RxJava for Android Developers

Timo Tuominen

MEAP

MANNING

MEAP Edition
Manning Early Access Program
RxJava for Android Developers
Version 10

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

welcome

Thank you for purchasing the MEAP for *RxJava for Android Developers*

Three years ago RxJava was at the beginning of what would end up transforming the entire Android landscape. Based on blog posts and presentations, a new way of creating Reactive Android applications was created. At this point in time the discussion is more on the details of whether it's Model-View-View Model, Model-View-Presenter or something else, but the trend itself is clearly towards View Models and Reactive Architectures. Even Google has finally come out with its own version at this year's I/O.

Instead of going through the endless documentation of RxJava from the technical point of view, this book explains in intuitive visual flows the motivation and thinking behind Reactive Applications on Android. The first part is about learning the basics of RxJava and cool asynchronous tricks we can do with it. The second part introduces the concept of a View Models and how it can be used to isolate the logic in the UI into a separate, testable, logic container. The third and final part goes over some more involved architectural patterns that allow us to build real life applications that are maintainable and easy to extend.

There will be many changes and refinements to how applications on Android are built, but this is where it started. We attempt to provide a solid starting point for understanding and applying Reactive Programming principles on the Android platform.

—Timo Tuominen

brief contents

introduction to reactive programming 1

In this chapter

- What to expect from this book
- How to use this book
- Why RxJava 2 for Android?
- Deep dive to RxJava 2 on Android

Perhaps you picked up this book because...

1 **Everyone's using RxJava and you have no idea why**

It's hard to name a large company that would do native Android and not use a reactive programming library such as RxJava 2. In this book we will focus on *why* it's hot and *what we can actually do with it*.

{ You may also have heard of Functional Reactive Programming (FRP), and it is indeed related to Rx. We will go through both concepts in this book. }

2 **You have used RxJava on Android and you want to learn more**

It's not uncommon these days to see snippets of RxJava code to solve a particular asynchronous problem. However, there is a whole world behind what sometimes looks like a simple utility.

{ The programming syntax used in Rx can seem like the entire point, but it is actually just a nice add-on. This is a book that teaches you how to think in Rx. }

3 **You have used RxJava and you hate it with a passion**

In the wrong hands RxJava can make traditional spaghetti code even worse. Just like with any power comes responsibility. We learn where to use RxJava and where not.

{ We will learn to design applications in sensible and extensible way. You can be reassured there is a way to maintain your Rx code. }

Whatever your reason, we want you to...

- learn through extensive illustrations and examples.
- understand a new way of seeing how applications work.
- figure out where to fit Rx in your day to day programming.

Don't read this book if...

You are very new to programming.

Rx is still quite a new paradigm, and it is not always a smooth sailing. Hopefully in the future this will change and everyone will start their programming journey with Rx.

or

"I just need to get it done."

The learning curve of reactive programming is a little steeper than usual. Cutting corners is not as easy as in the more traditional ways of writing applications. This is fundamentally double edged, but you will need a curious mind and a healthy dose of patience.

but

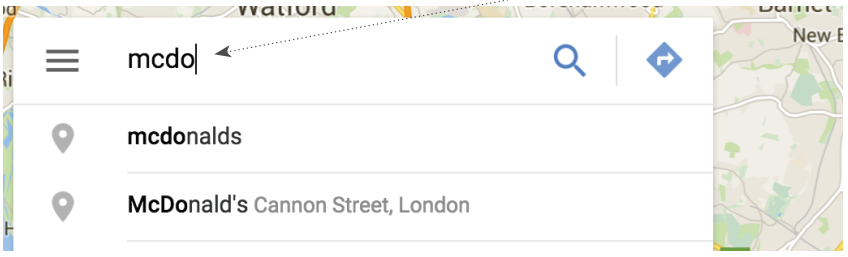
Continue reading if you want to learn how to properly make a delayed auto search field in five lines.

This is the example code we will learn to make in the first chapter. We don't expect you to be able follow just yet, but take it as a preview of how powerful RxJava can be.

```
RxTextView.textChanges(textInput)
  .filter(text -> text.length() >= 3)
  .debounce(150, TimeUnit.MILLISECONDS)
  .observeOn(AndroidSchedulers.mainThread())
  .subscribe(this::updateSearchResults);
```

We will spend the first chapter constructing this piece of code

The user writes values and 150 milliseconds after he writes the last letter we trigger a search



OOP, Rx, FP and FRP

RxJava is in the club of Reactive Programming libraries (http://reactivex.io/). They all start with "Rx", so sometimes they are together called Rx programming. To get an idea of where this sits let's take a recap of the popular paradigms.

OOP, Object Oriented Programming

The idea of OOP is that everything is an object, a thing, and can interact with other objects. Typically an object would encapsulate its state and allow outside actors modify it thought its member functions.

FP, Functional Programming

Functional programming is an old programming style which focuses on an almost mathematical precision in describing the program. It turns out that while the mental model of FP seems more complex than OOP, it seems to scale better when the complexity of the state needed increases.

FRP, Function Reactive Programming

The concept of FRP was introduced 1997, but it has become popular only in the recent years. It is a bit like an extension on top of FP, enabling an app to be built to “react” to input in a seamless way. How FRP does it is by declaring relationships between values.

Rx, Reactive Programming

Reactive Programming, or sometimes called Rx, is an umbrella term for all paradigms that use a data flow to construct the application. RxJava is our tool of choice to implement the data flow in practice. The previous term FRP can also be considered as Reactive Programming, but not always the other way around.

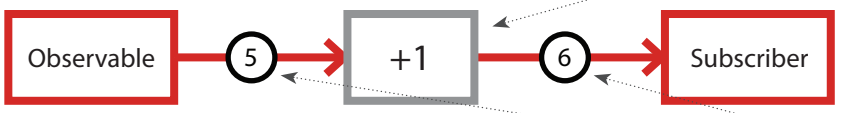
What about the Facebook React library?

React is the name of a Facebook UI library made on top of HTML5. While you can do Reactive Programming with the React library, it is not inherently “reactive” as we define it. It's really just a rendering engine.

Benefits of Rx

Rx, including RxJava, is not a typical technology that is made to replace another inferior technology. Rx more of a way to enable visual data flow thinking rather than a mechanical solution—and that is what we are going for in this book.

We will spend a lot of time to cover the basics, but at the core RxJava implements an publish - subscribe pattern. Observable (the publisher) sends a value and a subscriber consumes it. However, the trick is *we can modify the values on the way*.



The publisher and subscriber are decoupled and only connected with the operation

The data values travel from observables towards subscribers

Handling of asynchronous operations

Asynchronous programming is almost always listed as the first benefit of Rx, and it certainly is a major one. The tradiotional ways of programming make the handling of background operations a special case, which is often a tricky one.

In Rx, on the other, every operation is *potentially asynchronous*.

Asynchronous operations are not special cases but they are handled naturally as though nothing unusual happened. It is the synchronous execution that is the exception.

We could replace the addition of two in the previous picture with an arbitrary operation that can take as long as necessary and return a type that is suitable for our needs.

We will learn now to do to all of this in a bit!

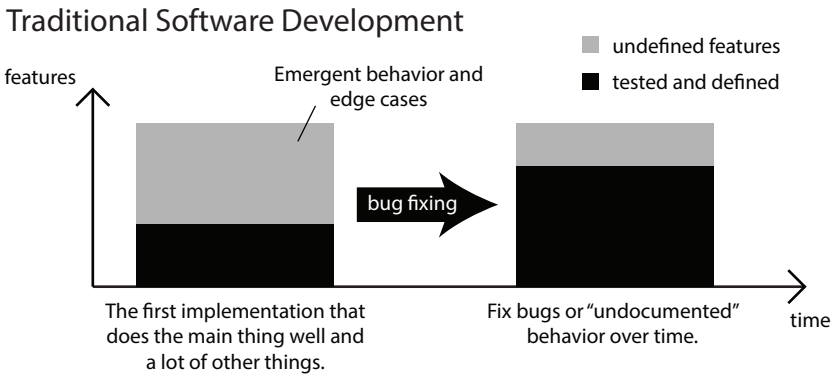
We are able to do this because the operations are decoupled—the next one does not know when the previous one is supposed to complete

The Subscriber has also been changed to accept a different type. Previously it was a number and now potentially a string.

The operation can be changed to be an asynchronous ones, for instance happening over the network.

What You Code is What You Get

In Rx you get what you asked for—and nothing else! The way Rx is written is inherently incremental. To understand what this means, we can have a look at how software development usually happens.



In short, we write an initial bunch of code that works most of the time on most people's machines. In this chart the black is what we really needed and the gray area are numerous corner cases and error conditions.

The gray code

We can have automated testing and quality assurance in place to mitigate the impact of the gray functionality, but it's still there. It's the part of code that works but we're not quite sure *how exactly*.

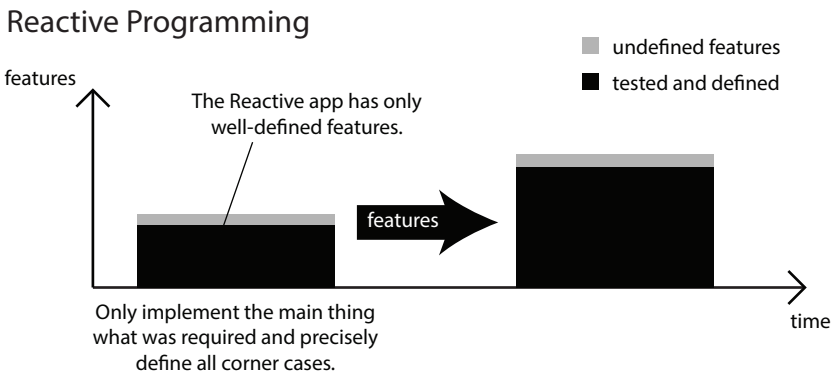
This is not because of bad or lazy developers, but because the tools are simply not that precise.

> But all features are defined in the code, no?
>
> In this context by features we mean functionality that can be fine-grained. Scenarios such as server failing to respond or the user clicking a button rapidly several times are seldom explicitly defined before a bug report comes in.
>
> In the long run in projects that require high quality standards typically a significant amount of time goes into fixing issues that are caused by potentially obscure edge cases.

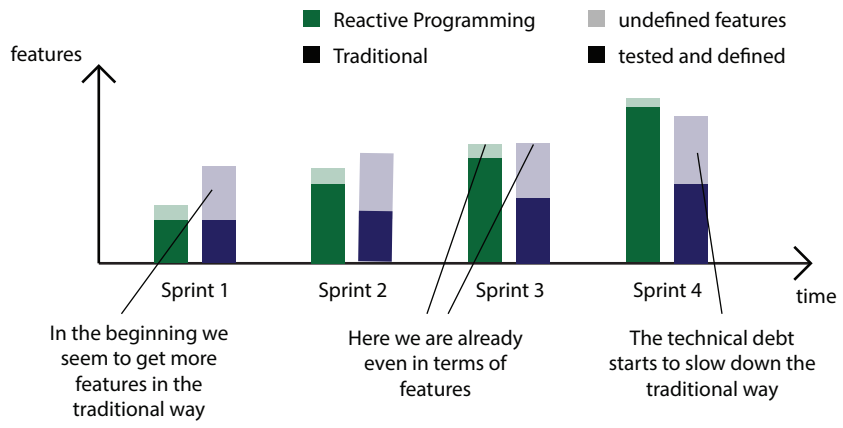
Reactive Programming Development

How this is different in Reactive Programming is we typically have less "free" features and we start having to ask questions like "what happens if the second server response does not match the first one" early on.

If you did not think of a scenario it will be obvious already when writing the code, *not when the user stumbles upon it*.



It seems that at first you get less bang for the buck, but over time you will see less impact from growing scale and technical debt. Reactive Programming is less impacted by scale and complexity than traditional ways. This is not a magic solution, though, but is simply *a result of us having to deal with the scale and complexity* instead of brushing it under the carpet.



The Reactive Landscape

In this book we will focus on Android and RxJava but that's not all there is to Rx programming. Here's a quick overview of what's out there.

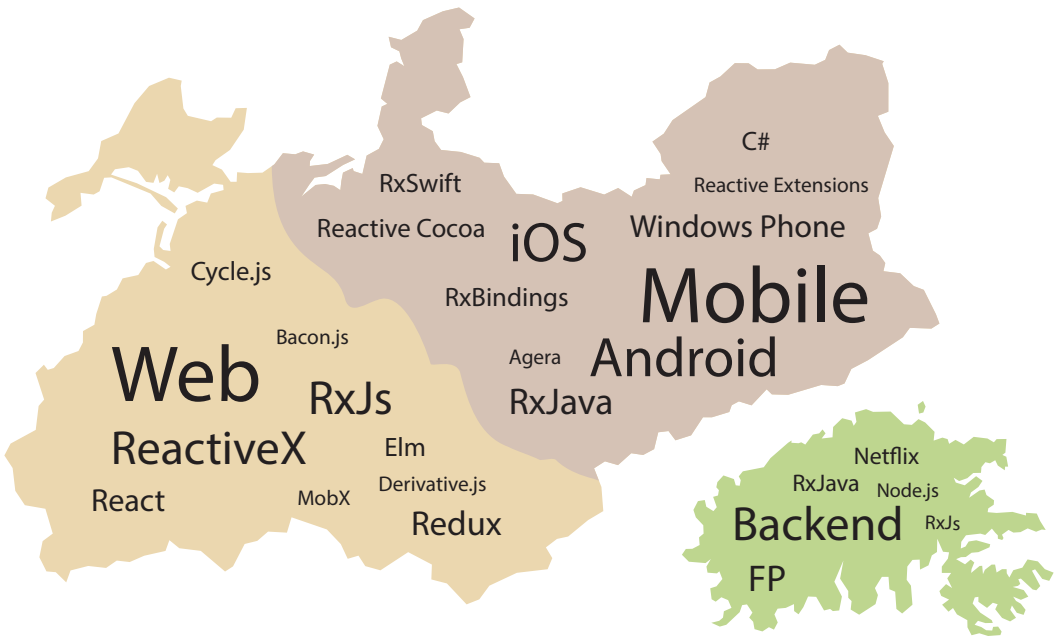
Mobile development

Rx started with Windows Phone, but Android and iOS have been catching up during the last years. The biggest problem used to be the lack of resources online, but the situation has much improved.

Android

RxJava is de-facto reactive library to use on Android, though some other options have started popping up. Google published an internal one called Agera and there are several others, such as Sodium.

On Android the challenge are the opinionated ways of constructing the apps from platform components, such as, Activities and Fragments, though working within this limitation has still yielded good results.



iOS

The way iOS apps are traditionally structured did not lend well to reactive programming, but nowadays there are well-known ways to go around it. Reactive Cocoa used to be the only serious reactive library on iOS, but its syntax is quite different from the more recent solutions. RxSwift, on the other hand, is more similar to RxJava and it is gaining popularity.

Windows Phone

Even if Windows Phone is becoming less used as a mobile platform, it gets a special mention as the best supported mobile platform when it comes to Rx. In fact Rx is a standard part of it. The Reactive Extensions of C# is considered to be the first implemention of the modern Rx.

Web and HTML5

JavaScript and the web has becomes one of the most innovative grounds for Reactive Programming. The biggest benefit of the web is that the browser is not very opinionated in how one should go about constructing a UI and therefore it is possible to create completely custom approaces.

We will not cover the whole scene of web here, since it would be difficult to be objective. There are many amazing frameworks and libraries to check out, each providing a different point of view to what Rx is and what it should be. (Or if it should be Rx at all, but that is another discussion).

The same principles covered in this book do apply to web too, though.

Backend systems and Rx

The most known backend system written partly with Reactive Programming is the Netflix streaming system. It seems, however, that most services can be made using the standard technologies and there has not been a high demand for Rx as such.

On the other hand, Functional Programming has become increasingly popular especially in heavy-duty systems. Languages such as Haskell and Go have found their way into the realities of many programmers.

While this book is more about UI programming and not the specialised functional languages, there are similarities in the way of thinking.

What do you need to know before we start?

With a suitable library Rx concepts can be applied to almost any platform and language. However, we need to pick one, and we pick Java as our language and Android as the platform. RxJava 2 is our Rx library (normally referred to as just RxJava). If you're curious about the reason we'll get to them in a bit.

This is not a book about learning Android development, though, so we will not go into too many details there. If you are completely new to Android we recommend you to see the appendix for a quick dive into Android development.

In terms of basic programming skills we do expect you to have some.

We expect you to ...

- have a basic understand of user interface development and the challenges it usually poses.
- be able to read Java code, including lambdas which were introduced in Java 8. (We have a small introduction if you're unfamiliar with them.)

You don't need to ...

- be an expert in reactive or object oriented programming patterns.
- have previous experience in functional programming (though it can be of use for understanding the concepts).
- know Git but it helps going through the example code online.

Java 8 Streams

While we will make extensive use of Java 8 lambdas, we do not touch other features that were introduced. In specific Java 8 Streams are not related to any streams we may or may not be talking about in this books.

About the Book

This book is divided into three parts. Part one is about getting a basic understanding of the RxJava library and what it can do. Here we dive into different roles it can have in an application on a lower level.

Part two is focused on Reactive Architectures—it is more about how we can organize our code to be robust an maintainable. Part two is the longest of the three, containing some core concepts when doing serious Reactive Programming.

The third, and last, part contains a few larger example projects that use the skills we have acquired. You can use them as a learning references for how Reactive Programming scales for real-life projects.

Online Code Examples

You can find most of the code featured in the book online at https://github.com/tehmou/Grokking-Reactive-User-Interfaces.

Coffee Breaks

Whenever you see a coffee cup it means it's time for a small exercise. Get you cup and see how you can apply what you have learned.

Most Coffee Breaks have the coding starting point defined online, as well as solutions.

To follow the chapter it is not compulsory you do the exercises, but we recommend you at least go through the given solutions.

What are the Grey Boxes?

These grey boxes usually answer questions you might already be thinking about. We use them to clarify misunderstandings as well to provide extra information that might be slightly off-topic.

RxJava 2 and Android

As mentoined, in this book we will use RxJava. It part of the "ReactiveX" family, which has a more or less standardised syntax. RxJava itself was originally a port of Reactive Extensions on Java, made by the development team at Netflix.

RxJava has found its way through the backend programming to UI applications build with Java—including Android. For the teaching purposes of this book we have chosen Java on Android for the examples. There are a few reasons for this:

Why Java?

- Java is a very common language that almost everyone has come across at some point in their developer life.
- Java is strongly and explicitly typed, which makes it easier to follow the data type conversions.
- Some purely functional languages are already very close to functional reactive programming and thus more suitable. However, they are still in the minority, especially in UI programming.

Why RxJava?

- Rx, or Reactive Extensions, has proven to be usable in production on a large scale. It is a safe pick for solving real-life problems.
- If you learn the Rx syntax on one language it is not difficult to apply your learnings on almost any other one. There is hardly a language or platform on which Rx would not have been ported.

RxJava 1 or 2?

In this book we are using RxJava version 2.x. There were a few major changes in the api. Most notably null values are no longer permitted, there are a few new Observable types and the way subscriptions are internally managed have changed. You should be able to follow the principles of the book even if you are working with RxJava 1 though.

Setting up the Android Environment

You do not need to have the development environment set up to follow the illustrations in this book, but it helps at least to check the implementation along the explanation. A lot of Rx is actually about the way we in the end translate the diagrams directly into code.

Android Studio

https://developer.android.com/studio/

Install the official IDE for Android is the Google Android Studio. It comes with the Android SDK, though it might ask to install additional parts.

Git

You can actually download the examples from the online repositories, though we encourage to use Git for managing your own coding as well. You can fork the examples, or just take a clone of an example repository.

Git is an industory-standard tool these day, so you can find plenty of material online if you are just learning to use it.

RxJava 2 Dependencies

Android Studio comes with everything set up except for the RxJava 2 library. All of our example codes have it included, but if you are starting a new project you can add it to the app/build.gradle file.

app/build.gradle

```
dependencies {
    ...

    // RxJava
    compile 'io.reactivex.rxjava2:rxjava:2.1.7'
    compile 'io.reactivex.rxjava2:rxandroid:2.0.1'

    // RxBinding wrappers for UI
    compile 'com.jakewharton.rxbinding2:rxbinding:2.0.0'
```

If this looks foreign to you just use the examples online!

Java 8 Lambdas

Even if you're familiar with Java you might still not be familiar with lambdas. They are an added feature that we can start using with a bit of configuration.

app/build.gradle

```
defaultConfig {
  ...
  compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
  }
```

This is the part you need to add to the configuration.

Do I need to do this configuration every time?

For new projects yes, though all our example projects already have this settings turned out.

What is a Lambda Function?

In short, a lambda is an unnamed inline function. If we take a sum function as an example we can see what this means.

```
int sum(int x, int y) {
  return x + y;
}
```

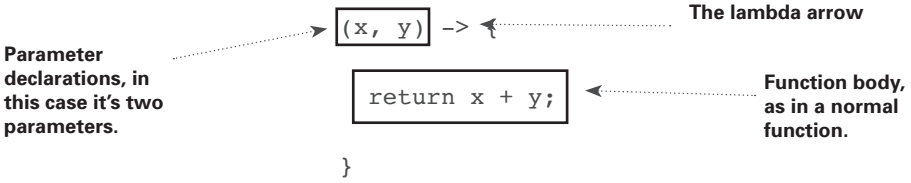
This is just a standard Java function (a method). If we write the same in a lambda form we can leave out the types altogether.

```
(x, y) -> {
  return x + y;
}
```

The types are *inferred* from what is given as the input and what is expected as the output.

The Anatomy of a Lambda Function

Just like a normal function, a lambda function declares the parameters it can take as well as the function body with a possible return value.



In case the lambda is just one return statement, it can be written even more concisely:

```
(x, y) -> x + y
```

This is *exactly the same* as the form we saw before, including the return value. However, if you want more than one statement, you will need to use a code block. *The above is a short notation for very simple lambdas.*

What's it Good for?

What's the point then? For instance, if we want to define an event handler function we can do it in a shorter way.

```
setOnClickListener(event -> { ... });
```

This saves us from the hassle of needing to define a separate named method just for the click.

We can also save function references into variables. The sum function we had is a function that takes two parameters and returns an integer. It can be saved as a BiFunction ("Bi" for "two").

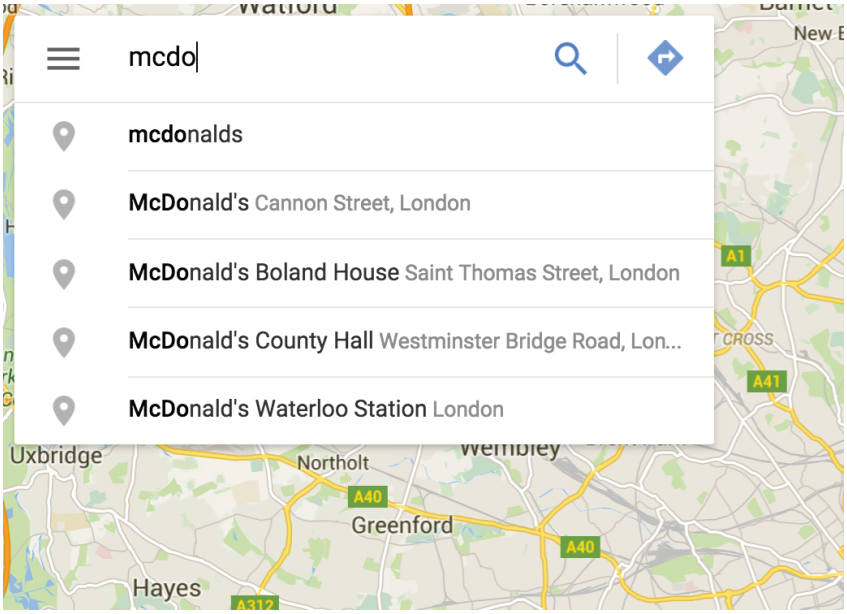
```
BiFunction<Integer, Integer, Integer> sum = (x, y) -> x + y;
```

First argument type · Second argument type · Return type · Lambda arguments · Lambda function body

After BiFunction come Function3 and Function4.

If this looks unfamiliar, don't worry. We will always go through the code to make sure it's all clear.

Deep dive into Rx: Live search



Screen shot of Google Maps web application doing a search (not our app).

To get a quick and dirty idea of what Rx is capable of let us have a look at an example: live search based on user input in a text field. You start typing in the text field and google helpfully shows you suggestions without needing to press a search button.

We want to trigger a search through a network API and show results based on the text user has written so far. However, we only want to do so when the user has written a minimum of 3 characters and has momentarily paused writing.

Conditions for triggering search:

1. **Ignore input shorter than 3 characters.**

2. **Only perform a search when the user has not been typing for 150 milliseconds.**

Project Set Up

We will not build a full maps client here, so we will use a simple project that has a dummy implementation. You can find the starting point of it in the online code examples.

A Quick Implementation

Our first task is to establish a basic connection between the editable text input and the list that renders the search results. If we assume we have an updateSearchResults function already in place the flow goes from text change to updated list in the UI.

If we first get started without RxJava, we can set a listener to the TextEdit component. On Android this is done with the `EditText.setOnChangeListener`. It allows to declare a function that is executed when ever the text changes (i. e. the user is typing). We will use or nifty lambdas here for brevity.

```
editText.setOnChangeListener(text -> {
  if (text.length() >= 3) {
    updateSearchResults(text);
  }
});
```

Everything in the middle is executed after each change in the EditText.

Here we have a handler that checks that the length of the input is more than 3 characters—if not, it just ignores the change event.

Filtering Text Changes Based on Time

The code in the event handler executes immediately, or in other words synchronously. However, our second condition of waiting for 150 milliseconds before executing the search is dependent on *time passed after it was called*. How do we deal with passing time? Threads?

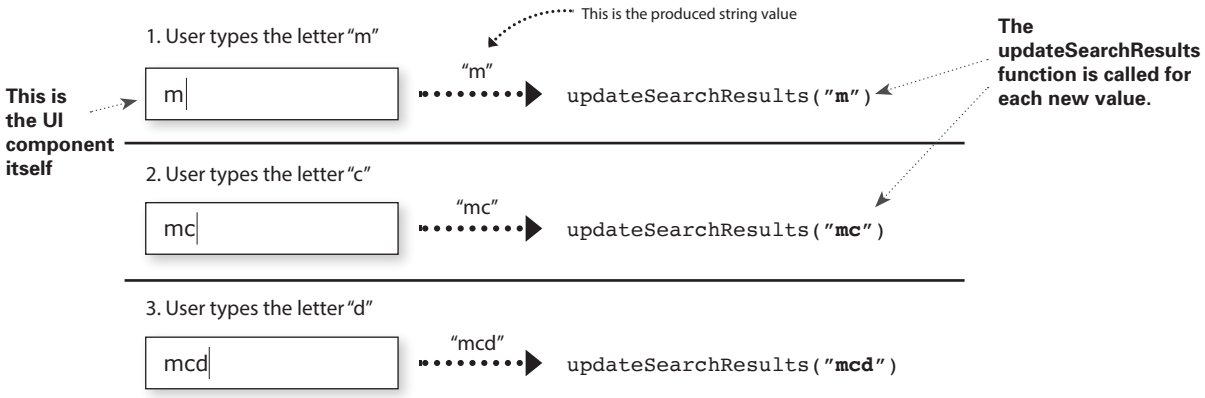
This is where things get traditionally tricky: to build such a system is no small feat with traditional Java. Fortunately handling time-based events is one of the technical problems Rx is good at.

Text input as a emitter of data

To get to the Rx implementation we will first will take a step back and reshape how we view the problem. We have two sides: the text input that *emits* text values and a function that we call with those values to make something happen.

> **How can data be "emitted"?**
>
> In this case when the text input changes it sends an event to all its listeners, informing of the state change. From the point of view of an individual listener, the text input just produced a new value of type String and we should do something with it.
>
> This may sounds a little strange, but it is the first step towards the thinking that enables us to use Rx.

The updateSearchResults function is the consumer of the data. It does something with it every time a new value arrives.

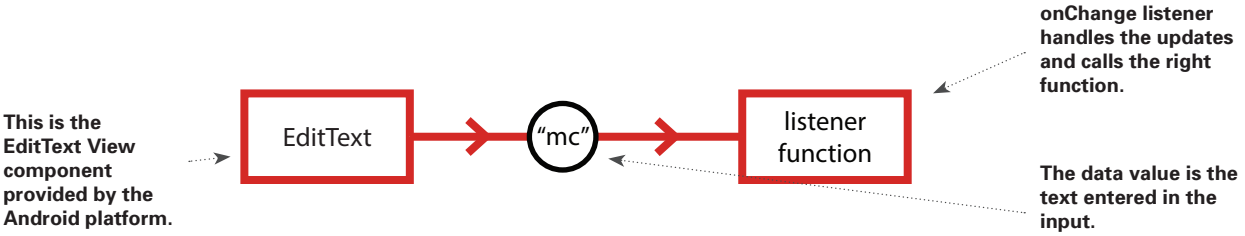


In the code we had before, the text input would give the new string value `text` to the event listener, which in turn would give it to the `updateSearchResults` function.

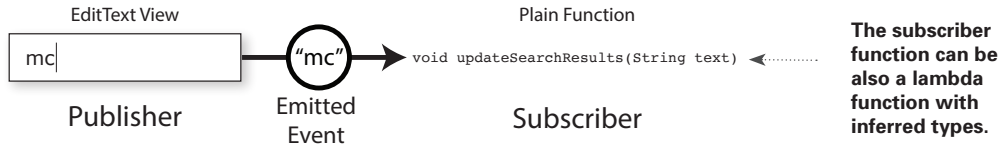
Since the user is not typing at infinite speed, these values would arrive at different times. This is how typing into a text input works—*whether you are using RxJava or not.*

The Publish-Subscribe pattern

As mentioned, at the core of RxJava is a pattern where we have event sources and functions that listen to them. An example of it is an EditText emitting (publishing) its new value whenever it changes. We will see next how to write this in RxJava way, but remember that in reality we will still always have an onChange listener attached to the EditText.



What this means in our Rx terminology is that the EditText *publishes* or *emits* and update and the listener functions are *subscribed* to it. This makes the listener functions *Subscribers*.



Notice that the code is still the same EditText and a change listener, we are simply interpreting it in a different way.

Should we emit the whole value or just the change?

If you think about it from the point of view of *change* event, could we not just emit the newest character and not the whole string?

We could indeed, but it would leave the Subscriber a big responsibility of aggregating the full string over time. In Reactive Programming we usually want to emit the *entire data value* to the Subscriber.

Text input as an Observable

Now we know that an EditText can be considered an emitter of string values over time. A producer like this in the Rx terminology is called an *Observable*—supposedly since the values it emits can be observed.

Converting UI View into Observables

We will now take into use the library RxBindings to create a proper Observable out of the EditText. What we want to do is wrap the text input into an observable—essentially hiding the implementation and making it a pure producer of data.

The counterpart of an Observable is the Subscriber, and it can be simply be a function, as we saw.

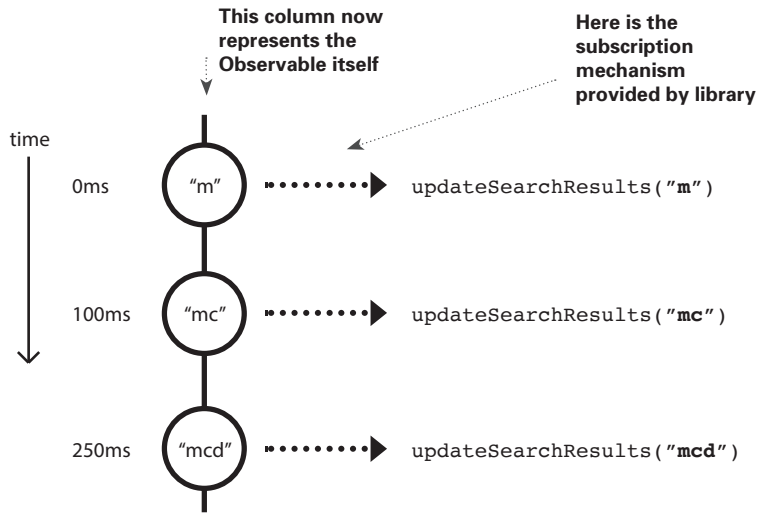
```
Observable<String> textObservable =
  RxTextView.textChanges(textInput);

textObservable
    .subscribe(this::updateSearchResults);
```

Create an observable based on the text input change events.

Establish a connection between the observable and the subscriber.

We can still use the same diagrams as before. `updateSearchResults` is called as before, though without the minimum limit of 3 characters (we'll get to that).



Filtering Observables

Now that we have everything set up we can start with real Rx. Everything we will do is based on the kind of publisher-subscriber pattern we described. There are emitter of data and the subscribers: what happens in between the two is the Rx logic.

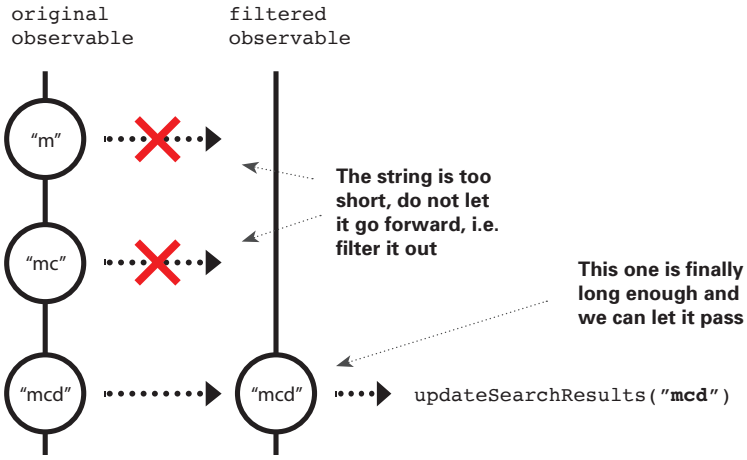
Most of the book is really about what goes in between.

With the decoupling of Observables and Subscribers we have created a convenient arrangement for our logic. *The Subscriber does not know where and when the data comes from, or how many times.*

We can in fact manipulate the data as it is *on the way* to the subscriber. For instance, we can disqualify data items based on certain conditions, such as whether the string is at least 3 characters long.

At the beginning of the book we were adding +1 to all values. We can also block any values from propagating altogether.

Technically we create a *new* Observable based on the original one, but one where the strings that are too short are missing.



The result for now will be the same as with the if-clause we had in the quick implementation. On the next page you can find the code.

The filter operator

Conveniently RxJava, as most Rx libraries, has a method to perform filtering. It works so that we call `.filter` on an observable and give it a filter function that validates the values *one by one*.

The filter function is called for each passing value. You have to be careful not to save state outside of the function itself.

The filter function is called for each item in the stream with the item as the argument. The function then evaluates the item and either lets it pass (returns true), or filters it out (returns false). The filtered values will simply disappear and never reach the Subscriber (in our case the `updateSearchResults` function).

To refresh your memory, here is the initial implementation we made a few pages back:

```
textInput.setOnChangeListener(text -> {
  if (text.length() >= 3) {
    updateSearchResults(text);
  }
});
```

Now we can write the same in RxJava, using the RxTextView utility to convert the TextView into an observable first. Notice that this code calls the `updateSearchResults` just like the code right above.

```
RxTextView.textChanges(textInput)
    .filter(text -> text.length() >= 3)
    .subscribe(this::updateSearchResults);
```

The filter operator sits between the TextView and the listener function, wrapped into the RxJava way of doing it.

Done! Great! It works! ..the same as before. Why the effort, you might ask? That brings us to our next topic.

Time decoupling of Observables and Subscribers

We have finally reached the point where Rx gets interesting and incredibly helpful. We will see how to deal with the second condition of filtering items by the point in time when they were emitted.

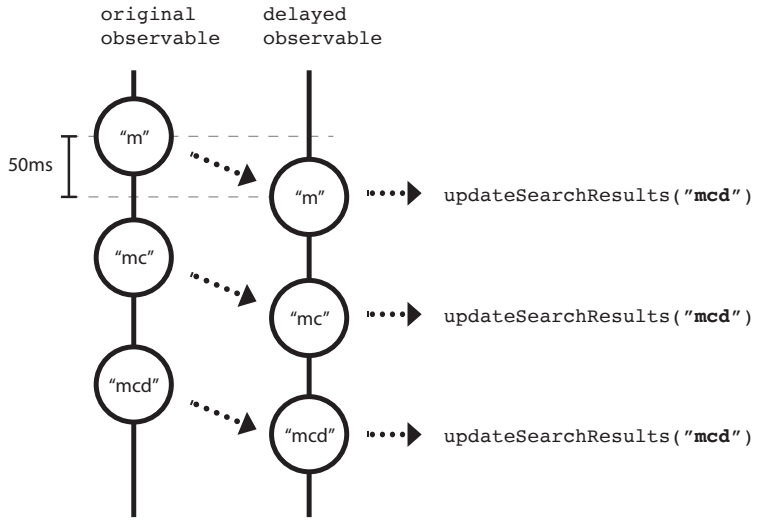
Our last code is already based on data items emitted by the TextView Observable and data items received by the listener function, the Subscriber. With the filter function we already saw that *these are not necessarily the same data items*. The Subscriber does not know what to expect, so we can simply delete some of them on the way.

The Subscriber is just a function. It can be literally be called at any time a new data item is available.

A less obvious aspect is since a Subscriber does not know anything about where the data comes from *a data item does not need to arrive at the same time it was sent*.

Bending time

It turns out with an Rx library we can easily manipulate time relationships of data emitted and data consumed. We could, for instance, delay all data items by 50 milliseconds. In our example it would mean that the search queries are always made 50ms later



For instance, the first marble could be described as such:

1. User types an "m" in the text input

2. The text input observable emits the string "m" into the stream

3. Because of the delay operation the Rx system makes the marble wait for 50ms

4. After 50ms has passed, the marble can continue its journey and in this case end up being consumed by the `updateSearchResults` *function.*

The code in RxJava is quite simple with the tools in the library. We tell our Rx system to delay all data items emitted from the source by 50ms, effectively creating a new Observable that is a bit late.

```
RxTextView.textChanges(textInput)
    .delay(50, TimeUnit.MILLISECONDS)
    .subscribe(this::updateSearchResults);
```

This is the operator that delays everything it gets its hands on

Can I use a delay with a negative number?

In case you had your hopes up, unfortunately delay operator does not work with negative values, and thus cannot receive data items from the future. There have been theories about time travel, and currently it is deemed possible, but only in case another end of a portal is set up in the future "first". It seems this has not yet happened.

Bending time in our benefit

Delaying everything is not usually desired behavior. However, there was the seed of the solution there. Let's have a look at our second condition again, the one about not triggering the update too often:

2. Only perform a search when the user has not been typing for 150 milliseconds.

Since emitting an item to the Subscriber triggers the search, this condition would seem to imply that we indeed have to send the last item with a 150ms delay. *But only in case no other item was emitted during the time it was waiting.*

If we received another item right after it would mean the user is still typing, and we want to let them finish.

The Bus Stop

Effectively what we have is a bus stop: The bus will only leave once everyone is on the bus and the driver has managed to close the doors. As long as there is still someone coming in the doors cannot close and the departure will not happen.

The time we have between the latest person stepping in and the doors closing is 150ms. As long as the user has not stopped typing for long enough the `updateSearchResults` bus has to wait.

Debounce: The Bus Stop Operator

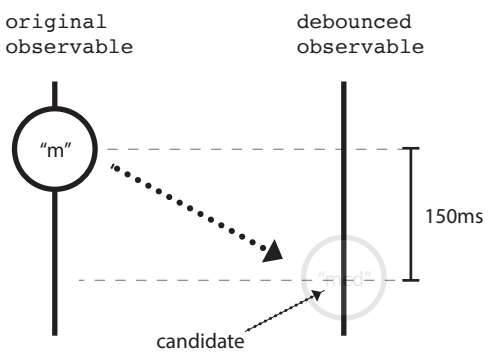
In this chapter we are going to cut it short and get to the more precise details of operators later. We are lucky to find out that what we described on the last page is actually a reasonably common problem in Reactive Programming and there already exists an operator for it. It is called debounce:

> Debounce: only emit an item from an Observable if a particular timespan has passed without it emitting another item.
>
> - ReactiveX Wiki

This may sound a bit abstract, but it's just how the wiki is written. Often seeing an example of *what you can do* with the operator makes it clearer. However, there might be more than one use for each, which is why the wiki is kept on a high level.

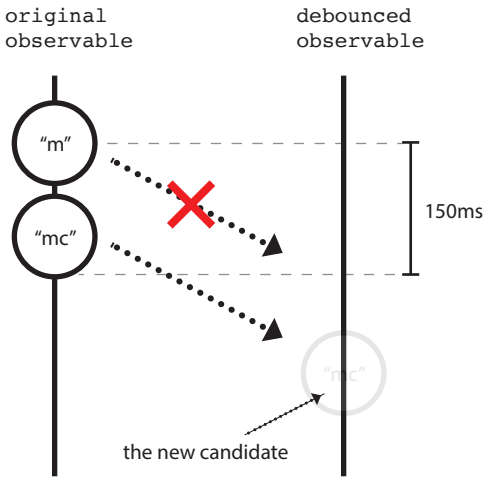
In our case we already have an example, so let's next see how it works from the user's point of view:

1 User starts typing "m" in the search box



The user pressed the "m" key and we receive the item in our Observable. What debounce does, it puts the "m" in the store as a potential candidate, but waits for 150ms before confirming it indeed should be let go forward.

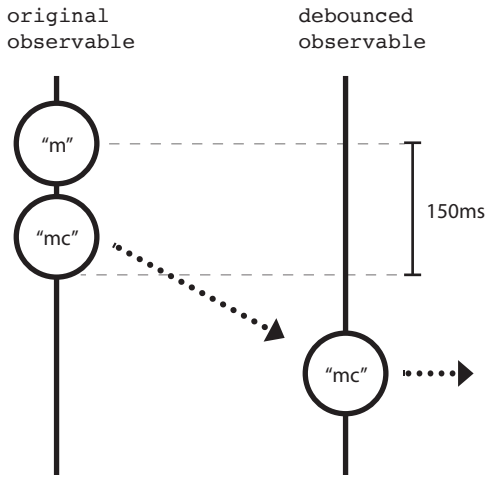
2 User types "c" immediately afterwards



User continues typing before 150ms has passed, which makes the previous candidate invalid! Instead debounce stores "mc" as the new candidate, and *resets the 150ms timer*.

Notice that here the observable always emits the full text in the field. This is how the library RxBindings works.

3 User stops typing



Finally the user stops typing for long enough for the 150ms timer to trigger. We can now send out the candidate we had, this time "mc".

Subsequent items will not affect it anymore, as it is already out there.

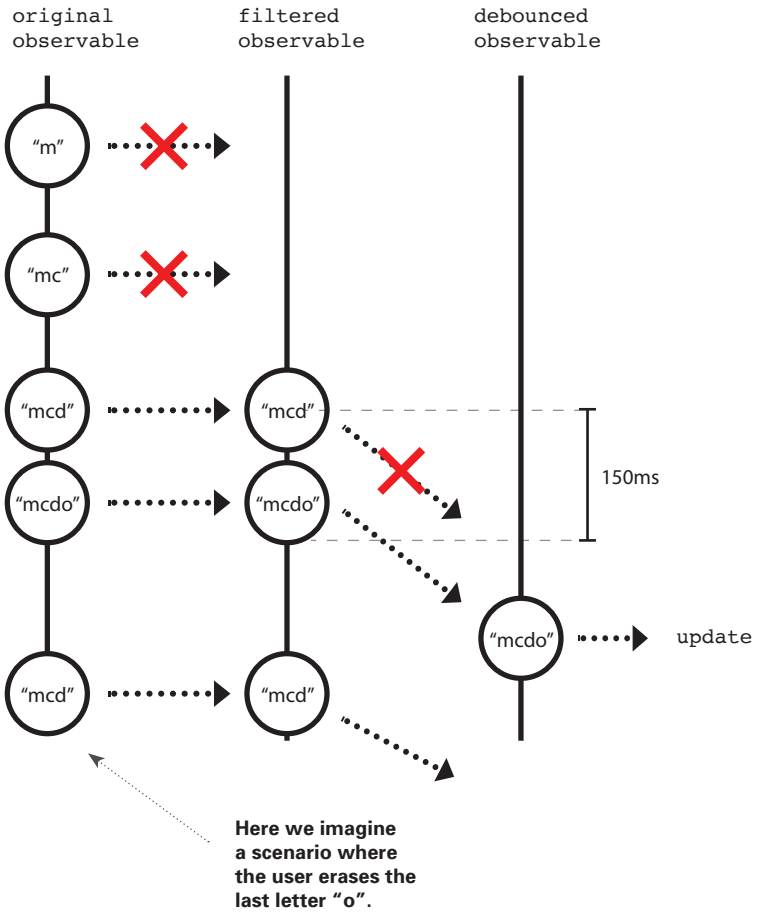
The code

When we apply the debounce to our previous code we actually get the final solution:

```
RxTextView.textChanges(textInput)
    .filter(text -> text.length() >= 3)
    .debounce(150, TimeUnit.MILLISECONDS)
    .subscribe(this::updateSearchResults);
```

This really does fulfill the conditions originally imposed on us: it filters out searches less than 3 characters and wait until user has not typed for 150ms before sending a search request.

The entire picture looks like this:



Putting it into Android

We have so far just seen code without specifying where to exactly put it in our traditional Android application. Skip this if you are not interested in the details. We have a couple of requirements here:

1. Run after UI is created

2. Run only once

Seemingly the most obvious place would appear to be either Activity onCreate on Fragment onViewCreated. Either will do—in this case we will use an Activity, since our UI is not very complex.

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        final EditText textInput =
           (EditText) findViewById(R.id.text_input);

        RxTextView.textChanges(textInput)
            .filter(text -> text.length() >= 3)
            .debounce(150, TimeUnit.MILLISECONDS)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(this::updateSearchResults);
    }

    private void updateSearchResults(String search) {
        ...
    }
}
```

We added a thread change in the middle, since we need to be sure that any operations that manipulate the UI are executed on the main thread. In this case *the debounce operator will automatically switch the thread to a background one* in order to not block the execution while we wait for more input. Most operators do not, but ones that are delayed do.

We will cover thread changes more extensively in the later chapters, but usually it is enough to call `.observeOn(AndroidSchedulers.mainThread())` just before doing a UI operation in the next step (or subscriber). This call will change the thread for anything *downstream*, meaning the steps that come after it are in the thread it defined.

Coffee Break

Make your own application now! Use the Coffee Break base in the online examples, or follow the instructions in the appendix to set up your dependencies.

Try to use other kinds of input components that the RxBinding library has. They all start with Rx, for instance:

- **`RxCompoundButton.checkedChanges`**
- **`RxRadioGroup.checkedChanges`**

These two return a boolean instead of a string.

This is a special case where are not obliged to use the observeOn. We know that the observable here is on main thread because it originates from the UI.

```
RxCompoundButton.checkedChanges(compoundButton)
    .subscribe(isChecked -> {
        myLabel.setText(
            isChecked ? "box checked!" : ":("
        );
    });
```

When looking at the functions RxTextView offers, we can also see ones that do not return observables. These are functions that you can invoke on a TextView, such as

```
Action1<CharSequence> text(TextView).
```

We can use them instead of the lambda syntax, if we want to. It is up to you which one you prefer, but we will continue using lambdas for brevity.

Exercise

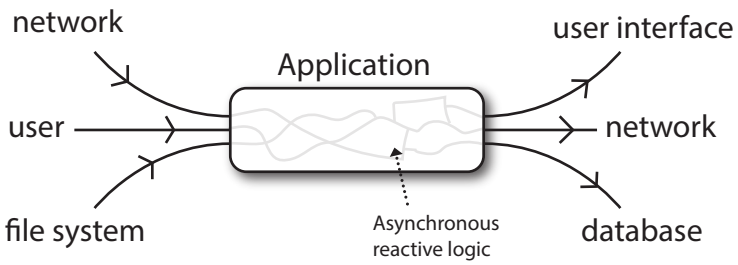
Make an EditText and a TextView. Have the TextView say "Text too long" after user has typed more than 7 characters. You can see the solution in the online code examples.

Principles of Reactive Programming

We have now seen a bit of Reactive Programming in action. This is just the tip of the iceberg, but we are starting to see the pattern here already:

- Program is seen as a data processing system—something goes in and something comes out. It might come out as text change or an animation on the screen.
- Data processing is done with small functions that take an input and produce an output (or an effect, such as debounce). An Rx library is the glue between these functions, giving the data from the previous function to the next.
- *Every function is considered asynchronous.* Running a function takes time. It might be just a millisecond or two, but there no such thing as "instant". We therefore only declare what should happen after a function has been executed but we do not hang around waiting for it.

The application becomes a big pipeline of different points of inputs and outputs. What happens in between is the isolated Rx logic. It models how we can calculate the output based on our inputs.



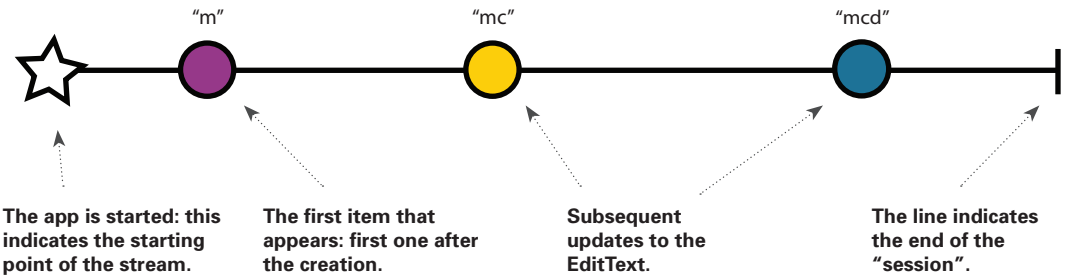
In out example the input was the user typing into the EditText. We wrapped the input into an Observable and pushed the generated data into the asynchronous Rx logic. The output in this case was shown in the `updateSearchResults`.

Events and Marble Diagrams

RxJava is good at handling events that occur at certain points in time. So good, in fact, that a new way of representing events has become a standard for it.

The Marble Diagram

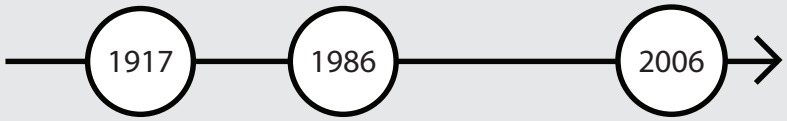
We already saw time-based diagrams: with circles that represent pieces of data that appear at certain points in time. The standard, however, is to rather put them horizontally. We also usually omit the actual data represented by the "marbles".



We will later see also different kinds of diagrams, but marbles are very popular with the *event-based problems* that Rx can solve. We will see many of those in the first part of this book.

Wait, is the diagram not the wrong way?

If you think about a traditional stack, the item on the right would be actually the first one to be emitted. However, the marble diagram is more of a "historical" time line of *what happened* on a time line. In these kinds of time lines the time flows from left to right.



Summary

Wow! That was almost 30 pages to write 5 lines. You are probably wondering if it was really necessary.. I mean, could we not just check Stack Overflow and copy the snippet?

The answer is yes and no. You can check the Internet and copy the snippet, *if you know what you are looking for*. Many problems become almost trivial once they are framed in the right way—in our case it took us a while to get to a point where we have a good representation of the input stream.

Learning the How to ask the Question

In Reactive Programming phrasing of the question is sometimes 90% of the work. Often your problem is not as unique as it first seems, and there might already be an Rx feature that solves it.

In this book we will focus on learning the thinking patterns that allow us to formulate our requirements in a way that makes sense.

Often we start by converting the inputs into streams and seeing how they can be combined. Sometimes it's about drawing the marble diagrams and checking the Rx wiki for similar ones (reactivex.io).

It may seem a little confusing at first, but it starts making sense after you get a few more practical example under your belt.

There are also not too many patterns you need in a typical app. After going through the examples of this book you should be well-equipped for most scenarios in the wild.