RxJava

Thinking in RX

Ivan Drizhiruk

What users want?

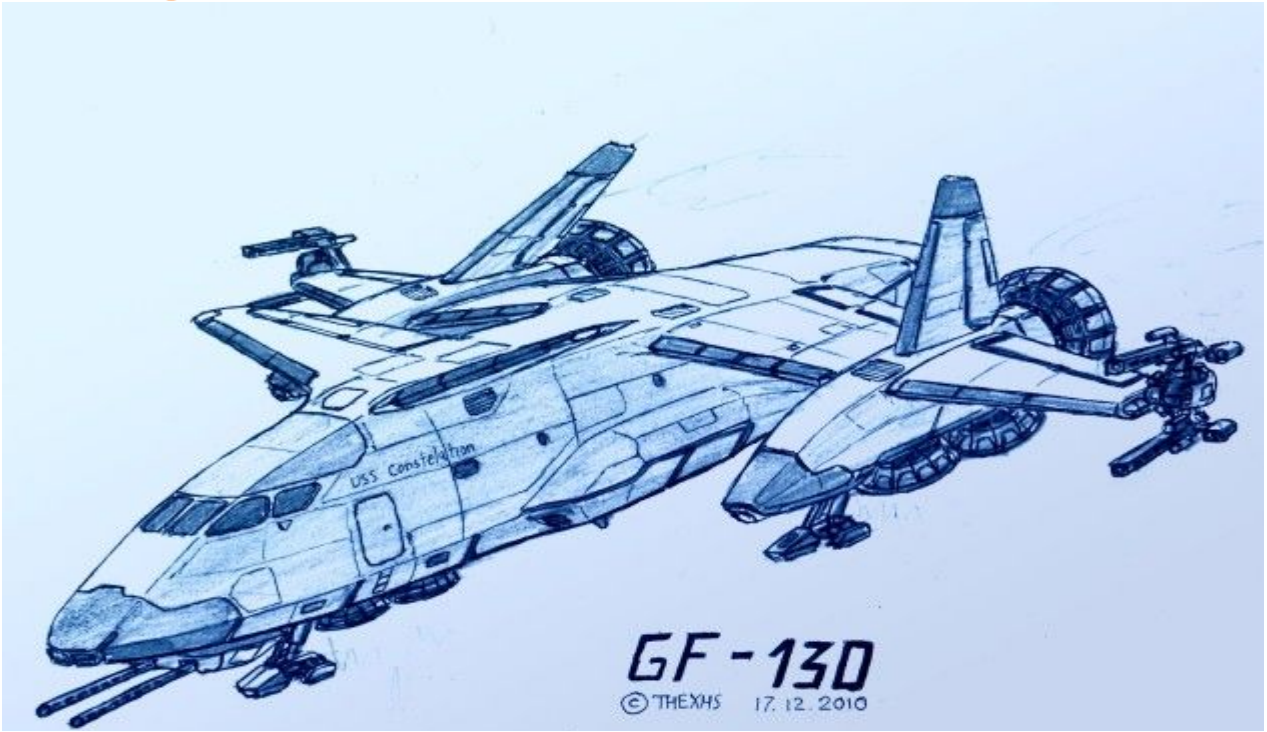
Give me data and now

AMY SCHWARTZ

I Can't WAIT!

I do not like to wait

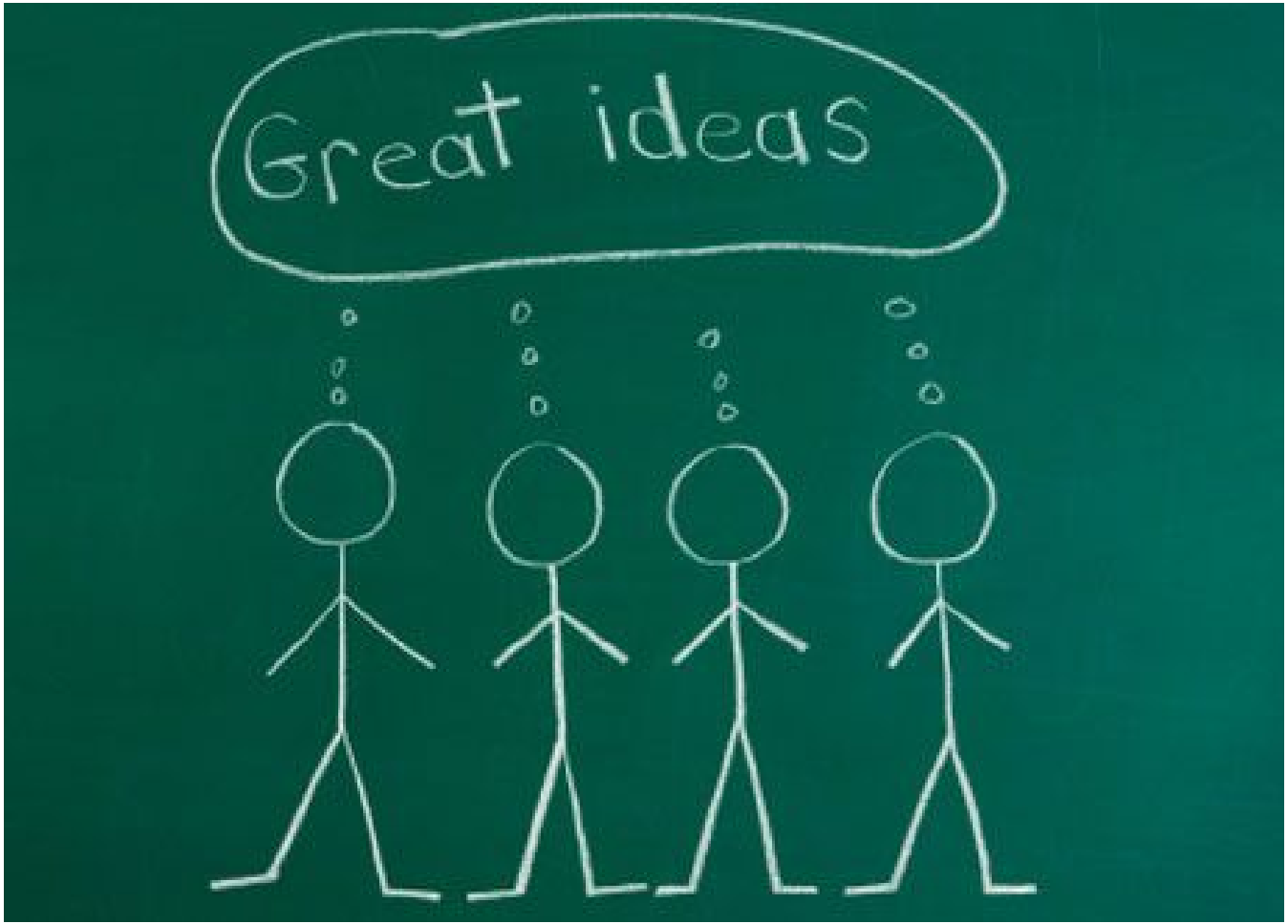
So the program must follow the following criteria



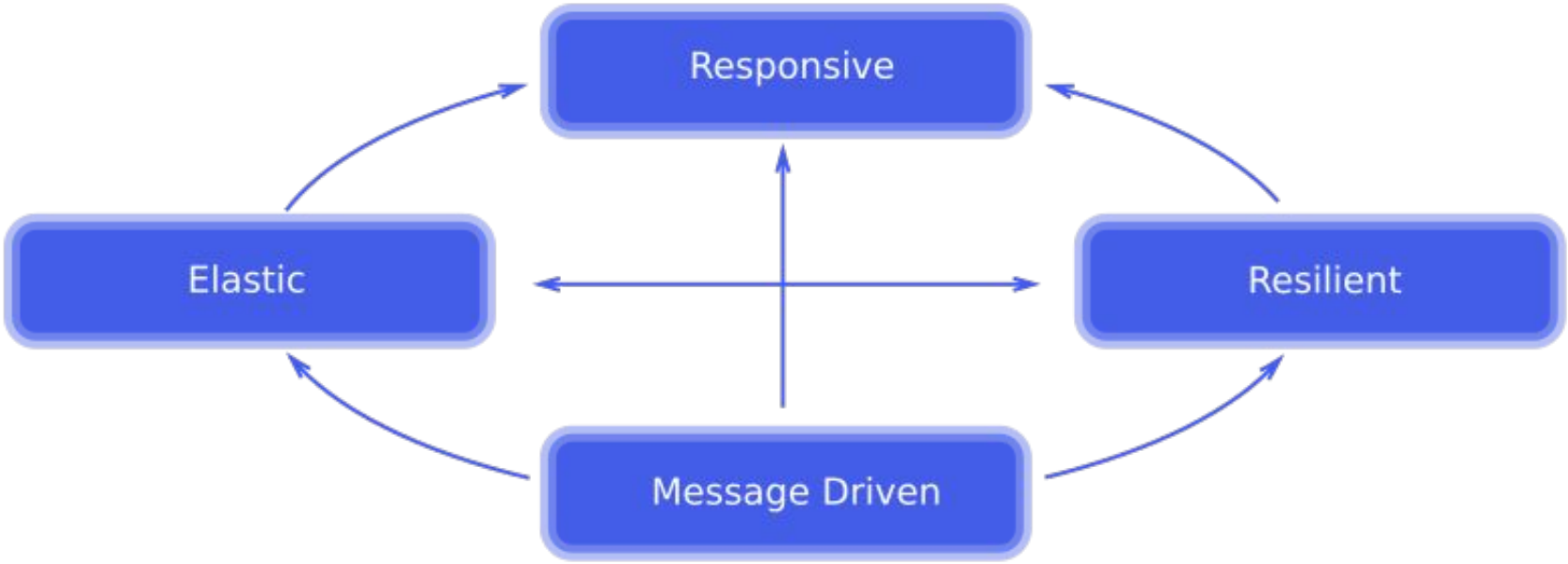
And programmers should understand requirements and choose best approach

Requirements to programs

- Reactivity
- Parallelizable
- Non blocking
- Composable
- Readable


Great ideas

The Reactive Manifesto



http://www.reactivemanifesto.org/

Publishing history:

- V1.0 - July 15th of 2013.
- V1.1 - September 23th of 2013.
- V2.0 - September 16th of 2014.

Asynchronous approach

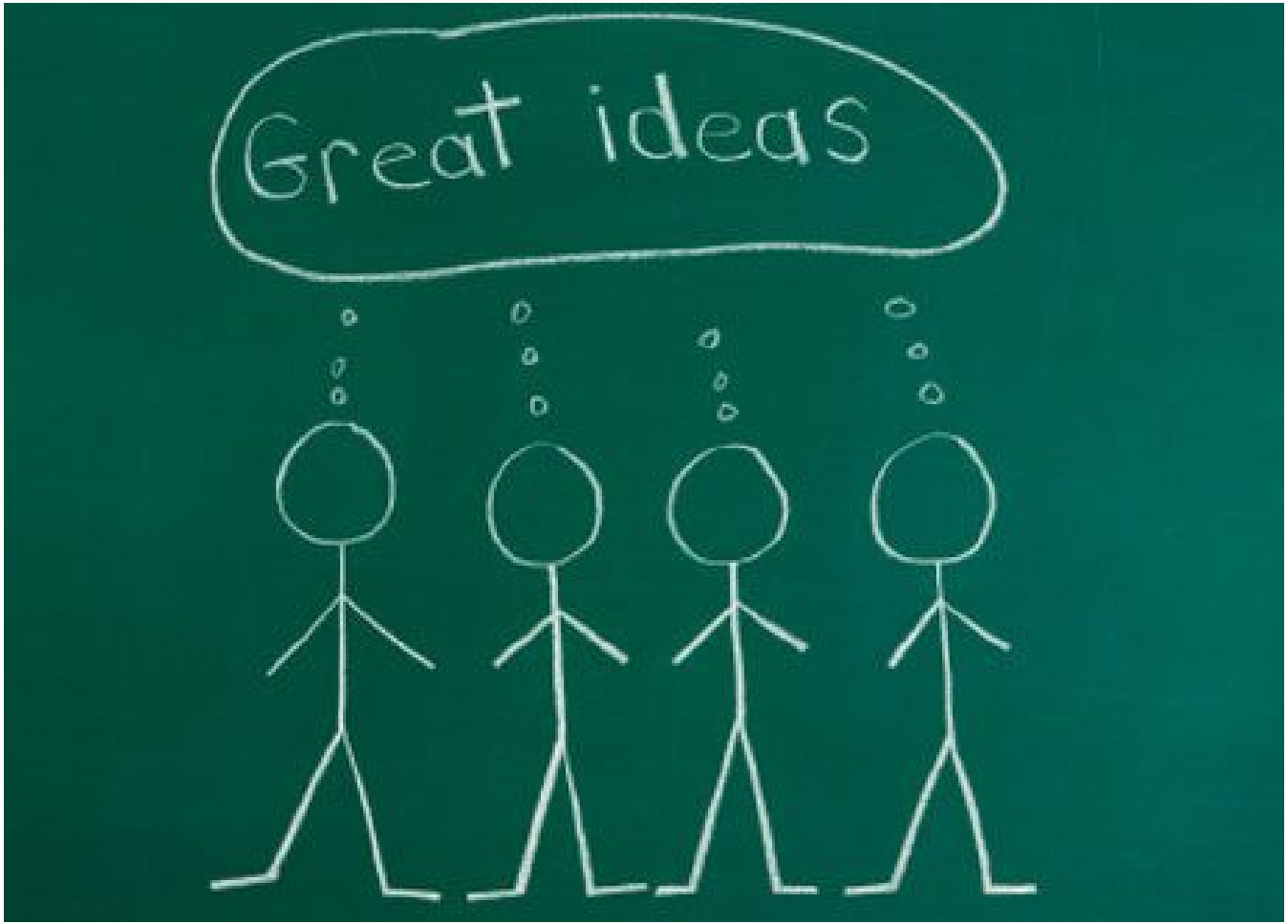
What we have?

Callback

- No composition
- Callback Hell

Futures

- Future.get() - blocks till all threads are complete
- CompletableFuture .supplyAsync(task) .thenAccept(action) - what if the tasks need to fetch millions of records?


Great ideas

Reactive Extensions

Reactive Extensions

Languages

- Java: RxJava
- JavaScript: RxJS
- C#: Rx.NET
- C#(Unity): UniRx
- Scala: RxScala
- Clojure: RxClojure
- C++: RxCpp
- Ruby: Rx.rb
- Python: RxPY
- Groovy: RxGroovy
- JRuby: RxJRuby
- Kotlin: RxKotlin
- Swift: RxSwift
- PHP: RxPHP

ReactiveX for platforms and frameworks

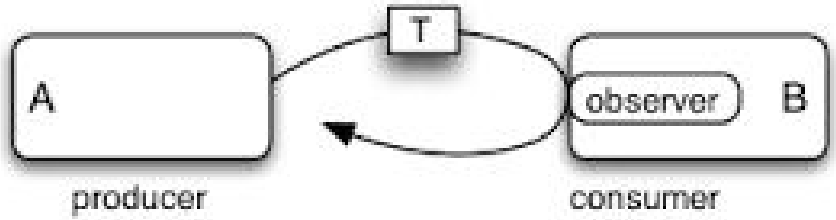
- RxNetty
- RxAndroid
- RxCocoa

http://reactivex.io/ -
- common standards

http://www.reactive-streams.org/
- standardize on the jvm

RxJava

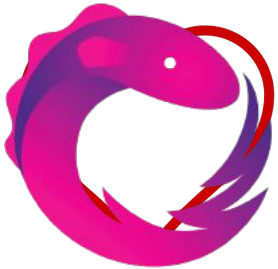
Observer



+

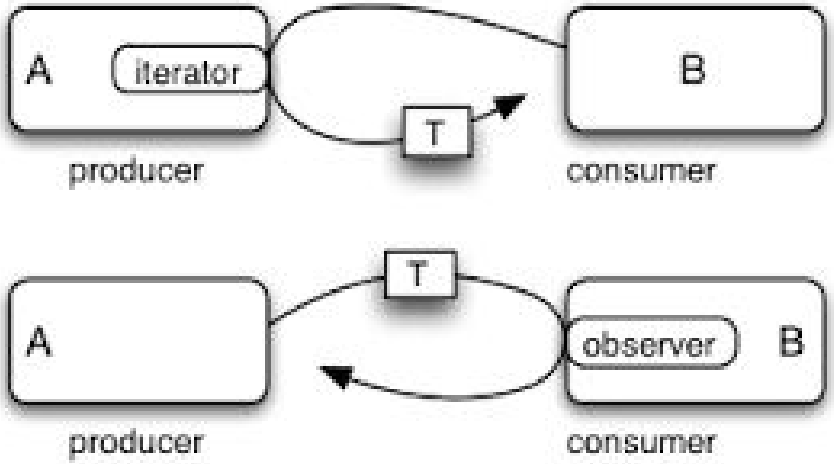
Steroids

=



It is like

Iterable - Iterator “Pull”



Observable - Observer “Push”

Main RxJava players

- Observable - producer of data
- Observer - consumer of the data
- Subscriber - connects observer with observable
- Operator - some action on data

Observer

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

Observable<T>

```
public final Subscription subscribe(
    Observer<? super T> observer)
```

How to use

```
Observer observer = new Observer() {
    public void onCompleted() { println("=> Completed"); }
    public void onError(Throwable th) { println(th); }
    public void onNext(Object obj) { println(obj); }
};

Observable observable = Observable.just(
    "Java", "Javascript", "C#", "Go", "Scala");

observable.subscribe(observer);
```

Operations

- Create
- Transformation
- Filter
- Count
- SideEffect
- Combine
- Live & Let Die and Retry

Create operators

Observable

- just
- from
- create

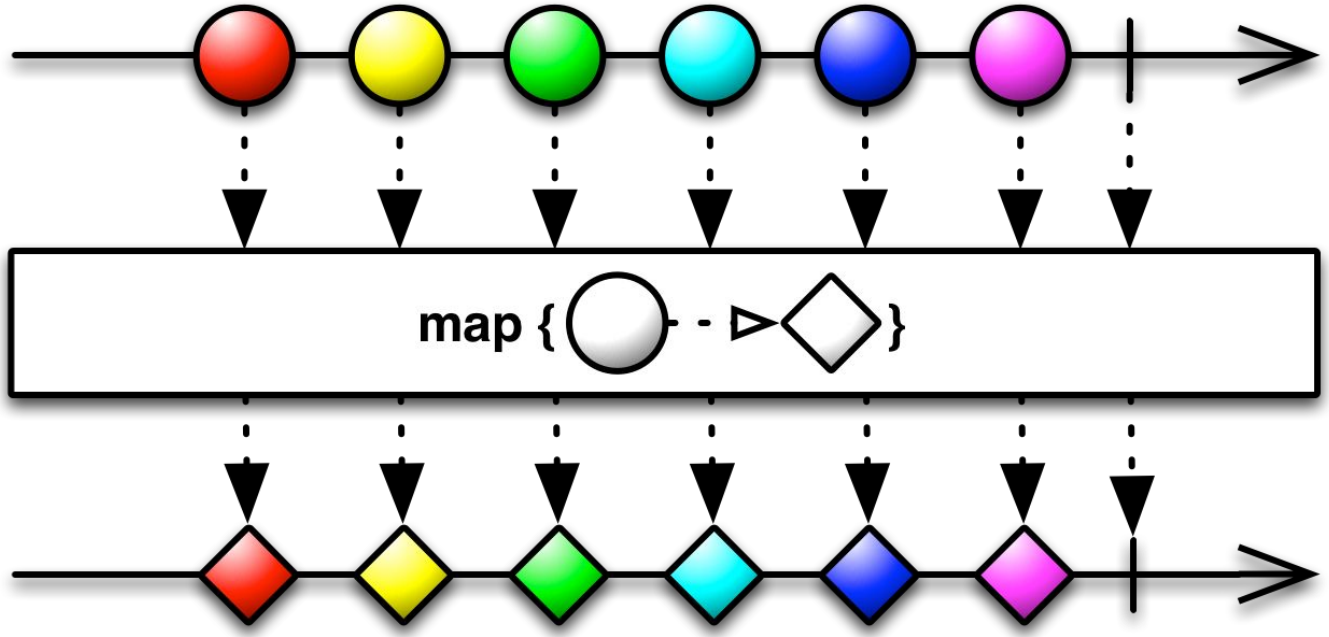
Transformation operators

Observable

- map
- single
- singleOrDefault
- flatMap
- reduce

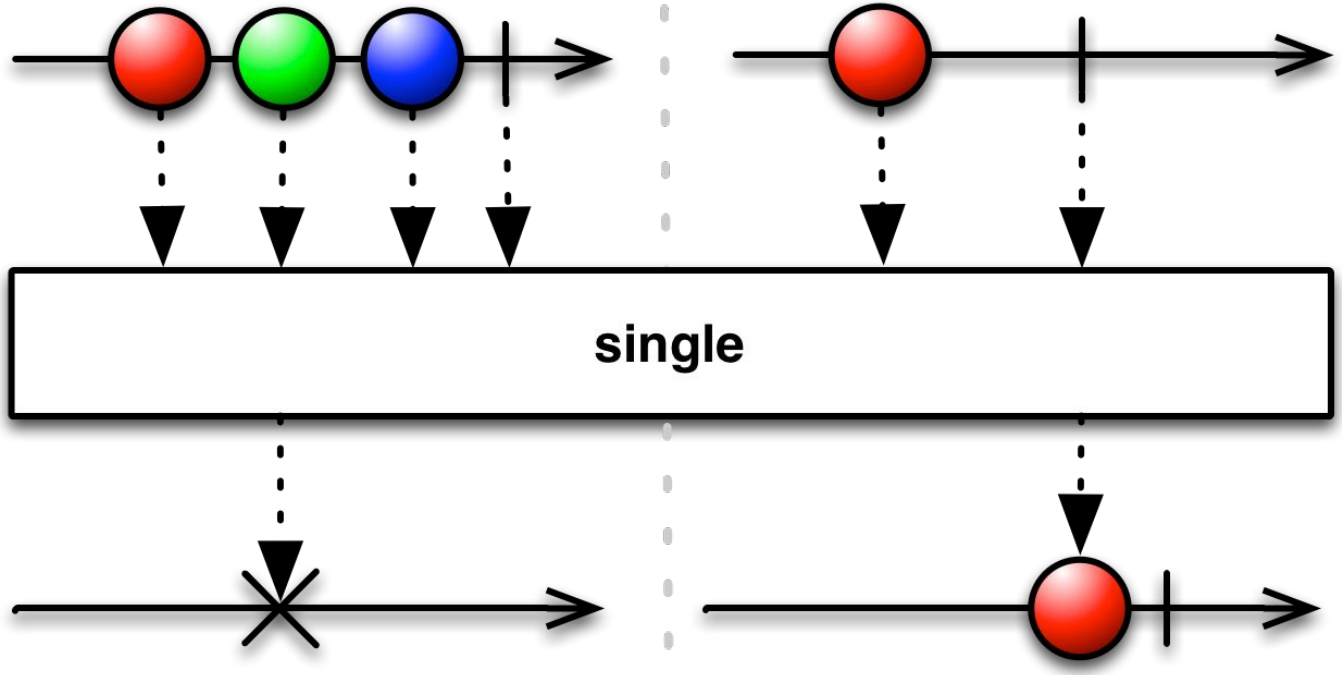
Transformation operators

Observable.map



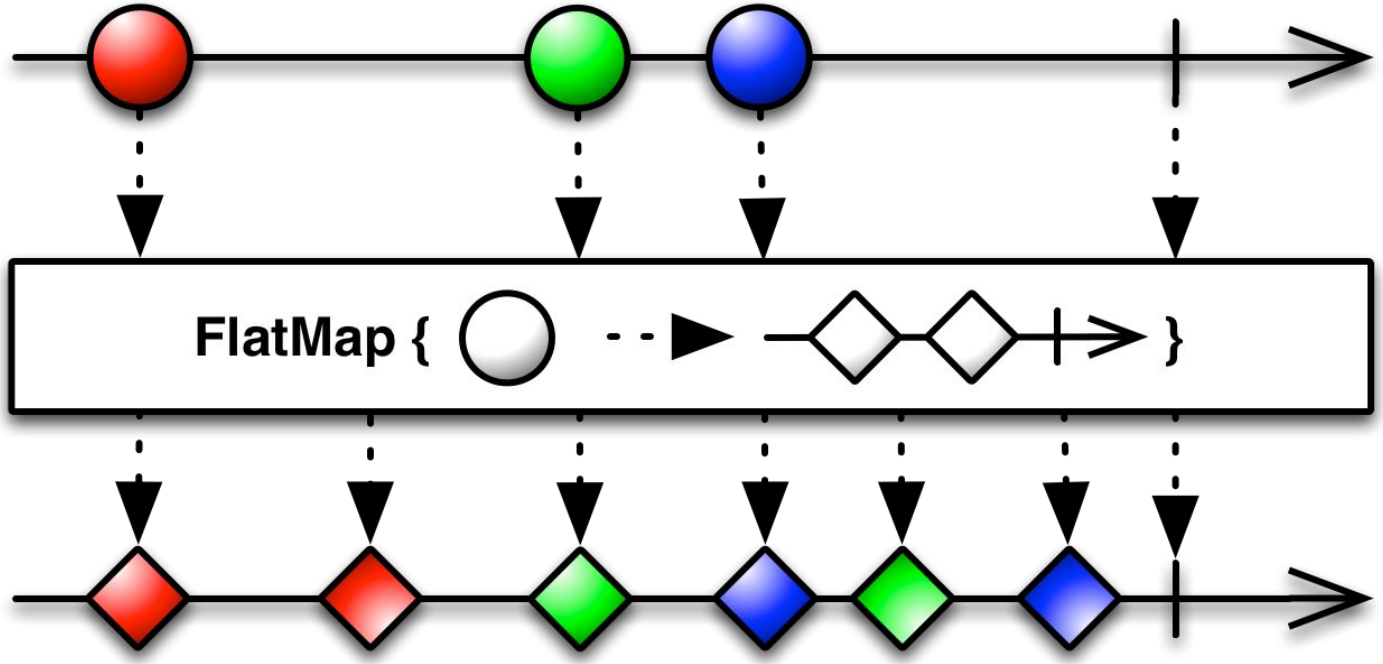
Transformation operators

Observable.single



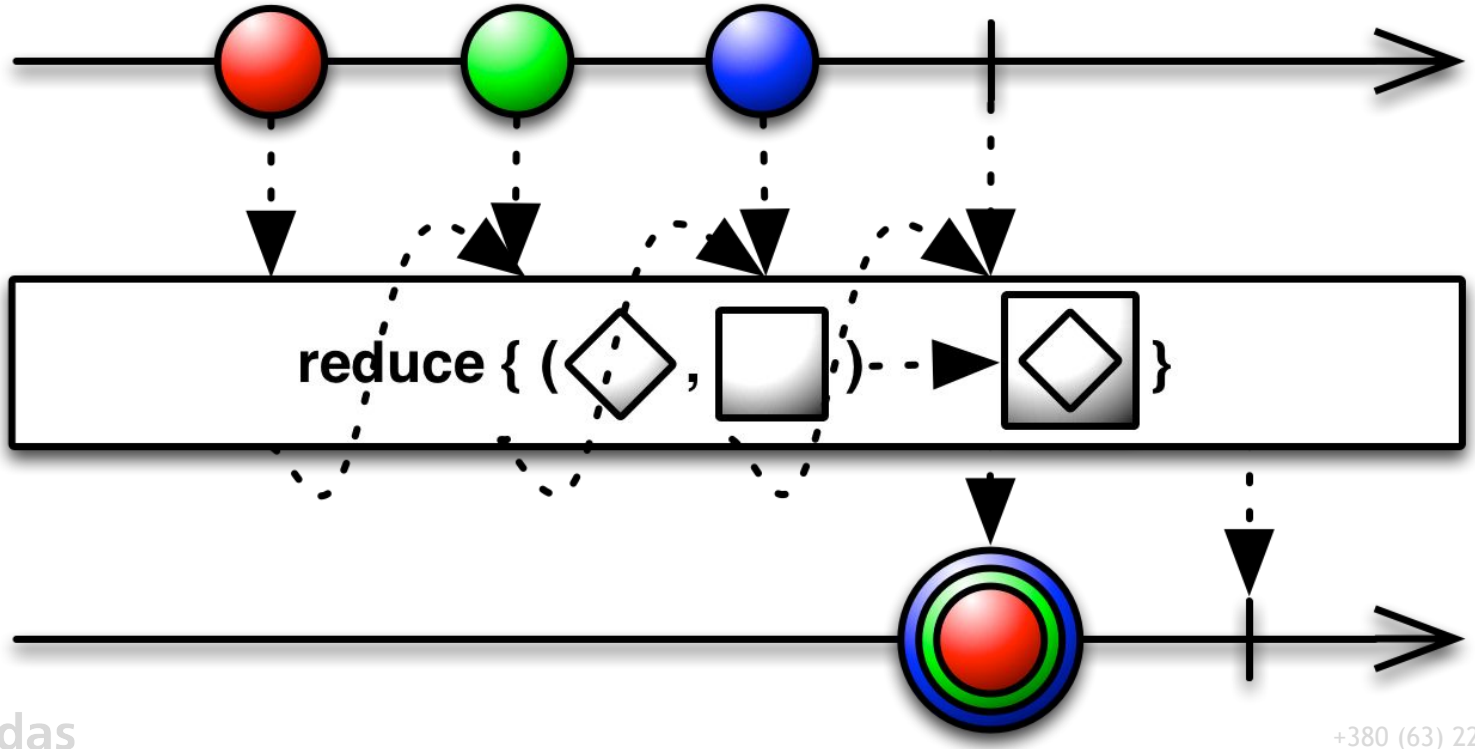
Transformation operators

Observable.flatMap



Transformation operators

Observable.reduce



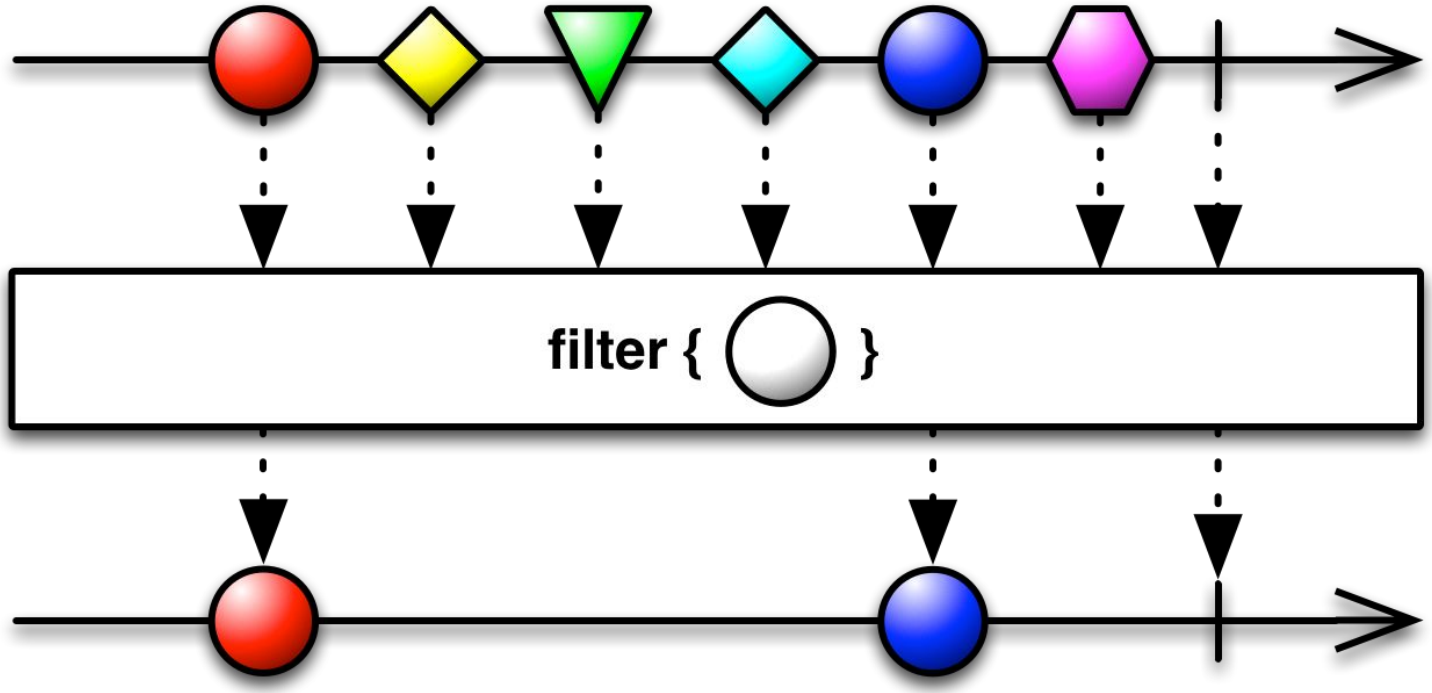
Filter operators

Observable

- filter
- take
- takeUntil

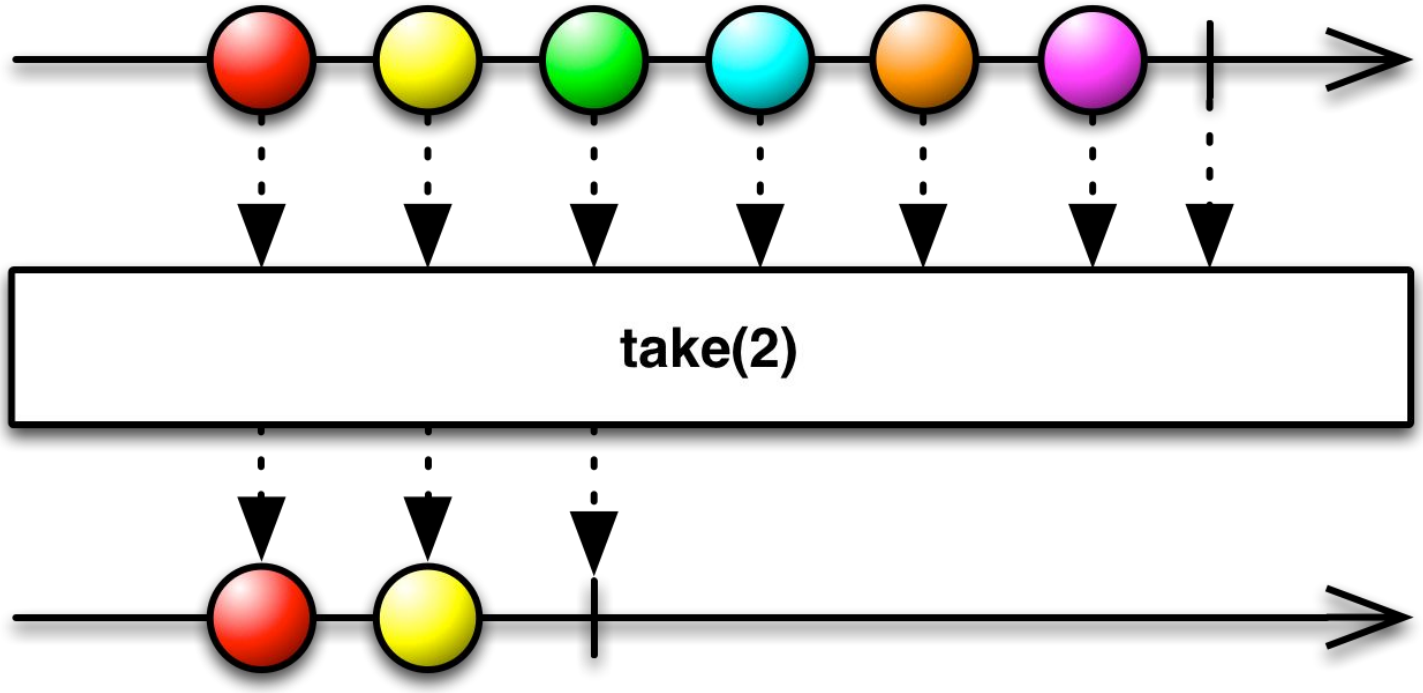
Filter operators

Observable.filter



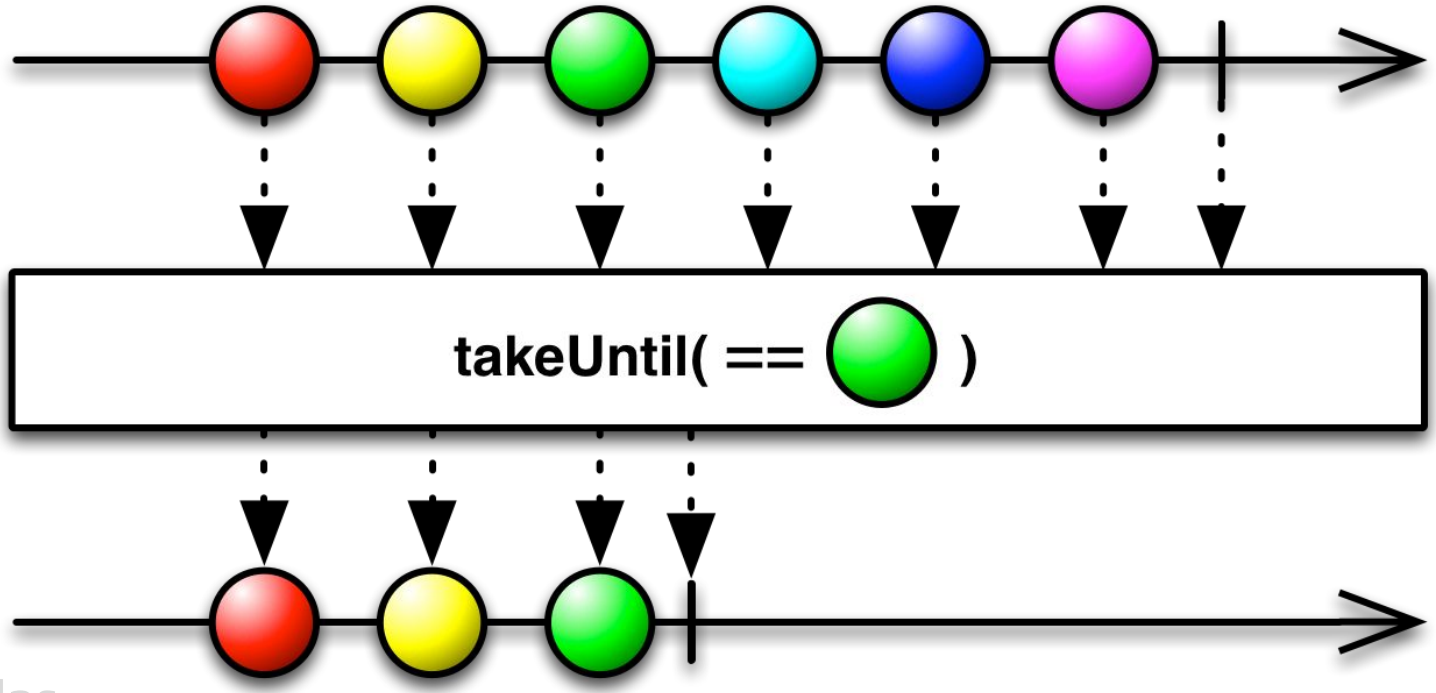
Filter operators

Observable.take



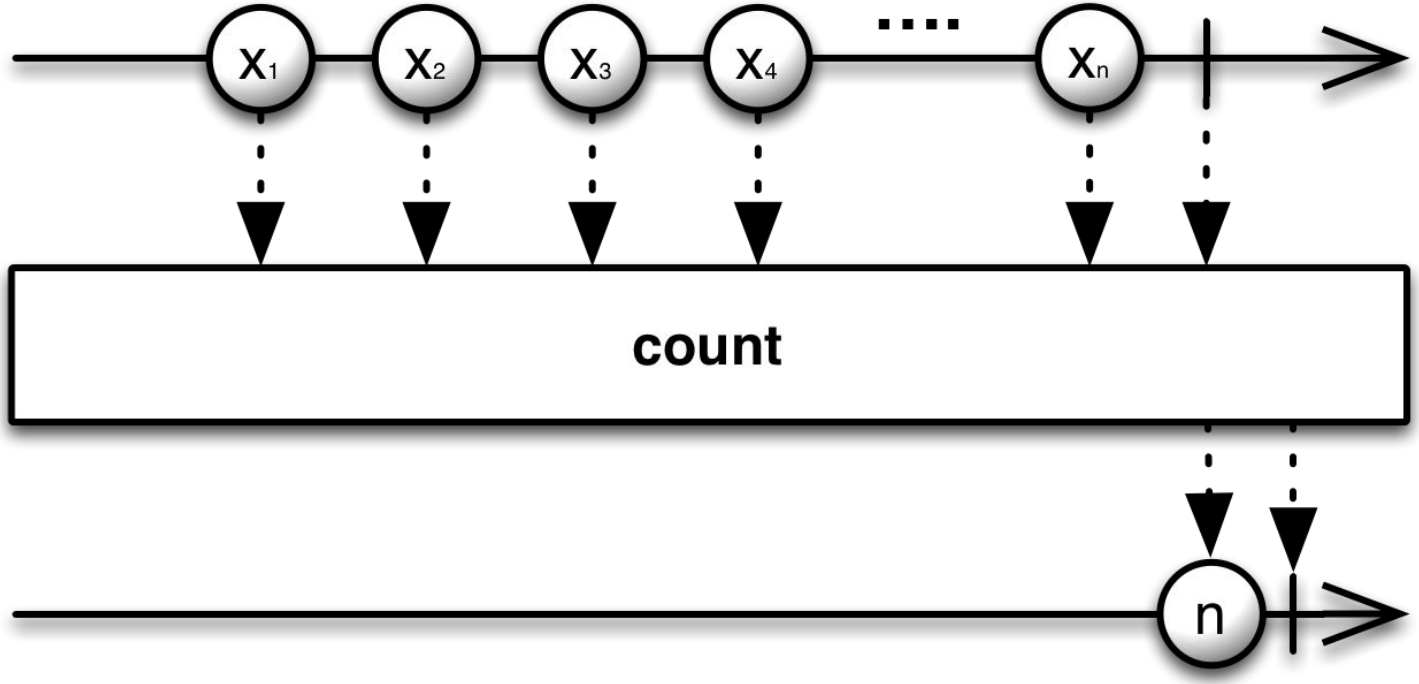
Filter operators

Observable.takeUntil



Count operators

Observable.count



Side effects operators

Observable.doOnXXX

- doOnNext
- doOnError
- doOnCompleted
- doOnEach

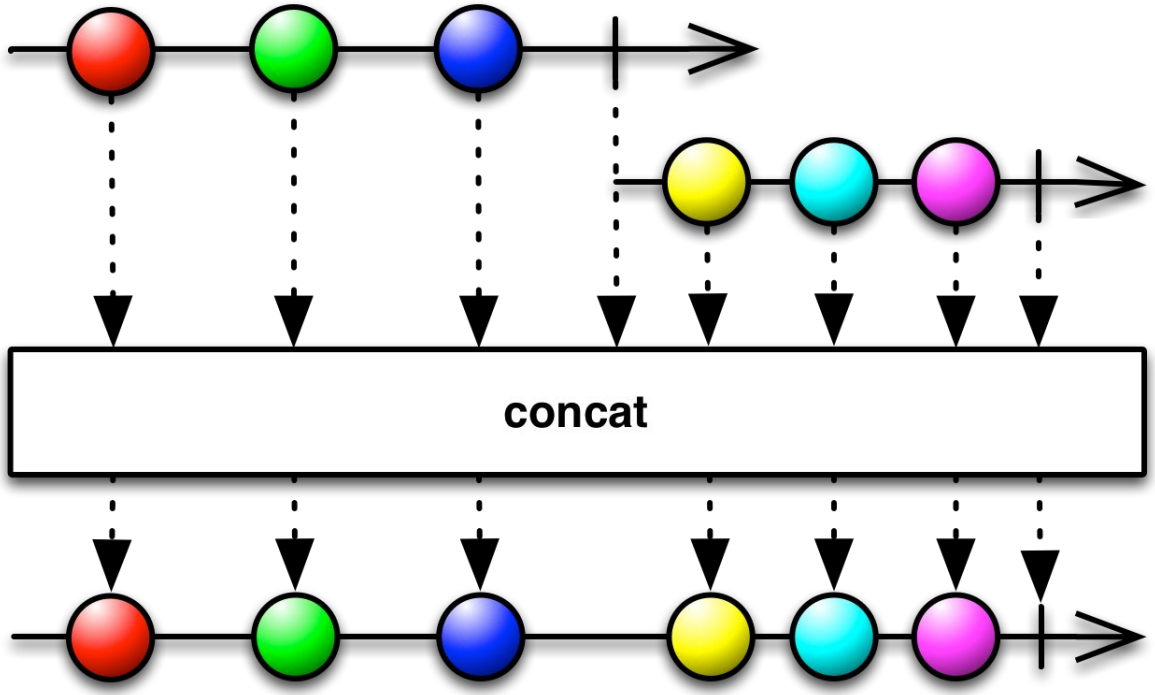
Combine operators

Observable

- concatWith
- mergeWith
- zipWith

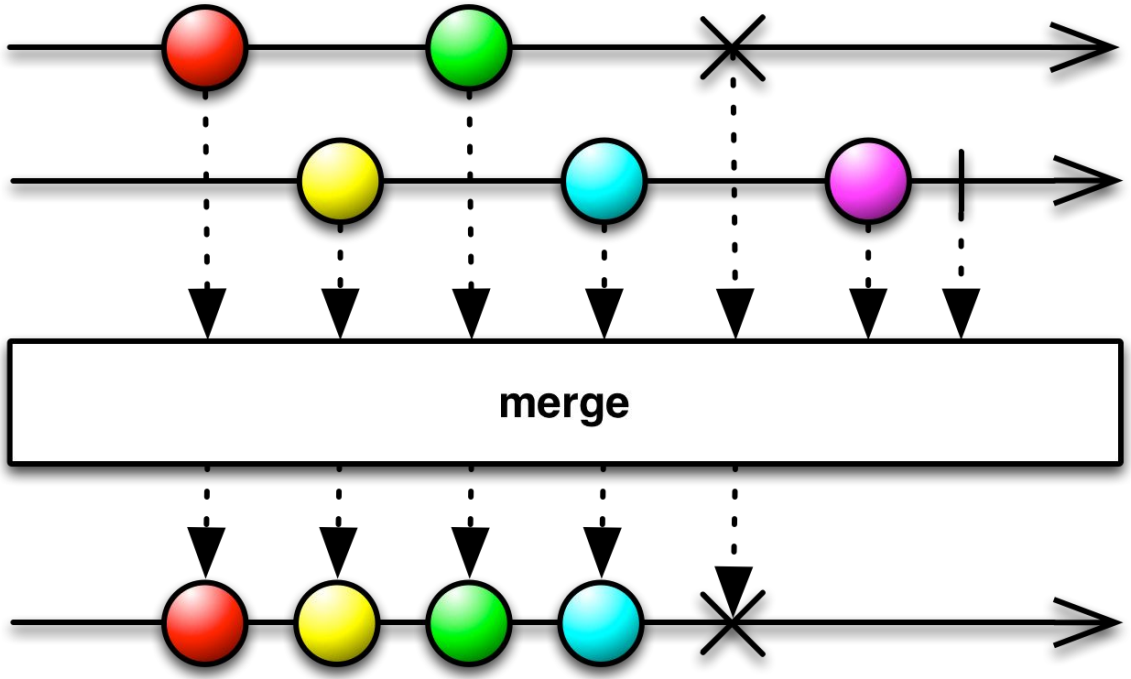
Combine operators

Observable.concatWith



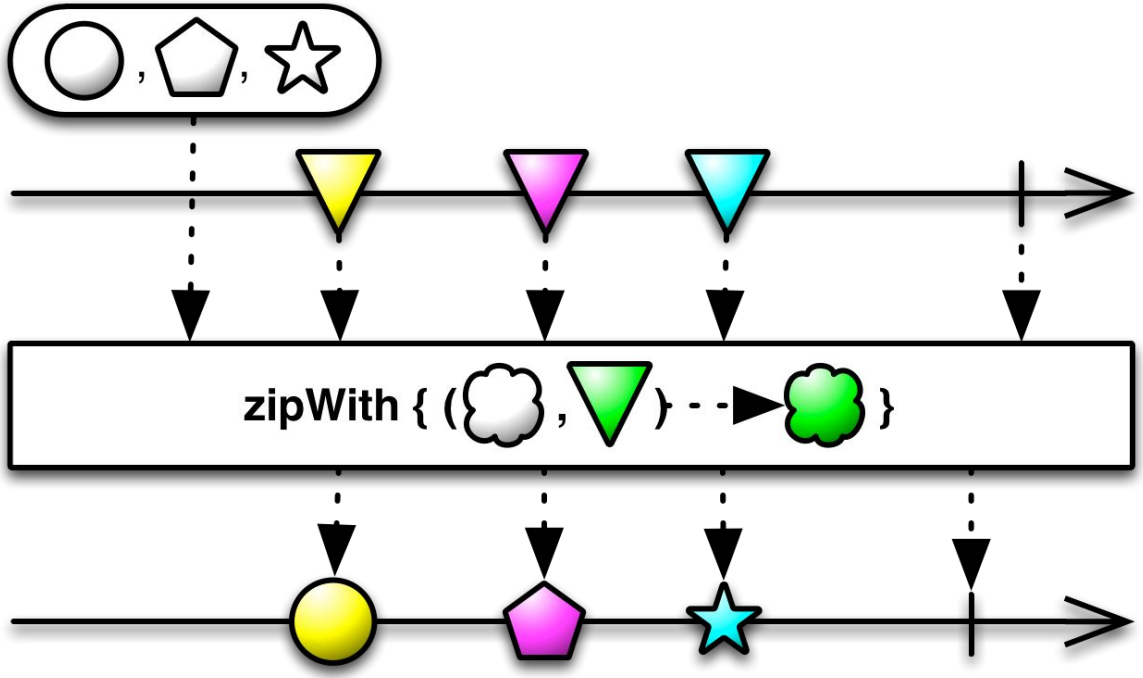
Combine operators

Observable.mergeWith



Combine operators

Observable.zipWith



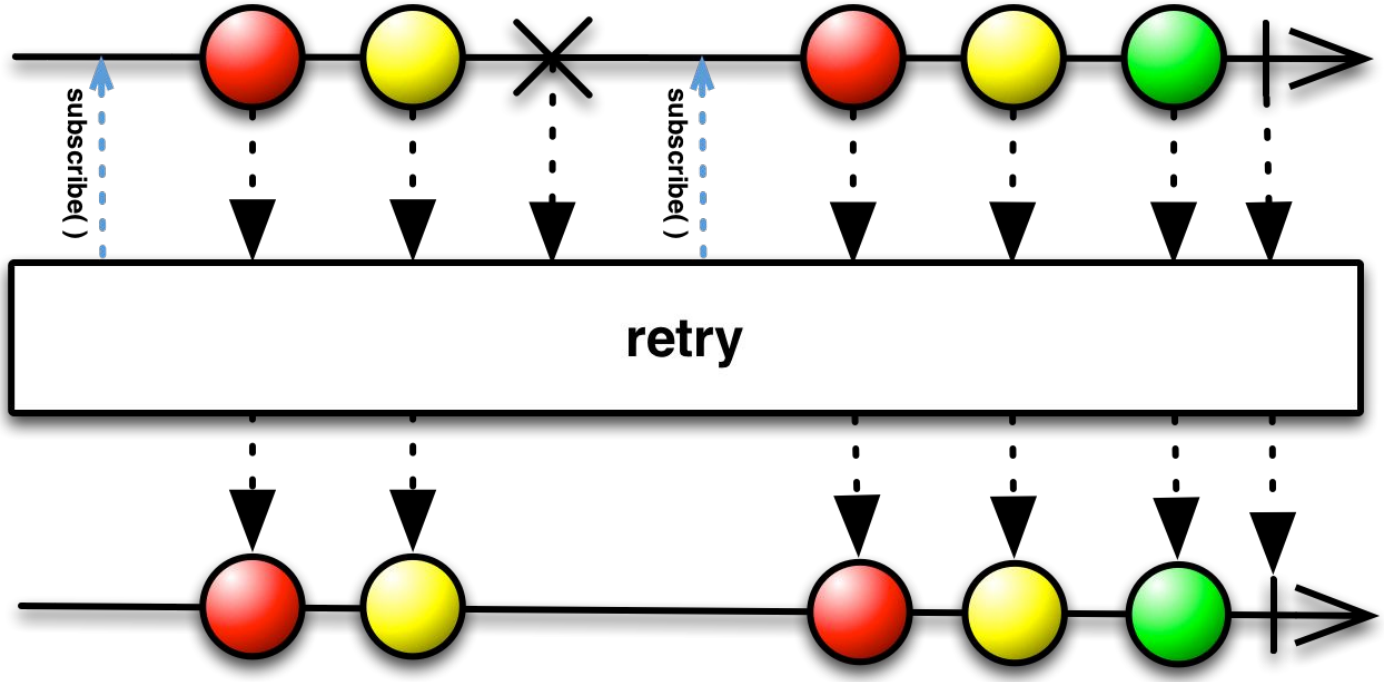
Live & Let Die and Retry operators

Observable

- retry
- onErrorReturn

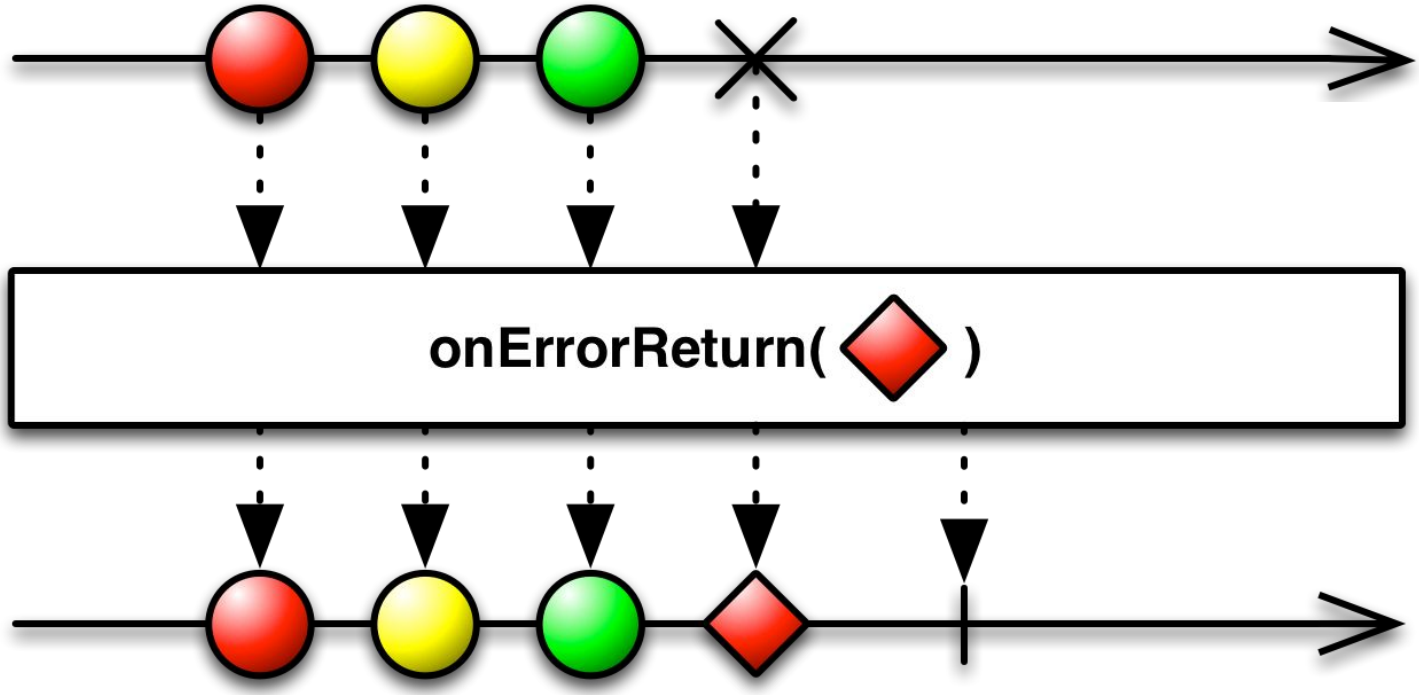
Live & Let Die and Retry operators

Observable.retry



Live & Let Die and Retry operators

Observable.onErrorReturn



Multithreading with Schedulers

Observable

- subscribeOn(Schedulers)
- observeOn(Schedulers)

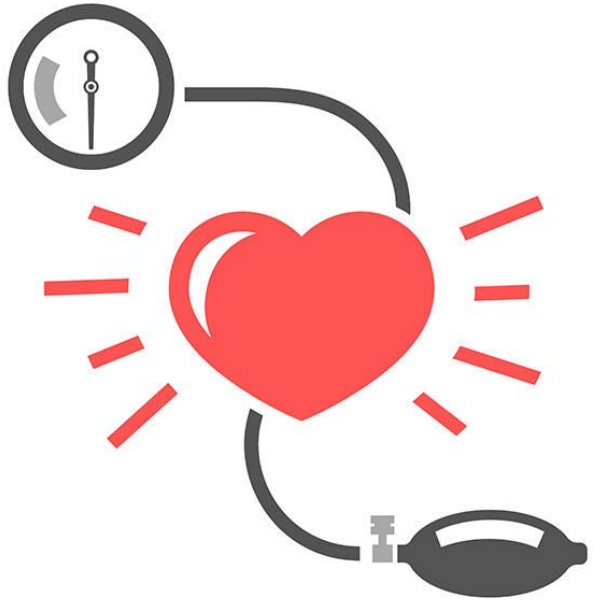
Schedulers

- immediate - executes work immediately on the current thread.
- trampoline - executes work on the current thread (after the current work)
- newThread - creates a new Thread for each unit of work
- computation - intended for computational work (depend from N of cores)
- io - intended for IO-bound work


PROGRAMMER
</CODE>

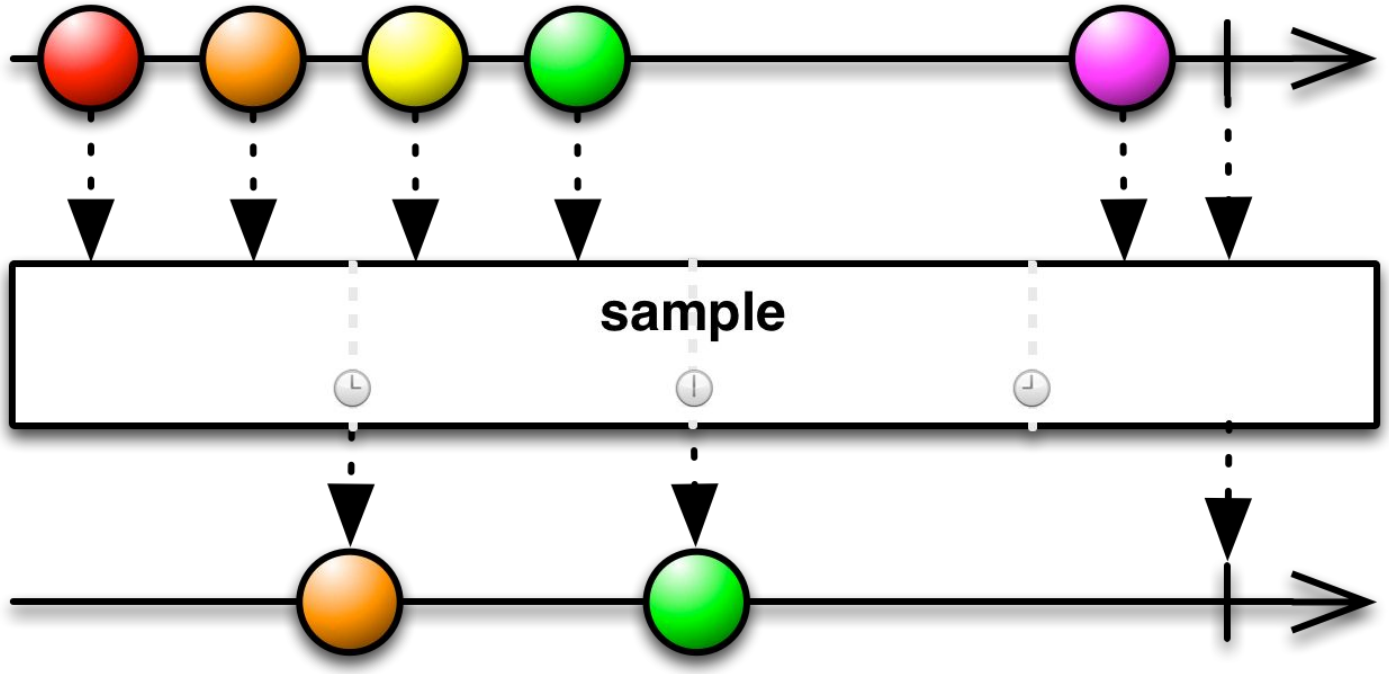
Back pressure

- Throuttle
  - throttleLast( ) or sample( )
  - throttleFirst( )
  - throttleWithTimeout( ) or debounce( )
- Buffer
- Windows



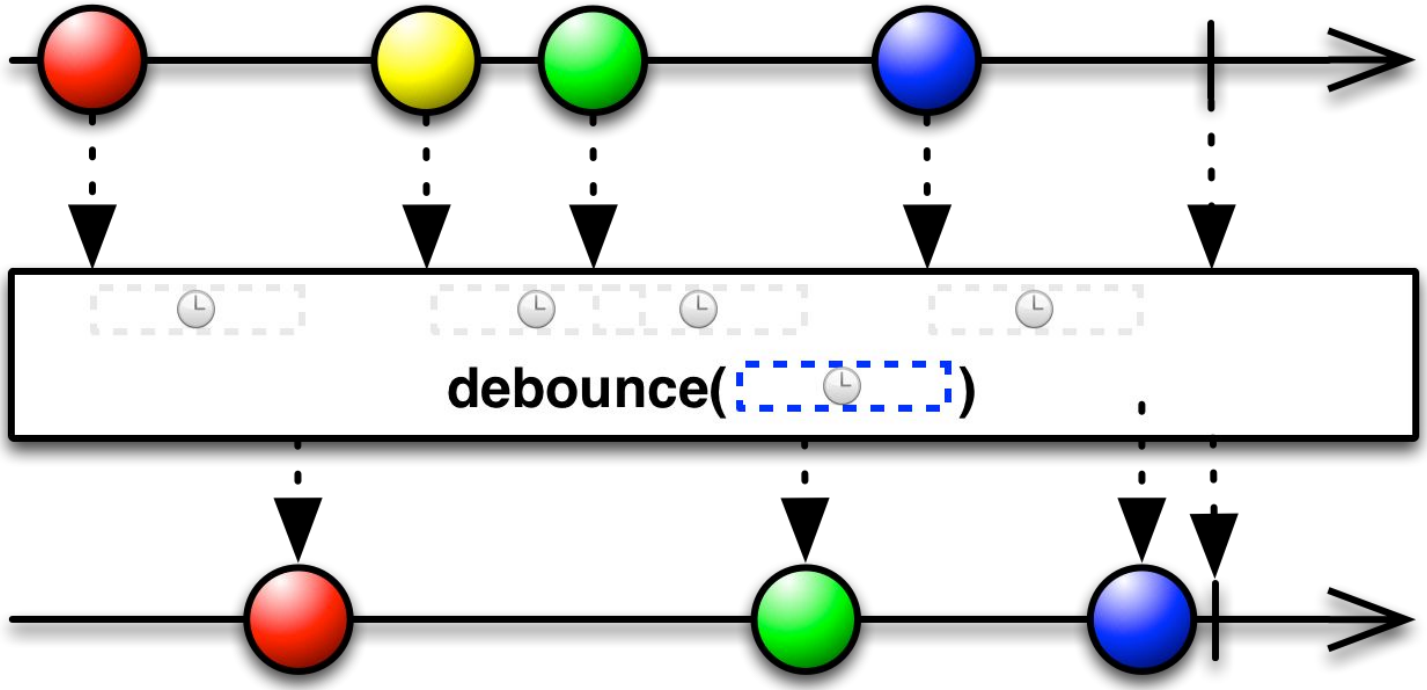
Back pressure

Observable.sample( ) or Observable.throttleLast( )



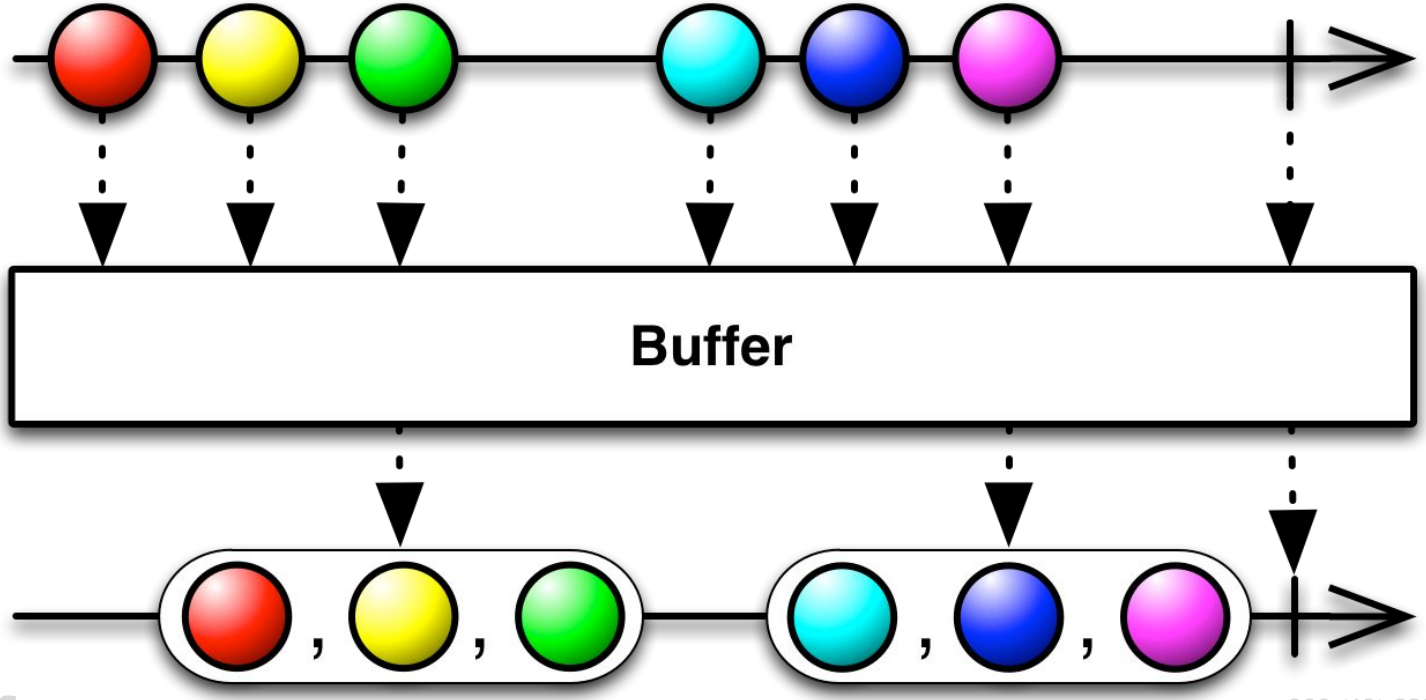
Back pressure

Observable.debounce( ) or Observable.throttleWithTimeout( )



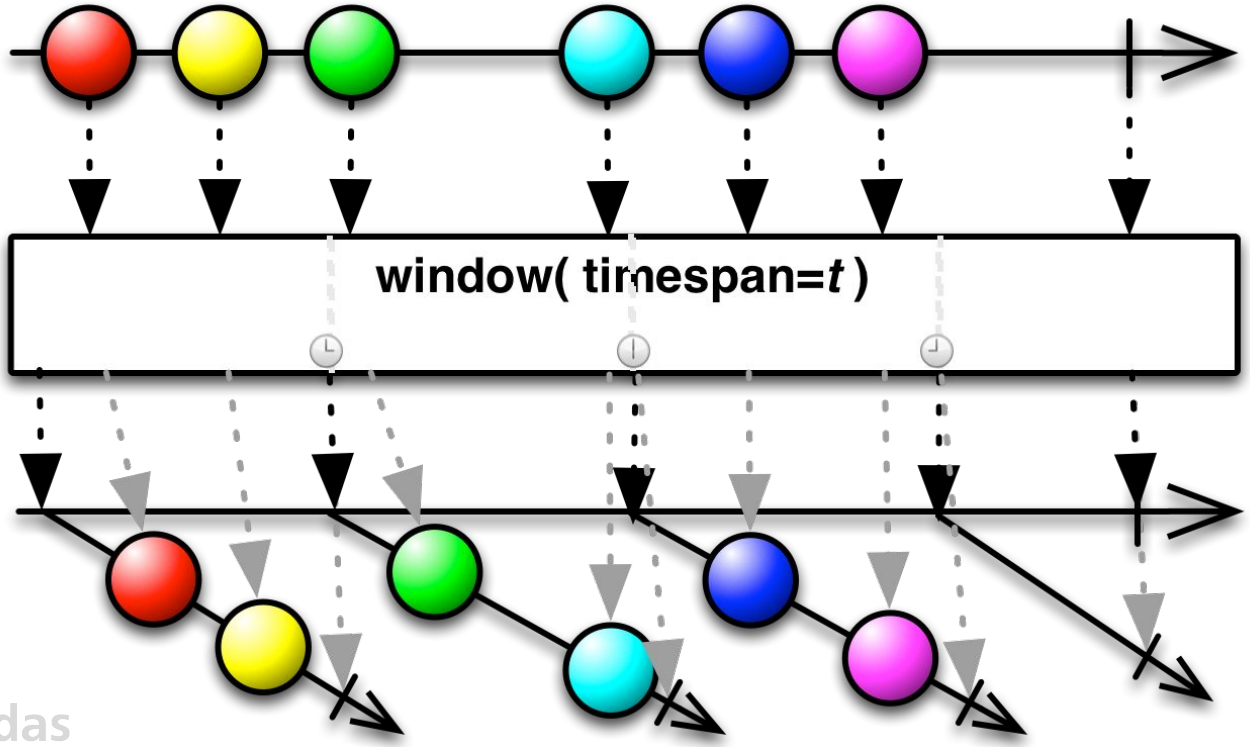
Back pressure

Observable.buffer( )



Back pressure

Observable.windows( )



Benefits

- Functional style of coding
- Intuitive understanding code
- Extensibility
- Composable
- Multithreading
- Nonblocking code
- Readable and understandable

When to use?

Should be used

- in anywhere you use events internally
- any situation when you need to run your code asynchronously

Should not be used

- To use for the sake of use

We use ReactiveX


USE REACTIVE PROGRAMMING
YOU MUST
memegenerator.net

Q & A

Materials

- http://jeeconf.com/program/reactive-thinking-in-java/
- https://www.youtube.com/watch?v=9o9dhWzOTa8
- https://habrahabr.ru/post/269417/
- https://www.google.com.ua/search?q=RXJava

This presentation materials

- https://github.com/IvanDrizhiruk/RXJava