Thank you for purchasing the MEAP for *Grokking ReactiveX*. We are excited to see the book reach this stage and look forward to its continued development and eventual release. This is a beginner-level book that covers what ReactiveX is and how it can be used to build flexible, loosely-coupled, and scalable systems. It is designed to be the first book you would read if you wanted to learn ReactiveX.

We are releasing the first two chapters to start. Chapter 1 covers the principles of reactive programming, and the key concepts of ReactiveX such as observables, observers, and subscriptions. Chapter 2 goes more in depth on the observable/observer relationship, and how schedulers approach multi-threading and concurrency.

Together, chapters 1 - 2 create a foundation that we will build on in for the remainder of the book. Our primary focus is making the material easy to follow. Every chapter introduces a new concept and then immediately shows how to use it.

Looking ahead, we will be covering basic and advanced Rx recipes, effective reactive programming, and advanced Rx techniques. By the end of this book, you will have expanded the types of problems you can solve with code.

I expect to have updates at least once a month, in the form of a new chapter or an update to an existing chapter.

I have read technical books that are boring, and I have read technical books that take big leaps of logic. This book aims to avoid both. As you're reading, I hope you will let me know about these issues in the [Author Online forum](https://forums.manning.com/forums/grokking-reactivex).

Thanks for reading!

—Ivan Morgillo
—Fabrizio Chignoli
—Saša Sekulić
brief contents

PART 1: UNDERSTANDING REACTIVE PROGRAMMING

1 Principles of reactive programming
2 Introduction to Rx
3 Basic Rx recipes: Reactive music player
4 More Rx recipes: Creating and handling observables
5 Transforming observable sequences
6 Backpressure and Custom Operators

PART 2: EFFECTIVE REACTIVE PROGRAMMING

7 Daily Backend Life with Rx
8 Daily Frontend Life with Rx
9 Rx and Mobile
10 Debugging and Testing

Appendix A: Future of Rx
Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when failure does occur they meet it with elegance rather than disaster.

— The Reactive Manifesto
Perhaps you picked up this book because...

1. You are curious about ReactiveX.
   You heard about ReactiveX, read about functional reactive programming and looked at a few books too. Maybe you were scared of the terminology and concepts. Yet it still holds a curiosity in you.

   In our hearts, we set out to write the least intimidating ReactiveX book ever. You will be the judge of that, of course, and not us.

2. You tried to learn reactive programming before.
   Many of us have tried learning ReactiveX more than once, and still don’t get it. We might think to have understood a key concept and the next thing is just so abstract we cannot make our head around it. This is not fun.

   We believe that the majority of learning comes from real life examples, not from abstract concepts. We will try to put as many concrete examples as possible throughout this book.

3. For you, the jury is still out.
   The majority of us have been programming for many years in an object oriented programming language like C#, Java or Ruby.
   We have heard the buzz of ReactiveX, read some blog entries, and tried a bit of code. Still, we cannot see how it makes our programming life better.

   We want to show you that using ReactiveX can help you handle concurrency issues in an elegant and simple way, by abstracting away all the complexities.

Whatever your reason, we want you to...

- learn through experimentation and play
- never fear asking a silly question
- feel yourself growing to new levels
Don’t read this book if...

You are an expert ReactiveX programmer.
Perhaps you have been using ReactiveX in C#, JavaScript or Java for quite some time now, and now all the intricacies. You might however still find this book interesting because we will show real life examples.

or

You need to know the inner workings.
Throughout the book we will explain some of the core concepts behind ReactiveX, but we will not get into the details of how exactly the mechanism is implemented. We will however expose some key concepts that might be useful for your job, but we are not aiming at making you rewrite the library for your language of choice.

but

What if you think that ReactiveX is just syntactic sugar and that conventional concurrent programming is still the best way to develop?
We are strongly convinced that ReactiveX is not only a powerful set of tools, but also that its design will help you create very clear, simple and testable applications which will be much more expressive than with other techniques.
The concepts conveyed by ReactiveX and functional reactive programming will be of valuable knowledge, and they will make you a better programmer regardless of the fact that you will use them or not. They will teach you, if none at all, a different way to look at problems and a different paradigm for solving them.
Certainly, the decision is yours to make: give the book a chance or give it a miss. The comforting thought is this is not a life or death decision. It’s ok to change your mind, anytime.
What do you need to know before we start?

We assume that you have been developing software in any of the popular languages such as Java or C#. If you can satisfy yourself with this quick checklist, then you should be able to follow along comfortably.

You need to ...

- be familiar with object orientation concepts like classes, objects and inheritance.
- understand what an anonymous inner class is in Java or what a lambda function is in C#
- be able to read Java or C# code. We will not go into anything too specific for a given framework, but a basic understanding of collection classes in your language of choice will help you out
- have a basic understanding of what a thread is and what it does
- have some working experience on real life scenarios that will involve database access and remote REST API calls
- if you are interested in the mobile part of this book have a basic understanding of the Android SDK and how an Android application is built

You don’t need to ...

- be an expert in object orientation
- be a Java or C# master
- be a concurrency programming guru, besides knowing what might be the most common pitfalls
- know anything about ReactiveX or functional reactive programming
What programming language will we use?

As ReactiveX is a universal concept we will try to convey such aspect through the code. Due to this fact we will try to use pseudocode whenever possible. Such code should be easy to understand by anybody with a bit of experience on a modern programming language.

We thought a lot about the language to use for the book and eventually come to the conclusion that the concepts expressed by ReactiveX are so universal that there isn’t actually a single language in which these concepts are expressed, but many. A look at the Github section for ReactiveX will show you that there are porting in the majority of programming languages available, and for many different platforms. In order to truly express such universality we decided to show you the concepts rather than how to use ReactiveX on a specific language.

Such pseudocode will evolve during the book and become more and more detailed as you will increase your understanding of ReactiveX. Those already familiar with C# or Java, however, should not have any problem as we took inspiration from these two languages.

**WARNING** This is not a Java or C# language book but you might learn some of both as a side effect of learning functional reactive programming.
What this book expects of you

To get the most out of this book, you may need to make an adjustment to the way most software development books are read. Don’t expect to read from cover to cover, like a bedside read. Instead, keep the book at your desk, working with it next to your keyboard and a few sheets of blank paper to write on. Shift perspective from being a receiver of thoughts to being an active participant.

Some tips for learning with this book

1. Create a space to learn.
2. Do the exercises.
3. Get up and move around.
4. Don’t rush.

1. Create a space to learn.
   - Keep paper nearby and a few pencils or pens of different color.
   - Sharpies or flipchart markers are good too. We want to create a space that radiates information, not a dull, sterile place.

2. Do the exercises.
   - Make a commitment to do every exercise. Feel the code as you type it, letting your subconscious mind absorb what you do physically. Resist the temptation to cut ‘n paste code.

3. Get up and move around.
   - Write code by hand on a vertical surface then stand back and look at your code from a distance. Apart from the physical movement, we are changing perspectives too.

4. Don’t rush.
   - Work at a pace that is comfortable. It’s ok to not have a consistently steady pace too. Sometimes we run, and other times we crawl. Sometimes we do nothing. Rest is important.
What to expect from this book

We want to have a fair chance of learning ReactiveX. We don’t want to get bogged down with theory and abstract examples. We also want to enjoy learning functional reactive programming. We should not feel frustrated, confused and irritated every step of the way. We want to provide as many real life examples and solutions as possible, so that the concepts learned will not just be a nice line on a resume, but part of a toolset to use during every day tasks.

At the risk of being flamed

We will deliberately err on the side of practical usage as opposed to theoretical correctness. We will explain concepts and techniques in a way that may incite functional reactive programming flame wars. This is a deliberate learning strategy to keep us moving forward as opposed to stalling at one theoretical concept. There will be a time that warrants theoretical perfection. For now, we shall focus a practical appreciation.
Let’s start!

ReactiveX (or Rx for short) is in its essence a library that has been successfully translated into different languages. What makes it truly unique, however, is its functional approach towards something that should be well known in computer science: the observable pattern.

Such take in what is apparently a simple matter opened an entire world of possibilities for developing concurrent applications in a much simpler and powerful way than ever before.

Throughout the following chapters you will learn how ReactiveX can be used in real life scenarios to help you improve not only the way you will code, but also the way you will think about coding and interacting with the data.

Be warned, however

Rx is one nice solution for many problems you might have in everyday tasks that will require concurrency.

As it is for many libraries and toolsets, however, it is not a one size fits all solution. Some complex tasks can be solved quite elegantly by applying Rx (this is the main reason we love it, by the way), but some others might simply not be suitable.

Part of this book is about telling you under what circumstances it might be ideal to apply Rx for solving a problem, and when it might be overkill instead.

Rx can, by its own nature, play well along other technologies that you might already be familiar with, such as promises in JavaScript, Tasks in C# or Futures in Java. In a sense it is a complement to such technologies, not a competitor.

Rx has its own concurrency abstraction, called Schedulers, that account for all your multi threading needs and will provide a way to interoperate smoothly with other concurrency paradigms you might already use or know.
Do not be scared

If the words “functional” or “observable” mean nothing to you, there is no need to worry. You will progressively get acquainted with these concepts throughout the rest of this book. Many of the details of Rx will unfold smoothly from chapter to chapter in progressive detail, starting from the key concepts up to some of the inner workings. You are, after all, learning a new way of expressing yourself in code.

This is just the beginning...

Do not expect to grok Rx in just one chapter, though. This introduction will help you understand some of the key concepts beneath the library and some of the reasons why it has been designed this way. After this chapter you will have a good understanding of the theory beneath Rx and its overall philosophy. These will come in handy on the subsequent chapters where we will tackle more practical scenarios. First of all we will need to know why Rx has been built and exactly what kind of problems it tries to solve.
Owning a workshop

Imagine owning a workshop that builds wooden lamps. You buy your wood from suppliers, carve it, paint it, put all the wiring and switches then you package the product and deliver it to a store for selling.

Being on your own

You are in charge of everything from start to finish, so it makes sense to build one lamp at a time. You start with the spare parts and move them along the different tasks needed for creating one single lamp, then you pick up another set of spare parts and start over again.

The equivalent in software

In software this is represented by a single class running on one thread, who is in charge of managing all the data needed for a specific work.

As with the workshop, such thread is responsible for reading, checking, transforming and presenting the data.
Owning a workshop

Some or all of these tasks might be delegated to different classes in the same way you might have different specialized desks in your workshop for specific tasks.

In the end, however, your main class will pass the data to these classes, get it back and pass it to the next one, more or less like you will move your lamp across different desks.

Business increases

Sooner or later you will realize that such way of working is not sustainable anymore, however, as you cannot keep up with the increasing demand. You will need to hire someone that can help you, and rethink the way you create your artifacts.

The same can be said for software development. A simple program might work well for a few users, but when the demand increases, the application will become slow and unresponsive. The higher the number of concurrent users the slower will be the application. You will need to rethink it with a more scalable approach in mind.
Growing your business

On an assembly line work is compartmentalized, so there are no interferences.

The job every person has to do is just one task out of many, so training people is much simpler. New models can be built by using already existing parts of the line for common tasks, and when the demand gets higher, you can add more people doing the same job.

What about software development?

Instead of having one big monolithic class doing everything, we can split the work onto different specialized classes, that will each handle just a tiny bit of the work. We can at that point exploit modern multi-core architectures, by running these classes on different threads, so that they can execute their tasks in parallel.
The power of a conveyor belt

As in a real life assembly line, we will need the equivalent of a conveyor belt, that will move our data across the different handlers, which will be part of our infrastructure.

In its essence, reactive programming can be defined like this: a way to design your programs as highly effective assembly lines. With this paradigm, your data moves across your algorithm, not the other way around, exactly like your lamp moves across different workers on the conveyor belt. In order to achieve such goal you will need small pieces of code, each performing a simple operation on your data. The sum of all the tiny operations you will apply on your data will define the outcome of your system, in the same way the sum of all the tiny operations you will apply on the lamp will define the final product.

Do not despair, however, if you think this might be too much of a change in your programming habits. Chances are you are already using some of these techniques. In order to prove it we will start by talking about a technique that will help us understand the power of reactive programming: iterations and higher order functions.
Handling big sequences

The majority of modern programming languages already have the notion of an iterator, therefore you might already be familiar with the concept, but for those not so familiar, let’s provide some insight on what it’s used for and why. Imagine that you have to read data from some very big file, such as a huge log file, and that you have to extract just certain lines (maybe just the errors).

A log file is just a long sequence of lines of text, hence we can scan them one line at a time without having to keep all of them in memory.

You just need to keep track of the one you are currently reading.

Moving one line at a time

You cannot possibly held all the lines in memory, as they might be too many and could not fit the amount of physical memory you have. A log file is a long sequence of lines of text, though, which means that you can read just one line at a time and move forward on the file until you finish your extraction. Every time you will have just one line in memory, not the whole file, regardless of the size of the original file.
What do you need to iterate?

An iterator can be defined as a mechanism that allows you to move over an arbitrary sequence of data (such as the lines in a log file), while at the same time keeping track of the current element in that sequence. You can think of it as the bouncing ball in a sing-along that will show you the word to sing next. No matter how complex or long the song is, as long as you remember the melody you can follow the song without having to learn it by heart.

What defines an iterator?

Every language has its own nuances on the implementation of an iterator, but all of them will need three things at least: the current element in the sequence (the one where the ball is), a way to move to the next one, and a way to tell you that there is no more data (the song has ended).

In order to iterate over a sequence we just need three things: the current item, a way to move to the next one and a way to check if there are more items available.

SPEAKING OF DIFFERENCES In C# an iterator is called IEnumerator, whereas in Java is called Iterator. Whilst the C# implementation separates the current element from the move operation (with a Current property and a MoveNext() method), in Java such operations are combined together with a next() method that will also return the last element read.

THINK ABOUT IT! A database cursor is pretty much an iterator over a subset of rows, as it allows you to scan all the elements of a very big list. Can you think of other examples that can be reconducted to an iterator?
Anatomy of `foreach`

If you are using a language developed in the last 20 years, chances are you already used the `foreach` statement in one way or another. What you might not be aware is how the `foreach` works on your language of choice.

Despite different implementations, the concept is quite simple: the `foreach` cycles through all the elements of a collection starting from the first up to the last. At this point of the reading, it might not surprise you to know that the `foreach` statement is most likely using an iterator in order to do this. Let’s examine in detail what a `foreach` does when we request it to scan a sequence of elements.

First of all, the `foreach` statement will need to know that an object can be iterated, so it will check if it can provide an iterator to use. An iterable is any object that can provide an iterator over a sequence of elements. On every request it will provide a new iterator to the requestor, that will keep track of its own current position on the sequence. Think of an iterable as a factory that builds iterators on request.

```
foreach asks the iterable for an iterator
```

1. Request

```
foreach
```

```
iterable
```

2. Create

```
iterable creates a new iterator and gives it back to the foreach
```

```
iterator
```

3. Returns

```
foreach uses the iterator for scanning the sequence of elements
```

The three main phases used by the `foreach` statement are request, create and return.
Scanning the elements

Once the `foreach` has an `iterator`, it can use it for scanning the elements, by requesting to the `iterator` the current element, perform whatever operation has been instructed to, then move to the next element and repeat the whole process until the `iterator` will tell the `foreach` that there are no more elements in the sequence.

Sugar coating

By the diagram it should be clear that the `foreach` is syntactic sugar over the code that is needed to scan an `iterator`. Whenever we use a `foreach` we don’t have to write such boilerplate code, because it will be written for us by the compiler. We can just write our specific block of code that will be applied to every element in the sequence. It doesn’t matter whether we are scanning an in memory list, a database table or the lines of a file, the `foreach` won’t care and will just execute our code.

**SPEAKING OF DIFFERENCES** While in Java the `foreach` statement looks for an actual `Iterator`, in C# anything that has a `bool MoveNext()` method and a `<any> Current { get; }` property can act as an `IEnumerator`. The reason lies beneath the fact that in C# 1.0 there weren’t generics, and the compiler was meant to avoid the cost of boxing and unboxing an item while scanning it, if it were over a value type.
Time for a quick recap

There are many things we learned so far, so let’s take some time and recollect all our findings so far:

- **Iterators** are a way to scan sequences of data without having to keep the whole sequence in memory.

- **Iterables** provide a way to iterate a sequence by creating iterators on every request.

- Every **iterator** is responsible of keeping track of its current position, so that different iterators on the same iterable will be independent.

- **foreach** is just a way to decouple the iteration logic from the specific logic we will write to handle our data.

With all these concepts in mind, we can now move forward and start thinking about how to exploit the benefits an iterator has over a list more effectively.
Playing smarter?

As working with lists is an unavoidable task in software development, chances are that over time you developed a series of utilities functions that will help you avoid writing the same code repeatedly.

A practical example

As an example, let's think of an expense report. Every expense needs to fall into a specific category such as lodging, meals and other. Each of these categories will need its own subtotal on the report. If you had to write the software chances are that your utilities library will include a sum function that given a list of expenses will return the total, and a forCategory function that given a list of expenses and a category ID will return back just those that belong to that specific category.

<table>
<thead>
<tr>
<th>Date</th>
<th>Description of Expense</th>
<th>Airfare</th>
<th>Lodging</th>
<th>Transportation</th>
<th>Meals &amp; Tips</th>
<th>Miscellaneous</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/03/2015</td>
<td>Lunch with client</td>
<td>$24.30</td>
<td>$24.30</td>
<td>$36.97</td>
<td>$155.00</td>
<td>$4,55</td>
<td>$0.00</td>
</tr>
<tr>
<td>12/03/2015</td>
<td>Dinner</td>
<td>$36.97</td>
<td>$36.97</td>
<td>$24.30</td>
<td>$24.30</td>
<td>$155.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>12/03/2015</td>
<td>Hotel Room</td>
<td>$155.00</td>
<td>$155.00</td>
<td>$4,55</td>
<td>$24.30</td>
<td>$155.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>13/03/2015</td>
<td>Coffee</td>
<td>$4,55</td>
<td>$4,55</td>
<td>$24.30</td>
<td>$155.00</td>
<td>$4,55</td>
<td>$0.00</td>
</tr>
<tr>
<td>13/03/2015</td>
<td>Lunch with client</td>
<td>$24.30</td>
<td>$24.30</td>
<td>$24.30</td>
<td>$4,55</td>
<td>$155.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>13/03/2015</td>
<td>Hotel Room</td>
<td>$155.00</td>
<td>$155.00</td>
<td>$4,55</td>
<td>$24.30</td>
<td>$155.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>14/03/2015</td>
<td>Coffee</td>
<td>$4,55</td>
<td>$4,55</td>
<td>$24.30</td>
<td>$155.00</td>
<td>$4,55</td>
<td>$0.00</td>
</tr>
<tr>
<td>14/03/2015</td>
<td>Dinner</td>
<td>$36.97</td>
<td>$36.97</td>
<td>$131.64</td>
<td>$441.64</td>
<td>$36.97</td>
<td>$0.00</td>
</tr>
</tbody>
</table>

Avoiding scans

Chances are that your library will consist of different methods using lots of foreach statements. After all you will have to scan the whole lines on the expense report in order to find what you want or to sum them. You will have to scan the lines on the report over and over multiple times, but as the elements are in memory this has never been a big issue. Now that we have a better understanding of how iterators and foreach work, let's see if we can refactor our methods to leverage this knowledge.
Iterators are smarter!

Consider the `forCategory` function we discussed earlier. It will return the list of expenses that belongs to a specific category out of an expense report. In pseudo code it can be defined as follows

```plaintext
foreach expense in report {
    if the expense category is the given category
        add the expense to the category list
}
return the category list
```

In essence we have a method that, given a list, will return another list of elements filtered. The signature of such method is most likely like this

```plaintext
list<of expenses> forCategory(list<of expenses>, category)
```

We just learned, however, that `foreach` uses iterators for moving across a sequence of elements. Let’s forget for a moment that it will require an `iterable` in order to request the correct iterator, what about changing the signature of our `forCategory` method so that it will better reflect such thing?

```plaintext
iterator<of expenses> forCategory(iterator<of expenses>,
                                       category)
```

Change in perspective

This new signature, whilst being very similar to our original one, is starting to show us a change in perspective: we are not returning a list anymore. We are returning a way to scan a sequence of expenses given another way to scan a sequence of expenses. We changed the focus from the what (a list of data) to the how (how to scan such sequence). Let’s see now how we can rewrite the code in our method provided this new signature.
Generalizing iterations

Before delving into the code of our method let’s take a look again at how the `foreach` uses an `iterator`

![Diagram of foreach flowchart]

The key of the flowchart is in the “move to the next element” part. Let’s take our previous pseudo code and replace the `foreach`, that will move across all the elements, with what we would do with an `iterator` by moving one element at a time.

```cpp
while there are still expenses in our report {
    if the expense category is the given category
        set the given category as the current element
    else
        move to the next expense in our report
}
```

This code can be put in the method that will advance the current element of our `iterator`. At first it will check if there is a current element in the original `iterator` for the expenses, then it will check if such current element belongs to the category we need to show. If this is the case we will set the current element of our own `iterator` to this expense. If this is not the case, we will need to move to the next expense and perform the check again up until there are no more elements in the original expense sequence or we met a new one.

We just changed the whole perspective of our code. Once again, our method will not return a sequence anymore, it will return a **way to scan a sequence** given a source sequence to scan and a **criteria** to apply on every element of the original sequence.
Passing a function as an argument

When writing a library of utility methods (or better said in our case, of iterators) one important aspect is to have them generic enough that they can be reused in other projects. The `forCategory` method serves this purpose for the expense report, but is limited to filter expenses. What if instead of categories we will need to filter by payment method? We can clearly write another iterator doing just that, but it will be very close to what we already have. We will need a way to express the same concept in a more generic way.

Generalizing iterations, again

Let’s take a look at our `forCategory` function, but rewritten so that our pseudo code could will handle any item for any condition

```java
while there are still items in our iterable {
    if the item satisfies a given condition
        set the item as the current item
    else
        move to the next item in our iterable
}
```

Code written this way can check an item against any condition. What this means is that we will need to pass this condition as an argument to our function. Thing is, this argument is a function itself; most specifically whatever function that given an item returns a boolean value could be used as an argument to our former `forCategory` method.

```
iterator< of items > filter(iterator< of items >, function condition)
```

It is interesting to notice that the function we are passing to our `filter` method knows nothing about how or when the iterator will be used. What this means is that we just need to write some very simple functions in our code, one for every criteria we will need. Such functions can then be either used on a single item or on a sequence.
Combining functions

We have defined a generic \texttt{filter} function, whose work is to iterate over a sequence and apply whatever condition function we are passing as argument.

Now that we have defined our generic \texttt{filter} function we can use it to retrieve any item from our list based on any condition we want. We might for example have a function that will return \texttt{true} if an expense belongs to the lodging category:

\begin{verbatim}
boolean isLodging(expense)
\end{verbatim}

We can also have other business requirements that request to extract all the expenses being paid cash, because they will need to go on a different approval workflow inside the expense system. As we can write any function that given an expense returns a \texttt{boolean} we can also write such function:

\begin{verbatim}
boolean isCash(expense)
\end{verbatim}

Now, what if we want to extract just the expenses belonging to the lodging category that have been paid in cash? We can obviously create a new function that will combine the two:

\begin{verbatim}
boolean isLodgingAndCash(expense) {
    return isLodging(expense) and isCash(expense)
}
\end{verbatim}
Composing through iterators

There is nothing wrong in creating a new function that combines two other functions, we do this all the time, but as we will need that in the context of an iterator there is a better way for combining the two. We can actually combine them by composing two distinct iterators together.

\[
\text{filter}(\text{expenses, isLodging}) \\
\text{the result of the filter is an iterator} \\
\text{that we can pass to the second filter} \\
\text{filter}(..., \text{isCash})
\]

The benefit of such mechanism with respect to returning a list is that we are still evaluating one element at a time. Chaining the filters in this way will just chain the functions we are passing as an argument.

\[
\text{filter}(\text{filter}(\text{expenses, isLodging}), \text{isCash})
\]

Iterators in this sense provide a way to compose functions together that will act on sequence of elements.

Combining operators

In the beginning of this chapter we talked about having a conveyor belt that will move our data across different elements. An iterator with respect to this is a simple conveyor belt that moves our items from one function to another. It will factor out the logic for iterating over a sequence from the logic our application will need to apply. We don’t have to care about how the function is applied, as the iterator will take care of this for us. We saw before that the filter function is technically an operator, hence we can say that we can combine different operators together in order to create new ones. This powerful mechanism will be at the base of ReactiveX. Before proceeding we will need to know what an observable is, though.
Sharpen your pencil!

So far we learned a lot about what an iterator is and how we can chain them together in order to build complex workflows starting from very simple building blocks.

So far we saw that we can filter an iterator by using a simple function that will return a boolean value. The possibilities of the mechanism are however endless.

Before we go through the next paragraphs for a theoretical introduction over what an observable is and how it relates to an iterable let’s take some time to put into practice some of the things we learned so far.

- can you think of an operator that will convert from one type into another? What will be the signature of the function we will need to pass? How will it work? Try to write the pseudo code for such operator

- what about an operator that will return an aggregation of all the items, such as the sum we discussed? Can you think of a generic way to write that?

- can you think of a way in your language of choice for removing the clutter of nesting operators together when composing them? so that instead of

        total = sum(filter(filter(expenses, isLodging), isCash))

    you will have something like

        total = expenses.filter(isLodging).filter(isCash).sum()
Solution to our exercises

Converting a type into another is basically mapping from one item to another of a different kind, so we can call our function map. As for the function any function that takes an item as a parameter and returns a different kind of item as a result can be used. The definition is therefore as follows:

```
iterator<of items> map(iterator<of items>,
                       function mapping)
```

the algorithm for this is even simpler than filter, as we will just have to process all the items

```
while there are still items in our iterable {
    set mapping(item) as the current item
    call the function we passed as an argument
}
```

Sum has a similar way of working, but it needs a function that given any item will return a number that we can use to accumulate

```
total = 0
while there are still items in our iterable {
    total = total + extract(item)
}
```
What is an observable?

In order to better define what an observable is let’s start with an example. You work for a startup company that provides cab services throughout the town. Every ride is charged by the mile or by the time spent when the cab is idle.

You need to compute and display the running amount for the fare. In order to compute the fare properly you will clearly need to know the speed of the vehicle at any given moment. Lucky for you somebody already wrote a class that interfaces with all the car internal sensors, therefore you have a way to access such information.

The problem is that you are not in control of anything. A car moves, accelerates and decelerates at the driver’s will, not yours. It might be idle for minutes or hours during traffic jams, or it can run on a highway for many hours at the same speed. Asking every moment the current speed to the sensor from this point of view makes little sense.

A better way to handle this would be to have the sensor tell you any change in speed it might have. This will make you an observer of speed values. On the other hand the speed sensor will be an observable of values.

A cab has two different fare systems based on the speed of the car and the fact that it is idling or not in the traffic.

In real life the taxi-meter is observing the speed of the car in order to compute the fare, hence we can think of the odometer as an object exposing speed values to be observed. In other words an odometer is an observable of speed values.
## Reacting to changes

Being an observer will mean that whenever a new speed is available the system will tell you so. As you will have to do something based on whatever speed you will receive, in a sense you will need to react to that change accordingly. In your case whenever there is a speed change, your reaction to such change will be to calculate the new fare and display that to the end user.

In order to calculate the fare you will have perform a series of operations on every single value you receive. Such series of operations will change based on the speed value you will get. The difference with a conventional system is that the data is driving your chain of reactions, not the other way around. What we are building here is the equivalent of an assembly line that will transform speed values into a fare values, and we need to define the single tasks in our virtual conveyor belt that will need to react to the data passed to them at a given moment.

### PARADIGM SHIFT

We cannot stress enough that reactive programming will shift the focus on the data and the changes that will occur to it over time, rather than on classes and the interaction along them. This might prove to be hard to understand at the beginning, because it might be quite far from what you’ve been used to do. Do not despair, however, as during the following chapters there will be plenty of other examples to help you in this journey.
Observing changes

As the speed in this case will be a series of values over time that will get notified to you let’s start by representing them like that.

If we write down the speed values we get over time we will end up with a sequence of values, pretty much like what we had on our previous paragraphs. An excerpt of such values might be like the following:

When and how to observe?

When there is a red light or any stop the fare system will need to compute the total number of seconds spent while idle, and apply a specific fare. On the other hand when the cab is moving (such as in the start of the ride) it needs to compute the distance covered, which is the fraction of miles per hour covered in that period of time.

An observer, however, is not bound to observe everything every time. In our case, for example, the cab might be moving without a passenger for whatever reason, and during that time there will be no use of the fare system.

The observer will need to ask for the observable data only when needed and not every time. In order to do this the observer will require a subscription mechanism to tell the observable that the data is needed or that is not needed anymore.

OBSERVABLE DONE RIGHT Those familiar with design patterns will recognize that what we described so far is just the standard observable pattern you might have read in other books. What makes ReactiveX different is the fact that observables can be chained together in a way similar to the one we used for iterables. We will see that the subscription will play a significant role in this game.
Introducing the subscription

The act of subscribing to an observable will provide us with a subscription. This is the way for the observable to keep track of who is observing him. This is what we will receive whenever a user enters into our cab, which will trigger the fare system.

As soon as the ride has finished, the observer will not need to receive any other data, therefore it would be safe for him to unsubscribe from the observable. In our case this is done by means of the subscription. What this means is that you will not tell the observable to unsubscribe, but you will just tell the subscription to do so.

Speaking of differences In C# a subscription is actually an object implementing the IDisposable interface. The reason for this is that IDisposable was already used by the .Net framework for resource management and had a similar meaning, therefore the designers or Rx leveraged the interface in order to provide a cleaner, more idiomatic design.
Can we “iterate” an observable?

Now that we have our observer and observables in place it is time to implement our fare system.

The first implementation that comes to mind is straightforward: whenever we receive a speed value, we will compute the partial fare with a specific formula, then add it to the running fare. We have two specific algorithm in place, so we will have to find a way to tell them apart based on the speed of the cab.

The running fare of our cab will be the running total of all the partial fares that we will compute for every speed value we will receive from the observable.

Push versus pull sequences

In our previous paragraphs we talked about iterators and how they can be used to handle sequences of data one element at a time, allowing us to focus just on the single function to apply on every element rather than on the scan logic. This case is not different. There is still a sequence, the only thing is that we are not in control of this one. In this case we don’t pull the data, because we are not requesting it, we will be pushed new values every time by the observable. What this means is that we can try to apply the same concepts we learned before to this new concept, after all we are still handling one value at a time here too.
Anatomy of an observer

Now that we have a better understanding of the relationship between an observer and an observable we can try to infer what the observer’s contract could be. We know that observer and observable are similar to iterable and iterators, because they act over sequences of data in a similar way, but can their contract be similar?

An observer is being pushed some data whenever a change occurs, where an iterator pulls the data whenever it needs that. In a sense, there is some sort of symmetry among the two that we can exploit in order to define the other characteristics an observer might have.

<table>
<thead>
<tr>
<th>Iterator</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keeps track of the current element in a sequence</td>
<td>Receives the next element in a sequence</td>
</tr>
<tr>
<td>Notifies when a sequence has no more elements</td>
<td>Should receive a notification that there will be no more elements</td>
</tr>
<tr>
<td>Might raise an exception in case something goes wrong while iterating over a sequence</td>
<td>Should receive an error when there has been an error while sending an element notification</td>
</tr>
</tbody>
</table>

This symmetry between observer and iterator is quite interesting, because it allows us to leverage our knowledge about how to compose iterators together in order to form new iterators. As they are so similar we can definitely compose observers together in order to form new observers. Such composition will work in a similar way, but could be applied to notifications rather than sequences. Let’s try to apply the same example we saw for our expense list to the fare calculator.

ABOUT DUALITY Sometimes you might hear that an observable is the dual of an iterable and vice versa. Besides the mathematical implications that such an affirmation will convey, the point is that the two can be interchanged and mixed together, because they will in a sense share the same roots. ReactiveX provides operators that will convert from an iterable to an observable and vice versa. Such operators will be shown in greater detail in the following chapters.
Composing through observable

The fare type we are using can be one of two different types, depending on the actual speed of the cab. This is similar to the category on our expense report. In the expense report we needed to filter different categories in order to compute their corresponding subtotal. In our fare system we need to filter different fare types in order to compute their corresponding subtotal. Can we create a filter function that will work for the observable world?

As observables are just push sequences whose values can be handled one at a time, we can create a filtering observable that will return the speed values belonging to one specific fare type. We will then have to subscribe to the observable that will subscribe to the original sequence and filter it out for us. The signature of this filter is basically the same we used for iterables, just for observables.

```javascript
observable<of items> filter(observable<of items>, function condition)
```

We are not interested to these values for the time being

Observables, being symmetric to iterables, can benefit from the same composition mechanism we saw earlier. This is great news as we will see in the following chapters. For the time being pat yourself on the back, as you are now ready to tackle the observable world from a more technical perspective.
Summary

- Reactive programming is a way to create concurrent applications that can operate over sequences of data over time, without having to worry about the intricacies of multi-threaded programming.

- Iterables and their corresponding iterators are a convenient way to scan large sequences of data one element at a time. They are used by the `foreach` statement to scan a sequence and apply a block of code over every item.

- An operator is any method that will accept one function as a parameter, or that will return a function as its result value.

- We can create operators that work over iterables and iterators. The `filter` operator, for instance, is a function that, given a condition, will return an `iterable` that when scanned will just yield the values satisfying the condition passed.

- We can combine different functions together by chaining operators. The result will be a sequence of values that when iterated will respect all the rules defined by the functions we used. For instance, we can apply two `filter` operators over a sequence instead of directly combining the conditions together in a new function.

- An observable and its corresponding observer is the dual of an iterable, in the sense that it operates over sequences of values received over time rather than by pulling data from a static sequence. Such duality will allow to write operators for the observables in a way similar to the one used for iterables.

- Operators written for the observable can be composed too.