Rx.NET IN ACTION

Tamir Dresher

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Rx.NET in Action
Version 11

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Rx.NET in Action*. I'm excited to see the book reach this stage and look forward to its continued development and eventual release. This is an intermediate level book that is aimed at any developer who writes applications that handle events of any kind – either inter-application (such as Client-Server) or intra-application (such as GUI).

When I first started to play with *Rx*, there weren't too many resources available to help users understand what it does and how it works. There were blogs, tutorials and some useful online guides, but none of them gave the full picture and talked about why I should even use it. Five years have passed since I began experimenting with Rx and things haven't changed much. For someone who is new to Rx, it's still hard to find this knowledge.

Rx.NET in Action is my attempt to produce a comprehensive, highly informative, deep dive into the *Rx* library. *Rx.NET in Action* covers both the fundamental building blocks of how *Rx* works with asynchronous and event-based programs, and also the advanced patterns of using Rx to create reactive applications with ease.

The book is divided into three parts. Part 1 will introduce to you the meaning of being reactive and will show you the difference between the traditional style of writing event-based applications and the reactive style. In addition, we will touch on the foundations that *Rx* is based on, including functional programming and LINQ.

Part 2 will introduce you to the core ideas of Rx. You will learn how to create observables and observers (representing the event-producer and the consumer in Rx) so we can create queries on them as if they were a simple data-store. Then we'll see how we can correlate different observables and deal with time and asynchronous code execution.

Part 3 will continue to introduce more query capabilities together with solutions to problems you may encounter in your day-to-day work. We will see advanced uses of Rx and ways to extend Rx to make it a better fit to your application domain, as well as ways to test it.

I'm anxious to know your thoughts and suggestions on what's been written and what you'd like to see in the rest of the book. Your feedback is invaluable in improving *Rx.NET in Action.*

Thanks again for your interest and for purchasing the MEAP!

—Tamir Dresher

brief contents

1

Reactive programming

This chapter covers

- Being Reactive
- Thinking about events as streams
- Introducing Reactive Extensions (Rx)

The Reactive programming paradigm has gained increasing popularity in recent years as a model that aims to simplify the implementations of event-driven application and to provide a simple mechanism for working with asynchronous code execution.

Reactive Programming[1] concentrates on the propagation of changes and their effects – simply put, how to react to changes and create data flows that depend on them.

With the rise of application such as Facebook, Twitter and the like, where every change (like a status update) happening on one side of the ocean is immediately observed on the other side, and a chain of reactions is happening inside the application instantly, it shouldn't come as a surprise that a simplified model to express this reaction-chain is needed. Today, modern applications are highly driven from changes happening in the outside environment, such as GPS location, battery and power management, social networking messages, and so on, and also from changes from inside the application, such as web calls responses, file reading and writing, timers, and more. To all of those events the applications are reacting accordingly, such as changing the displayed view or modifying stored data.

[1] This book is about Reactive Programming and not about Functional Reactive Programming (FRP). FRP can operate on continuous-time, which is not supported by Rx that can only operate on discrete points of time. More info can be found at the FRP creator keynote https://github.com/conal/talk-2015-essence-and-origins-of-frp

We see the necessity for a simplified model for reacting to events in many types of applications: robotics, mobile apps, health care, and more. Reacting to the events in a classic imperative way leads to cumbersome, hard to understand, and error-prone code because the coordination between events and data changes is the responsibility of the poor programmer who needs to do so manually and work with isolated islands of code that can change the same data and might happen in an unpredictable order and even at the same time. Reactive programming provides abstractions to events and to states that are changed over time so that we can free ourselves from managing the dependencies between those values when we create the chains of execution that run when those events occur.

Reactive extensions (Rx) is a library that provides the Reactive programming model in the .NET applications. Rx makes the events handling code simpler and expressive by using declarative operations (in LINQ style) to create queries over a single sequence of events. Rx also provides methods that are called combinators (combining operations) that allow joining different sequences of events to handle patterns of event occurrence or correlation between them. As of the time of this writing, there are more than 600 operations (with overloads) in the Rx library—each one encapsulates a recurring event processing code that otherwise you'd have to write yourself.

This book's purpose is to teach you why and how you should embrace the Reactive programming thinking and how you can leverage Rx to build event-driven applications with ease and, most importantly, fun. The book will go step by step in teaching the various layers that Rx is built upon, from the building blocks that allow creation of reactive data and event streams, through the rich query capabilities that Rx provides, and the Rx concurrency model that allows you to control the asynchronicity of your code and the processing of your Reactive handlers. But first we need to understand what being Reactive means, and what's the difference between the traditional imperative programming and the Reactive way of working with events.

1.1 Being Reactive

As changes happens in an application your code needs to react to them: that's what being reactive means. Changes comes in many forms, the most simple one is a change of a variable value that we are so accustomed to in our day-to-day programming. The variable holds a value that can be changed at a particular point of time by some operation. For instance, in C# you can write something like this:

```
int a = 2;
int b = 3;
int c = a + b;
Console.WriteLine("before: the value of c is {0}",c);

a=7;
b=2;
Console.WriteLine("after: the value of c is {0}",c);

Output:
```

```
before: the value of c is 5
 after: the value of c is 5
```

In this small program both of the printouts show the same value for the **c** variable. In our imperative programming model the value of **c** is five, and it will stay five unless you override it explicitly.

Sometimes you want **c** to be updated the moment **a** or **b** changes. In Reactive programming we introduce a different type of variable that are time-varying, meaning the value of the variable isn't fixed to the value that was assigned to it, but rather the value changes by reacting to changes that happen over time.

If you look again at the little program we wrote, when it's running in a Reactive programming model the output will be:

```
Output:
before: the value of c is 5
 after: the value of c is 9
```

"Magically" the value of c has changed. This is due to the change that happened to its dependencies. Just like a machine that is fed from two parallel conveyers and produce an item upon entrance of an input from either side

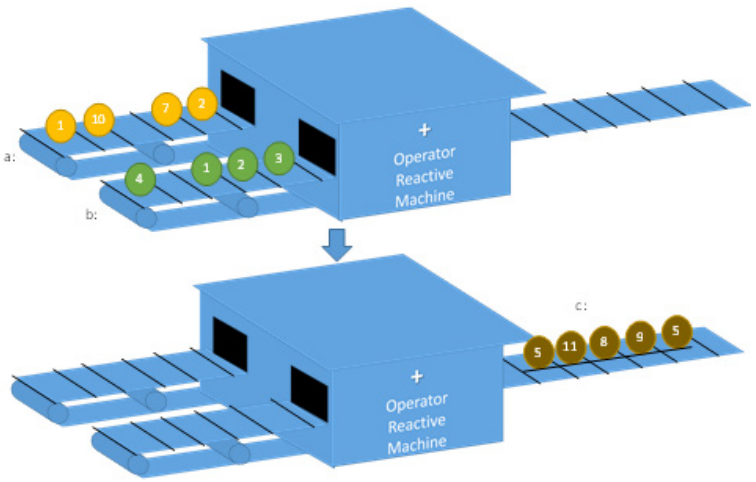

Figure 1.1 A Reactive representation of the function c=a+b. As the values of a and b are changing, c's value is changing as well. When a was 7 and b was 2, c automatically changed to 9 and when b changed to 1, c became 8 because b's value was still 7.

You might find it surprising, but you worked with such Reactive application for years. This concept of reactiveness is what makes your favorite spreadsheet application so easy and fun

to use. When you create this type of equation in a spreadsheet cell, each time you change the value of cells from which the equation is made of, the result in the cell changes automatically.

1.1.1 Reactiveness in your application

In a real-world application, we can spot possible time-variance variables in many circumstances, for instance, GPS location, temperature, mouse coordinates, or even text-box content. All of those hold a value that is changing over time, to which the application reacts, and are therefore time-variant.

It's also worth mentioning that time itself is a time-variant, its value is changing all the time.

In an imperative programming model such in C# you'd use events to create the mechanism of reacting to change, but that can lead to hard-to-maintain code because the code that involves different events is scattered among different code fragments.

Imagine a mobile application that helps users find discounts and specials in shops that are located in their surrounding area—let's call it "Shoppy". The Shoppy architecture is described in figure 1.2.

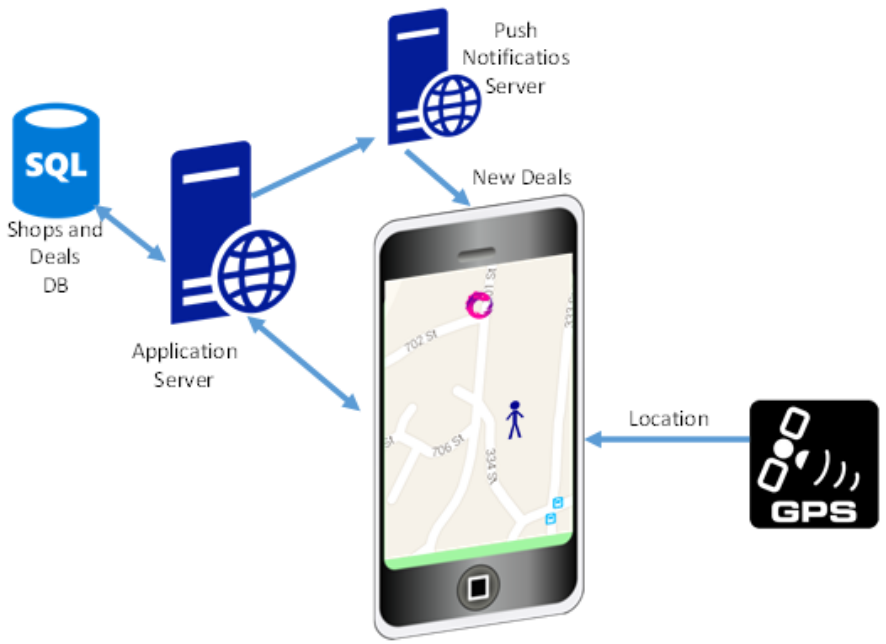

Figure 1.2 The Shoppy application architecture. The mobile app receives the current location from the GPS and can query about shops and deals via the application service. When a new deal is available, the application service sends a push notification through the push notifications server.

One of the great features we want from Shoppy is to make the size of the shop icon bigger on the map as the user is getting closer (from a certain minimal radius). We also want the system to push new deals to the application when updates are available.

Figure 1.3 The Shoppy application view of the map. When the user is far from the Rx shop, the icon is small (on the left) and as the user gets closer the icon gets bigger (on the right).

In this scenario we could say that the *store location*, *myLocation,* and *iconSize* are time-variant, and so for each store the icon size could be written as:

```
distance = store.Location – myLocation;
iconSize = (MINIMAL_RADIUS / distance)*MinIconSize
```

Because we've used time-variant variables, each time that there's a change in the `myLocation` variable, it will trigger a change in the `distance` variable, and the application will eventually react by making the store icon appear bigger or smaller depending on the distance from the store. Note that for simplicity, I didn't handle boundary check on the minimum allowed icon size, and the option that distance might be zero or close to it.

This is very a very simple example, but as we'll see as we continue, the great power of using the Reactive programming model lies in the ability to combine and join, as well as partition and split the *stream* of values that each of the time-variant variables is pushing. This is because Reactive programming lets you focus on the purpose of what you're trying to achieve rather than on the technical details of making it work. This leads to a simple and readable code and eliminates most of the boilerplate code (such as change tracking or state

management) that distracts you from the intent of your code logic. When the code is short and focused it's less buggy and easier to grasp.

We can now stop talking theoretically and see how we can bring Reactive programming into action in .NET with the help of Rx.

1.2 Introducing Reactive Extensions (Rx)

Now that we covered what Reactive Programming is it's time to get to know our star: **Reactive Extensions** which is often shortened to **Rx**. Reactive Extensions is a library that was developed by Microsoft to make it easy to work with streams of events and data. In a way, a time-variant value is by itself a stream of events; each value change is a type of event that we subscribe to and updates the values that depend on it.

Rx makes it easy to work with the streams of events by abstracting them as **observable sequences,** which are also the way Rx represents the time-variant values. *Observable* means that you as a user can observe the values it carries, and *sequence* means an order exists to what's carried. Rx was architected by Erik Meijer and Brian Beckman and drew its inspiration from the functional programming style[2]. In Rx a stream is represented by **observables** that you can create from .NET events, tasks, collections, or create by yourself from another source. Using Rx you can query the observables with **LINQ** operators and control the concurrency with **schedulers**[3]**;** that's why Rx is often defined in the Rx.NET sources[4] as Rx = Observables + LINQ + Schedulers.

[2] https://en.wikipedia.org/wiki/Functional_programming
[3] A Scheduler is unit which holds an internal clock and is used to determine when and where (Thread, Task, and even machine) notifications are emitted
[4] https://github.com/Reactive-Extensions/Rx.NET

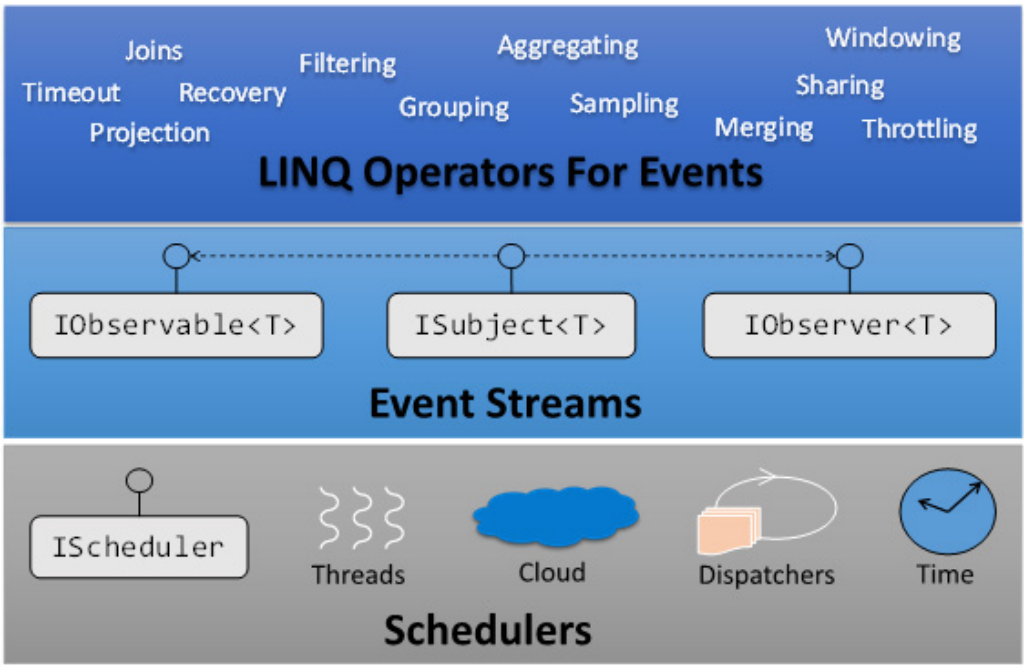

Figure 1.4 The Rx Layers – In the middle are the key interfaces that represent event streams, on the bottom are the schedulers that control the concurrency of the stream processing. Above all is the powerful operators library that allows creating an event processing pipeline in LINQ style.

We will explore each component of the Rx layers throughout this book, as well as the interaction between them, but first a short history about Rx origins.

1.2.1 Rx History

I believe that to get full control on something (especially technological) you should know the history and the behind-the-scenes of the subject. Let's start with the Rx logo. The Rx logo is of the electric eel that you see in figure 1.5, the eel was Microsoft Labs' Volta project logo. The Volta project was an experimental developer toolset for creating multi-tier applications for the cloud when the term cloud wasn't defined and just started to form. Using Volta you could specify which portion of your application needs to run in the cloud (server) and which on the client (desktop, JavaScript or Silverlight) and the Volta compiler will do the hard work for you. Soon it became apparent that a gap exists in transferring events arising from the server to the clients, and because .NET events aren't first-class citizens they cannot be serialized and pushed to the clients, soon the observable and observer pair was formed (though they weren't called that at the time). Rx isn't the only technology that came out of project Volta, an IL

(intermediate language) to JavaScript compiler was also invented then and is the origin of Microsoft Typescript. The same team that worked on Volta is the one that brought Rx to life.

Figure 1.5 Rx electric eel logo – inspired from the Volta project that Rx was born in its development.

Since its release in 2010 Rx has been a success story and has been adopted by many companies. Its success has been seen in other communities outside the .NET, and it was soon being ported to other languages and technologies. Netflix, for example, uses Rx extensively in its service layer and is responsible for the JavaRx port[5]. Microsoft also uses Rx internally to run Cortana—the intelligent personal assistant that's hosted inside every Windows Phone device—when you create an event an observable is created in the background. As of the time of this writing, Rx is supported in more than 10 languages, including JavaScript, C++, Python, Swift and more. The Reactive Extensions are now a collection of open-source projects. You can find information about them as well as documentation and news on the ReactiveX homepage (http://reactivex.io/). The Reactive Extensions for .NET are hosted under GitHub repo at https://github.com/Reactive-Extensions/Rx.NET.

Now that we covered a bit of history and survived to tell we can start exploring the Rx internals.

1.2.2 Rx on the client and server

Rx makes a good fit with event driven applications. This makes sense because events as we saw earlier are the imperative way to create time-variant values. Historically, event driven programming was mostly seen in client side technologies because of the user interaction that was implemented as events. For example, you might have worked with OnMouseMove or OnKeyPressed events. For that reason, it's no wonder that you see many client applications using Rx; furthermore, there are even client frameworks that are based on Rx, such as ReactiveUI (http://reactiveui.net/). Despite that, let me assure you that Rx isn't client-side only technology. On the contrary, many scenarios exist for server side code that Rx will fit perfectly. Furthermore, as I said before, Rx is used for large applications such as Microsoft Cortana, Netflix, and CEP (complex event processing) using Microsoft StreamInsight. Rx is an

[5] http://techblog.netflix.com/2013/02/rxjava-netflix-api.html)

excellent library for dealing with messages that the application receives, and it doesn't matter if it's running on a service layer or a client layer.

1.2.3 Observables

Observables are how the time-variant values (which we defined as observable sequences) are implemented in Rx. They represent the "push" model in which new data is pushed (or notified) to the observers.

In one sentence observables are defined as the source of the events (or notifications) or, if you prefer, the publishers of a stream of data. And the push model means that instead of having the observers fetch data from the source and always checking if there's a new data that wasn't already taken (the "pull" model), the data is delivered to the observers when it's available.

Observables implement the `IObservable<T>` interface that resides in the `System` namespace since version 4.0 of the .NET Framework, as shown in the following listing.

Listing 1.1 The IObservable interface

```
public interface IObservable<T>
{
    IDisposable Subscribe(IObserver<T> observer);  (1)
}
```

(1) Subscribes an observer to the observable sequence.

The `IObservable<T>` interface has only one method `Subscribe` that allows observers to be subscribed for notifications. The `Subscribe` method returns an `IDisposable` object that represents the **subscription** and allows the observer to unsubscribe at any time by calling the `Dispose()` method. Observables hold the collection of subscribed observers and notify them when there's something worth notifying. This is done using the `IObserver<T>` interface, which also reside in the `System` namespace since version 4.0 of the .NET Framework, as shown in the following listing.

Listing 1.2 The IObserver interface

```
public interface IObserver<T>
{
    void OnNext(T value);  (1)
    void OnError(Exception error);  (2)
    void OnCompleted();  (3)
}
```

(1) Notifies the observer of a new element in the observable sequence.
(2) Notifies the observer that an exception has occurred.
(3) Notifies the observer that the observable sequence has completed and no more notifications will be emitted.

The basic flow of using IObservables and IObservers is visualized in the sequence diagram in figure 1.6.

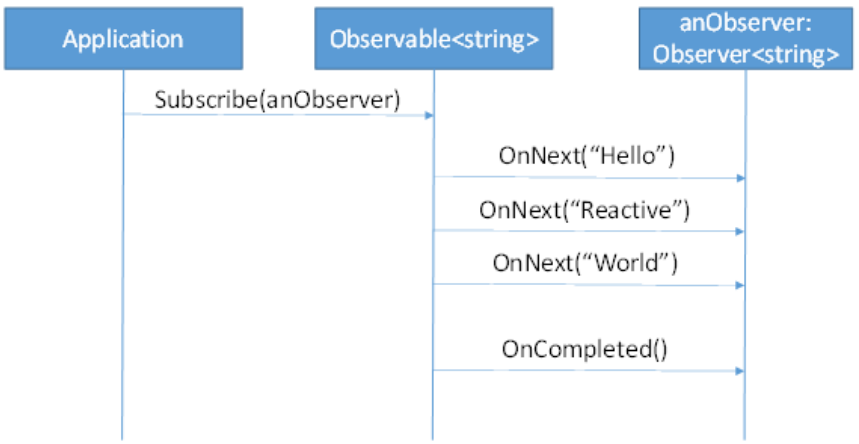

Figure 1.6 Sequence diagram of the happy-path of observable and observer flow of interaction. In this scenario an observer is subscribed to the observable by the application, the observable "pushes" three messages to the observers (only one in this case), and then notifies the observers it has completed.

Observables don't always complete; they can be providers of a potentially unbounded number of sequenced elements (such as an infinite collection). This case also includes the scenario that the observable is "quiet", meaning it never pushed any element and never will. Observables can also fail—the failure can be after the observable has already pushed elements and it can happen without any element ever being pushed.

This observable algebra is formalized in the following expression:

OnNext(t)* (OnCompleted() | OnError(e))?
* - zero more times. ? – zero or one times. | - an OR operator.

When failing, the observers will be notified using the `OnError` method, and the exception object will be delivered to the observers to be inspected and handled. After an error (as well as after completion) no more messages will be pushed to the observers. The default strategy Rx uses when the observer doesn't provide an error-handler, is to escalate the exception and causing a crash. We will learn about the ways you handle errors gracefully in chapter 10.

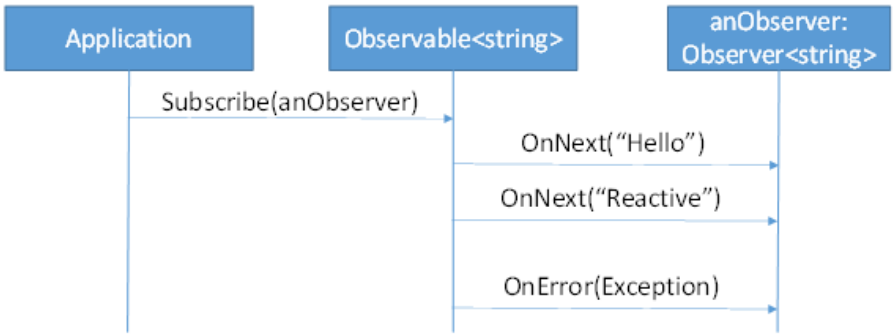

Figure 1.7 In the case of an error in the observable, the observers will be notified through the OnError method with the exception object of the failure.

The observer design pattern

In certain programming languages events are sometimes offered as a first-class citizen, meaning that you can define and register to events with the language-provided keywords and types and even pass it as a parameter to functions.

For languages that don't support events as first-class citizens, the observer pattern is a useful design pattern that allows adding events-like support to your application. Furthermore, the .NET implementation of events is based on this pattern.

The observer pattern was introduced by the Gang of Four (GoF) in their book "Design Patterns: Elements of Reusable Object-Oriented Software". The pattern defines two components: subject and observer (not to be confused with IObserver of Rx). The observer is the participant that's interested in an event and subscribes itself to the subject that raises the events. This is how it looks in a UML (Unified Modeling Language) Class Diagram:

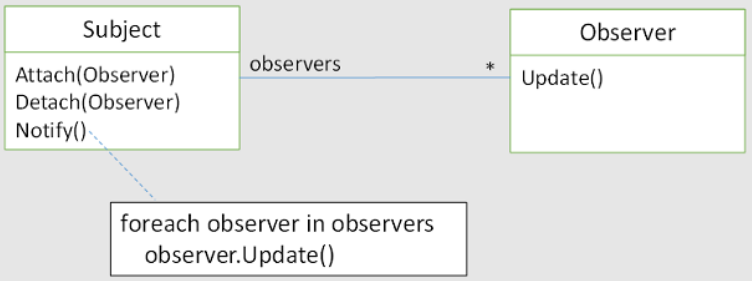

Figure 1.8 The Observer design pattern class diagram.

The observer pattern is useful but it has several problems. The observer only has one method to accept the event. If we want to attach to more than one subject or more than one event we need to implement more update methods. Another problem is that the pattern doesn't specify the best way to handle errors, and it's up to the developer to find a way to notify of errors, if at all. Last but not least is the problem of how to know when the subject is done, meaning that there will be no more notifications, which might be crucial for right resource management. The Rx IObservable and IObserver are based on the Observer Design Pattern but extends it to solve the shortcomings it carries.

1.2.4 Operators

Reactive Extensions also bring a rich set of **operators,** in Rx an operator is a nice way to say operation, but with the addition that it's also part of a DSL (Domain Specific Language) that describes event processing in a declarative way. The Rx operators allow us to take the observables and observers and create pipelines of querying, transformation, projections, and other event processors you may know from LINQ. It also includes time-based operations and Rx-specific operations for queries, synchronization, error handling, and so on.

For example, this is how you subscribe to an observable sequence of strings that will show only strings that begin with "A" and will transform them to uppercase:

```
IObservable<string> strings= ...  (1)

IDisposable subscription =        (2)
    strings.Where(str => str.StartsWith("A"))  (3)
        .Select(str => str.ToUpper())          (4)
        .Subscribe(...);          (5)

//Rest of the code
:
subscription.Dispose();           (6)
```

(1) observable of strings that will push strings to observers.
(2) Saving the *subscription* enables us to unsubscribe later.
(3) Allows only strings that start with "A" to be passed to the observer. I used a lambda expression here as a short way to create a method inline.
(4) The string is transformed to uppercase before continuing.
(5) An observer is subscribed to receive strings that passed through the filtering and transformation.
(6) When we no longer want to receive the strings, we dispose the subscription.

> **NOTE** Don't get scared if you don't understand all the syntax and the meaning of each keyword. I'll explain all of them in the next chapters

In this very simple example we can see the declarative style of the Rx operators – say what you want and not how you want it – and so the code reads like a story. Since I wanted to focus on the querying operators in this example, I didn't show how the observable is created. In fact, there are many ways to create observables – from events, from enumerables, from asynchronous types and more. Those will be discusses in chapters 4 and 5. For now, you can assume that the observables were created for us behind the scenes.

The operators and combinators (operators that combine multiple observables) can help us create even more complex scenarios that involve multiple observables.

For example, to achieve the resizable icon for the shops in the Shoppy example we can write the following Rx expressions:

```
IObservable<Store>  stores = ...  (1)
IObservable<Location>  myLocation = ...  (2)
IObservable<StoreIconSize> iconSize =
              from store in stores    (3)
              from currentLocation in myLocation    (4)
```

```
                let distance = store.Location.DistanceFrom(currentLocation)  (5)
                let size = (MINIMAL_RADIUS / dist) * MIN_ICON_SIZE    (5)
                select new StoreIconSize { Store=store , Size=size };  (5)

iconSize.Subscribe( iconInfo => iconInfo.Store.Icon = iconInfo.Size);  (6)
```

1. an observable that delivers the information about stores in the system.
2. the observable that carries the information on our current geo-location.
3. we want to handle each store and assign it the store variable (which is of type `Store` since `stores` is an observable of `Store`).
4. similar to the `stores` observable. By writing this, we get all the pairs of stores and the current location, every time the location changes.
5. the `let` keyword allows us to create a new variable for each pair of store and location. We create two variables in that way to calculate the distance from the store and then the size of the store icon.
6. the lambda expression acts as the observer's OnNext implementation and will be called every time a store icon has a new size.

As you can see, even without knowing all the fine details of Reactive Extensions, it already seems that the amount of code we needed to implement this feature from our Shoppy application is small and easy to read. All the boilerplate of combining the different streams of data was done by Rx and saved us the burden of writing the isolated code fragments that were needed to be written to handle the events of data change.

1.2.5 The composable nature of Rx operators

Most of the Rx operators have the following format:

```
IObservable<T> OperatorName(argumants)
```

Note that the return type is an observable. This allows the composable nature of Rx operators, and as such it allows us to add operators to the observable pipeline and keep the encapsulation of what happens to the notification since emitted from the original source.

Another important takeaway from the fact that operators returns observables is that from the observer point of view, an observable with or without operators that are added to it is still an observable, as shown in figure 1.9

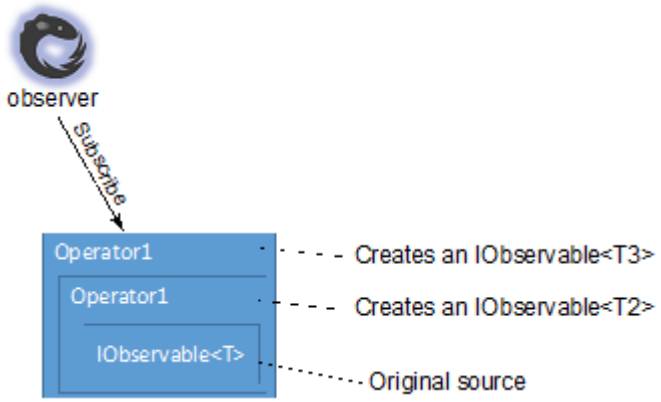

Figure 1.9 the composable nature of Rx operators allows the encapsulation of what happens to the notification since emitted from the original source

Since the addition of operators to the pipeline can be made not only when the observable is created, but also when the observer is subscribed, it gives us the power to control the observable even if we don't have access to the code that created it.

1.2.6 Marble diagrams

"A picture is worth a thousand words." That's why when explaining reactive programming and Rx in particular, it's important to show the visualization of the execution pipeline of the observable sequences. In this book I'll use Marble diagrams to help you understand the different operations and their relations.

In Marble diagrams we use a horizontal axis to represent the observable sequence, each notification that's carried on the observable is marked with a symbol, usually a circle, but other symbols will be used from time to time to distinguish between different values. The value of the notification will be written inside the symbol or as a note above it as shown in figure 1.10.

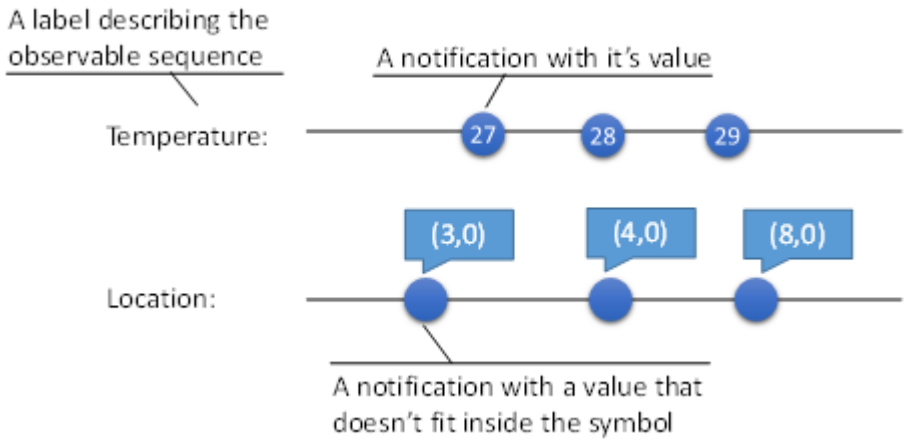

Figure 1.10 Marble diagram with two observable sequences.

In the marble diagram the time goes from left to right and the distance between the symbols shows the amount of time that passed between the two events: the longer it is, the more time has passed, but only in a relative way. There's no way to know if the time is in seconds, hours, or another measurement unit. If such information is important, it will be written as a note.

To show that the observable has completed, we use the **|** symbol. To show that an error occurred (which also ends the observable), we use **X.**

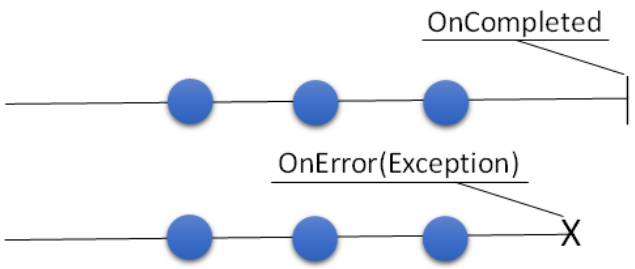

Figure 1.11 Observable can end because it has completed or because an error occurred.

To show the output of an operator (or multiple operators) on an observable we can use an arrow that shows the relation between the source event and the result.

Remember that each operator (at least the vast majority of operators) returns observables of their own, so in the diagram I'm writing the operator that is part of the pipeline on the left side and the line that represent the observable returned from it on the right side.

For example, figure 1.12 shows how the marble diagram will look for the example we saw previously of an observable sequence of strings that shows only the string that begins with "A" and transforms them to uppercase.

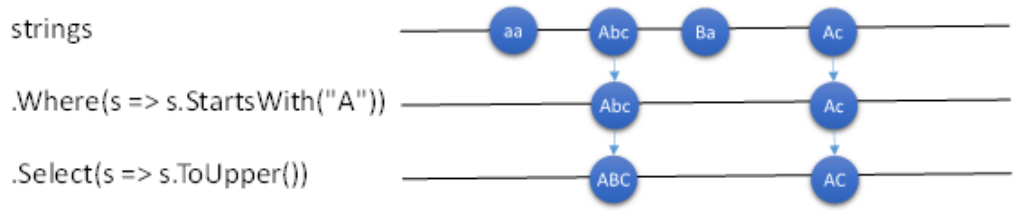

Figure 1.12 Marble diagram that shows the output of different operators on the observable.

Marble diagrams will be used in this book to show the effects of operators as well as examples of combining different operators to create observables pipelines. A common question that may arise for you at this point is how observable sequences relate to non-observables sequences, which are commonly addressed as enumerables. The answer is on the next page.

1.2.7 Pull model vs push model

Non-observable sequences are what we normally call enumerables (or collections), which implement the `IEnumerable` interface and return an iterator[6] that implements the `IEnumerator` interface. When using enumerables we pull values out of the collection, usually with a loop. Rx observables behave differently: instead of pulling, the values are pushed to the observer. Tables 1.1 and 1.2 show how the pull and push models correspond to each other. This relation between the two is called the duality principle[7].

Table 1.1 How IEnumerator and IObserver correspond to each other

IEnumerator	Iobserver
MoveNext() – when false	OnCompleted()
MoveNext() – when exception	OnError(Exception exception)
Current	OnNext(T)

[6] https://en.wikipedia.org/wiki/Iterator
[7] This observation was made by Erik Meijer http://csl.stanford.edu/~christos/pldi2010.fit/meijer.duality.pdf

Table 1.2 How IEnumerable and IObservable correspond to each other

IEnumerable	Iobservable
IEnumerator GetEnumerator(void)	IDisposable Subscribe(IObserver) [8]

Observables and observers fill the gap .NET had when dealing with asynchronous operation that needs to return a sequence of values in a push-model (pushing each item in the sequence). Unlike Task<T> that provides a single value asynchronously, or IEnumerable that gives multiple values, but in a synchronous pull-model. This is summarized in table 1.3.

Table 1.3 Push model and pull model data types

	Single value	Multiple values
Pull/Synchronous/Interactive	T	IEnumerable<T>
Push/Asynchronous/Reactive	Task<T>	IObservable<T>

Since there is a reverse correspondence between observables and enumerables (the duality), it means that you can move from one representation of a sequence of values to the other. A fixed collection, such as List<T>, can be transformed to an observable that emits all its values by pushing them to the observers. The more surprising fact is that observables can be transformed to pull-based collections. We will dive into the details of how and when you should make those transformations in later chapters. For now, the important thing to understand is that because you can transform one model to the other, everything that you can do with a pull-based model can also be done with the push-based model. So when you face a problem you can solve it in the model that does that in the easiest way and transform the result if needed.

The last point I'll make here is that because we can look at a single value as if it was a collection of one item, we can by the same logic see that we can take the asynchronous single item—the Task<T>—and look at it as an observable of one item and vice versa. Keep that in mind, because it an important point in understanding that "everything is an observable".

1.3 Reactive systems and the Reactive Manifesto

So far, we've discussed how Rx adds reactiveness to an application, but many applications aren't standalone, but rather part of a whole system that is composed from more applications (desktop, mobile, web), servers, databases, queues, service buses, and other components that you need to connect together to create a working organism. The Reactive Programming

[8] There's one exception to the duality here since the dual of the GetEnumerator parameter (which is void) should have been transformed to the Subscribe method return type (and stay void) but Instead IDisposable was used.

Model (and Rx as an implementation of that model) simplifies the way the application handles the propagation of changes and the consumption of events, thus making the application reactive. But how can we make a whole system reactive?

As a system, reactiveness is defined by being responsive, resilient, elastic, and message-driven, these traits of reactive systems are defined in the **Reactive Manifesto**.

The Reactive Manifesto (http://www.reactivemanifesto.org/) is a joint effort of the software community to define what reactiveness is, by collaborating the experience of developing software throughout the years. The Reactive Manifesto is a short text that was set to define the best architectural style for building a reactive system. You can join the effort of raising the awareness to reactive systems by signing the manifesto yourself and spreading the word. It's important to understand that the Reactive Manifesto didn't invent anything new; Reactive applications existed long before it was published. An example is the telephony system that's existed for decades – this type of distributed system needs to react to a dynamic amount of load (the calls), recover from failures and stay available and responsive to the caller and the callee 24/7, and all this by passing signals (messages) from one operator to the other.

The manifesto is here to put the Reactive Systems term on the map and to collect the best practices of creating such a system. Let's drill into those concepts.

1.3.1 Responsiveness

When you go to your favorite browser and enter a URL, you expect that the page you were browsing to will load in a short time. When the loading time is longer than a few milliseconds you get a bad feeling (and even get angry). You might even decide to leave that site and browse to another website. If you're the website owner, you lost a customer because your website wasn't **responsive**. Responsiveness of a system is determined by the time it takes for the system to respond the request it received—a shorter time to respond means that the system is more responsive.

A response from a system can be a positive result, such as the page you tried to load or the data you tried to get from a web service or the chart you wanted to see in the financial client application.

Response can also be negative like an error message that specifies that one of the values you gave as input was invalid.

In any of those cases, if the time that it took for the system to respond was reasonable, we can say that the application was responsive.

Reasonable time for a response is a problematic thing to define because it depends on the context and on the system we're measuring. For a client application that has a button it's assumed that the time it takes the application to respond to the button click will be a few milliseconds, but for a web service that needs to make a heavy calculation one or two seconds might also be reasonable. When you're designing your application you need to analyze the operations you have and define the bounds of the time it should take for an operation to

complete and respond. Being responsive is the goal that Reactive systems are trying to achieve.

1.3.2 Resiliency

Every once in a while your system might face failures. Networks disconnect, hard drives fail, electricity shuts down, or an inner component experiences an exceptional situation. A resilient system is one that stays responsive in the case of a failure. In other words, when you write your application you want to handle failures in a way that doesn't prevent the user from getting a response.

The way you add resiliency to an application is different from one application to another, For one application, it might be catching an exception and return the application to a consistent state, for another application, it might be adding more servers so that if one of the servers crashes, another server will compensate and handle the request. A good principle you should follow to increase the resiliency of your system is to avoid a single point of failure. This can be done by making each part of your application isolated from the other parts; isolation might be separating into different AppDomains, different processes, different containers; or different machines. By isolating the parts, you reduce the risk that the system will be unavailable as whole.

1.3.3 Elasticity

The application that you're writing will be used by a number of users, hopefully a large number of users. Each user will make requests to your system that may result in high load that your system will need to deal with. Each component in your system has a limit on the load level it can deal with, and when the load goes above that limit, requests will start failing and the component itself may crash. This situation of increasing load can also be a Distributed Denial of Service (DDoS) attack that your system is experiencing. To overcome the causes of overload your system needs to be elastic and span instances as the load increases and remove instances as the load decreases. This kind of automatic behavior is much more apparent since the cloud entered our lives. When running on the cloud you get the illusion of infinite resources and with a few simple configurations you set your application to scale up or down depending on the threshold you define. You only need to remember that a cost is associated with running extra servers.

1.3.4 Message driven

If we look at responsiveness, resiliency, and elasticity we talked about earlier we can say that responsiveness is our goal, resiliency is the way we ensure we keep being responsive, and elasticity is one method for being resilient. The missing piece of the puzzle of Reactive systems is the way that the parts of a system communicate with each other to allow the concepts of reactiveness we've explored. Asynchronous message passing is the

communication message that best suits our needs, because it allows us to control the load level on each component without limiting producers—normally with an intermediate channel such as a queue or service bus. It allows routing of messages to the right destination and resending of failing messages in case a component crashes. It also adds transparency to the inner system components because users don't need to know the internal system structure except the type of messages it can handle. Message driven is what makes all the other Reactive concepts possible. Figure 1.12 shows how the message driven approach using a message queue helps in leveling the rate of the message processing in the system and how it enables resiliency and elasticity.

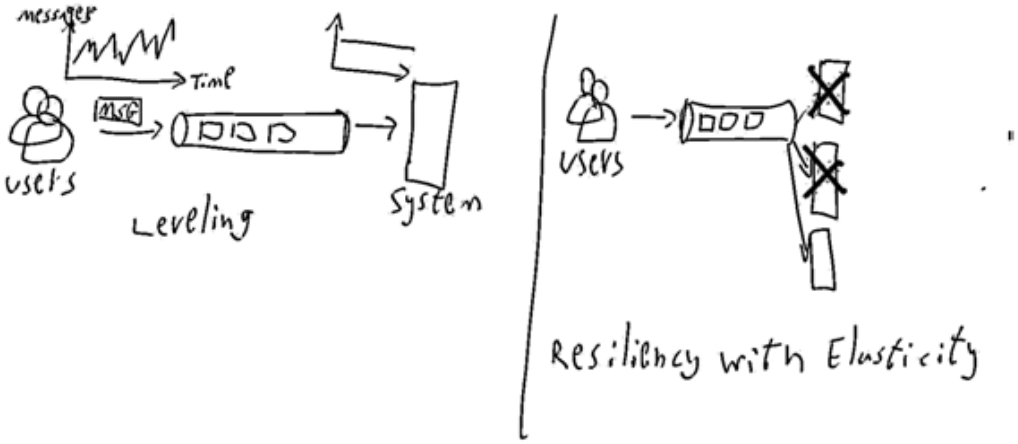

Figure 1.13 Message Driven relation to Load leveling and Elasticity. On the left, we see that although messages are arriving at a high frequency, the processing by the system is leveled to a constant rate, and the queue is filling faster than it's emptied. On the right we see that even if the processing worker roles has crashed users can still fill the queue, and when the system recovers and adds a new worker, the processing continues.

In the figure, the participants are communicating in a message driven approach through the message queue. The client sends a message that is later retrieved by the server. This asynchronous communication model gives us greater control on how we make the processing in the system – controlling the rate and dealing with failures. There are many implementations for message queuing with different feature sets. Some of them allow the persistence of the messages, which provides durability, and some also give a "transactional" delivery mode that locks the message until the consumer signals that the processing completed successfully. No matter which message queue (or message driven platform) you choose, you'll need to somehow get ahold of the messages that were sent and start processing them. This is where Rx fits in.

1.3.5 Where is Rx

In a Reactive system the Reactive Extensions library comes into play inside the applications that compose it, and relates to the concept of message driven. Rx isn't the mechanism to move the messages between the applications or servers, but rather it's the mechanism that's responsible for handling the message when it arrives and passing it along the chain of execution inside the application. It's important to state that working with Rx is something you can do even if you're not developing a full system with many components. Even a single application can find Rx useful for reacting to events and the types of messages that the application may want to process. The relations between all the reactive manifesto concepts and Rx are captured in Figure 1.14.

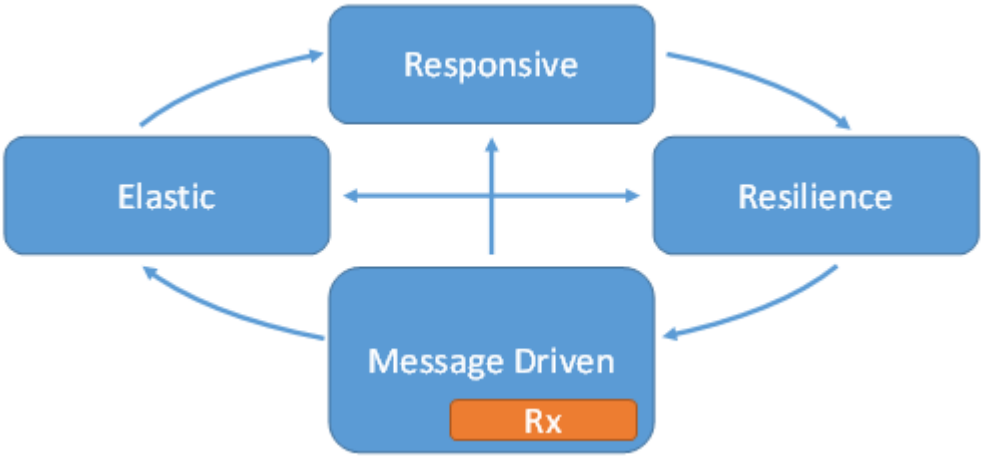

Figure 1.14 The relation between the reactive manifesto core concepts. Rx is positioned inside the Message driven concept since it provides abstractions to handle messages as they enter the application.

To get a fully Reactive system, all the concepts we see in the diagram must exist. Each one can be implemented differently in different systems. Rx is one way to allow easier consumption of messages, so it is shown as part of the Message Driven block. Rx was introduced as a way to handle asynchronous and event-based programs, like in the case of messages, so it's important that we explain what it means to be asynchronous and why it is important.

1.4 Asynchronicity

Asynchronous message passing is a key trait of a Reactive system, but what exactly is asynchronicity means and why is it so important to a Reactive application?

Our lives are made up of many asynchronous tasks. You may not be aware of it, but your everyday activities would be annoying if they weren't asynchronous by nature.

To understand what asynchronicity is we first need to understand non-asynchronous execution, or synchronous execution.

Merriam Webster definition to synchronous

Happening, existing, or arising at precisely the same time.

Synchronous execution means that you have to wait for a task to complete before you can continue to the next task. A real-life example of synchronous execution could be the way you approach the staff at the counter, decide what to order while the clerk waits, wait until the meal is ready, and the clerk waits until you hand the payment. Only then you can continue the next task of going to your table to eat. This sequence is shown in Figure 1.15.

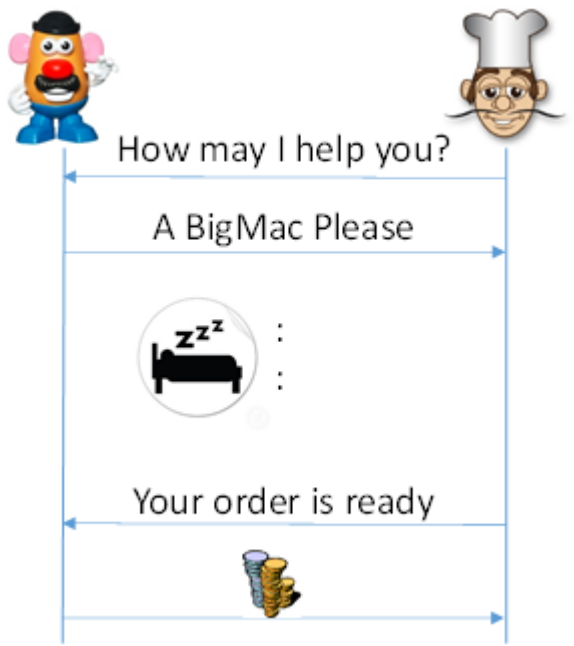

Figure 1.15 Synchronous food order in which every step must be completed before going to the next one.

This type of sequence feels like a waste of time (or, better said, a waste of resources), so imagine how your applications feel when you do the same for them. The next section will demonstrate this.

1.4.1 It's all about resource utilization

Imagine how your life would be if you had to wait for every single operation to complete before you could do something else. Think of the resource that would be waiting and utilized at that time. The same issues are also relevant in computer science:

```
writeResult=LongDiskWrite();
response=LongWebRequest();
entities=LongDatabaseQuery();
```

In this synchronous code fragment, `LongDatabaseQuery()` won't start execution until `LongWebRequest()` and `LongDiskWrite()` completes. During the time that each of these methods are executed the calling thread is blocked and the resources it holds are practically wasted and cannot be utilized to serve other requests or handle other events. If this was happening on the UI thread then the application would look frozen until the execution finish.

If this was happening on server application, then at some point we might run out of free threads and requests would start being rejected. In both of those cases the application stops being responsive.

The total time it takes to run the code fragment above is

$$\text{total\_time} = \text{LongDiskWrite}_{\text{time}} + \text{LongWebRequest}_{\text{time}} + \text{LongDatabaseQuery}_{\text{time}}$$

The total completion time is the sum of the completion time of its components.

If we could start an operation without waiting for a previous operation to complete, we could utilize our resources much better, this is what **asynchronous execution** is for.

Asynchronous execution means that an operation is started, but its execution is happening in the background and the caller isn't blocked. Instead, the caller is notified when the operation is completed. In that time the caller can continue to do useful work.

In the food ordering example, an asynchronous approach will be similar to sitting at the table and being served by a waiter.

First, you sit at the table, the waiter comes to hand you the menu and leaves. While you're deciding what to order the waiter is still available to other customers. When you decided what meal you want, the waiter comes back and take your order. While the food is being prepared you're free to chat, use your phone, or enjoy the view. You're not blocked (and neither is the waiter). When the food is ready, the waiter brings it to your table and goes back to serve other customers until you request the bill and pay.

This model is asynchronous, tasks are executed concurrently, and the time of execution is different from the time of the request, this way the resources (such as the waiter) are free to handle more requests.

Where the asynchronous execution happens?

In a computer program, we can differentiate between two types of asynchronous operations, IO based and CPU based.

CPU-based operation means that the asynchronous code will run on another thread and the result will be returned when the execution on the other thread finishes.

IO-based operation means that the operation is made on an IO device such as a hard drive or network. If network is the case, a request was made to another machine (by using TCP or UDP or other network protocol) and when the operating system on your machine gets a signal from the network hardware by an interrupt that the result came back then the operation will be completed.

The calling thread in both of the cases is free to execute other tasks and handle other requests and event.

There's more than one way to run code asynchronously, and it depends on the language that's used. Chapter 4 will show the different ways this can be done in C# code and will dive deeper into bits and bytes of each one. But for now let's look at one example of doing asynchronous work using the .NET implementation of Futures – the `Task` class.

The asynchronous version of the code fragment above will look like the following code:

```
taskA=LongDiskWriteAsync();
taskB=LongWebRequestAsync();
taskC=LongDatabaseQueryAsync();

Task.WaitAll(taskA, taskB, taskC);
```

In this version, each of the methods return a `Task<T>`. This class represents an operation that's being executed in the background. When each of the methods is called, the calling thread isn't blocked, and the method returns immediately and then the next method is called while the previous method is still executed. When all the methods were called we wait for their completion by using the `Task.WaitAll(…)` method that gets a collection of tasks and blocks until all of them completed. Another way we could write this is:

```
taskA=LongDiskWriteAsync();
taskB=LongWebRequestAsync();
taskC=LongDatabaseQueryAsync();

taskA.Wait();
taskB.Wait();
taskC.Wait();
```

This way we get the same result, we wait for each of the tasks to complete (while they're still running in the background). If a task was already completed when we called the `Wait()` method then it will return immediately.

The total time it takes to run the asynchronous version of the code fragment is shown as:

$$total\_time = MAX(LongDiskWrite_{time}, LongWebRequest_{time}, LongDatabaseQuery_{time})$$

Because all of the methods are running concurrently (and maybe even parallel) the time it takes to run the code will the time of the longest operation.

1.4.2 Asynchronicity and Rx

Asynchronous execution isn't limited to only being handled by using `Task<T>`. In fact, in chapter 5 you'll be introduced to other patterns that are used inside the .NET framework to make asynchronous execution.

Looking back at the Rx representation of time-variant variable – the `IObservable<T>` – we can use it to represent any asynchronous pattern, so when the asynchronous execution completes (successfully or with an error) the chain of execution will run and the dependencies will be evaluated. Rx provides methods for transforming the different types of asynchronous execution (like `Task<T>`) to `IObservable<T>` .

For example, in the Shoppy app we want to get new discounts not only when our location changes, but also when our connectivity state changes to online, like in the case our phone lost signal for short period of time and then reconnected. The call to the Shoppy web service is done in an asynchronous way, and when it completes we want to update our view to show the new items.

```
IObservable<Connectivity> myConnectivity=...
IObservable<IEnumerable<Discount>> newDiscounts =
    from connectivity in myConnectivity
    where connectivity == Connectivity.Online
    from discounts in GetDiscounts()  (1)
    select discounts;

newDiscounts.Subscribe(discounts => RefreshView(discounts));  (2)

private Task<IEnumerable<Discount>> GetDiscounts()
{
    //Send request to the server and receives the collection of discounts
}
```

(1) **GetDiscounts() is returning a Task which is implicitly converted to an observable.**
(2) **RefreshViews() will display the discounts**

In this example we're reacting to the connectivity changes that are carried on the `myConnectivity` observable. Each time there's a change in connectivity, we check to see if it's because we're online and, if so, we call the asynchronous `GetDiscounts` method. When the method execution is complete, we select the result that was returned. This result is what will be pushed to the observers of the `newDiscounts` observable that was created from our code.

1.5 Events and streams

In a software system, an event is a type of message that's used to indicate that something has happened. The event might represent a technical occurrence, for example, in a GUI application, we might see events on each key that was pressed or mouse movements. The

event can also represent a business occurrence such a money transaction that was completed in a financial system.

An event is raised by an **event source** and consumed by an **event handler**.

As we saw events are one way to represent time-variant values. And in Rx, the event source can be represented by the observable, and an event handler can be represented by the observer. But what about the simple data that our application is using, such as the one sitting in the database or fetched from a webserver. Does it have a place in the Reactive world?

1.5.1 Everything is a stream

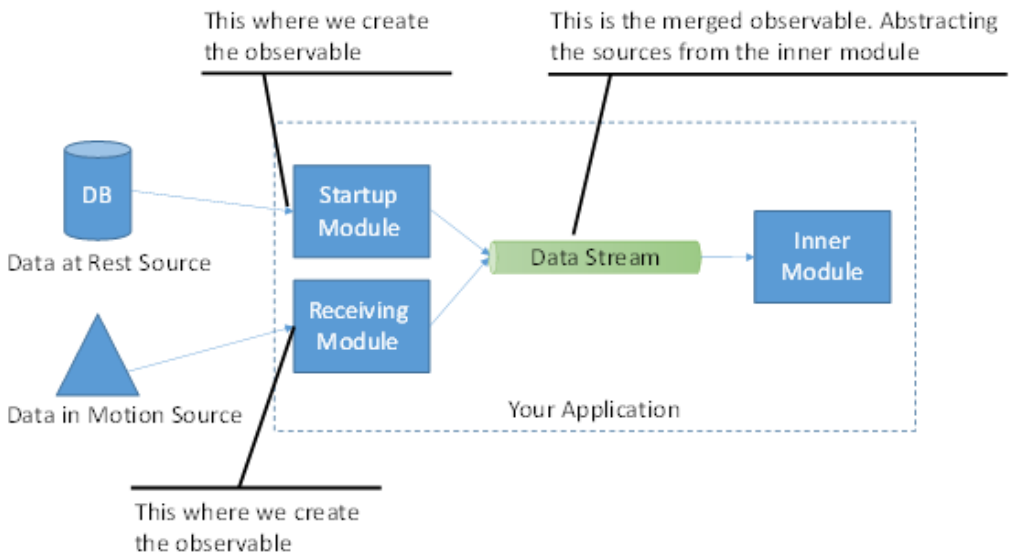

Figure 1.16 Data-in-motion and data-at-rest as one data stream. The connection points from the outside environment are a perfect fit for creating observables. Those observables can be merged easily with Rx to create a merged observable that the inner module can subscribe to without knowing what is the exact source of a data element.

The application you write will ultimately deal with some kind of data. Data can be of two types: data-in-motion and data-at-rest. Data-at-rest is data that's stored in a digital format and that you usually read from some persisted storage such as a database or files. Data-in-motion is data that's moving on the network (or other medium) and is being pushed to your application or pulled by your application from any external source.

No matter what the type of data you use in your application, it's time to understand that everything can be observed as a stream, even data at rest and data that looks static to your application. For example, Configuration data is data that's perceived as static, but even configuration changes at a point of time, either long time or short time. From your application

perspective, it doesn't matter, you want to be *reactive* and handle those changes as they happen. When you look at the data-at-rest as another data stream it makes it easier for you to combine both types of data together. For your application it doesn't matter where the data came from. For example, application startup usually loads data from its persisted storage to restore its state (the one that was saved before the application was closed). This state can, of course, change during the application run and the inner parts of your application that care about the state can look at the data stream that carries it, when the application starts, the stream will deliver the data that was loaded, and when the state changes the stream will carry the updates.

A nice analogy I like to use for explaining streams, is a water-hose, only that this hose has data packets going through it, just like the one you see in Figure 1.17. When using a water hose, there are many things you can do with it, like putting filters at the end, adding different hose-heads that give different functionality. You can add pressure monitors on the hose to help you regulate the flow. The same thing is what you would wish to do with your data stream. You'll want to build a pipeline that let the information flow through it, to eventually give an end result that suits your logic, this include filtering, transformations, grouping, merging, and so on.

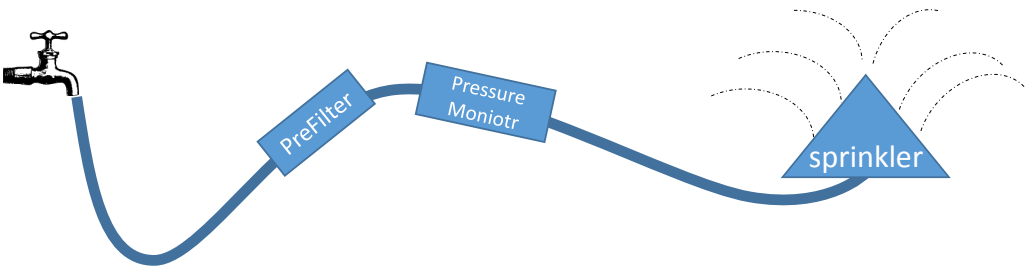

Figure 1.17 Data stream is like a hose, every drop of water is a data packet that needs to go through different stations until it reaches the end. Your data as well needs to be filtered and transformed until it gets to the real handler that does something useful with it.

The data and event streams are a perfect fit for Rx observables, when abstracting them with an IObservable we get the possibility to make composition of the operators and create the complex pipeline of execution. This is similar to what we did with the Shoppy example, where we had a call to a server to get the discounts as part of a of a more complex pipeline of execution that also made filtering (on the connectivity) and eventually refreshed view (like the sprinkler is splashing the water).

1.6 Summary

In this chapter, we covered what being Reactive means and how you can use Rx to implement reactive programming to your applications.

- In Reactive Programming we use time-variance variables that hold values that change by reacting to changes happening on its dependencies. We saw examples of such variables in the Shoppy example - `location`, `connectivity`, `iconSize` and so on.
- Rx is a library that was developed by Microsoft to implement reactive programming in .NET applications.
- In Rx, time-variance variables are abstracted by observable sequences that implement the IObservable<T> interface.
- The observable is a producer of notifications, and observers subscribe to it to receive those notifications.
- Each observer subscription is represented as IDisposable that allows unsubscribing at any time.
- Observers implement the IObserver<T> interface.
- Observables can emit a notification with a payload, notify on its completion, and notify on an error.
- After an observable notified an observer on its completions or about an error, no more notifications will be emitted.
- Observables don't always complete; they can be providers of potentially unbounded notifications.
- Observables can be "quiet", meaning they never pushed any element and never will.
- Rx provides operators that are used to create pipelines of querying, transformation, projections, and more in the same syntax that LINQ uses.
- Marble diagrams are used to visualize the Rx pipelines.
- Reactive systems are defined as being responsive, resilient, elastic, and message-driven. These traits of reactive systems are defined in the Reactive Manifesto.
- In a reactive system, Rx is placed in the message-driven slot, as the way we wish to handle the messages the application is receiving.
- Asynchronicity is one of the most important things in being reactive, since it allows us to better use our resources and thus makes the application more responsive.
- "Everything is a stream" is why Rx makes it easy to work with any source, even if it's a data source such as a database.

In the next chapter you'll get the chance to build your first Rx application, and we'll compare it with how you'd write the same application in the traditional event-handling way. You'll see by yourself how awesome Rx is.