RxJava
Observable streams

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

Reactive programming is a programming paradigm oriented around data flows and the propagation of change.
This means that it should be possible to express static or dynamic data flows with ease in the programming languages used, and that the underlying execution model will automatically propagate changes through the data flow.

```
var a = 10
var b = 5
var y = a*b
println("Product is " + y)
b = 6
println("Product is " + y)
```

Functional reactive programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. `map`, `reduce`, `filter`).

History and generations
by Dávid Karnok

0th generation is when you have `addListener`/`removeListener` and `update()` with or without an actual value.

- `java.util.Observable` and most GUI frameworks

1st generation is what the Microsoft folks invented back in ~2009. It's a step upwards with straightforward architecture and great composability. However, their concept has some shortcomings discovered in late ~2013.

- Rx.NET, Reactive4Java

2nd generation is what RxJava 1.x currently is. It fixes the synchronous cancellation problem, introduced some optional backpressure and the notion of lifting into a chain.

History and generations #2

3rd generation is the Reactive-Streams initiative with fleshed out, standardized APIs, designed ~2015. The architecture is reactive-push with backpressure.

- Reactor 1, 2 and Akka-Streams.

4th generation is the cutting edge of the field. It builds on Reactive-Streams and adds an adaptive push-pull option, in the form of operator-fusion, that allows both efficient synchronous and asynchronous use.

- Reactor 2.5

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

	single items	multiple items
synchronous	`T getData()`	`Iterable<T> getData()`
asynchronous	`Future<T> getData()`	`Observable<T> getData()`

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

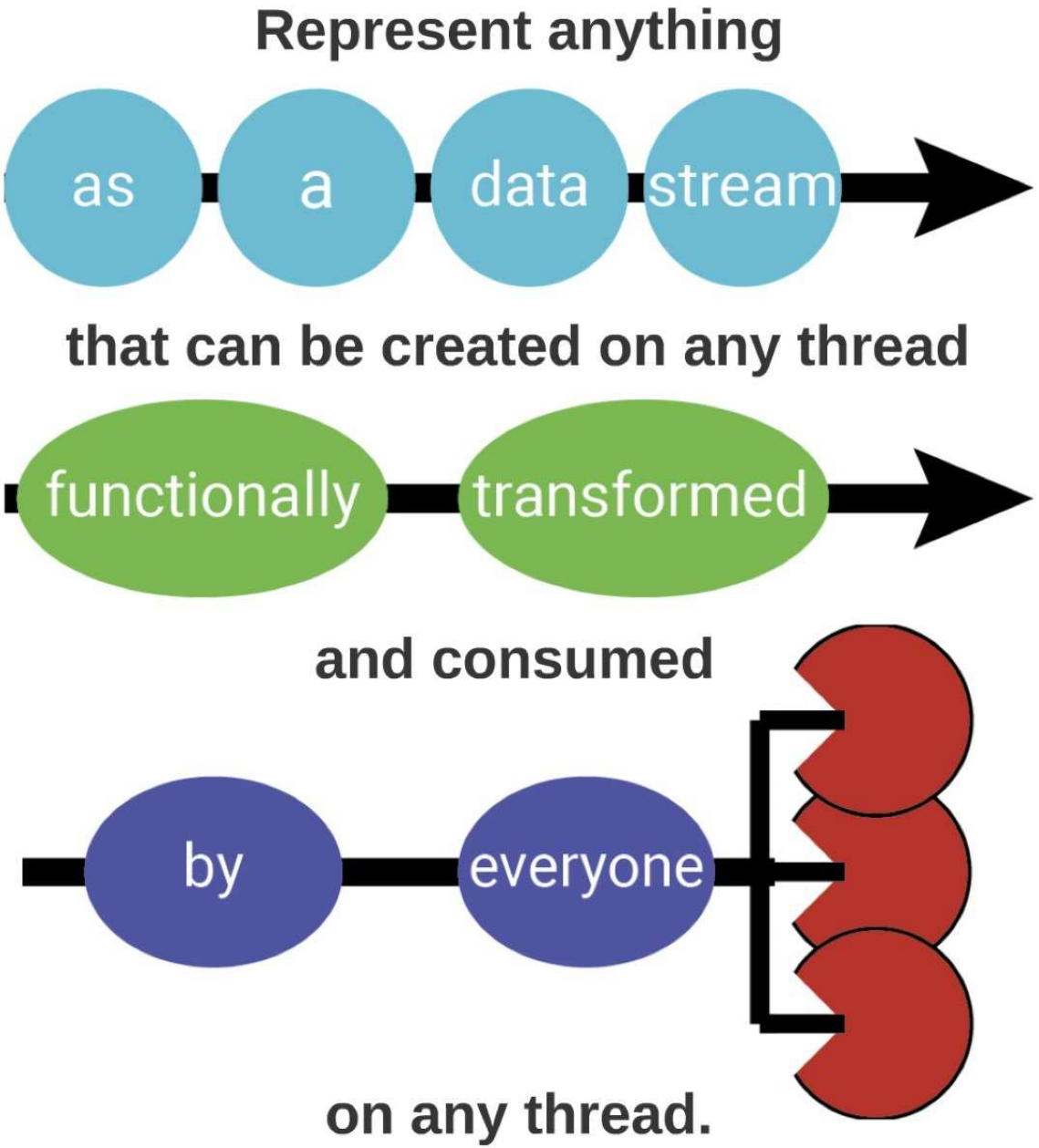
	single items	multiple items
synchronous	`Observable<T> getData()`	`Observable<T> getData()`
asynchronous	`Observable<T> getData()`	`Observable<T> getData()`

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

	single items	multiple items
synchronous	`Observable<T> getData()` `Single<T> getData()`	`Observable<T> getData()`
asynchronous	`Observable<T> getData()` `Single<T> getData()`	`Observable<T> getData()`

An Observable is the asynchronous/push “dual” to the synchronous/pull Iterable

event	Iterabe (pull)	Observable (push)
receive data	`T next()`	`onNext(T)`
discover error	`throws Exception`	`onError(Exception)`
complete	`!hasNext()`	`onCompleted()`


Represent anything
as
a
data
stream
that can be created on any thread
functionally
transformed
and consumed
by
everyone
on any thread.

ReactiveX (Reactive Extensions) is a combination of the best ideas from the Observer pattern, the Iterator pattern, and functional programming.

RxJava

RxJava is a Java VM implementation of Reactive Extensions.

Java 6+ & Android 2.3+

Java 8 lambda support

Polyglot (Scala, Groovy, Clojure and Kotlin)

Links

https://github.com/ReactiveX/RxJava

http://reactivex.io/documentation/observable.html

http://reactivex.io/tutorials.html

http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/

https://www.youtube.com/watch?v=QdmkXL7XikQ

https://github.com/google/agera/issues/20