take(5)
    .Subscribe(sbj);
```

```
Observable.Interval(TimeSpan.FromSeconds(2))
    .Select(x => "Second: " + x)
    .Take(5)
    .Subscribe(sbj);
```

```
sbj.SubscribeConsole();
```

Creates a  
subject of  
type string

Creates an observable that simulates the first chat room the subject is subscribed to. Chat room emits five notifications before completion, one every 1 second.

Simulates a second chat room which emits five notifications before completion, one every 2 seconds.

Subscribes an observer  
to the subject



After running this example, you'll get this output:

```
- onNext (First: 0)
- onNext (Second: 0)
- onNext (First: 1)
- onNext (First: 2)
- onNext (Second: 1)
- onNext (First: 3)
- onNext (First: 4)
- OnCompleted()
```

The output shows that after the five values are emitted by the first observable, a completion notification from the first observable is observed by the subject and then published to its observer. Afterward, no more notifications are received.

### CLASSIC MISUSE OF A SUBJECT

Typically, developers naively try to merge observables together by using a subject, but the built-in Merge operator should be used instead. The following listing shows a classic example of a subject misuse: the subject subscribes to multiple sources to merge them. And the surprisingly confusing result is that the resulting sequence isn't merged at all. The scenario here merges an enumerable that was fetched from a database and transformed to an observable (everything is a stream, remember?<sup>1</sup>) together with an observable of real-time notifications. The observable created from the enumerable completes first and, therefore, the rest of the notification won't be observed, making the result confusing.

#### Listing 7.4 The wrong way to merge observables

```
Subject<string> sbj = new Subject<string>();
sbj.SubscribeConsole();

//at some point later...

IEnumerable<string> messagesFromDb = ...
IObservable<string> realTimeMessages = ...

messagesFromDb.ToObservable().Subscribe(sbj);
realTimeMessages.Subscribe(sbj);
```


**Creates a subject and subscribes an observer**

**Fetches a collection of messages from the database**

**Creates an observable of messages that emits messages in real time**

**Converts the collection to an observable that synchronously emits all the messages to the subject subscribed to it and publishes its completeness**

**Subscribes the subject (because the previous observable already completed, none of this observable's notifications will be observed)**

**Don't do it. Use Merge instead.** 

In the example, you create a subject at the beginning of the application and subscribe an observer to it. (In a real application, the observer can be the screen that shows the messages.) Later, somewhere in the code (for example, after the initialization process), you subscribe the subject to two observables: the first is an enumerable of the

<sup>1</sup> Chapter 1 introduced the concept that everything is a stream.

items that the database loads (and transforms to the observable), and the second is the observable of the messages received in real time. This creates a simple implementation of a merge; however, the correct way to implement the merge is by using the Merge operator.

The first observable is created from a finite collection of messages because a finite number of messages are stored in the database. The moment the subject subscribes to it, all the messages are synchronously emitted, and then the `OnCompleted` method is called on `Subject`.

Calling the `OnCompleted` method at this point means the subject discards any message emitted afterward. This makes the subscription to the second observable useless, as it has no effect.

**TIP** As a general rule, use subjects (of any kind) with caution, and make sure you're not reinventing the wheel; instead, use the built-in Rx operators.

One problem with `Subject<T>` you may encounter is that if the source observable emits a value before an observer subscribes, this value will be lost. This is specifically problematic if the source always emits only a single notification. Luckily, `AsyncSubject` provides a remedy for those cases.

### 7.1.2 Representing asynchronous computation with `AsyncSubject`

You can add inner behavior to the way subjects handle source notifications. `AsyncSubject<T>` adds logic to your code that fits nicely with asynchronous emissions. This is useful when the source observable might complete before the observer has a chance to subscribe to it, as shown in figure 7.4. This behavior is often seen when dealing with concurrent applications, where order of execution can't be predicted.

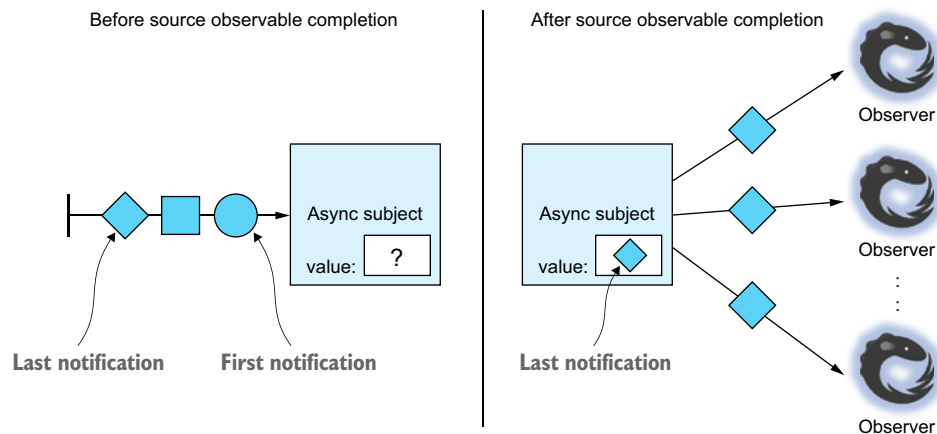


Figure 7.4 `AsyncSubject` emits only the last value to current and future observers.

Internally, `AsyncSubject` stores the most recent value so that when the source observable completes, it emits this value to current and future observers. For example, you can use `AsyncSubject` inside Rx to convert `Task` and `Task<T>` into observables. Listing 7.5 shows the conceptual implementation of this conversion. The Rx implementation for the `ToObservable` operator is different and includes performance optimizations and edge-case handling.

The code shows how to create an `AsyncSubject` and redirect each possible completion status for the task to the observable notifications. Even though the task is completed, the subject emits the notification to the observer.

**Listing 7.5 Converting Task<T> to an observable by using AsyncSubject**

```
var tcs = new TaskCompletionSource<bool>();
var task = tcs.Task;

AsyncSubject<bool> subj = new AsyncSubject<bool>();
task.ContinueWith(t =>
{
    switch (t.Status)
    {
        case TaskStatus.RanToCompletion:
            subj.OnNext(t.Result);
            subj.OnCompleted();
            break;
        case TaskStatus.Faulted:
            subj.OnError(t.Exception.InnerException);
            break;
        case TaskStatus.Canceled:
            subj.OnError(new TaskCanceledException(t));
            break;
    }
}, TaskContinuationOptions.ExecuteSynchronously);
tcs.SetResult(true);
subj.SubscribeConsole();
```

**Creates a Task from a TaskCompletionSource that you can control in the code**

**If the Task completes successfully, emits its result and then completes**

**Takes the exception that was thrown and notifies the observers**

**If the Task is canceled, notifies the observers with a TaskCanceledException**

**Sets the Task to completion before the observer subscribes**

**Sets the continuation to work on the same thread as the completed Task**

The program output shows that even though the Task completed before the observer subscribed, the observer is notified of the result:

```
- OnNext (True)
- OnCompleted()
```

Keep in mind that `AsyncSubject` emits only one value, and only after the source observable completes. Sometimes, however, you'll want to emit notifications as they come and preserve the ability to cache the latest value for future observers, as `AsyncSubject` does. For that, you need to use `BehaviorSubject`.

### 7.1.3 Preserving the latest state with BehaviorSubject

The type `BehaviorSubject<T>` is useful when you need to represent a value that changes over time, such as an object state. Say you need to store an object's possible states (`PreLoad`, `Loaded`, `Rendering`, and so forth).

Every observer that subscribes to `BehaviorSubject` receives the last value and all subsequent notifications, as shown in figure 7.5. Therefore, when creating an instance of `BehaviorSubject`, you pass an initial value (a default). You can also read the last (or initial) value through the `Value` property that `BehaviorSubject` exposes, making it ideal as a backing field for a state property that allows change notifications.

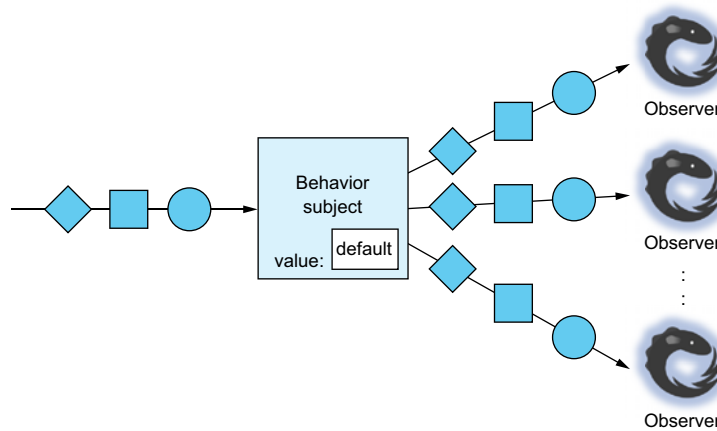


Figure 7.5 `BehaviorSubject` represents a value that changes over time. Observers receive the last (or initial) value and all subsequent notifications.

This example uses `BehaviorSubject` to maintain the state of the network connectivity while still making changes in the connectivity observable:

```

    Creates a BehaviorSubject that represents the
    connectivity state and initializes as Disconnected
    BehaviorSubject<NetworkConnectivity> connection =
    new BehaviorSubject<NetworkConnectivity> (
        NetworkConnectivity.Disconnected);
    connection.SubscribeConsole("first");
    //After connection
    connection.OnNext(NetworkConnectivity.Connected);
    connection.SubscribeConsole("second");
    Console.WriteLine("Connection is {0}", connection.Value);

```

If an observer subscribes before a connection is made, the subscriber receives the Disconnected value.

If another observer subscribes after a connection is made, the subscriber receives the cached Connected value.

Shows the last emitted or initialized `BehaviorSubject` value through the `Value`

Emits a notification with the Connected value which is cached inside `BehaviorSubject`.

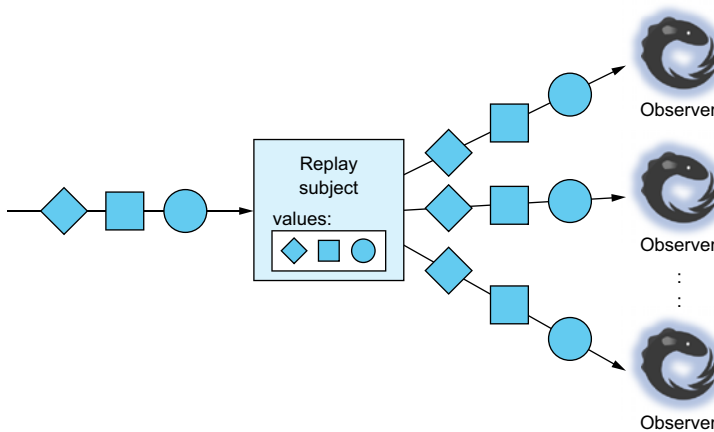
Running this example shows this output:

```
first - OnNext(Disconnected)
first - OnNext(Connected)
second - OnNext(Connected)
Connection is Connected
```

BehaviorSubject keeps a cache of one value only (the last one). For more than one value, use ReplaySubject.

### 7.1.4 Caching the sequence with ReplaySubject

ReplaySubject<T> is a subject that holds a cache of the notifications it observes inside an inner buffer, as shown in figure 7.6.



**Figure 7.6** ReplaySubject broadcasts each notification to all subscribed and future observers, subject to buffer trimming policies.

ReplaySubject lets you, for example, store notifications and replay them for various observable pipelines that you're testing, and compare the results to see which is the best. To prevent unwanted memory leaks, you can control the caching policy that limits the buffer size, time, or both.

Listing 7.6 shows how to limit ReplaySubject by time and size. This example uses Rx with a health sensor. Like Microsoft Band,<sup>2</sup> the client application connects to the sensor when started, but the user can add a heart-rate parameter to the UI later. To display a nice graph, you want to keep the last 20 readings from the last 2 minutes.

<sup>2</sup> A repository that adds Rx support to Microsoft Band can be found at GitHub (<https://github.com/Reactive-Extensions/RxToBand>).

**Listing 7.6 Limiting the ReplaySubject cache by time and size**

```

IObservable<int> heartRate = ...
ReplaySubject<int> sbj = new ReplaySubject<int>(bufferSize: 20,
    window: TimeSpan.FromMinutes(2));

heartRate.Subscribe(sbj);

// After the user selected to show the heart rate on the screen
sbj.SubscribeConsole("HeartRate Graph");

```

Gets an observable of the heart rate from Microsoft Band

Subscribes the subject when the application starts

If the user displays the heart rate onscreen, subscribes an observer to receive the cached readings and all the ones that follow

Creates ReplaySubject with a buffer size of 20 and notifications cached at 2 minutes

For the heart rate, I simulated five readings (70–74) and, instead of displaying a graph, I printed them onscreen:

```

HeartRate Graph - OnNext(70)
HeartRate Graph - OnNext(71)
HeartRate Graph - OnNext(72)
HeartRate Graph - OnNext(73)
HeartRate Graph - OnNext(74)
HeartRate Graph - OnCompleted()

```

Like everything that involves caching in software, you should be aware of the memory footprint it leaves and the cache invalidation you use. There's no way to manually clean the cache that `ReplaySubject` contains (nor access it and read it), so pay special attention when you use the unbounded version of `ReplaySubject`. You can free the cache's memory only by disposing of `ReplaySubject`.

Next, we'll talk about guidelines and best practices for subjects.

### 7.1.5 Hiding your subjects

You should be aware of a risk when working with subjects: it's easy to lose control of them. Suppose you have a class that holds an inner subject and then exposes it when a property returns an observable, as this example shows:

```

class BankAccount
{
    Subject<int> _inner = new Subject<int>();

    public IObservable<int> MoneyTransactions { get { return _inner; } }
}

```

Returns the subject instance

Although you expose the `IObservable` type only, the encapsulation can still be broken. That's because it's possible for a hostile or inexperienced developer to cast the observable back to a subject, as in this example:

```

var acct = new BankAccount();
acct.MoneyTransactions.SubscribeConsole("Transferring");

```

Makes a regular subscription as the class author intended

```
var hackedSubject = acct.MoneyTransactions as Subject<int>;
hackedSubject.OnNext(-9999);
```

← A hostile casting of the observable

← Your encapsulation is broken, and all the account money is taken.

After casting back to `Subject` (or `ISubject`), the code can now emit notifications from the outside. This will cause confusion and unwanted bugs.

#### HOW TO PROTECT FROM OUTSIDE EMISSIONS

Your subject was compromised because you returned an inner object that has accessible methods. To fix that, you need to return a different object—one that won't reveal the ability to reach your observers even by accident.

For that purpose, Rx provides the `AsObservable` operator. `AsObservable` creates a proxy that wraps your subject and exposes only the `IObservable` interface, so the observer can still subscribe, but no code can cast the observer to a subject, and no code can access the observers. This is demonstrated in figure 7.7.

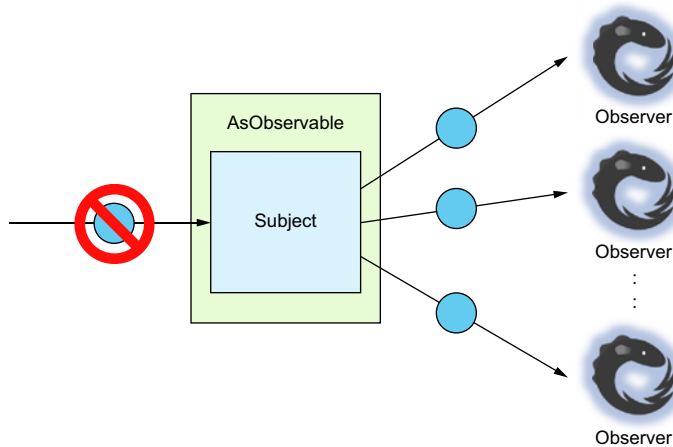


Figure 7.7 Instead of exposing your subject, use the `AsObservable` operator to create a proxy that hides the inner subject.

The following example proves that the observable returned by the `AsObservable` operator (the proxy) can't be cast to a subject:

```
Subject<int> sbj = new Subject<int>();
var proxy = sbj.AsObservable();
var subject = proxy as Subject<int>;
var observer = proxy as IObservable<int>;
Console.WriteLine("proxy as subject is {0}", subject == null
    ? "null"
    : "notnull");
Console.WriteLine("proxy as observer is {0}", observer == null
    ? "null"
    : "not null");
```

This, of course, prints the following:

```
proxy as subject is null
proxy as observer is null
```

Subject plays a big role in Rx operators and is a powerful tool if used correctly. Unfortunately, Subject can be used incorrectly. The next section provides a few guidelines that can help you decide whether Subject is the right object for you to use.

### 7.1.6 Following best practices and guidelines

One of the areas that causes a lot of debate in the Rx world is whether subjects are good or bad, and if using them is right or wrong. As Erik Meijer once said, “Once you start seeing yourself using Subject, something is wrong. Because subjects are stateful things.”<sup>3</sup>

But let’s set the record straight: subjects aren’t bad and, when used correctly, can be useful indeed. They’re used extensively inside the Rx code itself. It’s true, however, that some developers use subjects when they don’t need them. So when should you use a subject and when should you avoid them? The following list contains the points you should consider:

- Use the built-in factory methods such as `Observable.Create` whenever possible, instead of using a subject. Use a subject only if no suitable built-in factory method exists.
- Use a subject only if the source of the notifications is local (your code raises the notifications and not an external source); for example, to create a notifying property with `BehaviorSubject`.
- Use a subject for controlling an observable’s temperature (as you’ll learn next).
- Use a subject when creating an operator of your own that needs a notification’s hub.
- Don’t expose subjects; use `AsObservable` to prevent that from happening.

The important thing to remember is that before you create an operator, you should always check whether an operator that does what you intended to write by yourself already exists in Rx.

Dave Sexton wrote a wonderful blog post about the correct use of subjects that drills down into these guidelines (<http://mng.bz/Pv9>). I recommend reading it after you read the next section, where I’ll show one area that depends on subjects for its existence—controlling the observable temperature.

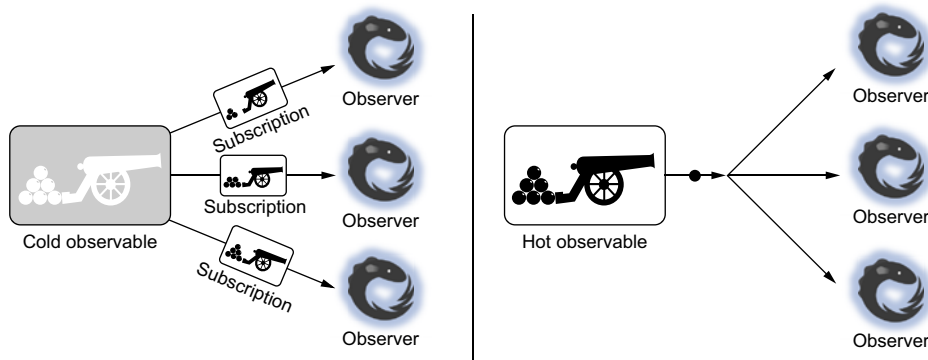
## 7.2 Introducing temperature: cold and hot observables

It may sound funny, but observables have a notion of temperature. Observables can be cold or hot, and each has different effects on your applications. A *cold observable* is *passive* and emits only when the observer subscribes; for each observer, a complete

---

<sup>3</sup> “RX: Reactive Extensions for .NET,” PDC 2009, <http://mng.bz/3qu4>, and Erik Meijer on Twitter, <http://mng.bz/WeiQ>.





**Figure 7.8** A cold observable is passive and starts emitting only when an observer subscribes. A hot observable is active, and its emissions are shared among all the observers.

sequence is generated. A *hot observable* is *active* and emits regardless of the observers. All the observers of the hot observable will observe the same emissions, so we say that the items are shared. Observables can also move from one temperature to the other with the techniques you'll learn in this section that will help make your observable queries predictable. Figure 7.8 summarizes the differences between hot and cold observables.

### 7.2.1 Explaining cold and hot observables

To understand the difference between hot and cold observables, I created the following simple program. It creates an observable that emits two string values with a short delay between them. Look at the following example and try to predict the output:

```
var coldObservable =
    Observable.Create<string>(async o =>
    {
        o.OnNext("Hello");
        await Task.Delay(TimeSpan.FromSeconds(1));
        o.OnNext("Rx");
    });

coldObservable.SubscribeConsole("o1");
await Task.Delay(TimeSpan.FromSeconds(0.5));
coldObservable.SubscribeConsole("o2");
```

**Emit the words Hello and Rx with a 1-second delay between the words**

**Subscribes two observers with a half-second delay between the subscriptions**

Many developers new to Rx find it surprising that the output of this small program shows the message of both observers intertwined:

```
o1 - OnNext (Hello)
o2 - OnNext (Hello)
o1 - OnNext (Rx)
o1 - OnCompleted()
o2 - OnNext (Rx)
o2 - OnCompleted()
```

You can see that the second observer receives the message *Hello* even though it subscribes after the first observer receives it.

For each observer that subscribes, the observable begins its work from the start and generates the entire sequence of notifications for that observer. You can also say that the observable isn't running until an observer subscribes to it. Those characteristics are typical for cold observables.

### 7.2.2 Cold observable

Here's my more formal definition of a *cold observable*:

*A cold observable is an observable that starts emitting notifications only when an observer subscribes, and each observer receives the full sequence of notifications without sharing them with other observers.*<sup>4</sup>

Most of the observables you've created thus far in this book are cold observables. When you use the operators `Create`, `Defer`, `Range`, `Interval`, and so on, you get an observable that's cold. From the observer's standpoint, if the observable it subscribes to is cold, then the observer can be certain that it hasn't missed any notifications.

### 7.2.3 Hot observables

Here's my formal definition of a *hot observable*:

*A hot observable is an observable that emits notifications regardless of its observers (even if there are none). The notifications emitted by hot observables are shared among their observers.*

The classic example of a hot observable is the one you create from an event, such as a mouse-move event. The mouse movement's observable sequence is "live," so even if there's no subscribed observer, the mouse movements still happen. And when there are multiple observers, they all get notified of the same mouse movement.

From the observer standpoint, if the observable it subscribes to is hot, then the observer might have already missed some notifications.

When learning about observable temperatures, it's typical to wonder whether the temperature is fixed or can somehow change. The next section answers just that.

## 7.3 Heating and cooling an observable

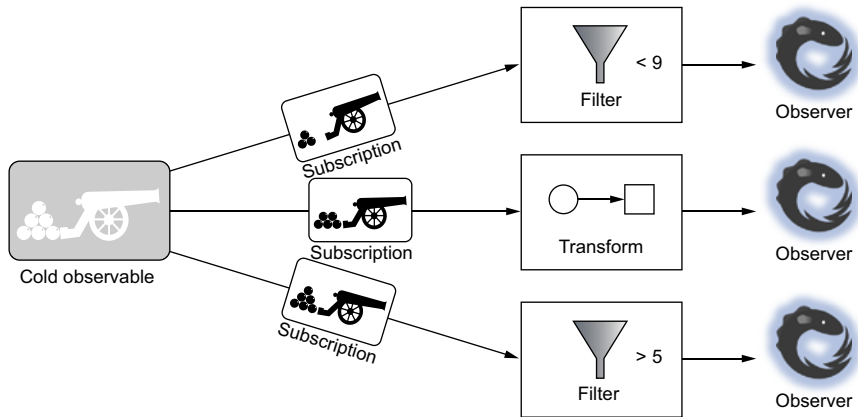
Now that you know what *cold* and *hot* mean in terms of observables, the next step is to figure out the ways to switch from cold to hot, or from hot to cold. In this section, you'll learn how and why you would want to perform the transformation from one temperature to the other.

### 7.3.1 Turning cold into hot

Suppose you want to create a few queries over an observable; for example, you want to filter certain elements with a few filter functions, and observe the ones that survived

---

<sup>4</sup> This doesn't mean the data carried inside the notification can't reference the same object (thus making them shared); rather, the notifications that carry the data are independent from one another.



**Figure 7.9** Even though each observer subscribes to the same observable, each observer receives a different sequence and the operator processes different elements.

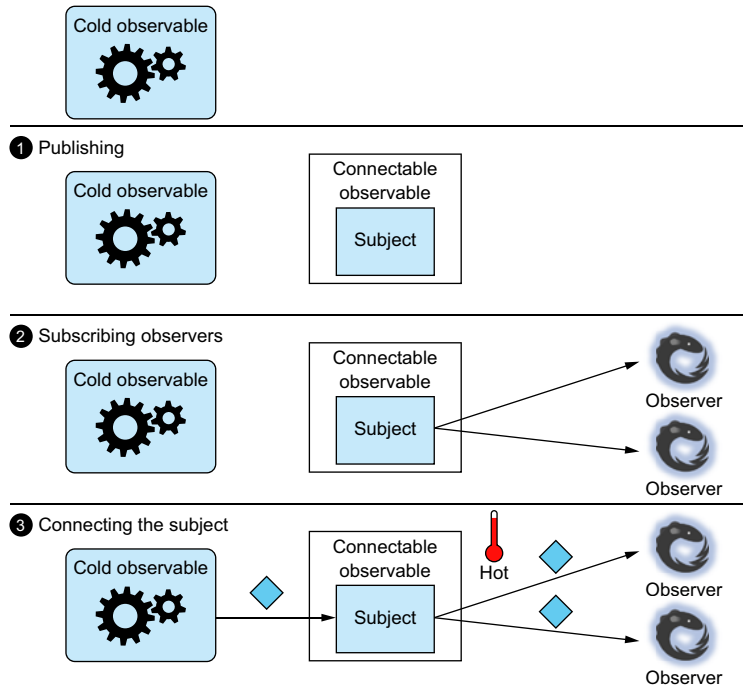
from each filter in a specific way. As a good practice, you'd probably encapsulate each observation (per each filter) in its own query, possibly in its own method. As mentioned previously, cold observables don't share their emissions between their subscribers, so multiple subscriptions, as in the case of the multiple queries, to a cold observable will result in different streams of elements for each one—shown as cannons in figure 7.9. The elements aren't shared and might be different in their values. This is exactly like calling a method twice, which could result in two different return values.

To overcome the possibility that multiple subscriptions will end up with different elements observed by each observer, you need to turn the cold observable into a hot observable, so that the observers will subscribe to the hot one instead, and you can then guarantee they'll observe the same notifications. You have to make sure that turning an observable from cold to hot won't cause you to lose any notifications. You'll have to take that into account inside your process, as you'll see next.

Conceptually, all it takes to make a cold observable hot is putting a proxy between the cold observable and the observers, and letting it broadcast all the notifications to the observers. Luckily, not so long ago, you learned about excellent types that can be programmed as those proxies: subjects. The process of turning an observable from cold to hot is shown in figure 7.10.

To turn a cold observable into a hot observable:

- 1 Create the subject that will be placed in front of the cold observable. The subject can now accept subscriptions from observers interested in the notifications of the cold observable.
- 2 Subscribe the observers that are interested in the notifications of the cold observable to the subject.
- 3 Subscribe the subject, as an observer, to the cold observable. This causes the cold observable to start emitting its sequence of notifications, which are broadcast by



**Figure 7.10** The steps for turning a cold observable into a hot observable. The order of the steps is important! After connecting the subject to the cold observable, data starts flowing and it is sent only once.

the subject to all of the observers. This is also the way to guarantee that you don't lose any notifications from the source observable.

Whenever you create an observable and know there will be more than one observable pipeline (and observers), you may want to make the observable hot. This may occur, for example, when you want to model periodic data retrieval from a web service as an observable and don't want each observer to initiate different calls to the web service. Instead, you want to make one call and share the retrieved data with all observers.

Don't be scared of this lengthy process. The code you need to write to turn the cold observable into a hot one is simple. The steps in figure 7.10 correspond to the Rx operators `Publish`, `Subscribe`, and `Connect`. First, I'll show the code that uses those operators and then I'll explain each operator.

#### Listing 7.7 Turning a cold observable hot

```
var coldObservable=Observable.Interval(TimeSpan.FromSeconds(1)).Take(5); ←
var connectableObservable = coldObservable.Publish();
```

Publishes the observable to let multiple  
observers share the notifications

Creates a cold observable that emits  
five notifications, one per second

```

connectableObservable.SubscribeConsole("First");
connectableObservable.SubscribeConsole("Second");

connectableObservable.Connect();

Thread.Sleep(2000);
connectableObservable.SubscribeConsole("Third");

```

**Subscribes two observers; both will share the same notifications.**

**Connects the inner subject to the source observable**

**Subscribes a third observer that will share ensuing notifications with the previous observers**

This small application creates a cold observable that emits five notifications, one every second. The application then makes the observable hot by converting it to a `ConnectableObservable` (more on that in a moment) and connects it to the source observable (by calling the `Connect` operator) after two observers subscribe. Then, after another 2 seconds, it subscribes another observer.

The output shows that all notifications are indeed shared between all observers:

```

First - OnNext(0)
Second - OnNext(0)
First - OnNext(1)
Second - OnNext(1)
Third - OnNext(1)
First - OnNext(2)
Second - OnNext(2)
Third - OnNext(2)
First - OnNext(3)
Second - OnNext(3)
Third - OnNext(3)
First - OnNext(4)
Second - OnNext(4)
Third - OnNext(4)
First - OnCompleted()
Second - OnCompleted()
Third - OnCompleted()

```

You can see that the same notification values are shared between the observers. A few new concepts have arisen here, so let's explore the first one: `ConnectableObservable`.

### 7.3.2 Using `ConnectableObservable`

To turn the cold observable to hot, you need a proxy around it. But you don't want the proxy to create a subscription to the cold observable before you finish setting all the observers you need (otherwise, you might miss some notifications). To help with that, Rx introduces the connectable observable. `ConnectableObservable` implements the `IObservable` interface and subscribes to the source observable only when explicitly told to do so by calling the `Connect` method.

#### Listing 7.8 The `IObservable` interface

```

interface IObservable<T> : IObservable<T>
{
    IDisposable Connect();
}

```

**Subscribes the observable wrapper to its source and returns a disposable object representing the subscription**

`IObservableObservable` is an observable by itself and can (and will) have observers. As long as the connection is established, all the observers will receive the notifications from the source observable.

To get an instance that implements the `IObservableObservable` interface, you need to call the `Publish` operator on your source observable. The `Publish` operator has a few overloads; each overload creates a `ConnectableObservable` with some tweaks, as you'll see next.

### 7.3.3 Publishing and multicasting

The `Publish` operator creates a `ConnectableObservable` wrapper around the source observable. This is a required step for allowing multicasting of the observable notifications. The `Publish` operator has a few overloads, so let's examine those one by one.

#### SIMPLE PUBLISH

This is the simplest overload:

```
IObservableObservable<TSource> Publish<TSource>(  
    this IObservable<TSource> source)
```

It creates a `ConnectableObservable` that holds a `Subject<T>` internally. So, from the moment you `Connect` it, all the observers share the same notifications. These are the code steps to follow:

```
var coldObservable= ...  
var connectableObservable = coldObservable.Publish();  
  
connectableObservable.Subscribe(...);  
:  
connectableObservable.Subscribe(...);  
  
connectableObservable.Connect();
```

**Publishes a cold observable by creating a `ConnectableObservable` that wraps it and holds a single subscription to it**

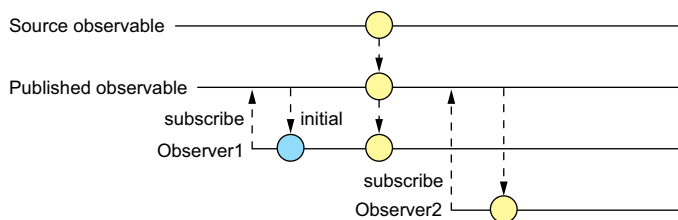
**Subscribes all observers interested in the shared notifications from the source observable**

**Subscribes the `ConnectableObservable` to the source observable**

In most cases, you'd like to subscribe all observers before calling `Connect`, so no observer will miss a notification; but that's not always the case. In case new observers subscribe later, it's important for you to note that they'll receive only the next notification that follows their subscription.

But you can tweak this behavior so that an observer will immediately receive the latest notification when it subscribes. This is done using the following overloads of `Publish`, which accept an initial value and create the `ConnectableObservable` with an inner `BehaviorSubject<T>`:

```
IObservableObservable<TSource> Publish<TSource>(  
    this IObservable<TSource> source,  
    TSource initialValue)
```



**Figure 7.11** Publishing an observable with an initial value. Observers receive either the last value that was emitted from the source observable or the initial value, if no notification was yet emitted.

The inner `BehaviorSubject<T>` this overload creates for the `ConnectableObservable` is initialized with an initial value, so every observer that subscribes before `Connect` was called will receive this value. Every observer that subscribes after `Connect` was called will receive the last value that was emitted from the source observable or the initial value, if no notification was yet emitted. This behavior is shown in figure 7.11

#### REUSING THE PUBLISHED OBSERVABLE TO CREATE A NEW OBSERVABLE

Things get a little interesting (and complex) when you need to combine the cold observable multiple times to create new observables. The following `Publish` overload is useful in these cases:

```
IObservable<TResult> Publish<TSource, TResult>(
    this IObservable<TSource> source,
    Func<IObservable<TSource>, IObservable<TResult>> selector)
```

← The cold source observable

← Selector function that can use the multicasted source sequence as many times as needed. Subscriptions made inside are deferred until the real subscription takes place.

Notice that this overload returns an observable and not a `ConnectableObservable`. With this overload, you can easily create observables that reuse the source observable. Consider the next example in which you want to use the `Zip` operator on an observable with itself. The `Zip` operator takes two (or more) observables and merges them by calling a function on the corresponding notifications. The normal expectation that developers have when they use the `Zip` operator on an observable with itself is that the two function arguments will be identical. This example shows why this expectation is false:

```
int I = 0;
var numbers = Observable.Range(1, 5).Select(_ => i++);
var zipped = numbers
    .Zip(numbers, (a, b) => a + b)
    .SubscribeConsole("zipped");
```

← Emits a sequence of numbers but causes a side effect on a shared variable.

← Because the “numbers” observable is cold, this results in the sequence of values in the form  $I + (i + 1)$  and not  $i + i$ .

In the example, you use an observable twice in order to create a new observable by using the `Zip` operator. Because the `numbers` observable is cold, the sequence is generated twice, and the side effect caused by incrementing the variable `i` happens twice per notification. Ultimately, what I did in this example is the same as if I had created two different observables that happen to use the same variable `i` and advance it independently (causing the side effect to be reflected in the other observable); thus the function arguments in iteration  $k$  will be with the values  $a = k$  and  $b = k + 1$ . You can see this effect in the output:

```
zipped - OnNext(1)           <— = 0 + 1
zipped - OnNext(5)           <— = 2 + 3
zipped - OnNext(9)           <— = 4 + 5
zipped - OnNext(13)
zipped - OnNext(17)
zipped - OnCompleted()
```

You can publish the source observable by yourself, but then it can be hard to decide when exactly to call `Connect`, especially if you want to share the zipped observable. To solve that, you want to defer `Connect` until the subscription happens. As the next example shows, the `Publish` operator can do this:

```
var publishedZip = numbers.Publish(published =>
    published.Zip(published, (a, b) => a + b));
publishedZip.SubscribeConsole("publishedZipped");
```

Calls the `Connect` method on the  
published `numbers` observable  
←

Now, the `numbers` observable is published, so the notifications are shared among all its observers. The same notification will be received both as `a` and `b`. The output is

```
publishedZipped - OnNext(0)
publishedZipped - OnNext(2)
publishedZipped - OnNext(4)
publishedZipped - OnNext(6)
publishedZipped - OnNext(8)
publishedZipped - OnCompleted()
```

## PUBLISHLAST

`ConnectableObservables`, created by the `Publish` operator, publishes the notifications from the source observable until it completes. At that point, `ConnectableObservable` completes as well.

Any observer that was late to subscribe won't see any values. This is especially bad when you have an observable that produces a single value, and that's the value you need. This source observable might even be a hot observable.

To help with that, Rx provides the `PublishLast` operator, which publishes only the last value of the source observable:

```
IObservable<TSource> PublishLast<TSource>(
    IObservable<TSource> source)
```



The `PublishLast` operator works similarly to the `Publish` operator, but instead of sharing all notifications from the source observable, the `ConnectableObservable` it creates will share only the last notification emitted before the source observable completes, both for existing observers and future ones. This is similar to working with an asynchronous type, as you saw earlier in this chapter, and `PublishLast` will create an `AsyncSubject<T>` that's used internally by the `ConnectableObservable`. Here's an example that shows it in action:

```
var coldObservable = Observable.Timer(TimeSpan.FromSeconds(5))
    .Select(_ => "Rx");

var connectableObservable = coldObservable.PublishLast();
connectableObservable.SubscribeConsole("First");
connectableObservable.SubscribeConsole("Second");
connectableObservable.Connect();

Thread.Sleep(6000);
connectableObservable.SubscribeConsole("Third");
```

**Simulates an asynchronous operation that takes a long time to complete**

**Shares the last value between all current and future observers**

**Subscribes an observer after the source observable completes**

Running this example shows that the last notification emitted by the source observable was shared among all observers:

```
First - OnNext (Rx)
First - OnCompleted()
Second - OnNext (Rx)
Second - OnCompleted()
Third - OnNext (Rx)
Third - OnCompleted()
```

### 7.3.4 Using Multicast

Both `Publish` and `PublishLast` are good for all of the common scenarios in which you need to heat a cold observable. But if you need more control or need to enforce policies on an internal subject used inside `ConnectableObservable` (for example, setting its buffer size and other configurations), then you need to use the `Multicast` operator. `Multicast` lets you pass the pending subject inside the `ConnectableObservable`

```
IObservable<TResult> Multicast<TSource, TResult>(
    this IObservable<TSource> source,
    ISubject<TSource, TResult> subject)
```

`Multicast` is a powerful low-level operator that's used to create other operators. All the `Publish` versions use `Multicast` in their implementations. For example, this implementation from the Rx source code for the `Publish` overload creates a `BehaviorSubject` for `ConnectableObservable`:

```
virtual IObservable<TSource> Publish<TSource>(
    IObservable<TSource> source,
    TSource initialValue)
```

```
{
    return source.Multicast(new BehaviorSubject<TSource>(initialValue));
}
```

As explained earlier, this `Publish` overload creates a `ConnectableObservable`. Every observer that subscribes to it, after its `Connect` method is called, will receive the last value emitted from the source observable or the initial value, if no notification was yet emitted. The implementation shows that in order to provide this behavior, `BehaviorSubject` is used as the underlying subject passed to the `Multicast` operator.

### 7.3.5 Managing the `ConnectableObservable` connection

After you connect `ConnectableObservable` to the source observable by calling the `Connect` method, you get back the subscription object that enables you to disconnect it whenever you want. What happens if you reconnect again? What if there are still observers? What if the observers are no longer there? To find the answers, keep on reading.

#### RECONNECTING

You can reconnect `ConnectableObservable` at any time. Doing so will cause the subscribed observers to see the notifications again. Reconnecting might be useful when you want to keep the observers but need to change the original source of the observable pipeline. For example, if the source observable is a chat server, and you know that server needs to be replaced, you can reconnect, which will cause the new server to be picked up again.

**Listing 7.9 Reconnecting `ConnectableObservable`**

Subscribes two observers to the connectable observable

```
var connectableObservable =
    Observable.Defer(() => ChatServer.Current.ObserveMessages())
        .Publish();
```

Creates and publishes an observable that connects to the current server and emits the messages coming from it

```
connectableObservable.SubscribeConsole("Messages Screen");
connectableObservable.SubscribeConsole("Messages Statistics");
var subscription = connectableObservable.Connect();
```

```
//After the application was notified on server outage
Console.WriteLine("-Disposing the current connection and reconnecting--");
subscription.Dispose();
subscription = connectableObservable.Connect();
```

Disposes of the connection to the servers without losing the current observers and reconnects to a new server

Connects the connectable observable to the source observable to connect to the server

In this example, the source observable is created using the `Defer` operator, which makes it a cold observable and, therefore, every observer shares the connection logic.

Because you publish it, the connection happens only once, and the notifications are shared among the observers.

The observer begins to receive notifications when you call `Connect` and stops receiving them when you dispose of the subscription object. When you call `Connect` a second time, an underlying connection to the new server is made (because `ChatServer.Current` points to the new server), and the observers receive the messages coming from it. This is shown in the program output:

```
Messages Screen - OnNext(Server0 - Message1)
Messages Statistics - OnNext(Server0 - Message1)
Messages Screen - OnNext(Server0 - Message2)
Messages Statistics - OnNext(Server0 - Message2)
Messages Screen - OnNext(Server0 - Message3)
Messages Statistics - OnNext(Server0 - Message3)
--Disposing the current connection and reconnecting--
Messages Screen - OnNext(Server1 - Message1)
Messages Statistics - OnNext(Server1 - Message1)
Messages Screen - OnNext(Server1 - Message2)
Messages Statistics - OnNext(Server1 - Message2)
Messages Screen - OnNext(Server1 - Message3)
Messages Statistics - OnNext(Server1 - Message3)
```

#### PERFORMING AUTOMATIC DISCONNECTION

If you dispose of the subscription object while there are still observers, you might see different results than expected. Moreover, when disposing of the subscription object, the subscribed observers won't see any notifications, and you have no way of telling that the `ConnectableObservable` is no longer connected.

If you keep the subscription when there are no observers, you're wasting expensive resources, and the source observable will keep pushing notifications for no reason. The best option is to make an automatic disconnect when there are no more observers. In addition, you should dispose of the subscription to the source observable.

To achieve this kind of automatic disconnect, you need to use the `RefCount` operator, which manages an inner counter for the number of subscribed observers and then disposes of the subscription when the count is zero.

The next example shows how to subscribe two observers to the observable and, when you unsubscribe them, no more notifications are emitted.

#### Listing 7.10 Automatic disconnection with `RefCount`

Prints a message to the console every time the observable emits a value

Creates an observable that emits a value every second

```
var publishedObservable = Observable.Interval(TimeSpan.FromSeconds(1))
    .Do(x => Console.WriteLine("Generating {0}", x))
    .Publish()
    .RefCount();
var subscription1 = publishedObservable.SubscribeConsole("First");
var subscription2 = publishedObservable.SubscribeConsole("Second");
```

Subscribes the two observers

Publishes with a reference count so that when the last observer unsubscribes, there will be no more notifications

<pre>Thread.Sleep(3000); subscription1.Dispose(); Thread.Sleep(3000); subscription2.Dispose();</pre>	<div style="border-left: 1px solid black; padding-left: 10px;"> <p><b>Waits 3 seconds before unsubscribing the first observer</b></p> <p><b>Waits 3 seconds before unsubscribing the second observer</b></p> </div>
--	---

As you can see from the following program output, after the second observer unsubscribes, no more notifications are emitted:

```
Generating 0
First - OnNext(0)
Second - OnNext(0)
Generating 1
First - OnNext(1)
Second - OnNext(1)
Generating 2
Second - OnNext(2)
Generating 3
Second - OnNext(3)
Generating 4
Second - OnNext(4)

Press any key to continue . . .
```

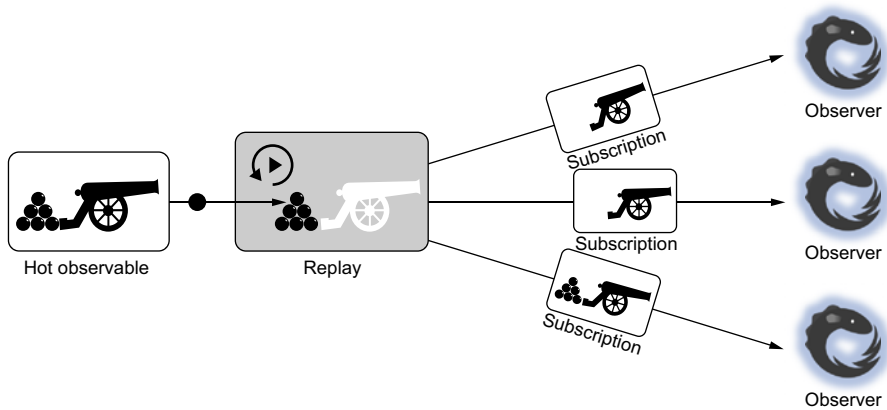
Using `RefCount` when publishing is a good practice that helps ensure that you're not keeping unneeded resources in use. Next you'll look at the other side of the temperature scale and see how to "cool" a hot observable to replay its emissions.

### 7.3.6 Cooling a hot observable to allow replaying

We defined a cold observable as an observable that generates the complete sequence of notifications for each observer that subscribes to it. Just as when you have a live broadcast that you want to watch later, it makes sense that if you could somehow record an observable and replay it later, each observer could subscribe when needed and be guaranteed to receive the entire recorded sequence. Therefore, you can conclude that a recorded observable is a cold observable.

It's important to note that if you have a hot observable, you can make it cold only from the moment you run the conversion. If by the time you make the conversion a notification is already emitted, you can't reproduce them.

To make an observable cold, you need to use the same tools that made a cold observable hot. The only difference is that, in addition to multicasting notifications as they happen, you need to store the notifications and replay them when an observer subscribes. This is what the `Replay` operator does (shown in figure 7.12), and it has many overloads to support doing just that. All of the overloads create a `ReplaySubject<T>` that you can use inside `ConnectableObservable`.



**Figure 7.12** Turning a hot observable to a cold observable is necessary when you want to capture emissions and replay them.

The Replay operator has many overloads that let you constrain both the time and the number of items to remember and replay. Here's an example that lets you replay the last two items for any observer that subscribes:

```

var publishedObservable = Observable.Interval(TimeSpan.FromSeconds(1))
    .Take(5)
    .Replay(2);
publishedObservable.Connect();
var subscription1 = publishedObservable.SubscribeConsole("First");
Thread.Sleep(3000);
var subscription2 = publishedObservable.SubscribeConsole("Second");

```

**Creates a connectable observable that replays the last two items**

**Connects to the source observable**

**Receives the last two values and all the subsequent ones**

**Waits 3 seconds before subscribing the second observable (meaning you missed three values)**

Running this application shows this output:

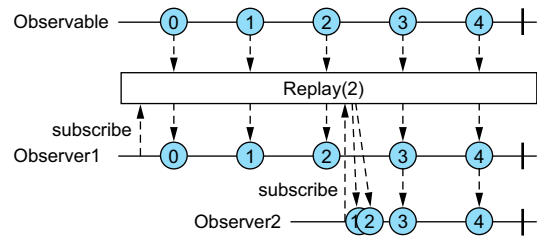
```

First - OnNext(0)
First - OnNext(1)
First - OnNext(2)
Second - OnNext(1) ← subscribing the second observable
Second - OnNext(2)
First - OnNext(3)
Second - OnNext(3)
First - OnNext(4)
Second - OnNext(4)
First - OnCompleted()
Second - OnCompleted()

```

The preceding results show how the Replay operator caches and then re-emits notifications from the source observable. Figure 7.13 shows the marble diagram.

It's important to understand the implications of the operators you use and how they might make an observable hot or cold. By using the operators you've seen in this chapter, such as Publish and Replay, you can control the temperature so that there will be no doubt about the results of the queries you write, therefore making your code more readable and predictable.



**Figure 7.13** Marble diagram showing the result of the Replay operator with a buffer size of two items

## 7.4 Summary

In this chapter, you've learned the definition of the observable temperature and the difference between cold and hot observables. You've also seen how to control the temperature by using special groups of Rx types called subjects.

Here are the important points of this chapter:

- A type that's both an observable and an observer is called a subject.
- Subjects implement the interface `ISubject<TSource, TResult>`, or `ISubject<T>` if the source and result are of the same type.
- Rx provides four built-in subjects: `Subject<T>`, `AsyncSubject<T>`, `ReplaySubject<T>`, and `BehaviorSubject<T>`.
- A subject broadcasts the notifications it receives to all its observers.
- Observables have a notion of temperature; they can be cold or hot.
- A cold observable emits the full sequence of notifications when the observer subscribes.
- A hot observable emits notifications regardless of its observers and may share the notifications among the observers.
- To make a cold observable hot, you use the `Publish` and `Multicast` operators to create a `ConnectableObservable` with an inner subject.
- Calling the `Connect` method on the `ConnectableObservable` subscribes it to the source observable, and the notifications are shared with all observers.
- To automatically unsubscribe the `ConnectableObservable` when there are no more observers, use the `RefCount` operator.
- The `Replay` operator renders a hot observable cold by replaying the notifications to the observers. You can limit the amount of memory used for replaying by specifying the number of items and/or time to keep the items in memory.

In the next chapter, you'll deepen your knowledge of the querying operators Rx has to offer.

# Rx.NET IN ACTION

Tamir Dresher



**M**odern applications must react to streams of data such as user and system events, internal messages, and sensor input. Reactive Extensions (Rx) is a .NET library containing more than 600 operators that you can compose together to build reactive client- and server-side applications to handle events asynchronously in a way that maximizes responsiveness, resiliency, and elasticity.

**Rx.NET in Action** teaches developers how to build event-driven applications using the Rx library. Starting with an overview of the design and architecture of Rx-based reactive applications, you'll get hands-on with in-depth code examples to discover firsthand how to exploit the rich query capabilities that Rx provides and the Rx concurrency model that allows you to control both the asynchronicity of your code and the processing of event handlers. You'll also learn about consuming event streams, using schedulers to manage time, and working with Rx operators to filter, transform, and group events.

## What's Inside

- Introduction to Rx in C#
- Creating and consuming streams of data and events
- Building complex queries on event streams
- Error handling and testing Rx code

Readers should understand OOP concepts and be comfortable coding in C#.

**Tamir Dresher** is a senior software architect at CodeValue and a prominent member of Israel's Microsoft programming community.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit [manning.com/books/rx-dot-net-in-action](http://manning.com/books/rx-dot-net-in-action)

“Keep a copy of this book handy to put Rx.NET into action!”

—From the Foreword by Erik Meijer, Inventor of Rx

“An excellent, deep journey towards true event-driven programming.”

—Stephen Byrne, Dell

“Thorough and comprehensive, with hundreds of code examples.”

—Edgar Knapp  
ISIS Papyrus Software

“An essential resource to take your reactive programming skills to the next level. A must-read.”

—Rohit Sharma, Morgan Stanley



\$49.99 / Can \$65.99 [INCLUDING eBook]

ISBN-13: 978-1-61729-306-1  
ISBN-10: 1-61729-306-7  
5 4 9 9 9



9 781617 129306

