
A Few
Words
About
RxJava
@ Saiimons
sidereo

I’m Simon Guerout.

C#O at Sidereo
s@sidereo.com
@saiimons
@sidereomobile
http://sidereo.com

So, What's Rx

Created by Microsoft for .NET

Functional Reactive Programming

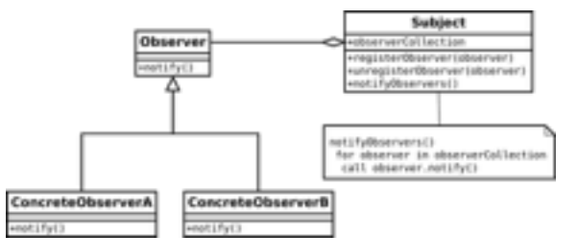
Now available for Java and Android

Reactive extensions are a framework created by Microsoft for .NET at the end of 2009.
It is a «library that allows programmers to write succinct declarative code to orchestrate and coordinate asynchronous and event-based programs »

The way you write code with this library might be referred to as functional reactive programming, we'll talk more about this during the presentation.

It is now available for Java and Android. The specifics for Android are helpers for thread and UI stuff.
Jake Wharton is a huge contributor for the Android version.

Functional Reactive Programming



Black boxes, input, output

Streams, streams, streams

Observer pattern

Think of functional programming as developing a stateless application.
You are writing black boxes, taking input data, processing it and delivery the output result.

Using this way of thinking, you have to put your mind in « stream mode » : everything is a stream.
Wether the user clicks a button, an API responds to a request or the GPS coordinate changes.

In the real world, Reactive extensions rely on an extended version of the Observer pattern.
This pattern is a pretty basic way off expressing the callback mechanism on an object.

1

Key Concepts

You have to learn these

Let me first introduce you to key concepts, which you have to know about if you want to start using Rx.

Classes And Methods

These, you will use :

Observable<SomeType>, the guy who emits the data.

Observer<SomeType>, the guy who listens to the data.

In order to use the framework, you must be aware that you will handle mainly 2 types :
The Observable, which is the implementation of the pattern, it is used to subscribe to the data output.
The Observer, which will be called when

- new data is available
- the end of the stream was reached
- an error occurred

Inside The Observer

```
public interface Observer<T> {
    void onCompleted();
    void onError(Throwable e);
    void onNext(T t);
}

public abstract class Subscriber<T>
    implements Observer<T>, Subscription { … }
```

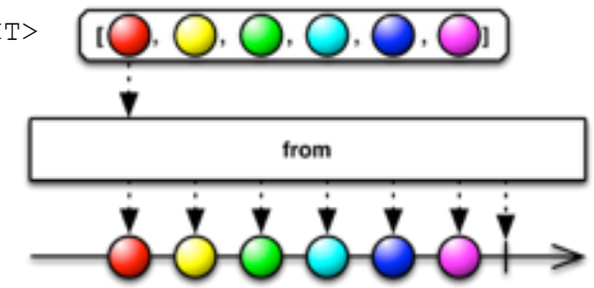
When creating an observer, you will have to write 3 functions :

- onNext, which will be called when new data is available from the Observable, this is where you can process the new data.
- onCompleted, when no more data will be delivered from the Observable, this is where you can free some resources.
- onError, which will be called when something failed.

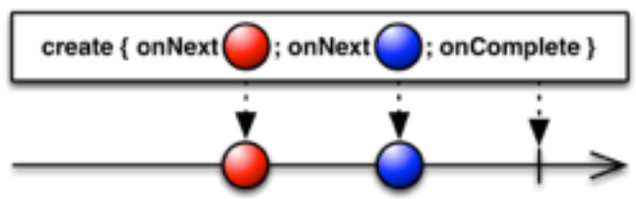
Mainly, you will extend the Subscriber class, it is handy, and it supports subscription (which will be covered later)

Inside The Observable

```
public final static <T> Observable<T>
from(T[] array)
```




```
public final static <T> Observable<T>
create(OnSubscribe<T> f)
```



There are basically 2 ways to start creating an Observable :

- Either you can set the list of values to be emitted directly.
- Or you can implement the OnSubscribe interface :
    - your implementation will be called each time a new subscriber wants your data,
    - you are responsible for calling onNext, onCompleted and onError when needed

This second case is very handy to wrap some calculation, and also work with a subscription.

Subscription

```
public interface Subscription {
    void unsubscribe();
    boolean isUnsubscribed();
}

public abstract class Subscriber<T>
        implements Observer<T>, Subscription {
    public final void add(Subscription s){ … }
}
```

The subscription is another essential part of Rx : it handles the fact that sometimes, you might want to stop listening to an observable.
It is a very handy interface, and works with a whole chain of calls (ie. if you have built a chain of observables, you only need to unsubscribe your own subscriber).

Another use for this interface, is when you are creating your own observable.
You can query the subscriber in order to know wether it is still active.
If you add a subscription to a subscriber, you will be notified asynchronously when this subscriber unsubscribes.

Operators

Map

Filter

Cast

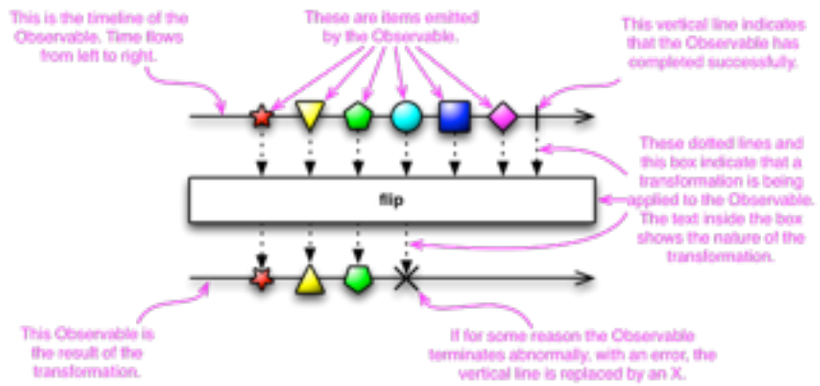
Merge

...

http://rxmarbles.com/

In order to finish introducing the key concepts, I'll have to talk about the Operators.
The operators are a set of tools allowing you to perform operations on the data emitted by an Observable.
There already is a set of very handy functions available with the framework, I will not cover each of the operator, but here are a few examples :
GO TO rxmarbles.com

Sum Up



In order to sum up the concepts :

- An observable emits data of a kind
- You can apply transformations on any observable
- You can subscribe (and unsubscribe) to any observable
- 3 main events : onNext, onCompleted, onError

2

Android + Rx

Some added sorcery

Libraries

RxAndroid

compile 'io.reactivex:rxandroid:1.0.1'

compile 'io.reactivex:rxjava:1.0.14'

RxLifecycle

compile 'com.trello:rxlifecycle:0.3.0'

compile 'com.trello:rxlifecycle-components:0.3.0'

There are basically 3 sets of libraries designed for Android.

RxAndroid : This module offers the integration of a few helpers, which are basically making it possible to run code in the Main thread or a Handler Thread.

RxLifecycle : This module gives you handy wrappers to handle the specificity of Activities, Fragments, …

Libraries

RxBinding-*

```
compile 'com.jakewharton.rxbinding:rxbinding:0.2.0'
compile 'com.jakewharton.rxbinding:rxbinding-support-v4:0.2.0'
compile 'com.jakewharton.rxbinding:rxbinding-appcompat-v7:0.2.0'
compile 'com.jakewharton.rxbinding:rxbinding-design:0.2.0'
compile 'com.jakewharton.rxbinding:rxbinding-recyclerview-v7:0.2.0'
```

RxBindings

This set of module is developed by Jake Wharton, and will offer a lot of basic features you would not want to write by yourself.
Basically this covers any callback system used in the UI.
For example, you can get the observable for the click events of any view.

Threads

Main thread vs . background threads is a recurrent issue.

AsyncTask or any similar API become heavily used.

Main thread, is the king, you end up « posting » a lot.

Observable

```
public final Observable<T> observeOn(Scheduler scheduler);
public final Observable<T> subscribeOn(Scheduler scheduler);
```

AndroidSchedulers

```
public static Scheduler mainThread();
```

One particularity of the Android system, is that many operations have to run in the Main (UI) thread.
As RxJava was built for a multithreaded environments, it is easy to overcome this, using :

- observeOn when you need the result to happen in a specific thread
- subscribeOn when you need to processing to happen in a specific thread

RxAndroid provides methods which will return a scheduler for the main thread (or for handler threads when needed).

Lifecycle

Memory leaks are bad.

Calling UI outside the main thread is bad.

Calling code when you Activity was destroyed is bad.

```
public class MyActivity extends RxActivity {
    @Override
    public void onResume() {
        super.onResume();
        myObservable
            .compose(bindToLifecycle())
            .subscribe();
    }
}
```

Using the Lifecycle components, you can extend Activity or AppCompatActivity, in order to be able to bind to a specific part of lifecycle.
For example : resume / pause, create / destroy.
Note that you will have to face weird casts on the compose depending on the data you observe.

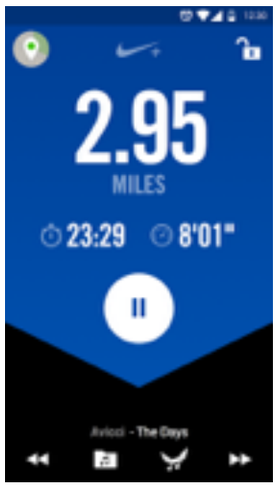
3

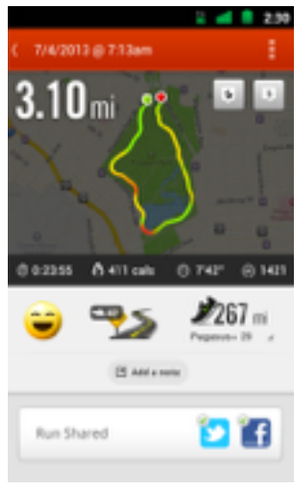
Real Life Example

‘Cause it’s always better to use it for real…

Let’s move on to a real world example, which will offer you a vision on how to use Rx and its extensions.

Location, Location, Location…





Let’s imagine you want to be able to listen to location changes from your application.
Let’s say, your user can activate or reactive location, and change the rate at which location is updated.
Let’s say you want your UI to be updated, and a database of the positions to be be populated.
(which would be a very common case for a fitness app)

If you were to make such a UI / app, which emitted data would you consider using ?

Players

GoogleApiClient : you will need it for using the FusedLocationApi

LocationRequest : this guy tells the FusedLocationApi which kind of location will be helpful for you.

Clicks : to the user's command you will listen, young padawan.

This game has 3 players :

- You need to set up a GoogleApiClient, and get it "connected" only when you need to listen to locations. This class is giving you access to a set of APIs controlled by the Google Play Services. It needs to be initialized in a separate thread, let's spare us some pain, and create an Observable.
- You will then have a few parameters to give to the LocationServicesApi : these are LocationRequests. They will change as your user selects one of the options available, so we might have an observable for this too.
- Your user will use a few buttons, which will be emitting events, we may need to convert.

=> Have a look a the code

What Did We Learn ?

Observables can be Hot or Cold, which roughly translates to active or passive.

Operators will help you transform emitted values.

Use Subscriptions it helps you avoiding mess.

Observables can be emitted it may save you time and complexity.

Android has its own extensions, use them!

First, please do not take this example as the perfect source.
It is merely an example, and I made some voluntarily specific uses of RX, so that you can see how many problems can be adressed.

The first thing I'd like to underline is the fact that the Observables can be Hot or Cold.
A Cold Observable is for example the Google Api Client one : it will only give a new GAC if you subscribe to it.
A Hot Observable is for example the View Click one : clicks will happen, wether there is a subscriber or not.

Subscriptions are the best tool when it comes to handling the lifecycle of your objects.
Use them, you will avoid memory leaks, and other bugs.

Emitted values can be transformed easily through black boxes se how I managed to get a Location Request from a ViewClickEvent.
This transformed value is then emitted to the next item in the chain.

Observables can be emitted : this is a tough one, but can be very helpful.
For example, we get many LocationObservables, and use the switch operator to select the latest one.
This helps wrapping up some APIs which are not letting you update their settings (or avoiding adding weird methods to do this).

I have used some of the awesome Android extensions for RxJava : Listeners for the view, Lifecycle sanity, …
USE THEM : you will save a lot of time, and a lot of boilerplate code.

Clone

https://github.com/saiimons/RxLocationTutorial

I’ll let you guys checkout the code of this little demonstration

Advantages / Drawbacks

Synthetic code

Functional slices of code

Android specifics

Reusable components

Replaceable blocks

Learning curve

Readability

Debugging

Advantages :

- If you like your code in synthetic, functional blocks, you will like Rx
- It has libraries helping you integrating with Android : USE THEM
- Your components can be reused / replaced : for testing or for evolutions of your app

Drawbacks :

- It can be painful to learn and discover all the Observable methods
- A neophyte will find it hard to understand (though you should definitely write documentation)
- Debugging can be a little bit tricky, because of some lengthy stack traces

Questions

```
Observable<Question> questions = Observable.create(people);

questions.subscribe(
    new Action1<Question>() {
        @Override
        public void call(Question question) {
            Simon.answer(question);
        }
    }
);
```

If you have any questions, now is the time !

Thank You!

Share if you liked this deck!

@Saiimons