Universitat de Lleida

Activity 2: Reactive programming. ReactiveX & JavaRx

Students
FERNÁNDEZ FLORENSA, VICTOR (47935487Z)
DKOUK EL FERROUN, CHAYMAA (49758978L)

Teacher
LERIDA MONSO, JOSEP LLUIS

Bachelor's degree in Computer Engineering
Distributed Computing
27Th of October 2022

Index

Index of Figures

1 Reactive Programming

1.1 Definition

Reactive programming is a declarative programming paradigm that consists in using asynchronous data streams, that is, an approach to problem-solving that allows us to manipulate a stream of data with functions asynchronously.

An asynchronous data stream can be best described as a stream of data where values are emitted, one after another, with a delay between them. The word asynchronous means that the data emitted can appear anywhere in time. Commonly, asynchronous streams are modeled as values on a time axis:

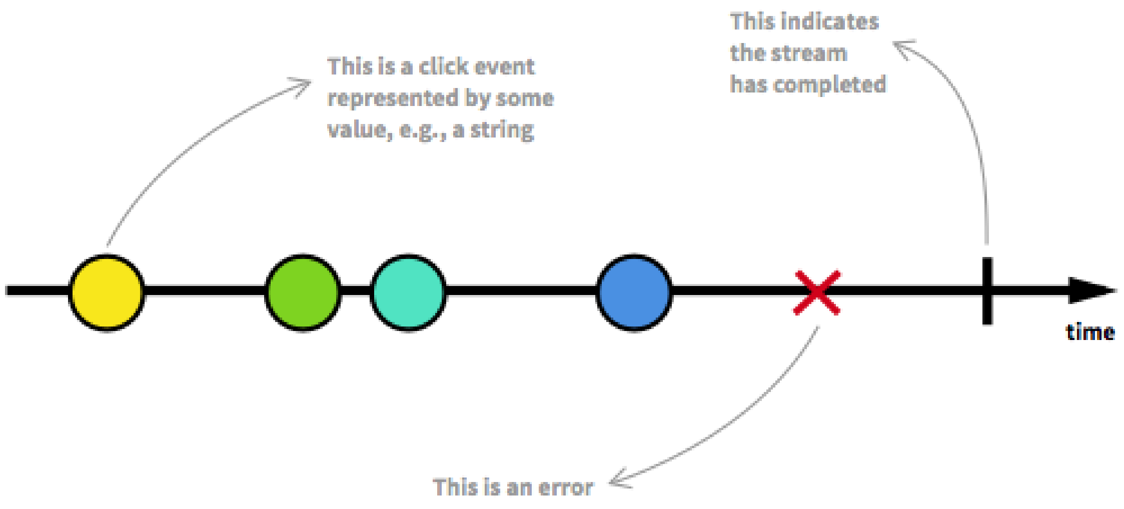
This is a click event represented by some value, e.g., a string
This indicates the stream has completed
time
This is an error

Figure 1: Model of an asynchronous data stream

The idea of using asynchronous data streams is something that existed way before the reactive programming paradigm. For example, conventional click events are basically an asynchronous event stream.

However, reactive programming takes this idea to another level. When using this paradigm, data streams are going to be the spine of your application. You will create data streams of anything and from anything that may change: click events, HTTP requests, ingested messages, availability notifications, changes on a variable, cache events, measures from a sensor, etc.

1.2 Use Cases

Like any programming paradigm, reactive programming isn't fitting for any problem and has some general patterns of use. Some examples where this paradigm could be applied are:

- Render a UI that combines data from multiple data sources
- Log data from a sensor
- Create a real-time model from stock prices

In each of these cases, there is some sort of data that can be processed as a stream and an opportunity for functional manipulation of that data stream asynchronously.

Although these type of problems can also be solved with other paradigms like the object oriented programming, reactive programming makes it easier and there's plenty of evidence that it is something worth to learn and integrate in our projects:

- There are many applications that should not block the UI, so it's very commonly used in the Android community.
- There is also some usage at Google for database requests.
- Microsoft, Netflix, GitHub, SoundCloud and others.

Figure 2: Logos from the top ranked enterprises using ReactiveX

1.3 Advantages and Limitations

One of the principal benefits of Reactive Programming is that it allows non-blocking execution (asynchronous) such that threads can do other useful work while the resource is occupied. These asynchronous communications provide scalability.

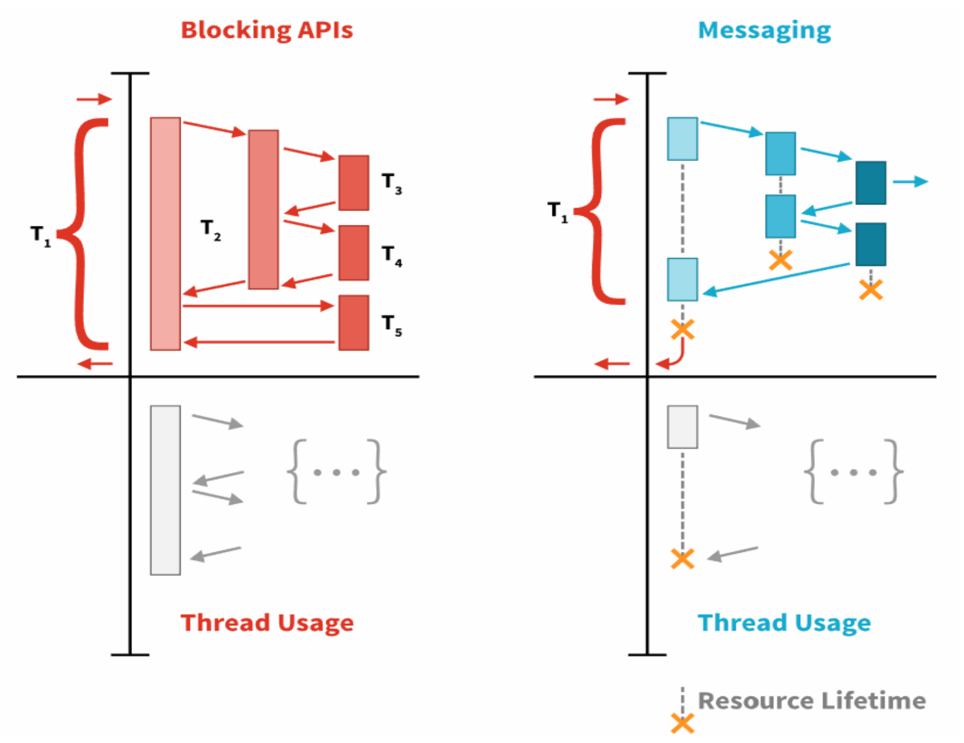

Figure 3: Blocking vs non-blocking execution

Another benefit is that it makes easy for developers to compose streams of data. Reactive Programming provides a set of functions to create, filter and combine data streams.

Additionally, this programming paradigm raises the level of abstraction of the code, enabling developers to focus on the business logic without to worry about every implementation detail.

Nevertheless, although it offers lots of advantages, it has a reputation of being difficult to learn. Also, because it is based on streams over time applications can become prohibitively memory intensive.

Moreover, because of the ability to compose functions this makes each stage not addressable. This means that if one function fails all the process needs to be restarted and the client notified. Hence, Reactive Programming makes resilience harder to achieve.

Also, Reactive Programming doesn't allow decoupling in space, which means it doesn't provide location transparency.

All in all, even though Reactive Programming is a very useful piece when constructing modern software, in order to reason about a system at a higher level one has to use another tool: Reactive Architecture—the process of designing Reactive Systems.

1.4 Reactive Systems

Up until now we've seen that we can write applications in a Reactive style by using Reactive Programming. However, using Reactive Programming does not build a Reactive System.

As defined in the reactive manifesto, Reactive Systems are an architectural style to build responsive distributed systems. A reactive system is mainly characterized by four properties:

- **Responsive**: it has to provide rapid and consistent response times.
- **Resilient**: the system stays responsive in case of failure.
- **Elastic**: the system must scale up and down, and be able to handle the load with minimal resources.
- **Message driven**: the components of a reactive system interact using asynchronous message passing.

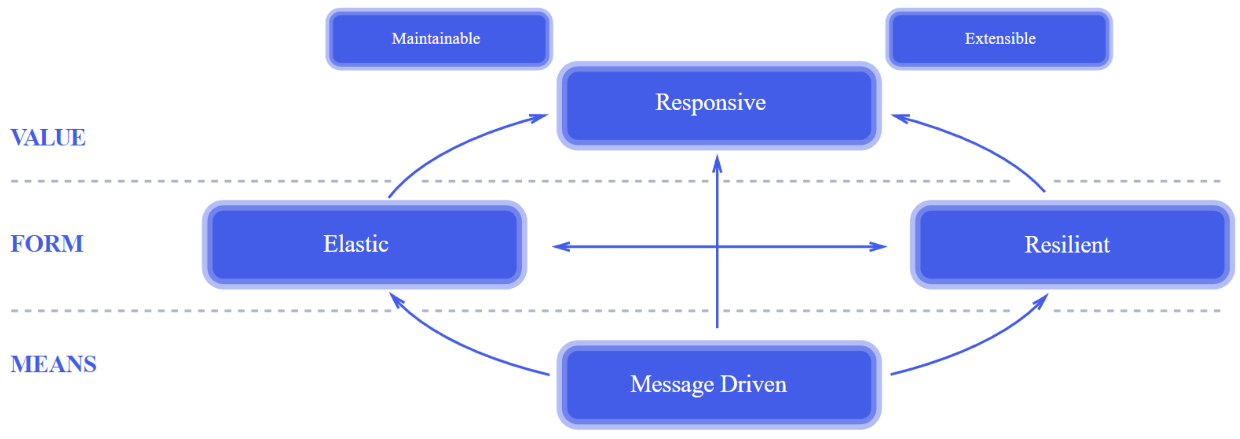

Figure 4: The tenets of the Reactive Manifesto

The main driver behind modern systems is the notion of Responsiveness: the acknowledgement that if the client/customer does not get value in a timely fashion then they will go somewhere else.

In order to facilitate Responsiveness, two challenges need to be faced: being Responsive under failure, defined as Resilience, and being Responsive under load, defined as Elasticity. The Reactive Manifesto prescribes that in order to achieve this, the system needs to be Message-driven.

2 ReactiveX

2.1 Introduction

In the official page, ReactiveX is defined as it follows:

"A library for composing asynchronous and event-based programs by using observable sequences."

In other words, ReactiveX is an extension to build reactive programs. Each language has its own library like JavaScript with RxJS, C# with Rx.NET, Python with RxPY, and so on. In this research, however, we will focus on JavaRx library.

You might wonder why to use JavaRx although Java already implements mechanisms to work with data streams in Java 8. As Jordan Jozwiak said in the CS50 Tech Talk 2018,

"One reason why JavaRx is still popular is because it adds a lot of functionality that isn't available and, in addition of that, although streams of data are pretty good on Java, they could be easier to work in terms of flexibility with the observer pattern."

In order to understand better how this set of methods will help us to build our programs, first we have to take a look at how the *observables* work.

2.1.1 Subject

We understand by Subject the source of the data. It could be a sensor, chat messages, economic indicators, a query on a database, and so on.

2.1.2 Observables

Observables fill the gap by being the ideal way to access asynchronous sequences of multiple items.

ReactiveX uses the Observer Pattern, which means Subject data sources will notify the Observers. Observables can "subscribe to" or "consume" this data. There can be multiple observers for a single subject.

This library also provides a collection of operators with which you can filter, select, transform, combine, and compose Observables.

Last but not least, in data streams errors will be propagated though the flow of data, and it's up to the consumer to handle it or ignore it.

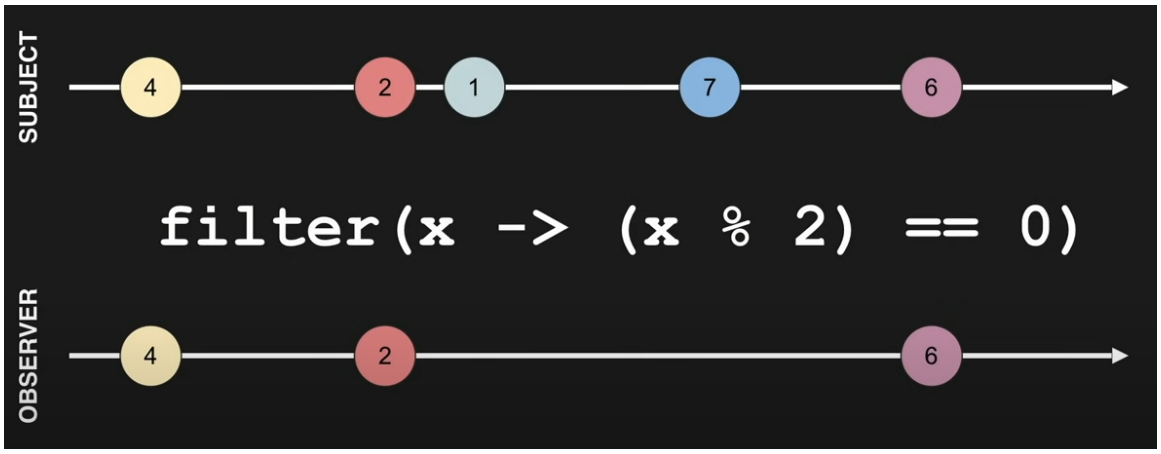

Figure 5: Diagram of an observer, obtaining filtered data from the subject.

2.2 JavaRX

Until now, we've gone through Reactive programming from a theoretical standpoint. In order to see a practical approach, we have created some examples to apply what we have learnt.

2.2.1 Asynchronous test of Reactive Programming

The goal of this example will be testing the asynchronous execution of Reactive Programming. The following program:

- Creates an observable that reads an element of the list fruits each 1.5 seconds.
- Initializes the observable. This one prints each item as it reads it.
- Finally it awaits until the observer finishes. This is necessary so that the main process doesn't exit before the child process (subscriber) has time to print the delayed data.

```java
public class Async_example {
    public static void main(String[] args) throws InterruptedException {

        //Source data
        List<String> fruits = Arrays.asList("Apple", "Orange", "Pear",
            "Banana", "Strawberry", "Lemon");

        //Observer with a timer: retrieves data each 1.5s
        Observable observer = Observable.fromIterable(fruits)
                        .concatMap(
                                item -> Observable.interval(1500,
                                    TimeUnit.MILLISECONDS)
                                .take(1)
                                .map(second -> item)
                        );

        System.out.println("I'm going to buy some fruits \n");

        //Subscriber
        observer.subscribe(
                item -> System.out.println("We got an: " + item)
        );

        System.out.println("I already made the order! \n");

        sleep(12000);
    }
}
```

Terminal Output

```
I am going to buy some fruits.

I already made the order!

I got an: Apple
I got an: Orange
I got an: Pear
I got an: Banana
I got an: Strawberry
I got an: Lemon
```

As you can see, we have added two prints: one before, and another after the subscription.

As expected, the printf("I already made the order!") shown on the output precedes all the items retreaded by the subscriber, so the observer and the subscriber are not blocking the main execution of the program.

2.2.2 Convert one observable to another

In this example, we will take the output of an observable, and we will transform the data with the chopFruit method in another observable. As it can be seen on the following class the Observable obs1 consumes the list fruits and the Observable obs2 executes the function chopFruit on each element consumed by obs1.

```java
public class MapFunction_example {
    public static void main(String[] args) {

        //Source data
        List<String> fruits = Arrays.asList("Apple", "Orange", "Pear",
            "Banana", "Strawberry", "Lemon");

        //Observer 1
        Observable obs1 = Observable.fromIterable(fruits);

        //Observer 2
        Observable obs2 = obs1.map(item -> chopFruit(item));

        //Subscriber
        obs2.subscribe(
                item -> System.out.println("I got a: " + item)
        );

    }
    public static String chopFruit(Object fruit){
        return "chopped " + fruit.toString();
    }
}
```

Terminal Output

```
I got a: chopped Apple
I got a: chopped Orange
I got a: chopped Pear
I got a: chopped Banana
I got a: chopped Strawberry
I got a: chopped Lemon
```

2.2.3 Combine two observables to a third one

In this example, we use the zip operator to join the outputs of observer1 and observer2 onto a tuple of two elements.

```java
public class ZipFuntion_example {
    public static void main(String[] args) {

        //Source data
        List<String> fruits   = Arrays.asList("Apple", "Orange", "Pear",
            "Banana", "Strawberry", "Lemon");

        List<String> vegetables = Arrays.asList("Pumpkin", "Broccoli",
            "Tomato", "Carrot", "Lettuce", "Onion", "Spinach");

        //Observer 1
        Observable obs1 = Observable.fromIterable(fruits);

        //Observer 2
        Observable obs2 = Observable.fromIterable(vegetables);

        //Observer 3
        Observable obs3 = Observable.zip(obs1, obs2, (BiFunction<String,
            String, String>)
                (s1, s2) -> String.format("(%s, %s)", s1, s2));

        //Subscriber
        obs3.subscribe(
                item -> System.out.println("I got a: " + item)
        );
    }
}
```

Terminal Output

```
I got a: (Apple, Pumpkin)
I got a: (Orange, Broccoli)
I got a: (Pear, Tomato)
I got a: (Banana, Carrot)
I got a: (Strawberry, Lettuce)
I got a: (Lemon, Onion)
```

Note that "Spinach" item doesn't have a pair, so it's not shown on the output.

3 Conclusions

Reactive programming opens a new paradigm for problem-solving which we think is very interesting and has special relevance nowadays, specially in order to build responsive applications. We see strong impact of this technology on the Android community, but also on most of the systems with sensors like the automotive sector. Furthermore, we have tried to put together reactive programming with what we have learnt on the theoretical lessons.

4 Bibliography

[1] *Reactive programming*, en, Page Version ID: 1116006149, oct. de 2022. adr.: `https://en.wikipedia.org/w/index.php?title=Reactive_programming&oldid=1116006149` (cons. 26-10-2022).

[2] *Java Reactive Programming: Everything You Need to Know*, en, nov. de 2020. adr.: `https://www.zibtek.com/blog/java-reactive-programming-everything-you-need-to-know/` (cons. 26-10-2022).

[3] *Asynchronous data streams - Architecting Angular Applications with Redux, RxJS, and NgRx [Book]*, en, ISBN: 9781787122406. adr.: `https://www.oreilly.com/library/view/architecting-angular-applications/9781787122406/8ac189ca-a3e4-4376-8a9c-aae8a60cf5df.xhtml` (cons. 26-10-2022).

[4] C. Escoffier, *5 Things to Know About Reactive Programming*, en, juny de 2017. adr.: `https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming` (cons. 26-10-2022).

[5] CS50, *Intro to Reactive Programming by Jordan Jozwiak of Google - CS50 Tech Talk*, oct. de 2018. adr.: `https://www.youtube.com/watch?v=KOjC3RhwKU4` (cons. 26-10-2022).

[6] L. Inc, *What is #Reactive? Read Reactive Programming vs Reactive Systems @jboner & @viktorklang | @lightbend*, en. adr.: `https://www.lightbend.com/white-papers-and-reports/reactive-programming-versus-reactive-systems` (cons. 26-10-2022).

[7] *The Reactive Manifesto*. adr.: `https://www.reactivemanifesto.org/` (cons. 26-10-2022).

[8] *ReactiveX*, en, Page Version ID: 1116010777, oct. de 2022. adr.: `https://en.wikipedia.org/w/index.php?title=ReactiveX&oldid=1116010777` (cons. 26-10-2022).

[9] *ReactiveX - Intro*. adr.: `https://reactivex.io/intro.html` (cons. 26-10-2022).

[10] Moldeo Interactive, *RxJS - ¿Qué es y Para qué usarlo?* Set. de 2018. adr.: `https://www.youtube.com/watch?v=zVONywULR_4` (cons. 26-10-2022).

[11] *Java Reactive Programming Tutorial - YouTube*. adr.: `https://www.youtube.com/watch?v=wUniksiWFcs&ab_channel=JavaCodeGeeks` (cons. 26-10-2022).

[12] *Programación Reactiva 4 conceptos para empezar - YouTube*. adr.: `https://www.youtube.com/watch?v=kkqn2Z2tl1k&ab_channel=C%C3%B3digoMorsa` (cons. 26-10-2022).

[13] Decoded Frontend, *Hot vs Cold Observable in RxJs (2021)*, des. de 2021. adr.: `https://www.youtube.com/watch?v=oKqcL-iMITY` (cons. 26-10-2022).

[14] *Java 8 Tutorial - 16 RxJava - YouTube*. adr.: `https://www.youtube.com/watch?v=dDOvE3GGzDM&ab_channel=MitoCode` (cons. 26-10-2022).

[15] J. Ramos, *Introducción a RxJava: Tutorial desde cero (Con ejemplos)*, es. adr.: `https://programacionymas.com/blog/introduccion-rx-java-tutorial-android` (cons. 26-10-2022).

[16] C. Á. Caules, *Introducción a RxJava y sus observables*, es, gen. de 2016. adr.: `https://www.arquitecturajava.com/introduccion-rxjava-observables/` (cons. 26-10-2022).

[17] *#5 RxJava - 3 ways to create Observables - YouTube*. adr.: `https://www.youtube.com/watch?v=0vqAcND_yas&ab_channel=MithuRoy` (cons. 27-10-2022).