Reactive Programming With RxJava

Creating Asynchronous,Event-Based Applications

By Mohammad Sianaki

Imperative and Functional Programming

- Java is an imperative,object-oriented language.
  - Sequence of instructions
  - Executed in the same order you write them
  - Execution leads to changes in the state of the program

Imperative and Functional Programming

```
List<Integer> input = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> output = new ArrayList<>();
for (Integer x : input) {
    if (x % 2 == 0) {
        output.add(x);
    }
}
```

- Define and create an input list.
- Define and create an empty output list.
- Take each item of the input list.
- If the item is even, add it to the output list.
- Continue until the end of the input list is reached.

Imperative and Functional Programming

- Functional Programming:
  - Is an alternative to imperative programming
  - The result of program derives from the evaluation of mathematical function
  - Without changing the internal program state
  - the result of the f(x) depends on the arguments passed to the function
  - Passing the same parameter x, you get the same result

Imperative and Functional Programming

- the blocks with which you build the program are not objects but functions.

```
var output = input.where( x -> x % 2 == 0);
```

Reactive Programming

- A general programming term that is focused on reacting to changes:
  - such as data values ,click events
- Often done imperatively
  - Callback is an approach to do that

- takes functional programming a little bit further, by adding the concept of data streams

Example of Reactive Programming

- x=y+z
- The value of x will be updated automatically without re-executing the statement(You MUST Implement a mechanism)

Streams of Data

- The main key to understand reactive programming

- a sequence of events, where an event could be user input (like a tap on a button), a response from an API request (like a Facebook feed), data contained in a collection, or even a single variable.

When You Need Reactive Programming?

- Processing user events such as: mouse clicks,keyboard typing,GPS signals changing over time as users move with their device
- Responding to and processing any and all latency-bound IO events from disk or network

Functional vs Imperative Reactive Programming

- Handling only one system event
  - Reactive-imperative programming with a callback is going to be fine
  - Reactive-functional programming is not going to give you much benefit
- Handling hundreds of different event streams which are all completely independent:
  - Still imperative approaches are going to be the most efficient

Functional vs Imperative Reactive Programming

- If you need to combine events,asynchronous responses from functions or network calls,etc
  - Imperative approach increase complexity
  - But reactive-functional programming begins to shine

Functional Programming with Java 8

- With the release of Java 8 some constructs of functional programming have been added,like:
  - Lambda Expression
  - Streams
- But with RxJava we can use Functional Programming since JDK 6.

Lambda Expressions

- Anonymous functions
- Using arrow symbol (->)
- Inputs are on the left and the function body is places at the right
- Lambda Expressions can be uses to replace anonymous inner classes that implement an interface with just one method

Lambda Expressions

```
Button button = ...
button.setOnClickListener(
new OnButtonClickListener() {
void onButtonClicked() {
// do something on button clicked
    }
  }
)
```

Lambda Expressions

```
Button button = ...
button.setOnClickListener( () -> // do something on button clicked )
```

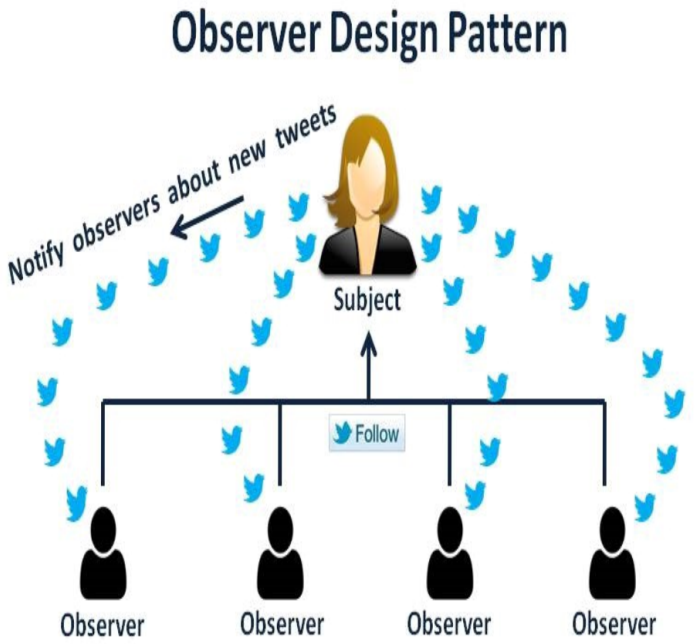
Streams

- Stream represents a sequence of objects from a source, which supports aggregate operations

```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);

//get list of unique squares
List<Integer> squaresList = numbers.stream().map( i -> i*i).distinct().collect(Collectors.toList());
```

The Observer Pattern



- Observer : an object that observes the changes of one or more subjects
- Subject : an object that keeps a list of its observers and automatically notifies them when it changes its state.

What is ReactiveX?

- ReactiveX is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming.
- a library that implements functional reactive programming in many languages:
  - RxJava,RxJS,Rx.NET,RxCpp,RxPY,RxSwift,RxPhp,RxKotlin,...
- uses “observables” to represent asynchronous data streams
- abstracts all details related to threading, concurrency, and synchronization

Why RxJava

- Avoid “callback hell”
- A lot simpler to do Async / threaded work
- A lot of operators that simplify work
- Very simple to compose streams of data
- Complex threading becomes very easy
- You end up with a more cleaner, readable code base
- Easy to implement back-pressure

RxJava Basics: Observable, Observer

- RxJava is all about two key components: Observable and Observer
- Observable: Observable is a data stream that do some work and emits data.
- Observer: Observer is the counter part of Observable. It receives the data emitted by Observable.
- Subscription: The bonding between Observable and Observer is called as Subscription. There can be multiple Observers subscribed to a single Observable.

RxJava Basics: Observable, Observer

- Operator / Transformation: Operators modifies the data emitted by Observable before an observer receives them.
- Schedulers: Schedulers decides the thread on which Observable should emit the data and on which Observer should receives the data i.e. background thread, main thread etc.

How to create Observable?

- There are plenty of static methods!
- Just,fromArray,fromCallable,fromFuture,range,...
- Examples:

```
Observable.just(1,2,3,4,5,6,7,8,9,10)

Observable.just(new Integer[]{1,2,3,4,5,6,7,8,9,10})

Observable.fromArray(new Integer[]{1,2,3,4,5,6,7,8,9,10,11,12})

Observable.range(1,10)
```

How to create Observer?

- Create an Observer that listen to Observable
- Observer<T> interface provides below methods:
  - onSubscribe()
  - onNext()
  - onError()
  - onComplete()

Make Subscription

- Make Observer subscribe to Observable so that it can start receiving the data
- There are two methods:
  - subscribeOn()
  - observeOn()
- Both method takes an scheduler as argument

Schedulers

- Schedulers jor role in supporting **multi threading** concept in android applications.
- Some of the most important Schedulers :
  - Schedulers.io()
  - AndroidSchedulers.mainThread()
  - Schedulers.computation()

So Far Now...

```
Observable<String>  animalObservable = Observable.just("Ant","Bee","Cat","Dog","Fox");
Observer<String>    animalObserver = new Observer<String>() {
            @Override
            public void onSubscribe(Disposable d) {
                Log.d(TAG, "onSubscribe");
            }

            @Override
            public void onNext(String s) {
                Log.d(TAG, "Name: " + s);
            }

            @Override
            public void onError(Throwable e) {
                Log.e(TAG, "onError: " + e.getMessage());
            }

            @Override
            public void onComplete() {
                Log.d(TAG, "All items are emitted!");
            }
        };

animalsObservable
                .observeOn(Schedulers.io())
                .subscribeOn(AndroidSchedulers.mainThread())
                .subscribe(animalsObserver);
```

Introducing Some Operators

- Map()
- Filter()
- SwitchMap()
- FlatMap()
- ConcatMap()
- Deboune()
- Zip()

Map() Operator

- It lets you transform every item of the emitted sequence with a specified function.
- http://rxmarbles.com/#map

Filter() Operator

- The filter operator uses a specified function to allow only some items of the source sequence to be emitted.
- http://rxmarbles.com/#filter

FlatMap() and conatMap()

- They merges items emitted by multiple Observables and returns a single Observable.
- FlatMap operator:
  - Preserves the order
  - Asynchronous is not maintained
- ConcatMap operator:
  - Interleaving items
  - Asynchronous

SwitchMap

- Nested Observable
- Discard the response and consider the latest one
- E.X: Googling “Reactive Programming

Debounce Operator

- Debounce operator emits items only when a specified timespan is passed.
- http://rxmarbles.com/#debounce

Zip() Operator

- The zip operator takes multiple observables as inputs and combines each emission via a specified function and emits the results of this function as a new sequence.
- http://rxmarbles.com/#zip

Project : RxJava Instant Search

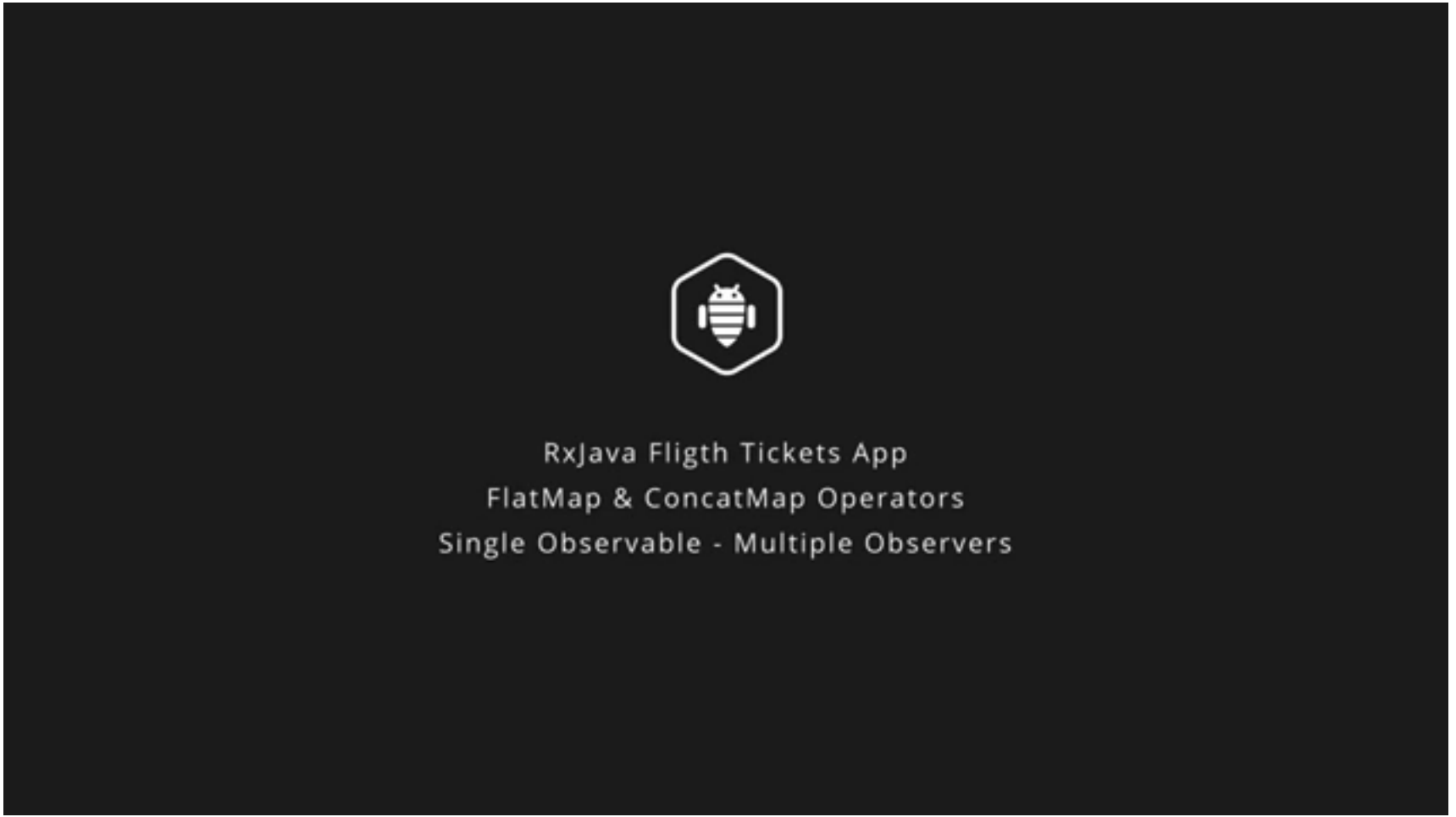
Project : RxJava Instant Search

- **CompositeDisposable** is used to dispose the subscriptions in onDestroy() method.
- **RxTextView.textChangeEvents** Triggers an event whenever the text is changed in search EditText.
- **debounce(300, TimeUnit.MILLISECONDS)** – Emits the search query every 300 milliseconds.
- **distinctUntilChanged()** avoids making same search request again

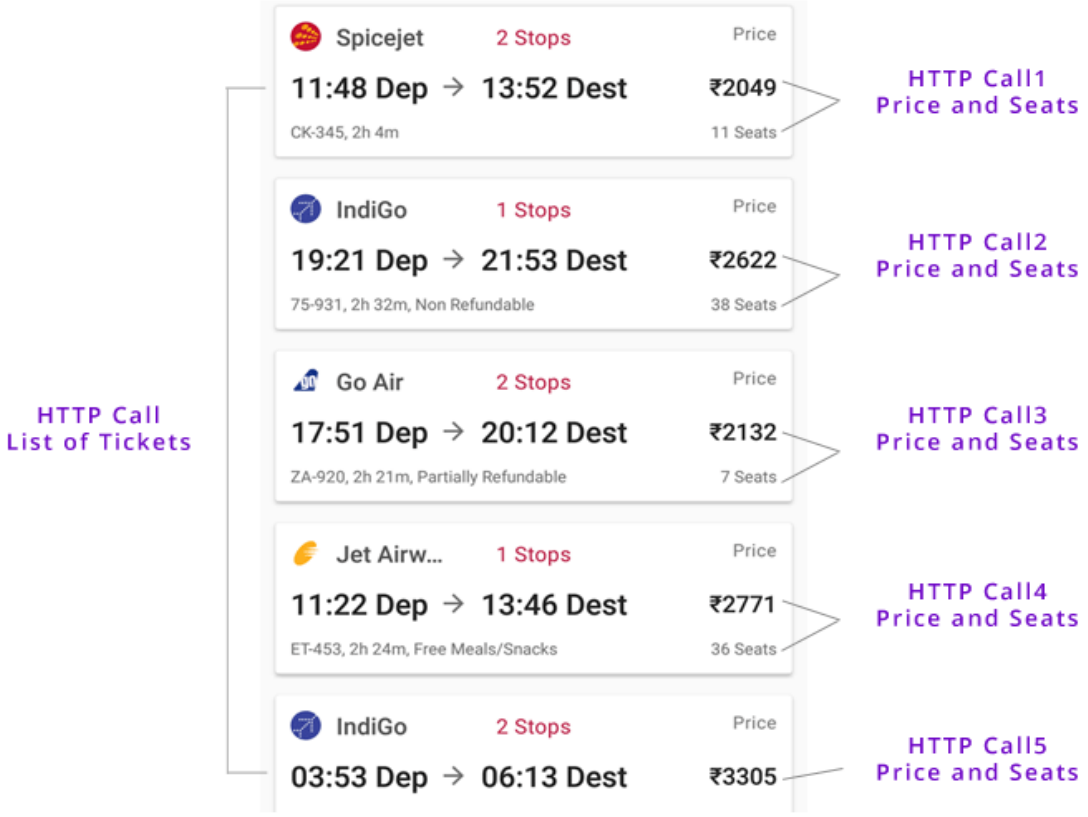
Project : RxJava Instant Search

- **fetchContacts()** fetches all contacts by making Retrofit HTTP call
- **searchContacts()** – Is an Observer that will be called when search query is emitted. By calling mAdapter.getFilter().filter(), the search query will filter the data on ArrayList.

Project : Flight Tickets App



Project : Flight Tickets App



Begin the Journey!

- MVP,MVVM Architecture
- Clean Architecture
- Flowable Backpressure
- Hot vs Cold Observables
- RxJava Subject
- And so many other things.

References

- Www.androidhive.com
- Www.reactivex.io
- Www.medium.com
- Reactive Programming with RxJava by Ben Christensen
- Reactive Java Programming by Andrea Magile
- Exploring RxJava for Android by Jake Wharton