Streams, Streams Everywhere!

An Introduction to Rx

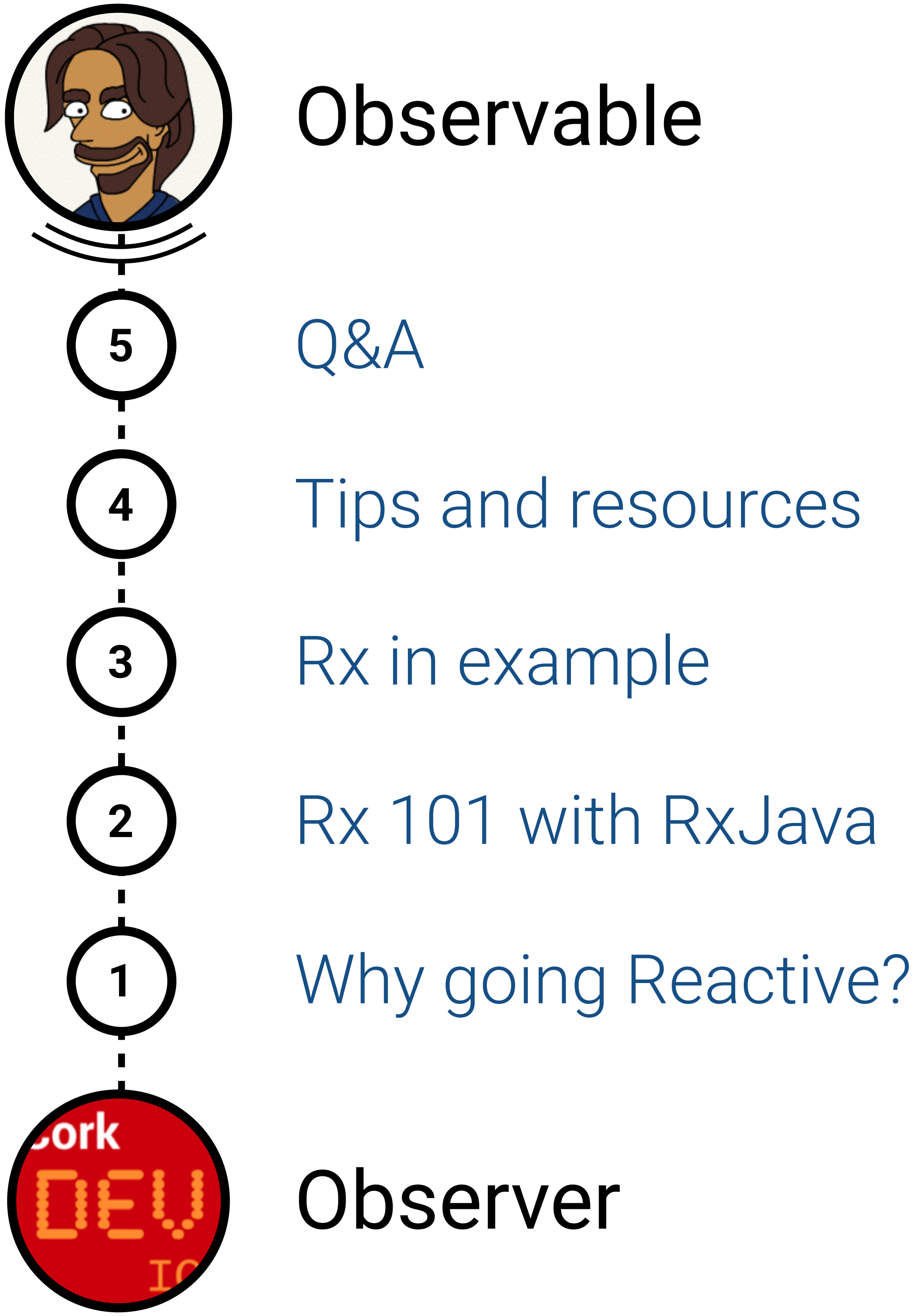
Andrzej Sitek

Corkdev.io - 15th December 2015

Let me introduce myself

```
<author>
   <name>Andrzej Sitek</name>
   <title>Android Developer</title>
   <company>Digisoft.tv / Progressive Apps</company>

   <email>andy@progressiveapps.co</email>
   <twitter>@andrzej_sitek</twitter>
   <google-plus>+AndrzejSitek</google-plus>
</author>
```

“Reactive” presentation ~~plan~~ *stream



Observable

5 Q&A

4 Tips and resources

3 Rx in example

2 Rx 101 with RxJava

1 Why going Reactive?

Observer

Why going Reactive?

Some driving factors:

- Users expect **real time** data
- **No-one wants to be blocked** waiting for results
- When working with results sets it is better to start processing individual results when they are ready
- The world has moved to **push** data model
- **Async programming** is very important but at the same time it’s often **difficult** and **error-prone**

Why going Reactive?

1

Reactive:

- A buzz word
- Pronunciation: /rɪ'aktɪv/
- ***“Showing a response to a stimulus.”***
- ***“Acting in response to a situation rather than creating or controlling it.”***

Why going Reactive?

Reactive manifesto:

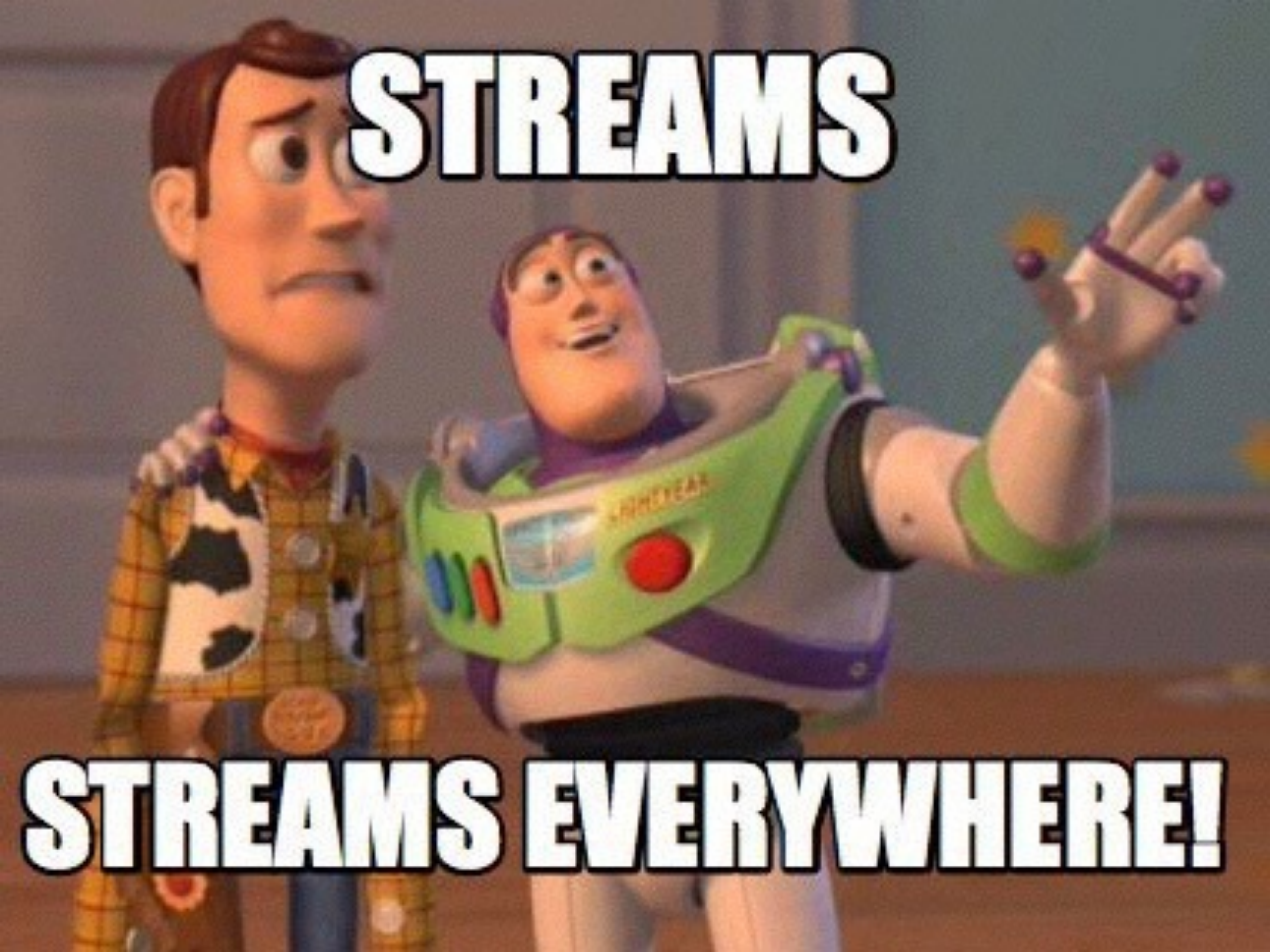
- **Responsive**:
  The system responds in a timely manner if at all possible

- **Resilient**:
  The system stays responsive in the face of failure

- **Elastic**:
  The system stays responsive under varying workload

- **Message Driven**:
  Reactive Systems rely on asynchronous message-passing

www.reactivemanifesto.org


STREAMS
STREAMS EVERYWHERE!

Why going Reactive?

Streams, streams everywhere!

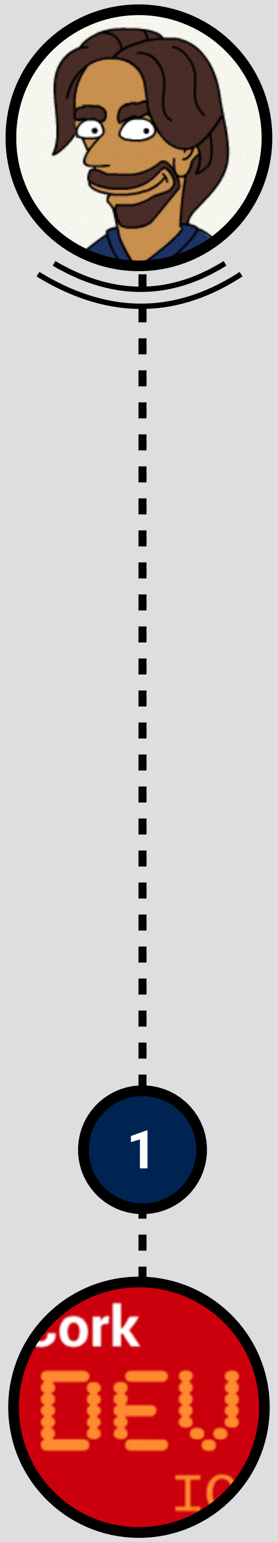
- Mouse clicks
- Keyboard events
- Tweets
- RSS feed
- A/V streams
- Stocks
- WebSockets
- Order status

Why going Reactive?

Reactive Programming:

- **RP**: *a paradigm oriented around asynchronous data flows and the propagation of change*
- Basically:
  - Reactive model is **Push** rather than **Pull**
  - Programming with **async data sequences**
  - Discrete values emitted over time
  - It’s not “Functional Reactive Programming”!

Why going Reactive?



ReactiveX:

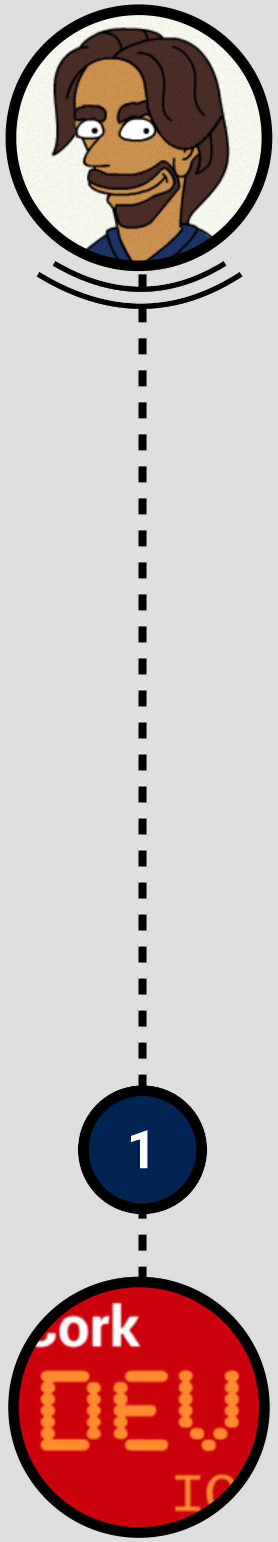
- Rx stands for **R**eactive E**x**tensions
- *"A library for composing asynchronous and event-based programs by using observable sequences."*
- Created by Erik Meijer (Microsoft) for the .NET
  - Version 1.0 released 17/11/2009
  - Version 2.0 released 15/08/2012
- Ported to different languages thereafter

Why going Reactive?

ReactiveX:

- Provides a collection of operators
- Observer pattern "on steroids":
  - Support sequences of data and/or events
  - Compose sequences using declarative way
  - Abstract away concerns about low-level stuff
  - Concurrent data structures and non-blocking I/O
  - Threading and Thread Safety

Why going Reactive?



Accessing sequences of data:

	single items	multiple items
sync	T getData()	Iterable<T> getData()
async	Future<T> getData()	

Accessing sequences of data:

	single items	multiple items
sync	T getData()	Iterable<T> getData()
async	Future<T> getData()	**Observable<T> getData()**

Why going Reactive?

Accessing sequences of data:

	single items	multiple items
sync	T getData()	Iterable<T> getData()
async	Future<T> getData()	Observable<T> getData()

Reactive X is about dealing with **Observables**!

Why going Reactive?

Observables are:

- **Composable**:
  They are intended for composing flows and sequences of asynchronous data unlike i.e. Java Futures

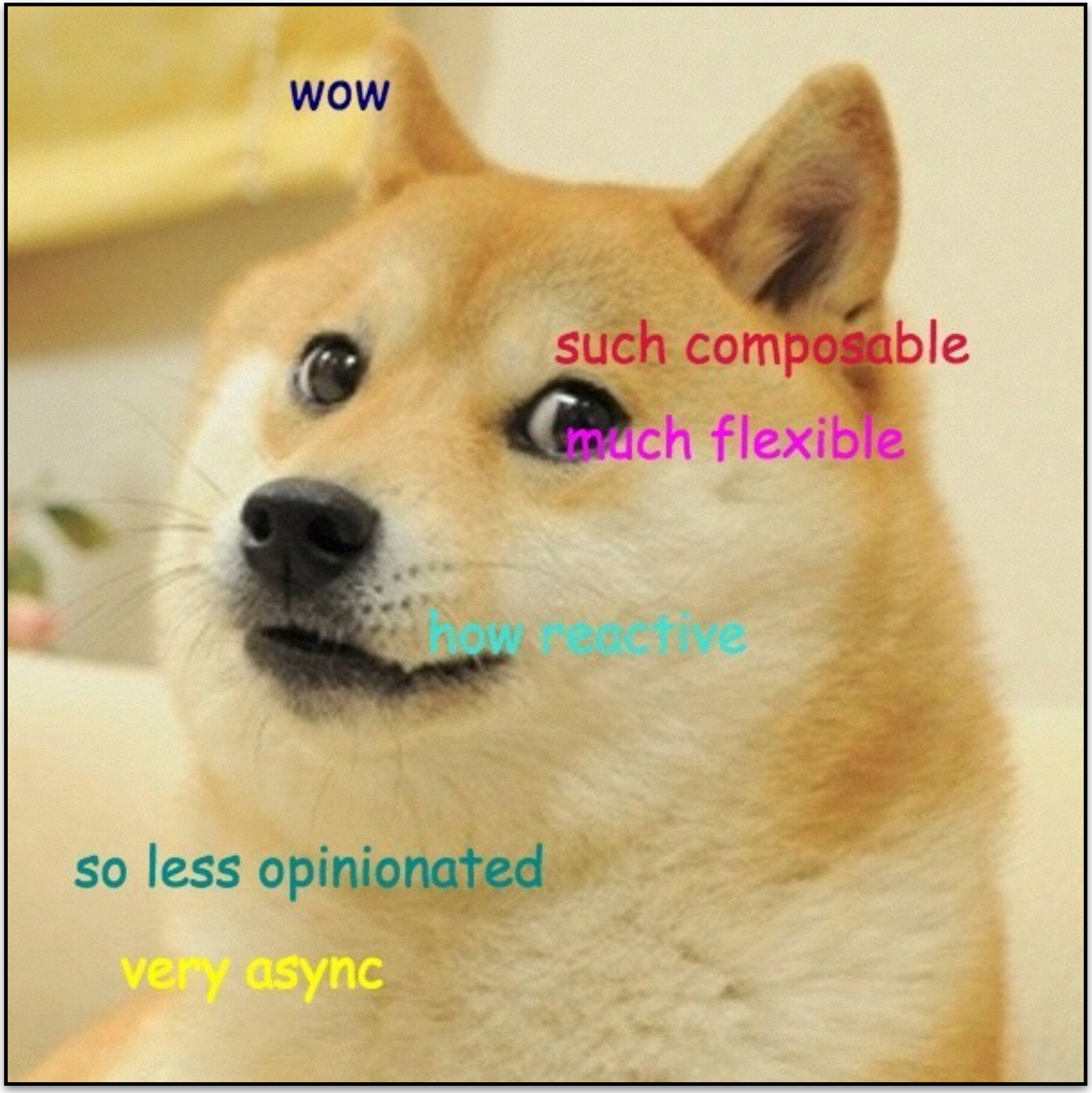
- **Flexible**:
  They support emission of single scalar value, but also of sequences of values or even infinite streams. They have the flexibility and elegance of their Iterable cousins

- **Less Opinionated**:
  They just don't care about the source of concurrency or asynchronicity. The underlying nature of their implementation might be changed without breaking the consumers

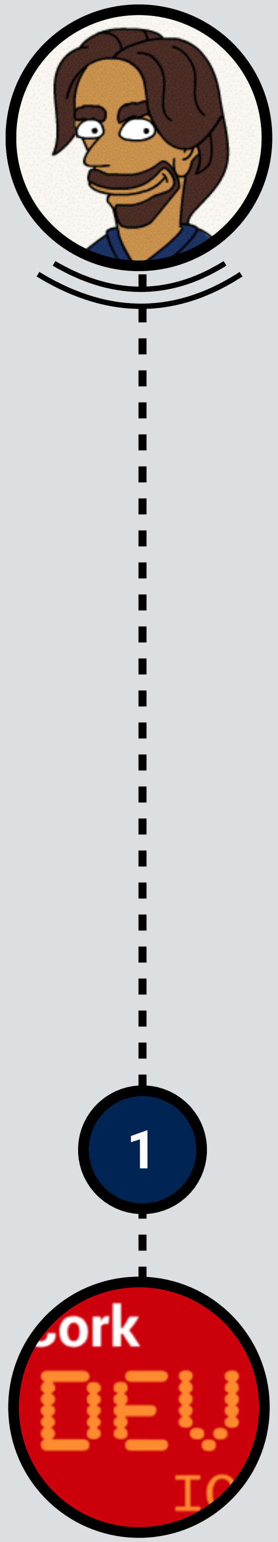
1

Why going Reactive?

1



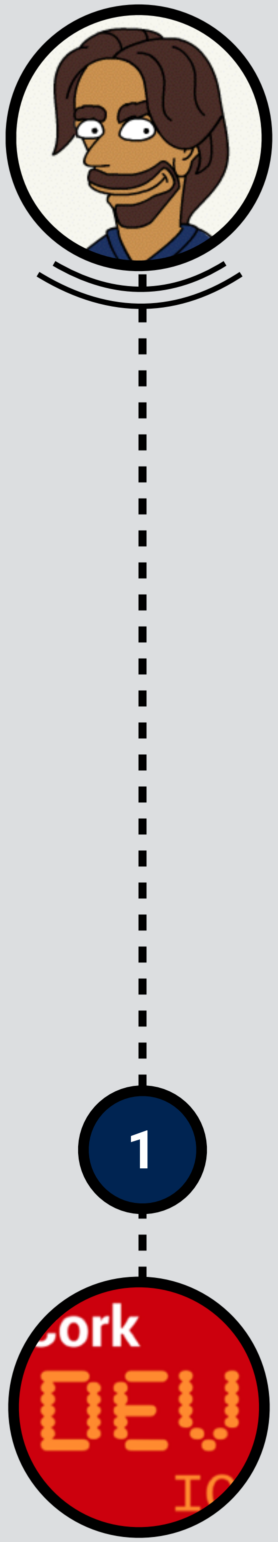
Why going Reactive?



Observables vs Iterables:

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

Why going Reactive?



Observables vs Iterables:

- Iterable is **synchronous** and **pull**
- Iterable: the consumer pulls the values from the producer (blocking the thread until they arrives)
- Observable is **asynchronous** and **push**
- Observable: the producer pushes values to the consumer whenever they are available
- Observable adds the ability for the producer to say:
  - *“There is no more data available!”*
  - *“An error has occurred!”*

Why going Reactive?

1

Why going Reactive?

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

Why going Reactive?

1

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

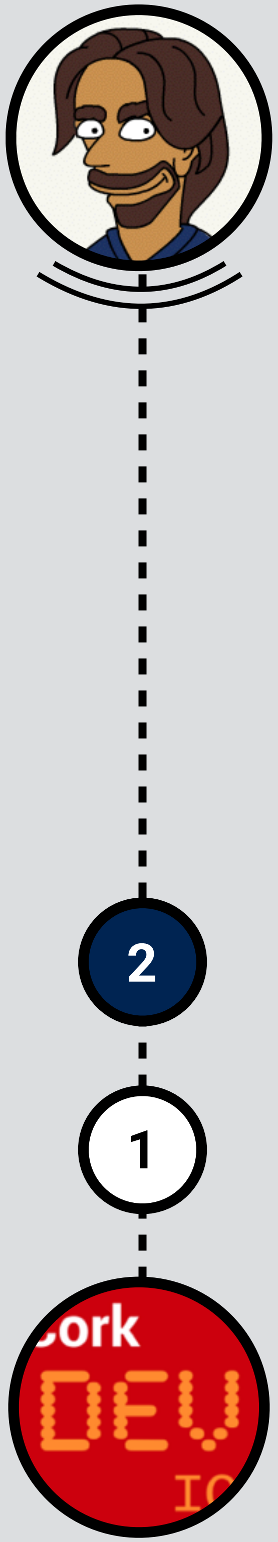
Understand it once - reuse the concepts in different languages

Rx 101*
with RxJava

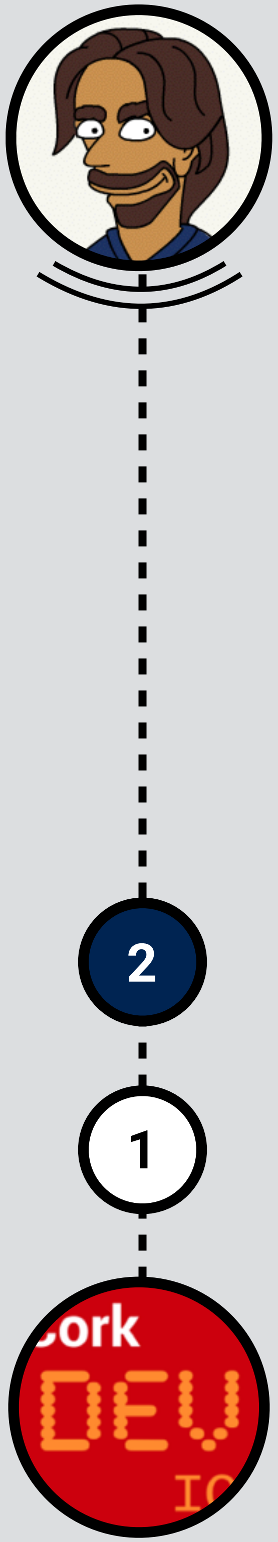
Rx 101*
with RxJava

* The road is steep, the road is long, the talk is relatively short.

RxJava:



- JVM implementation of ReactiveX
- Ported by Netflix
- Reached stable (1.0.0) version in November 2014
- Polyglot implementation to Scala, Groovy, Clojure and Kotlin
- Seems like just a library… but it’s also the concept in the way you code



RxJava:

- Lightweight (Zero Dependencies and < 800KB Jar)
- Java 6+ & Android 2.3+
- Java 8 lambda support
- Non-opinionated about source of concurrency
- Async or synchronous execution
- Virtual time and schedulers for parameterised concurrency

2

1

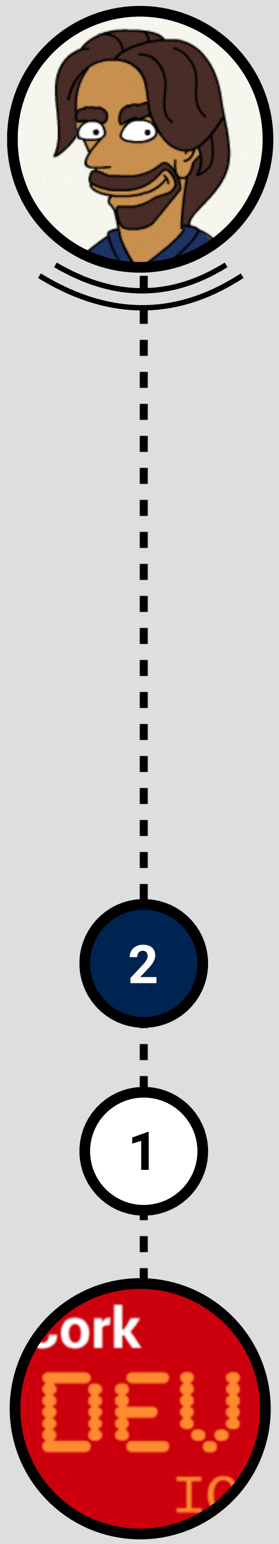
Cork DEV IO

RxJava:

https://github.com/ReactiveX/RxJava/

compile ‘io.reactivex:rxjava:x.y.z'

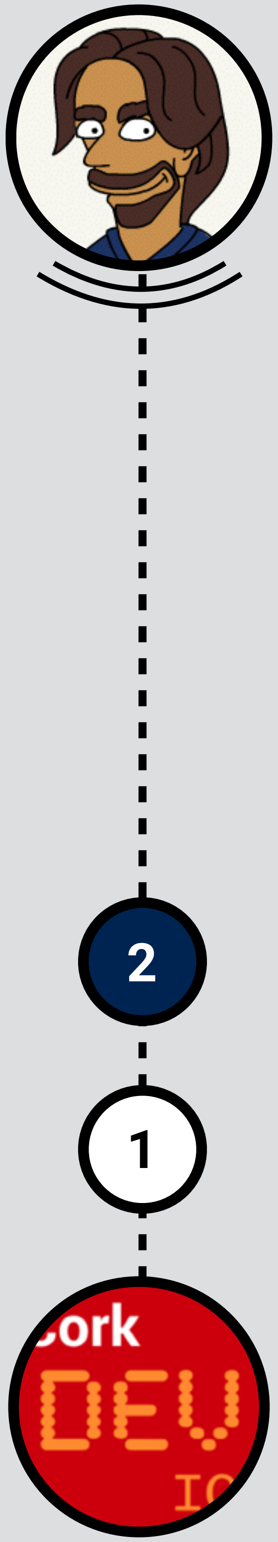
Current version: 1.1.0 released on 02/12/2015



Rx 101 topics:

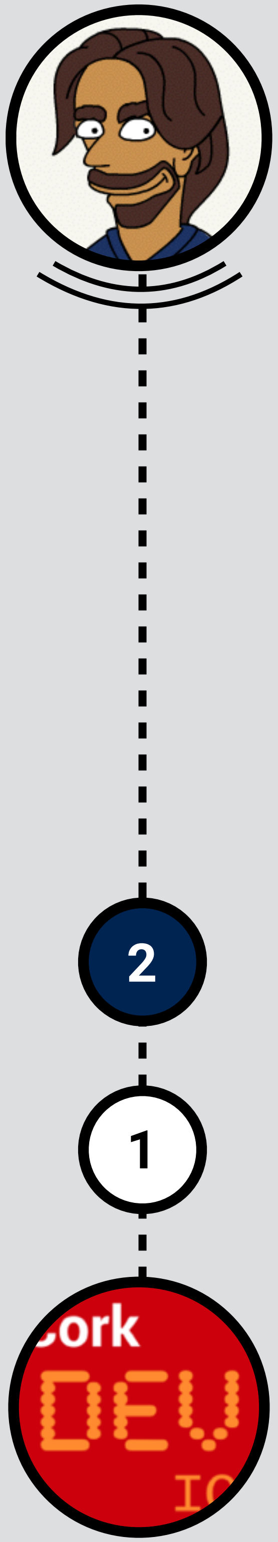
- Observable
- Observer
- Subscription
- Marble diagrams
- Operators
- Schedulers*

Observable



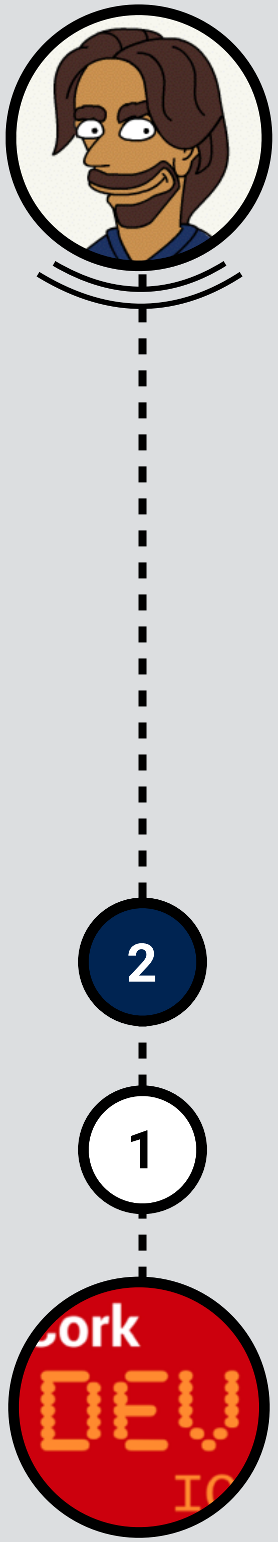
Observable:

- Emits zero or more items (values)
- Notifies the Observer using **onNext(T data)** method when the item is ready to be pushed
- Calls Observer's **onCompleted()** method when the sequence is finished
- Calls Observer's **onError(Throwable t)** method when a serious error happened
- It's either **onCompleted()** or **onError(Throwable t)**



Observable creation:

- Convert objects, lists, or arrays of objects into Observables that emit those objects:
  - **Observable.just(…);**
    For converting a single object
  - **Observable.from(…);**
    For converting lists, or arrays of objects
- Design your own Observable implementing:
  - **Observable.create(…);**
    For async i/o, computational operations, streams of data…

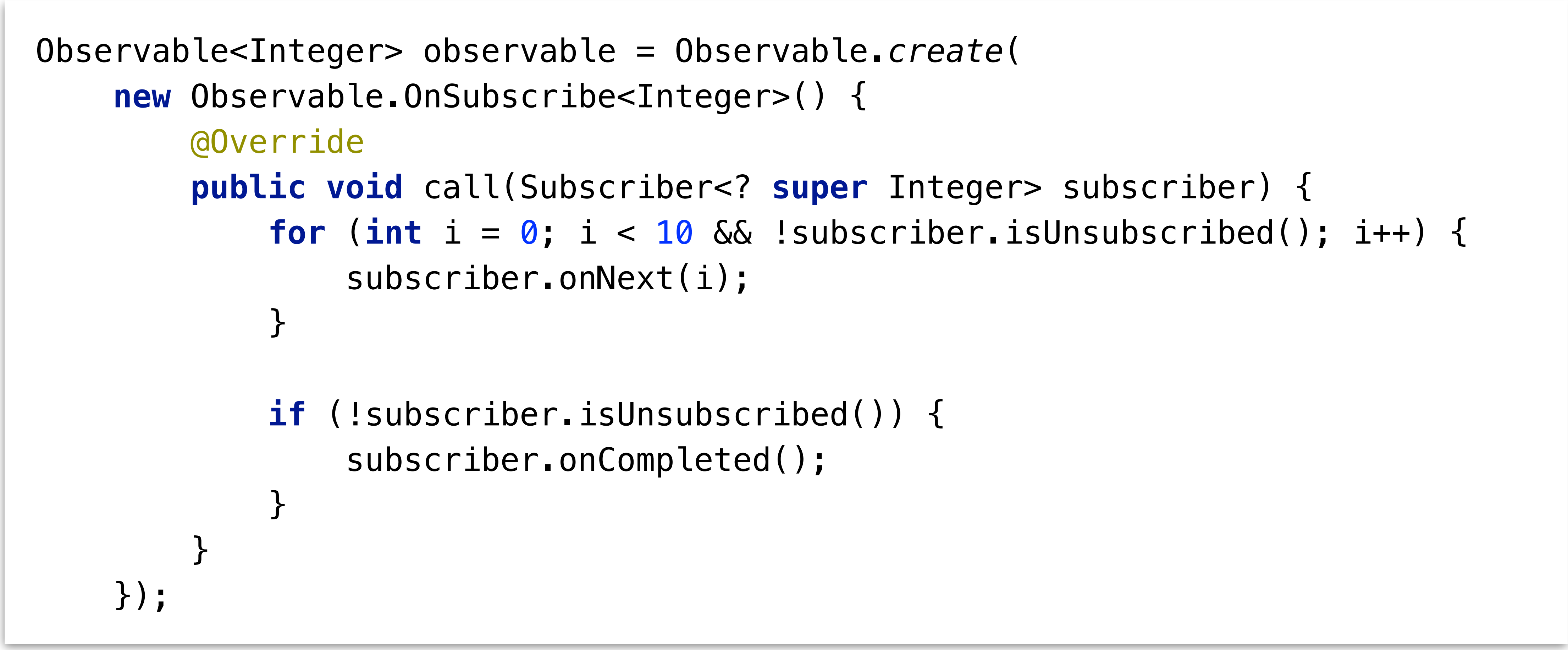


Observable creation:

- There are many more practical ways of creating Observables:

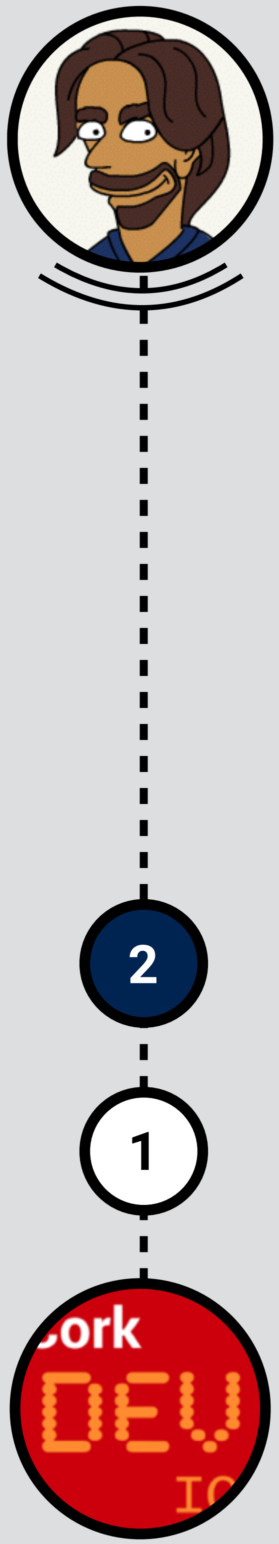
  https://github.com/ReactiveX/RxJava/wiki/Creating-Observables

Observable.create():

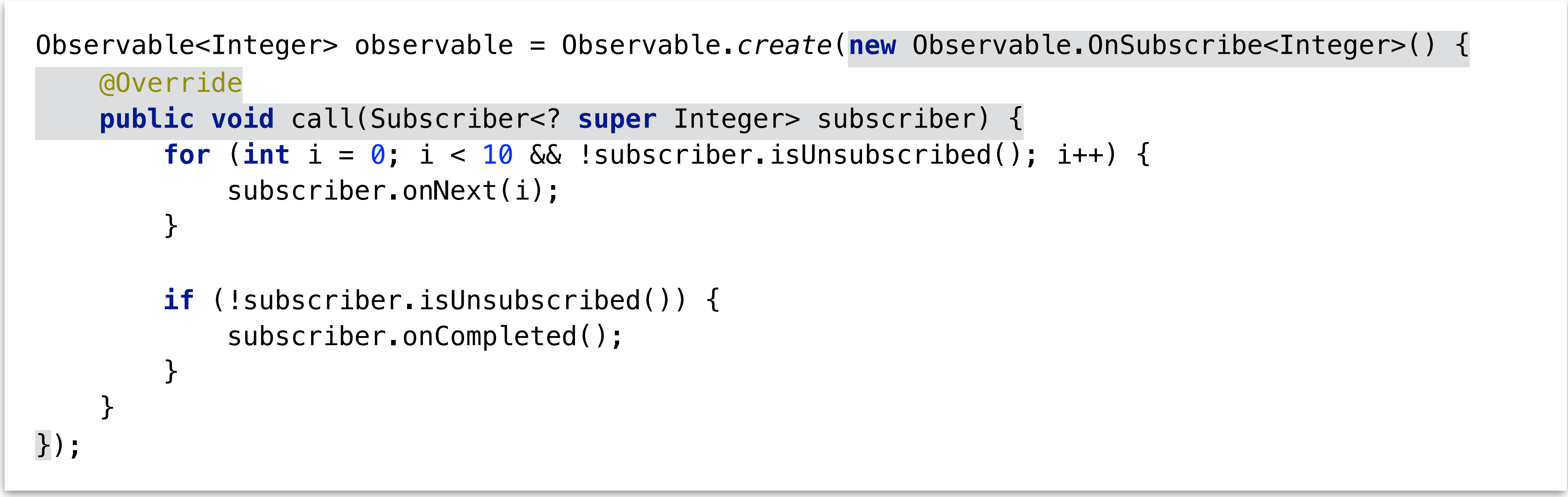


```java
Observable<Integer> observable = Observable.create(
    new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            for (int i = 0; i < 10 && !subscriber.isUnsubscribed(); i++) {
                subscriber.onNext(i);
            }

            if (!subscriber.isUnsubscribed()) {
                subscriber.onCompleted();
            }
        }
    });
```

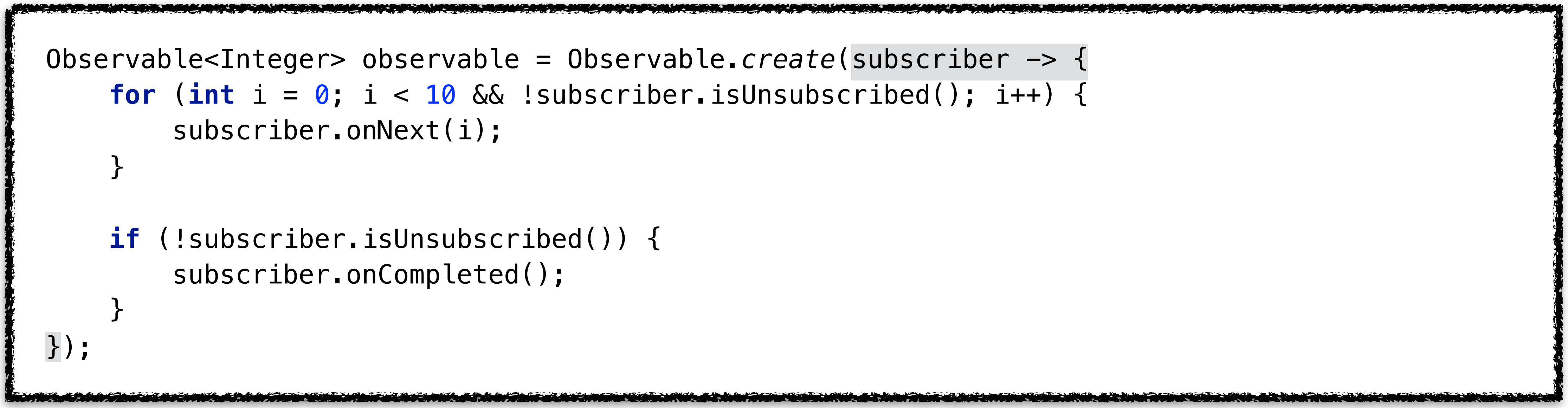
Observable.create():



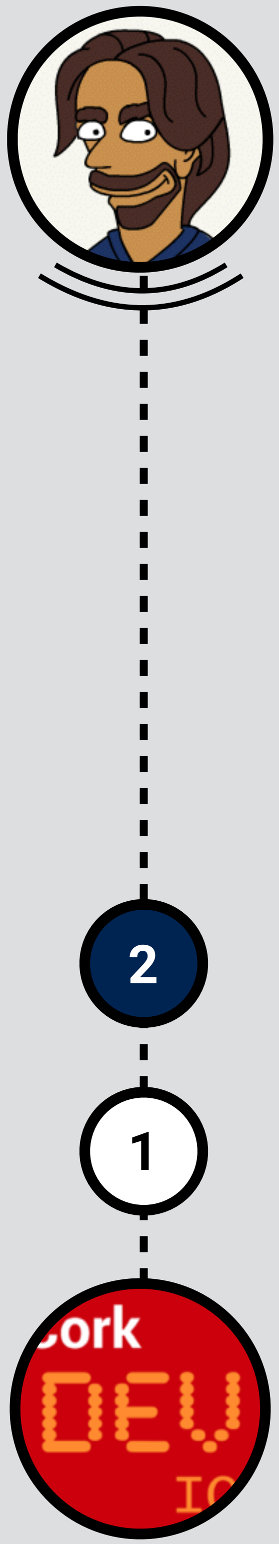
Using lambda expressions:



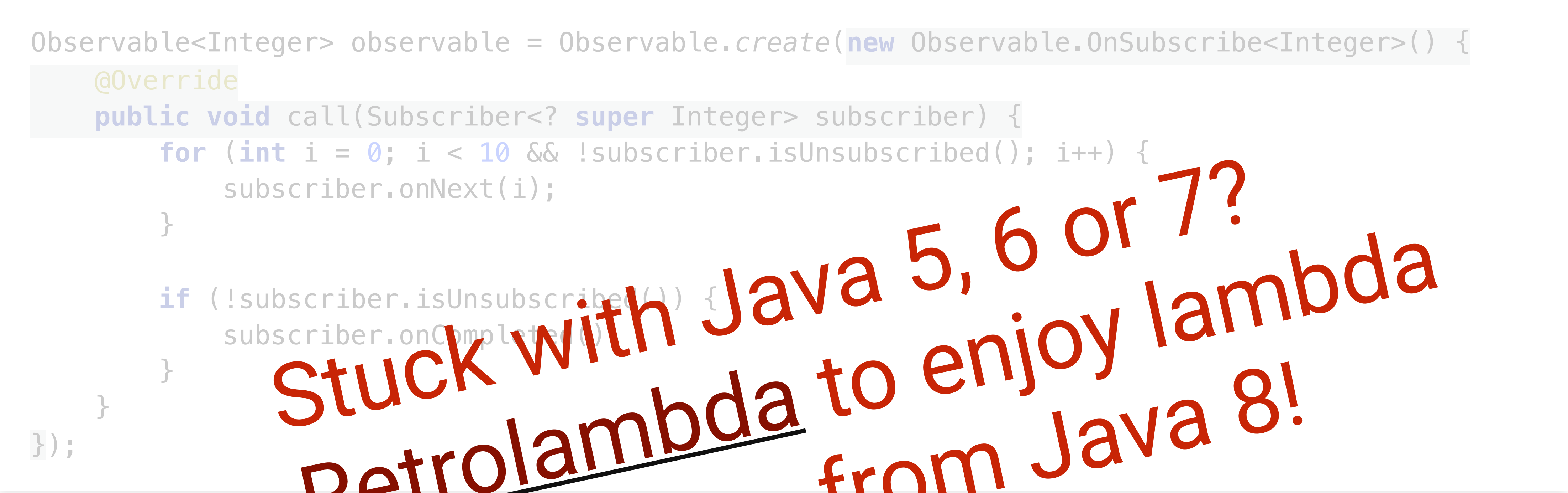
```
Observable<Integer> observable = Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for (int i = 0; i < 10 && !subscriber.isUnsubscribed(); i++) {
            subscriber.onNext(i);
        }

        if (!subscriber.isUnsubscribed()) {
            subscriber.onCompleted();
        }
    }
});
```



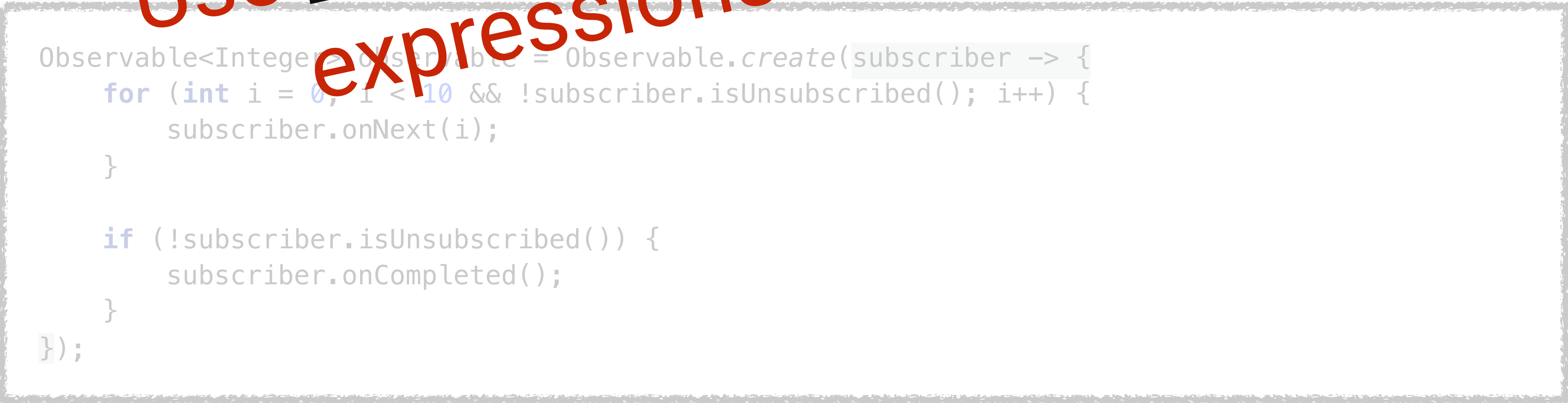
```
Observable<Integer> observable = Observable.create(subscriber -> {
    for (int i = 0; i < 10 && !subscriber.isUnsubscribed(); i++) {
        subscriber.onNext(i);
    }

    if (!subscriber.isUnsubscribed()) {
        subscriber.onCompleted();
    }
});
```



Using lambda expressions:



```
Observable<Integer> observable = Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for (int i = 0; i < 10 && !subscriber.isUnsubscribed(); i++) {
            subscriber.onNext(i);
        }

        if (!subscriber.isUnsubscribed()) {
            subscriber.onCompleted();
        }
    }
});
```

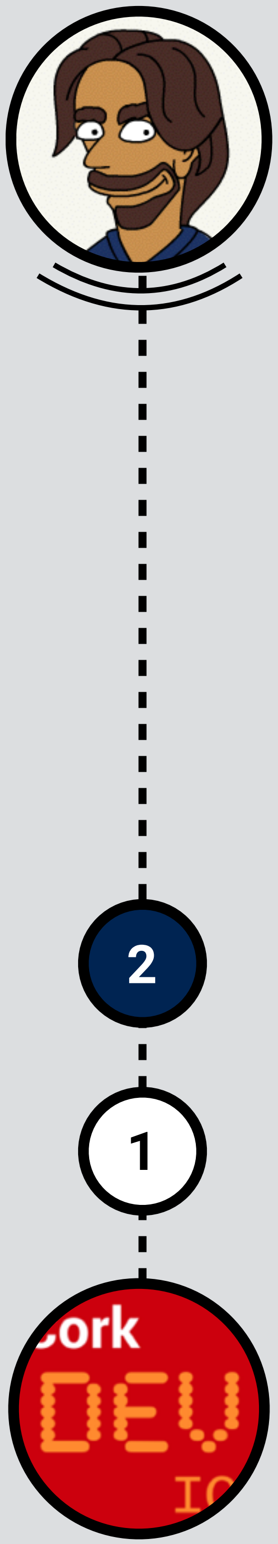


```
Observable<Integer> observable = Observable.create(subscriber -> {
    for (int i = 0; i < 10 && !subscriber.isUnsubscribed(); i++) {
        subscriber.onNext(i);
    }

    if (!subscriber.isUnsubscribed()) {
        subscriber.onCompleted();
    }
});
```

Stuck with Java 5, 6 or 7?

Use Retrolambda to enjoy lambda expressions from Java 8!

“Hot” vs “Cold” Observables:



“Hot” vs “Cold” Observables:

- “Hot” Observables:
  - May begin emitting items as soon as is is created
  - Observer who later subscribes may start observing the sequence somewhere in the middle
- “Cold” Observables:
  - Waits until an observer subscribes to it before it emits items
  - An observer is guaranteed to see the whole sequence from the beginning

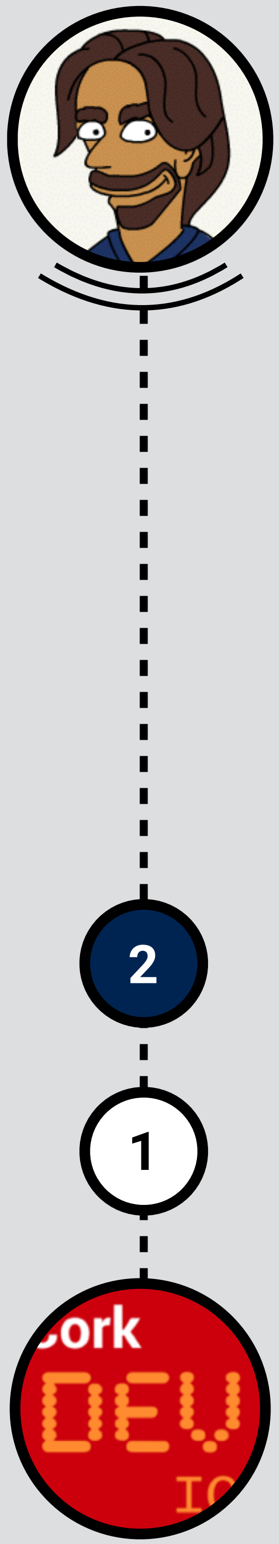
“Hot” vs “Cold” Observables:

- “Hot” Examples:
  - Stream of mouse click events
  - Tweets
- “Cold” Examples:
  - Network request
  - A/V Stream

Rx 101 with RxJava

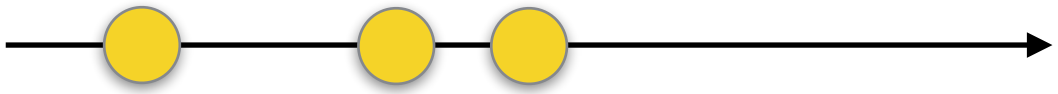
2

1

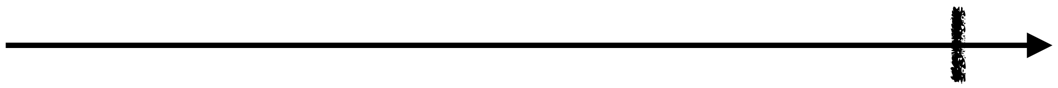


Observer<T> - the interface:

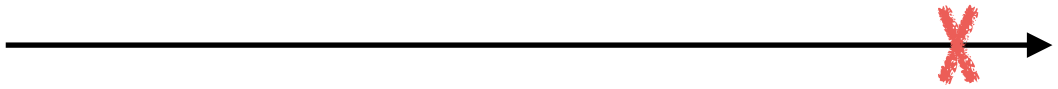
- Interface methods:
  - onNext(T data);



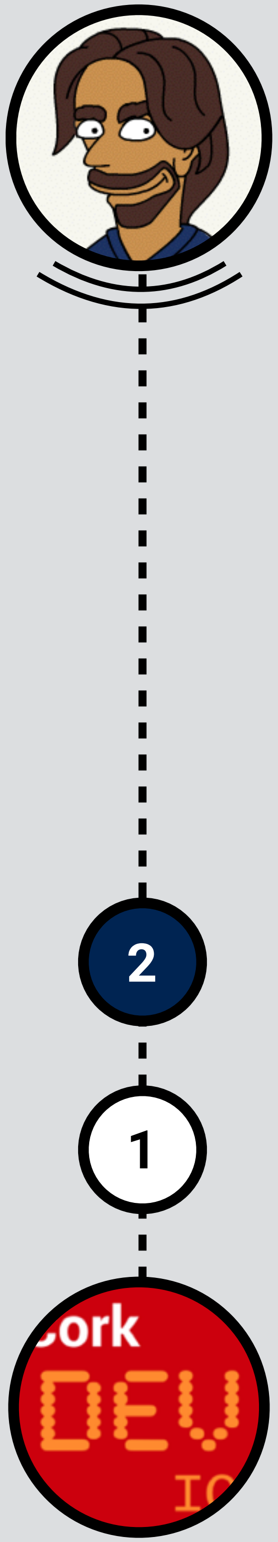
  - onCompleted();



  - onError(Throwable t);

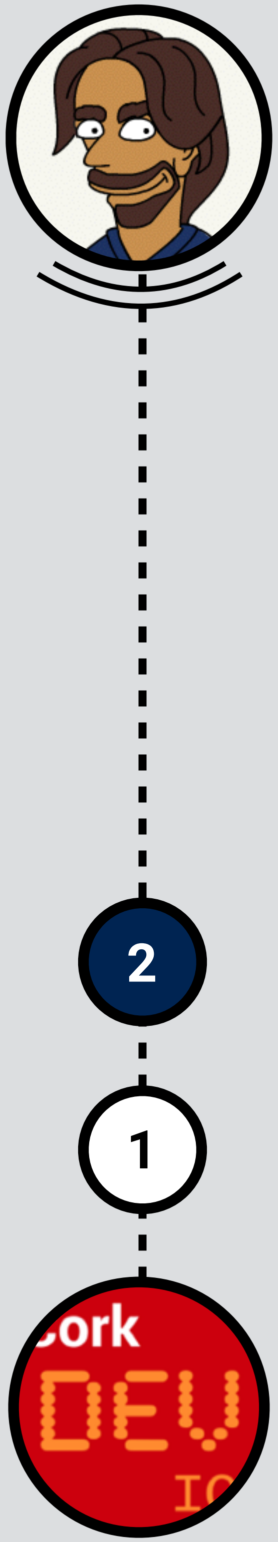


- Receives zero or more items
- You can override all of the interface methods (recommended for beginners) or just a single one

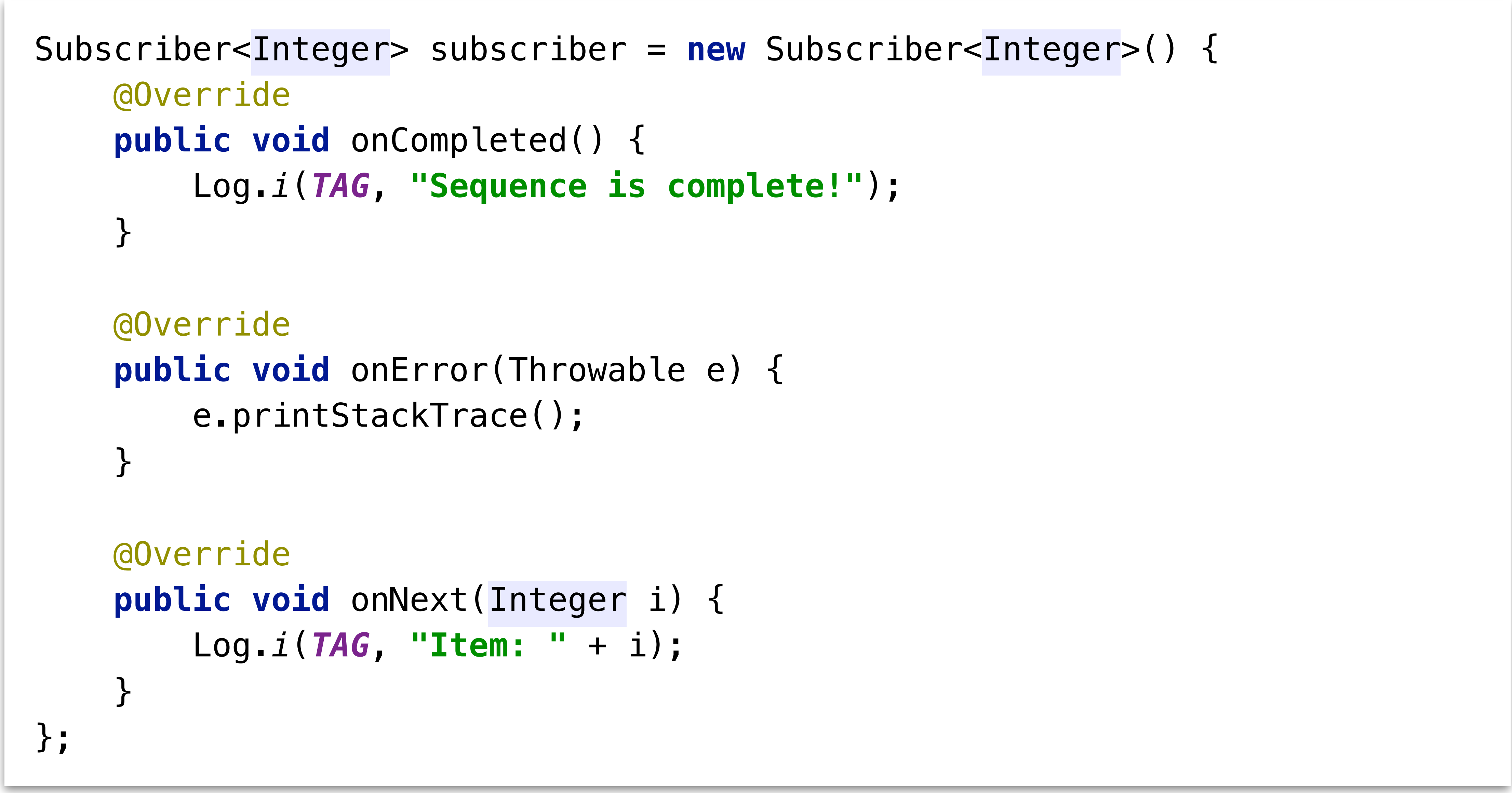


Subscriber - the implementation:

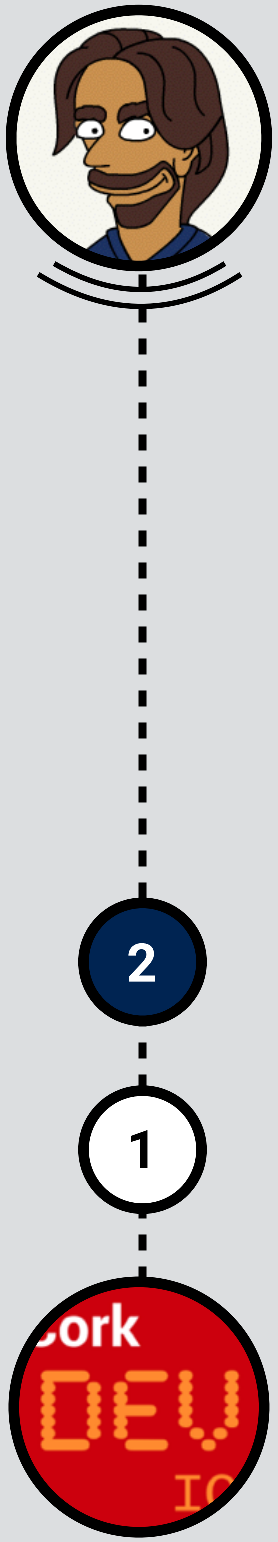
- *“A Subscriber is an Observer that can also unsubscribe from that data source.”*
- Subscriber class is the implementation of two interfaces: Observer and Subscriber
- It’s a common practice to pass the implementation rather then the interface
- It reduces the learning curve for developers new to Rx



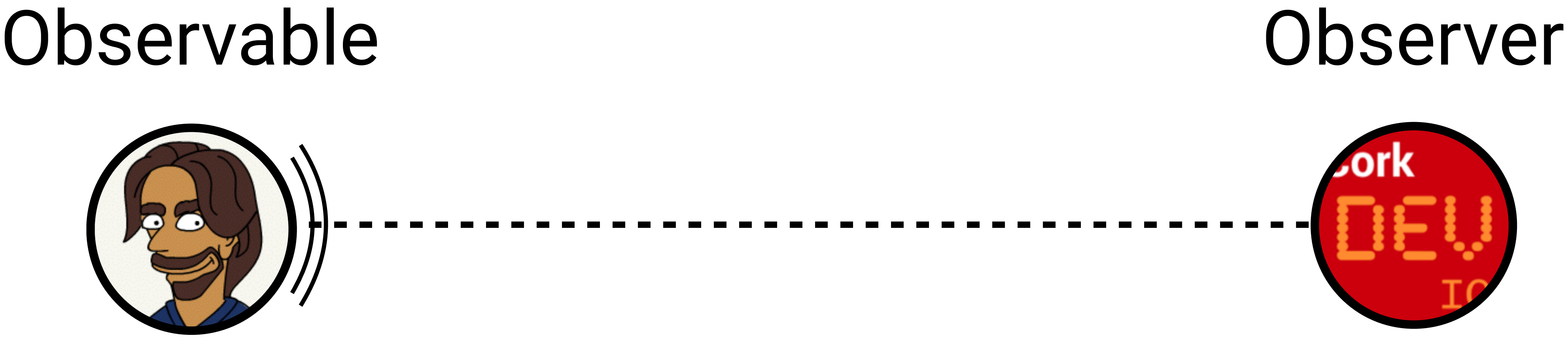
Creating a Subscriber:

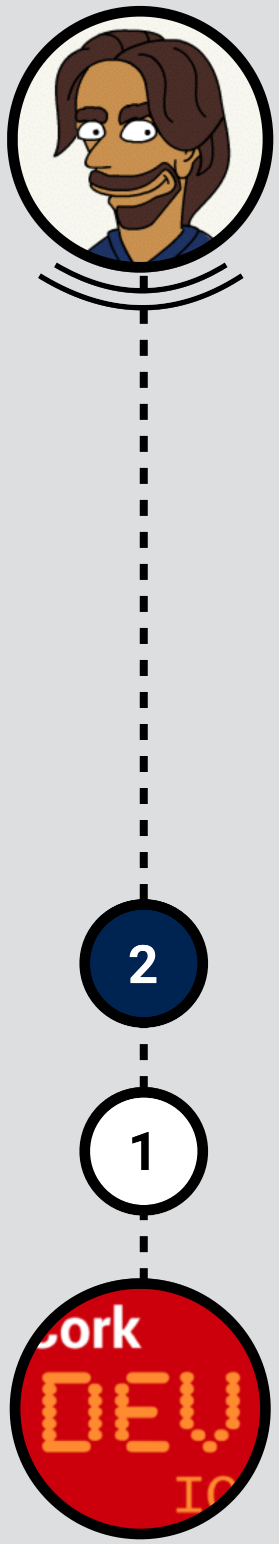


```java
Subscriber<Integer> subscriber = new Subscriber<Integer>() {
    @Override
    public void onCompleted() {
        Log.i(TAG, "Sequence is complete!");
    }

    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }

    @Override
    public void onNext(Integer i) {
        Log.i(TAG, "Item: " + i);
    }
};
```

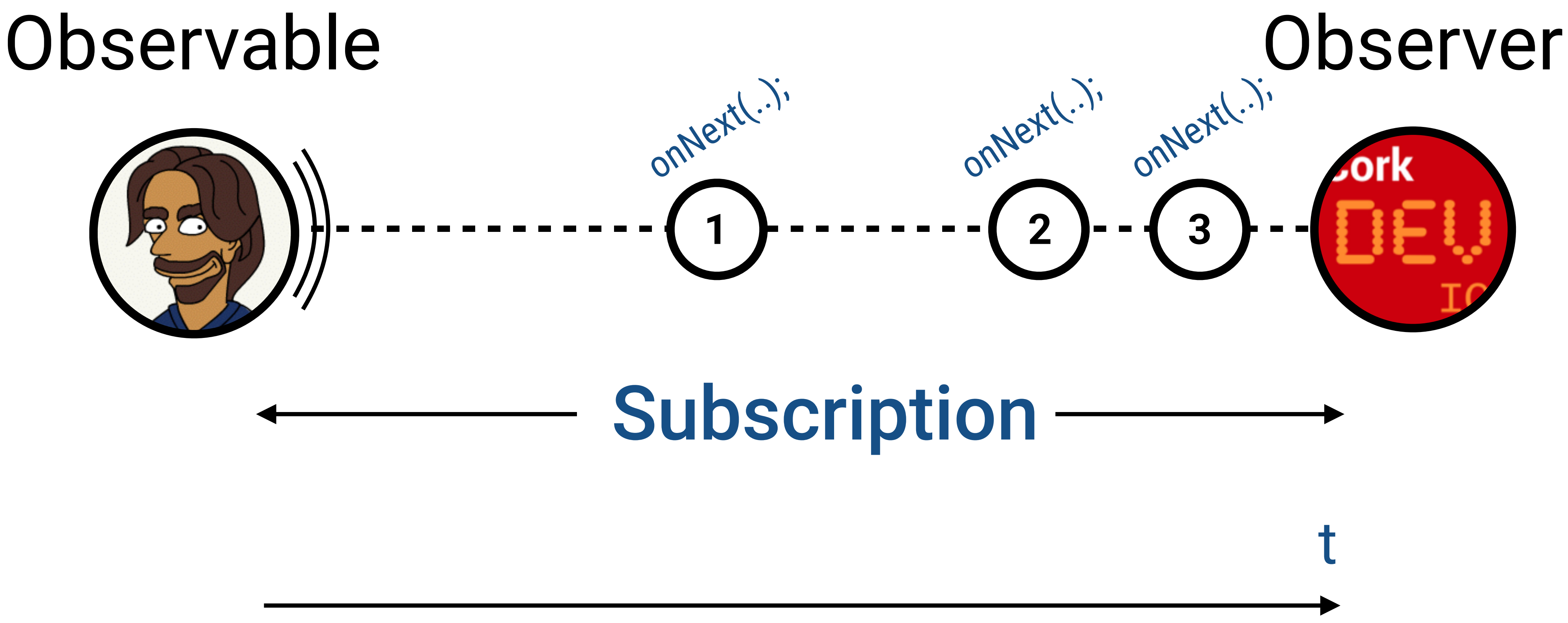


Subscription:





Subscription:


Observable

Observer

onNext(..);

onNext(..);

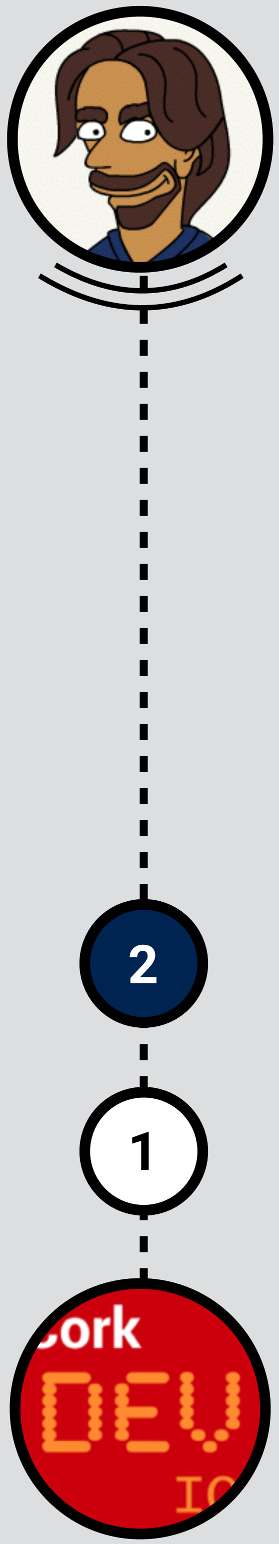
onNext(..);

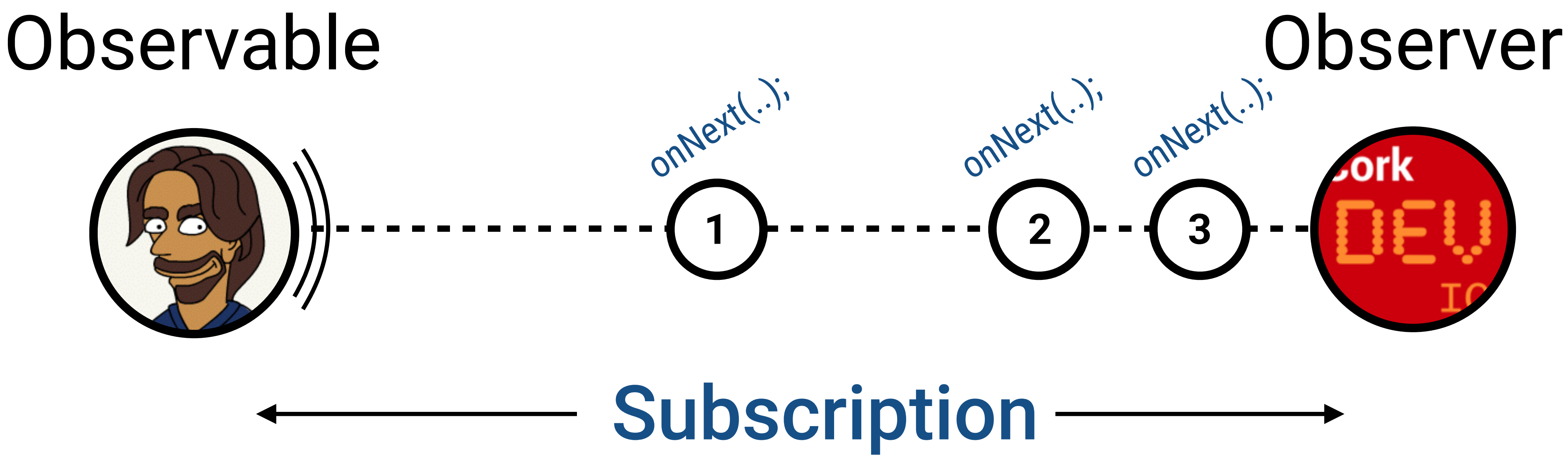
Subscription

t

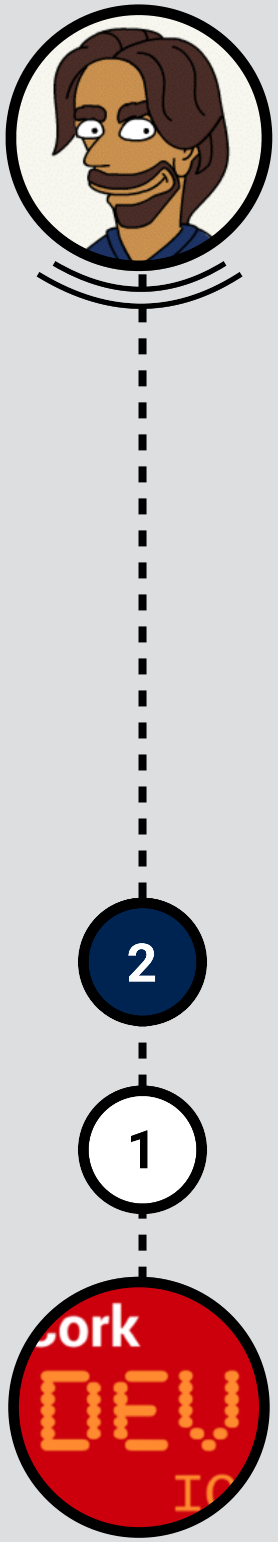
Subscription:

```
// Subscribing to the Observable
observable.subscribe(subscriber);
```





```
// Unsubscribing the Observer/Subscriber
subscriber.unsubscribe();
```

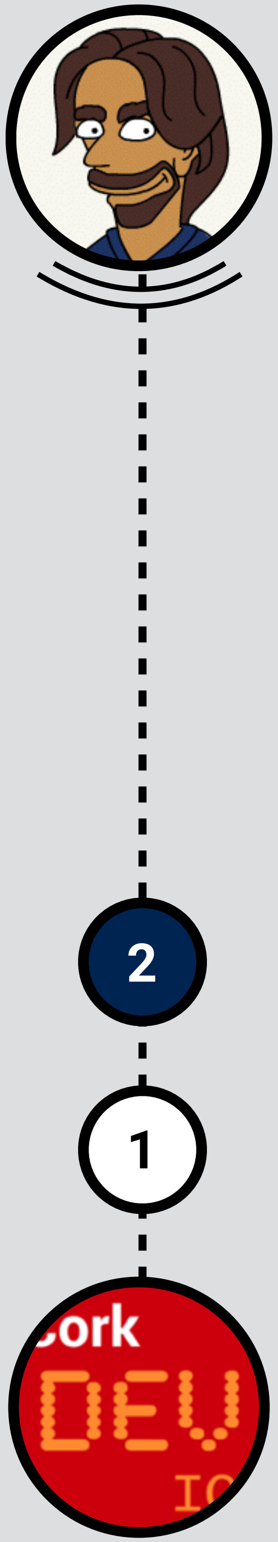


Subscriptions in examples:

```
// subscribe() method can implement only onNext()
Observable.just("Hello world!")
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                Log.i(TAG, "Greeting: " + s);
            }
        });
```

```
// Simplified using lambda expression
Observable.just("Hello world!")
        .subscribe(s -> Log.i(TAG, "Greeting: " + s));
```

```
// subscribe() with onNext() and onError()
Observable.just("Hello world!")
        .subscribe(
                s -> Log.i(TAG, "Greeting: " + s),
                t -> t.printStackTrace()
        );
```

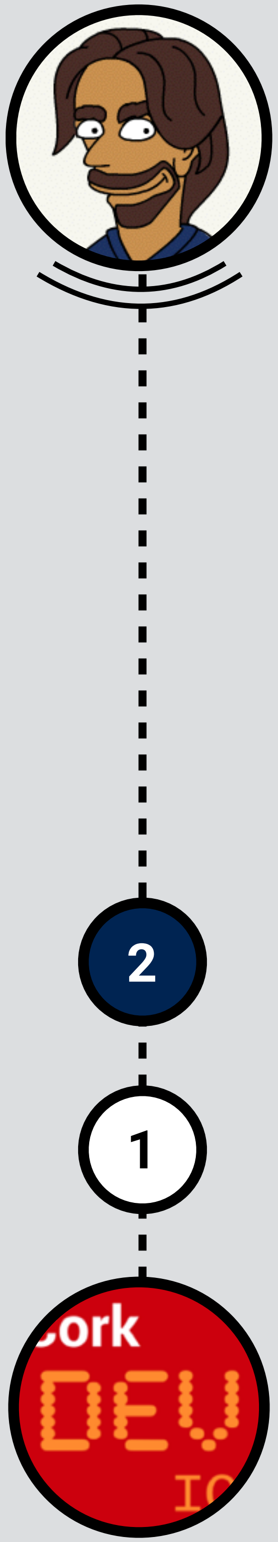


Subscriptions in examples:

```
// subscribe() method can implement only onNext()
Observable.just("Hello world!")
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                Log.i(TAG, "Greeting: " + s);
            }
        });
```

```
// Simplified using lambda expression
Observable.just("Hello world!")
        .subscribe(s -> Log.i(TAG, "Greeting: " + s));
```

```
// subscribe() with onNext() and onError()
Observable.just("Hello world!")
        .subscribe(
                s -> Log.i(TAG, "Greeting: " + s),
                t -> t.printStackTrace()
        );
```

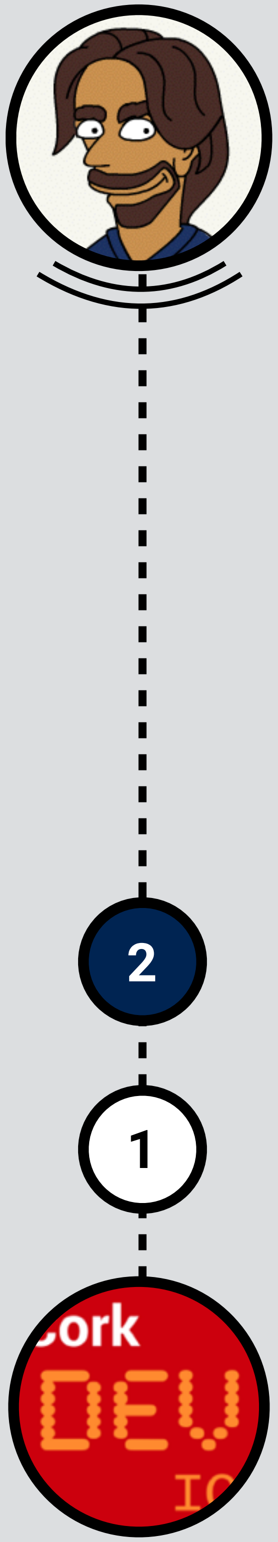


Subscriptions in examples:

```
// subscribe() method can implement only onNext()
Observable.just("Hello world!")
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                Log.i(TAG, "Greeting: " + s);
            }
        });
```

```
// Simplified using lambda expression
Observable.just("Hello world!")
        .subscribe(s -> Log.i(TAG, "Greeting: " + s));
```

```
// subscribe() with onNext() and onError()
Observable.just("Hello world!")
        .subscribe(
                s -> Log.i(TAG, "Greeting: " + s),
                t -> t.printStackTrace()
        );
```



Subscriptions in examples:

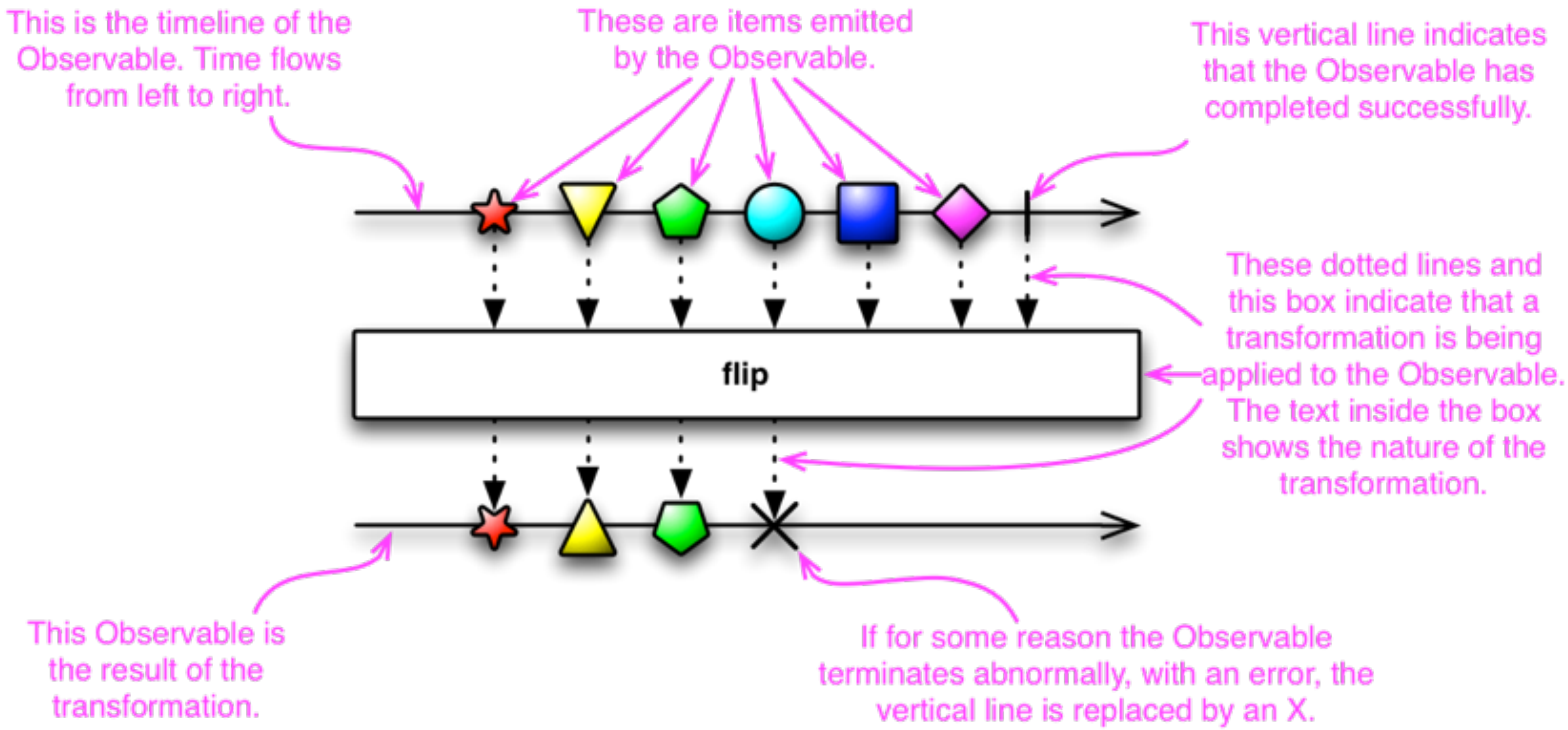
```
// subscribe() method can implement only onNext()
Observable.just("Hello world!")
        .subscribe(new Action1<String>() {
            @Override
            public void call(String s) {
                Log.i(TAG, "Greeting: " + s);
            }
        });
```

```
// Simplified using lambda expression
Observable.just("Hello world!")
        .subscribe(s -> Log.i(TAG, "Greeting: " + s));
```

```
// subscribe() with onNext() and onError()
Observable.just("Hello world!")
        .subscribe(
                s -> Log.i(TAG, "Greeting: " + s),
                t -> t.printStackTrace()
        );
```

Marble diagrams:

Marble diagrams:



This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.

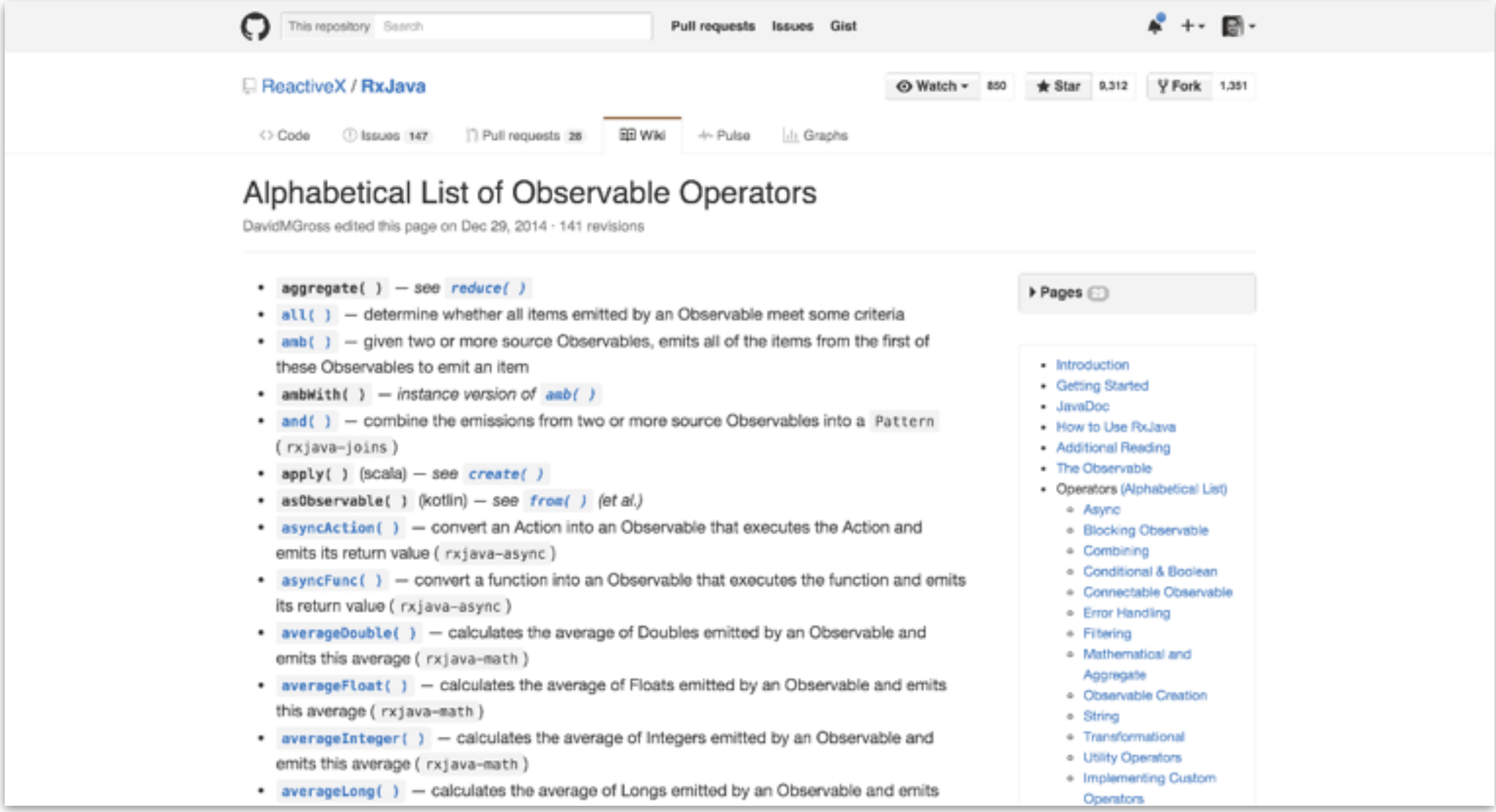
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Source: http://reactivex.io/documentation

Operators:

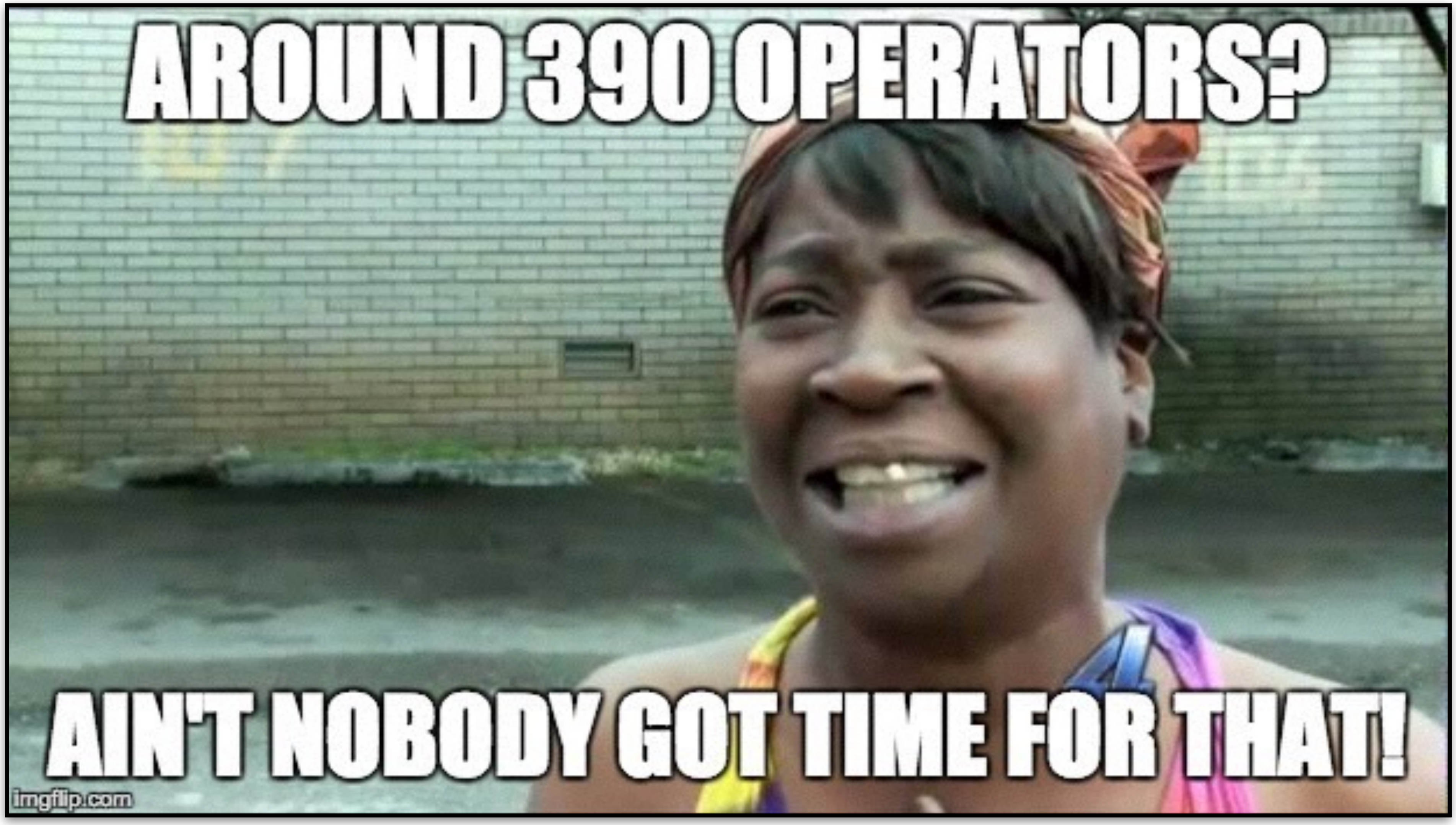


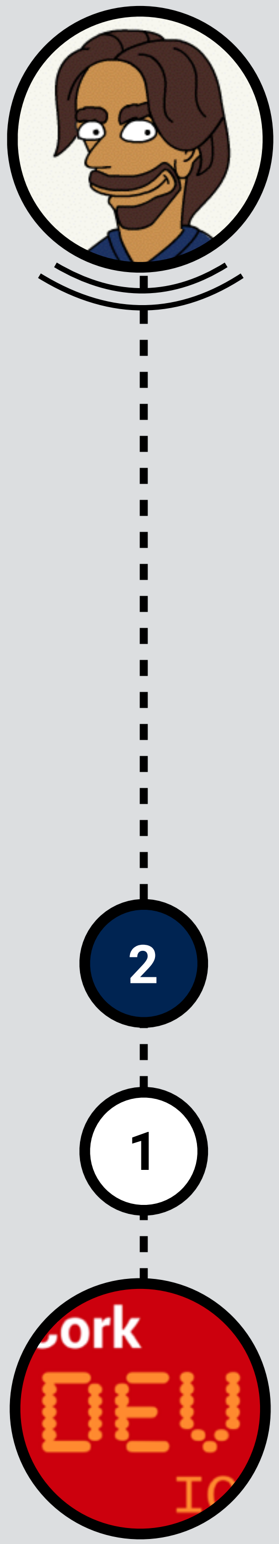
Alphabetical List of Observable Operators

DavidMGross edited this page on Dec 29, 2014 · 141 revisions

- `aggregate( )` — see `reduce( )`
- `all( )` — determine whether all items emitted by an Observable meet some criteria
- `amb( )` — given two or more source Observables, emits all of the items from the first of these Observables to emit an item
- `ambWith( )` — *instance version of* `amb( )`
- `and( )` — combine the emissions from two or more source Observables into a `Pattern` (`rxjava-joins`)
- `apply( )` (scala) — *see* `create( )`
- `asObservable( )` (kotlin) — *see* `from( )` *(et al.)*
- `asyncAction( )` — convert an Action into an Observable that executes the Action and emits its return value (`rxjava-async`)
- `asyncFunc( )` — convert a function into an Observable that executes the function and emits its return value (`rxjava-async`)
- `averageDouble( )` — calculates the average of Doubles emitted by an Observable and emits this average (`rxjava-math`)
- `averageFloat( )` — calculates the average of Floats emitted by an Observable and emits this average (`rxjava-math`)
- `averageInteger( )` — calculates the average of Integers emitted by an Observable and emits this average (`rxjava-math`)
- `averageLong( )` — calculates the average of Longs emitted by an Observable and emits

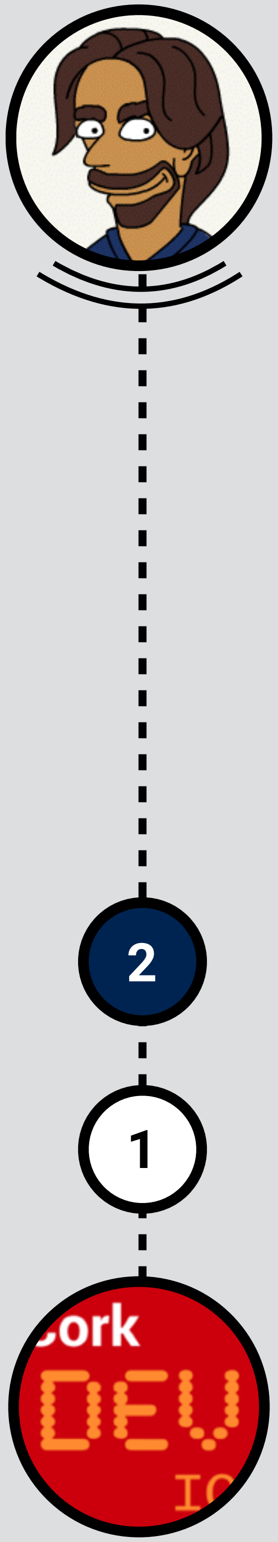
Operators:


AROUND 390 OPERATORS?
AIN'T NOBODY GOT TIME FOR THAT!
imgflip.com



Operator categories:

- Creating Observables
- Transforming Observables
- Filtering Observables
- Combining Observables
- Error Handling Operators
- Observable Utility Operators
- Conditional and Boolean Operators
- Mathematical and Aggregate Operators
- Backpressure Operators
- Connectable Observable Operators
- Operators to Convert Observables

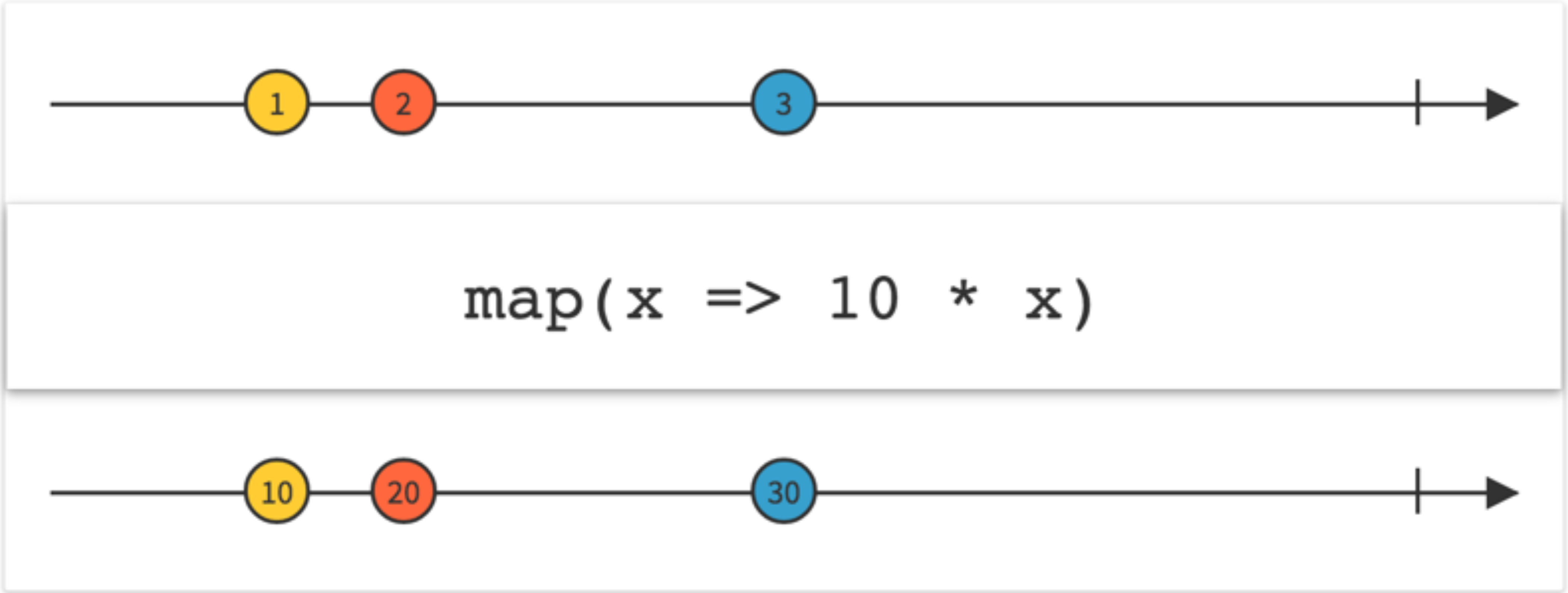


Operator categories:

- ~~Creating Observables~~
- Transforming Observables
- Filtering Observables
- Combining Observables
- Error Handling Operators
- ~~Observable Utility Operators~~
- ~~Conditional and Boolean Operators~~
- Mathematical and Aggregate Operators
- ~~Backpressure Operators~~
- ~~Connectable Observable Operators~~
- ~~Operators to Convert Observables~~

Transforming - **map**:

Transform the items emitted by an Observable by applying a function to each item

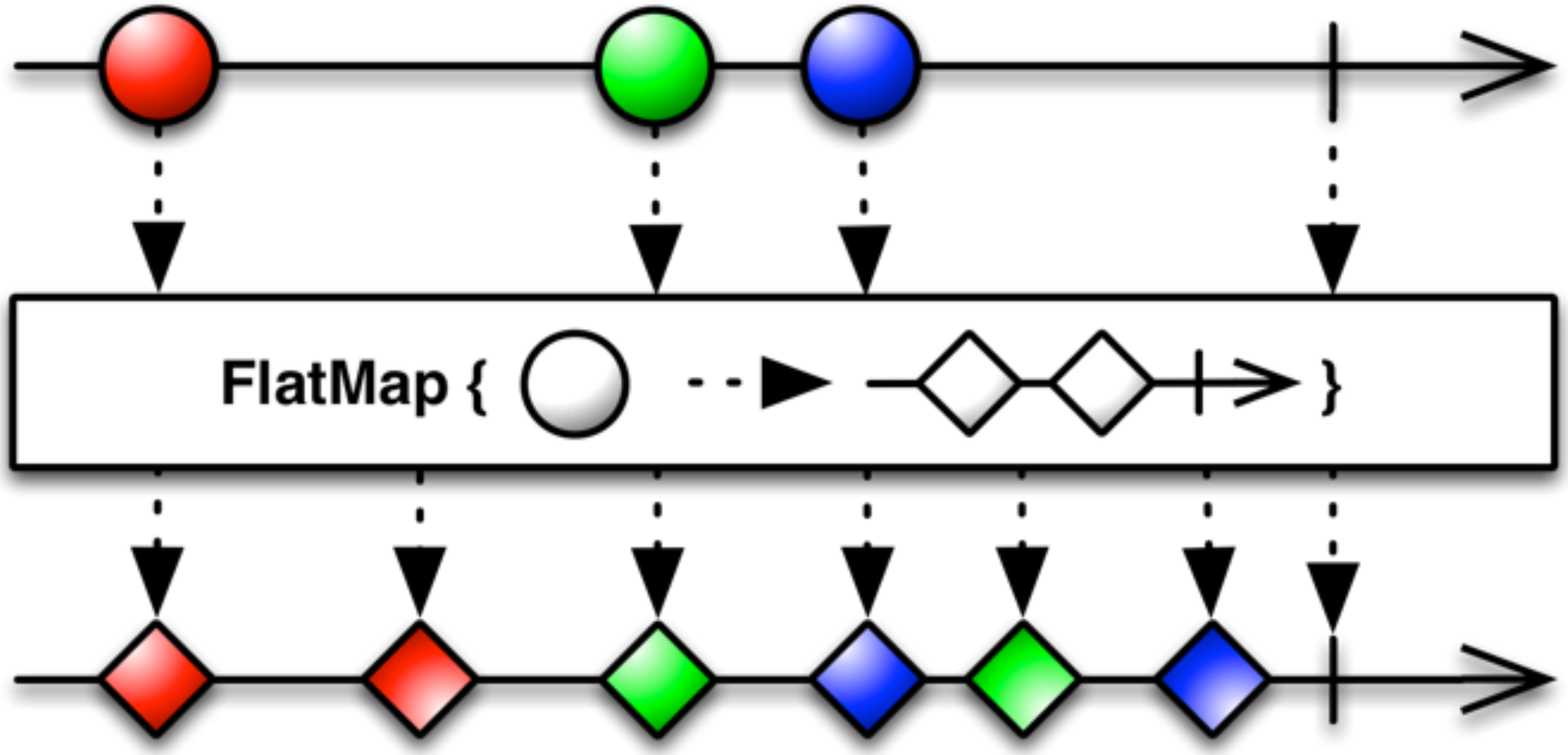


```
map(x => 10 * x)
```

Source: http://rxmarbles.com

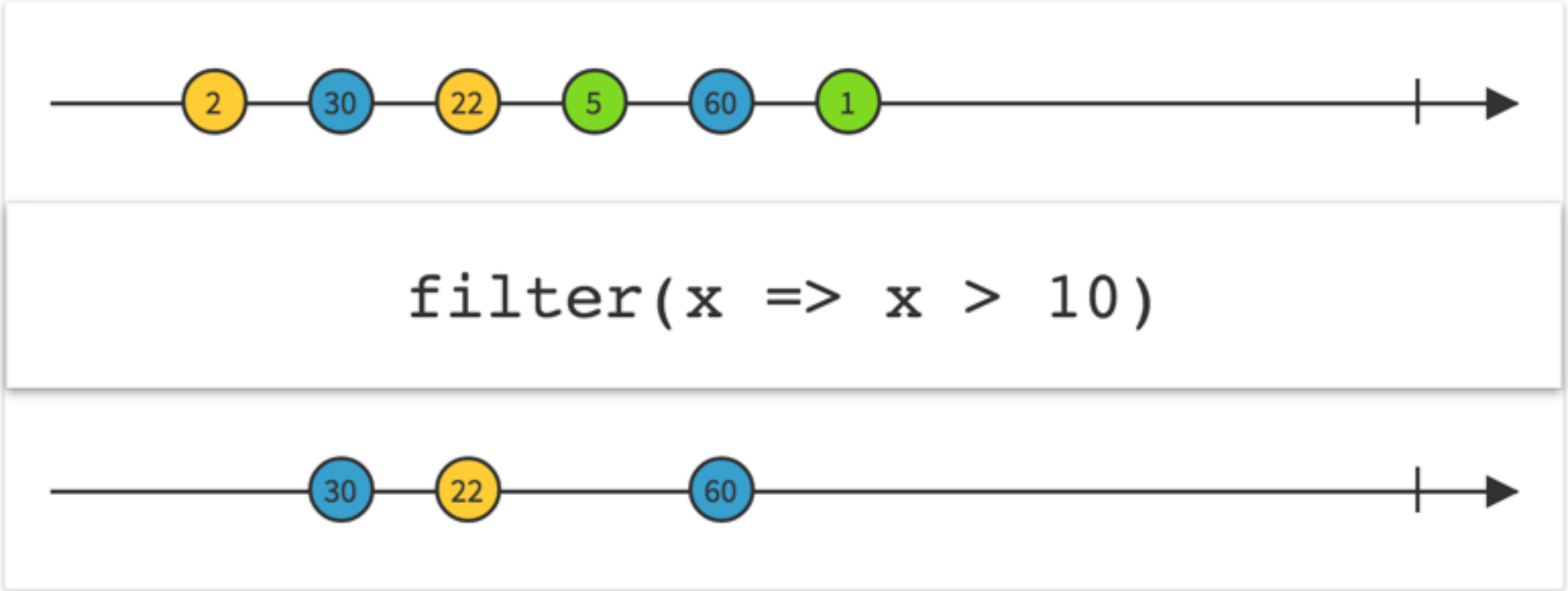
Transforming - **flatMap**:

Transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable



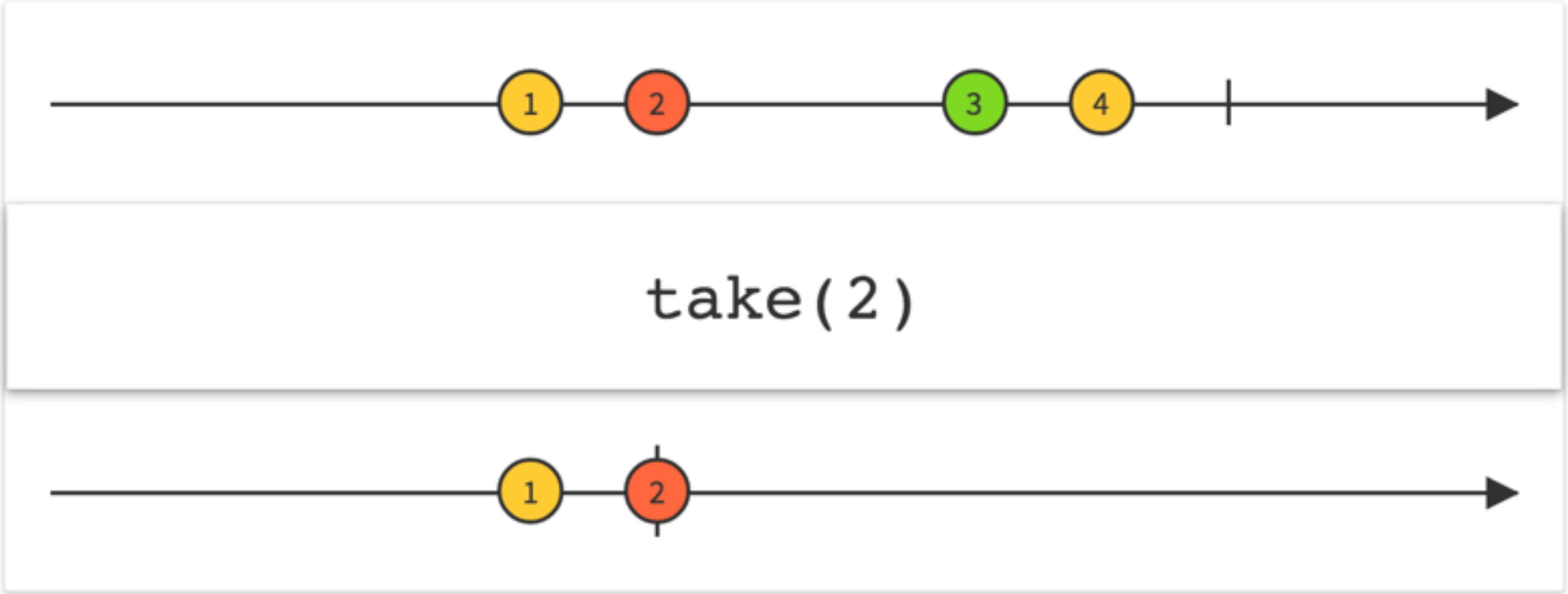
Filtering - **filter**:

Emit only those items from an Observable that pass a predicate test



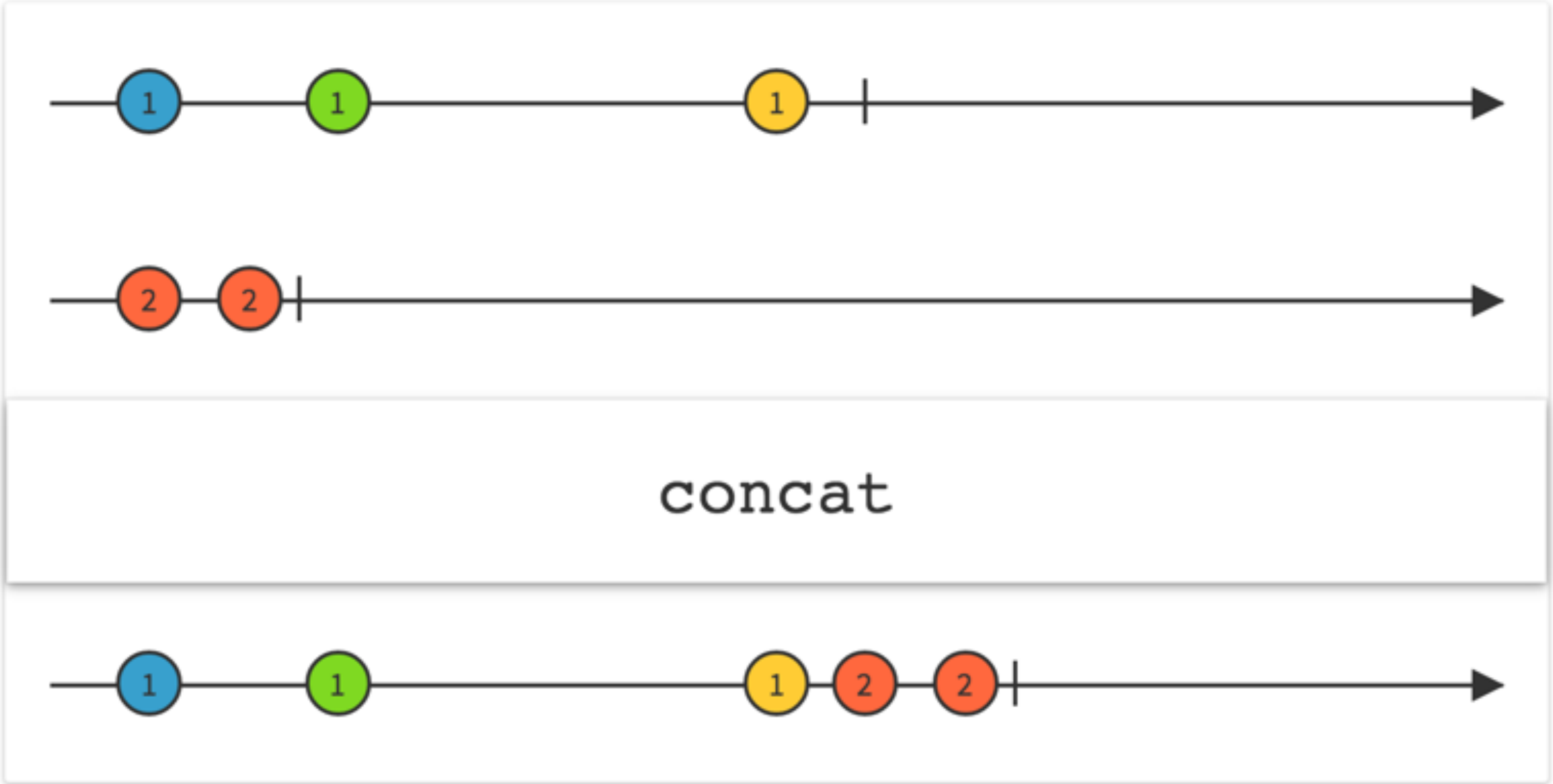
Filtering - **take**:

Emit only the first *n* items emitted by an Observable



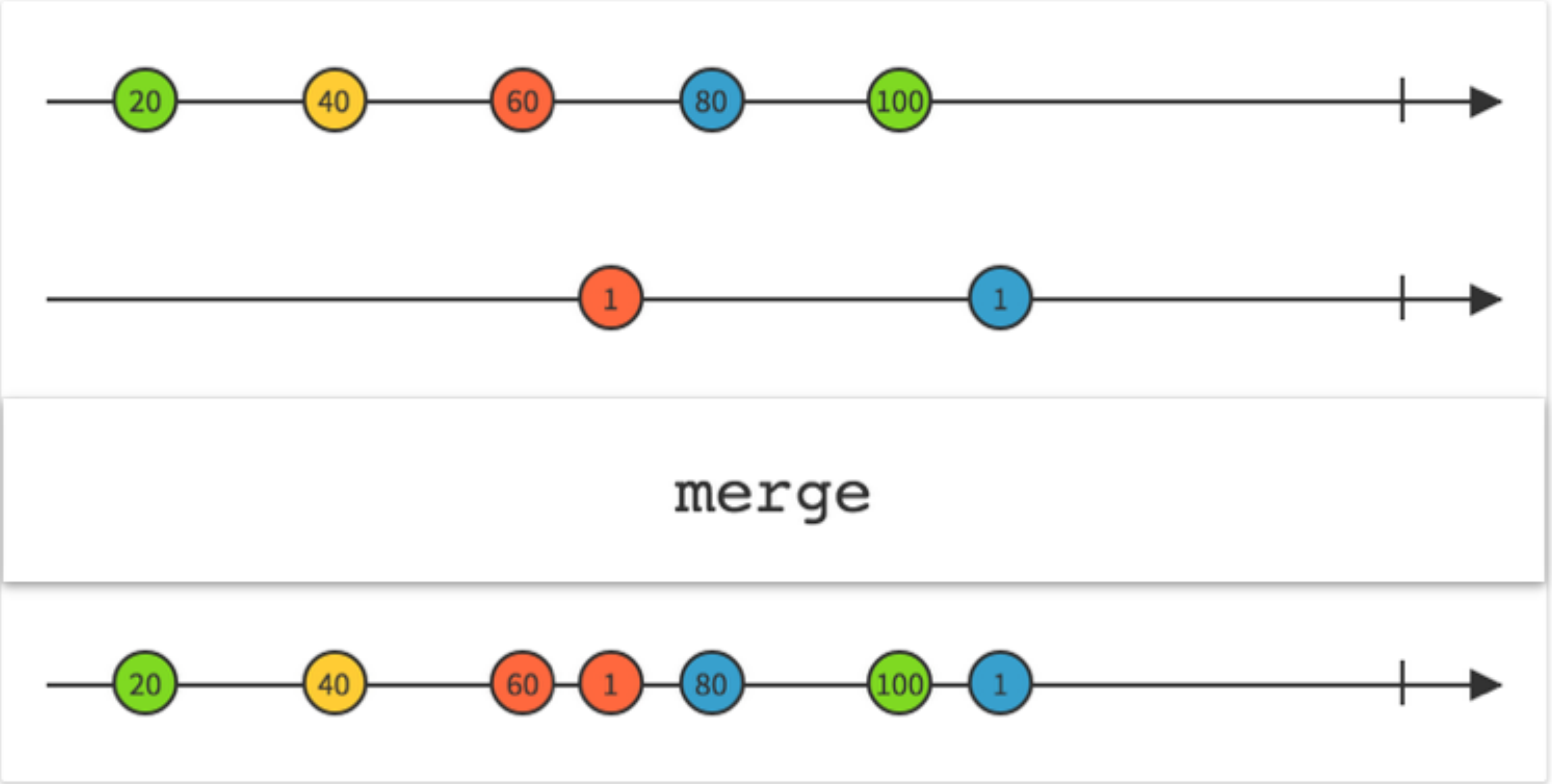
Aggregating - **concat**:

Emit the emissions from two or more Observables without interleaving them



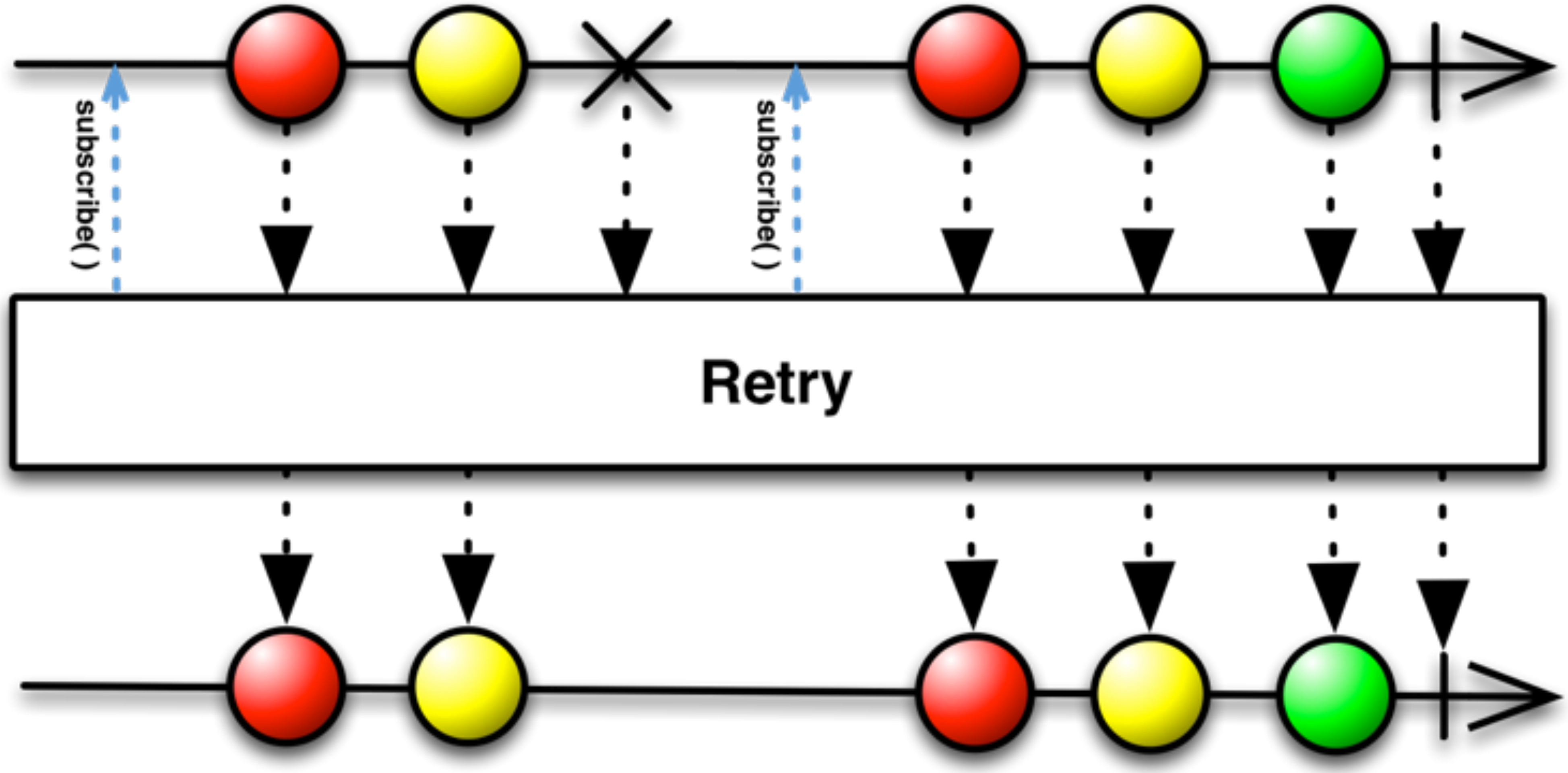
Combining - **merge**:

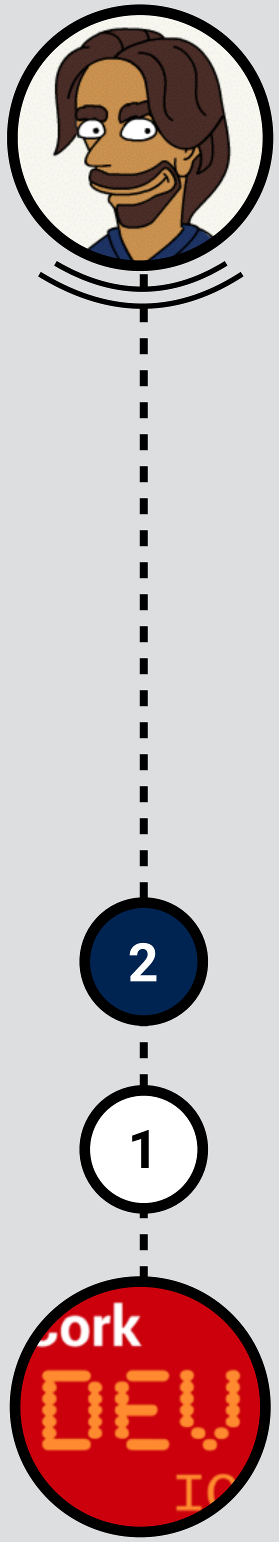
Combine multiple Observables into one by merging their emissions



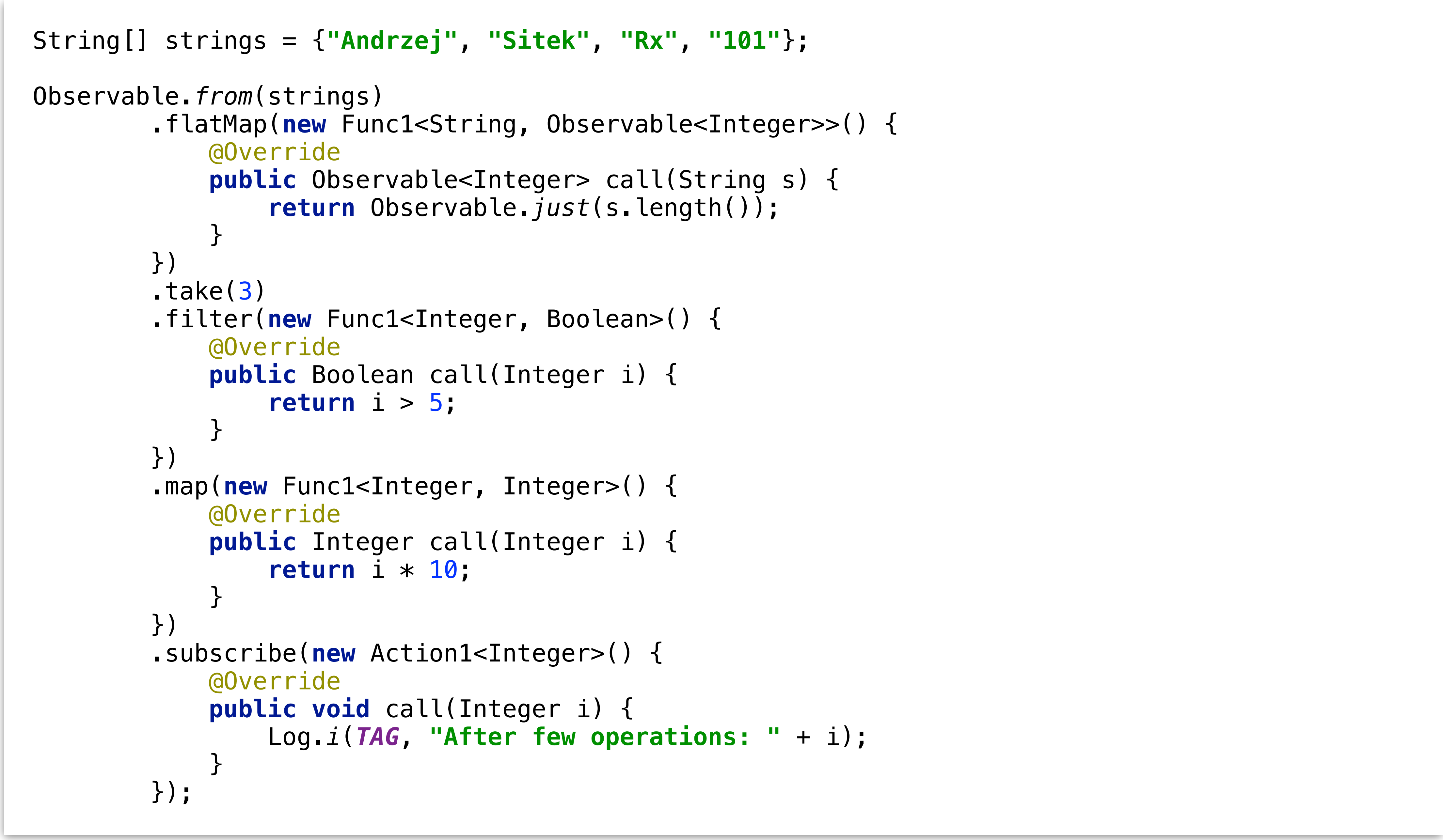
Error handling - **retry**:

If a source Observable emits an error, resubscribe to it in the hopes that it will complete without error





Operators - example:



```
String[] strings = {"Andrzej", "Sitek", "Rx", "101"};

Observable.from(strings)
        .flatMap(new Func1<String, Observable<Integer>>() {
            @Override
            public Observable<Integer> call(String s) {
                return Observable.just(s.length());
            }
        })
        .take(3)
        .filter(new Func1<Integer, Boolean>() {
            @Override
            public Boolean call(Integer i) {
                return i > 5;
            }
        })
        .map(new Func1<Integer, Integer>() {
            @Override
            public Integer call(Integer i) {
                return i * 10;
            }
        })
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer i) {
                Log.i(TAG, "After few operations: " + i);
            }
        });
```

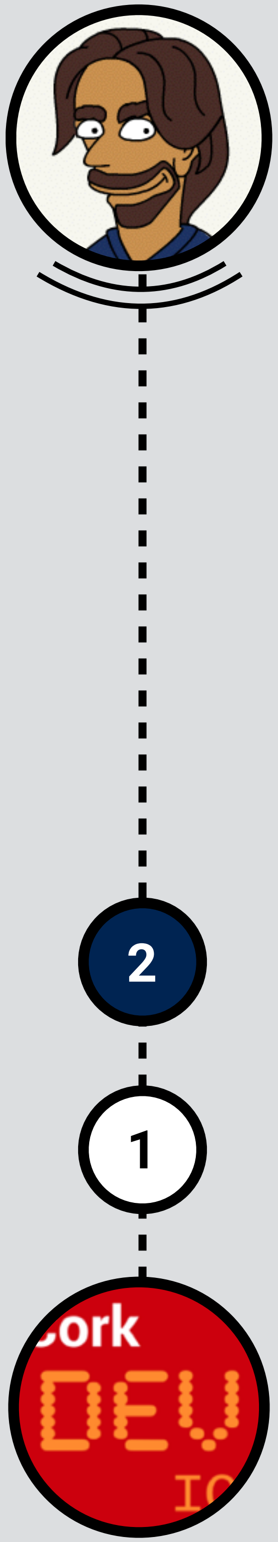
2

1

Operators - example:

```
String[] strings = {"Andrzej", "Sitek", "Rx", "101"};

Observable.from(strings)
        .flatMap(new Func1<String, Observable<Integer>>() {
            @Override
            public Observable<Integer> call(String s) {
                return Observable.just(s.length());
            }
        })
        .take(3)
        .filter(new Func1<Integer, Boolean>() {
            @Override
            public Boolean call(Integer i) {
                return i > 5;
            }
        })
        .map(new Func1<Integer, Integer>() {
            @Override
            public Integer call(Integer i) {
                return i * 10;
            }
        })
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer i) {
                Log.i(TAG, "After few operations: " + i);
            }
        });
```

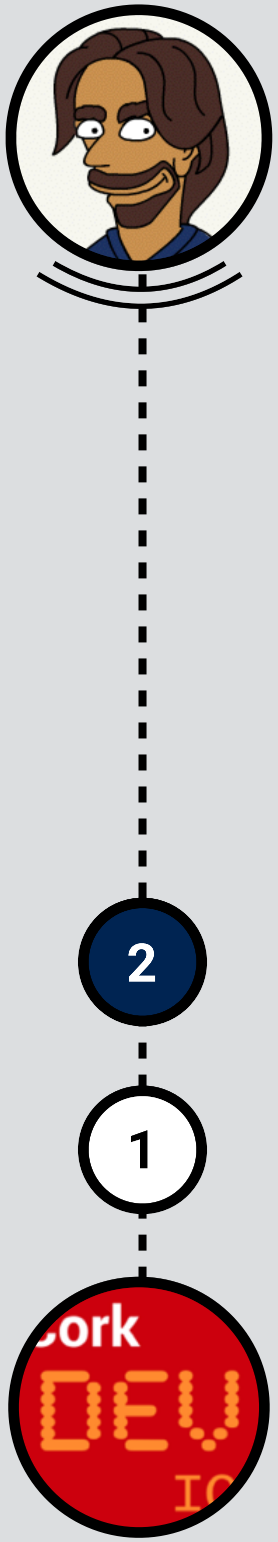
Lots of anonymous classes..
Use lambdas to reduce the boilerplate!



Operators - example simplified:

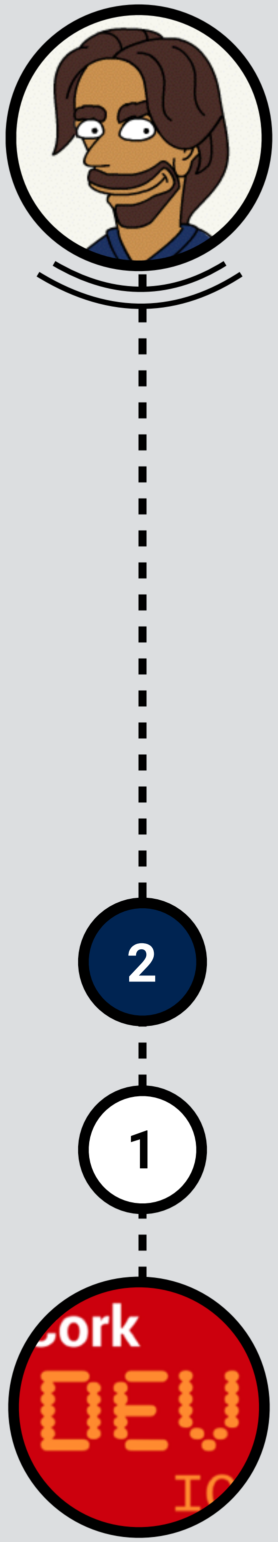
```
String[] strings = {"Andrzej", "Sitek", "Rx", "101"};

Observable.from(strings)
        .flatMap(s -> Observable.just(s.length()))
        .take(3)
        .filter(i -> i > 5)
        .map(i -> i * 10)
        .subscribe(i -> Log.i(TAG, "After few operations: " + i));
```

```
I/MainActivity: After few operations: 70
```



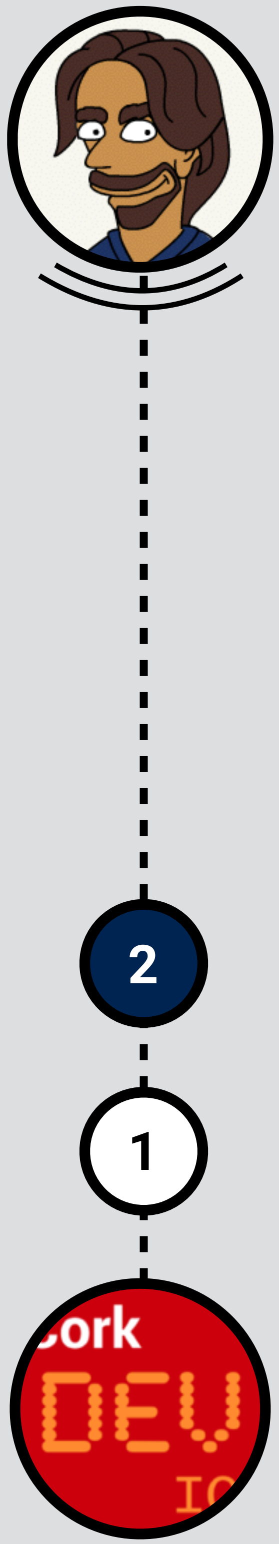
Schedulers:

- They provide **concurrency** for Observables
- Utility operators associated:
  - Observable.observeOn(Scheduler s);
    "perform work on that specific Scheduler"
  - Observable.subscribeOn(Scheduler s);
    "observe the results on that Scheduler"
- Provide different processing strategies such as Thread Pools, Event Loops, Handlers, etc.



Schedulers in RxJava:

Scheduler	Description
Schedulers.computation()	meant for computational work such as event-loops and callback processing
Schedulers.from(executor)	uses the specified Executor as a Scheduler
Schedulers.immediate()	schedules work to begin immediately in the current thread
Schedulers.io()	meant for I/O-bound work such as asynchronous performance of blocking I/O
Schedulers.newThread()	creates a new thread for each unit of work
Schedulers.trampoline()	queues work to begin on the current thread after any already-queued work



Schedulers - example:

```
Observable<String> observable = Observable.create(subscriber -> {
    Log.d(TAG, "Executing on: " + Thread.currentThread());
    subscriber.onNext("Schedulers Example");
    subscriber.onCompleted();
});

observable.subscribeOn(Schedulers.newThread())
        .observeOn(Schedulers.io())
        .subscribe(s -> {
            Log.i(TAG, "Observing on: " + Thread.currentThread());
            Log.i(TAG, s);
        });
```

```
D/MainActivity: Executing on: Thread[RxNewThreadScheduler-1,5,main]
I/MainActivity: Observing on: Thread[RxCachedThreadScheduler-1,5,main]
I/MainActivity: Schedulers Example
```

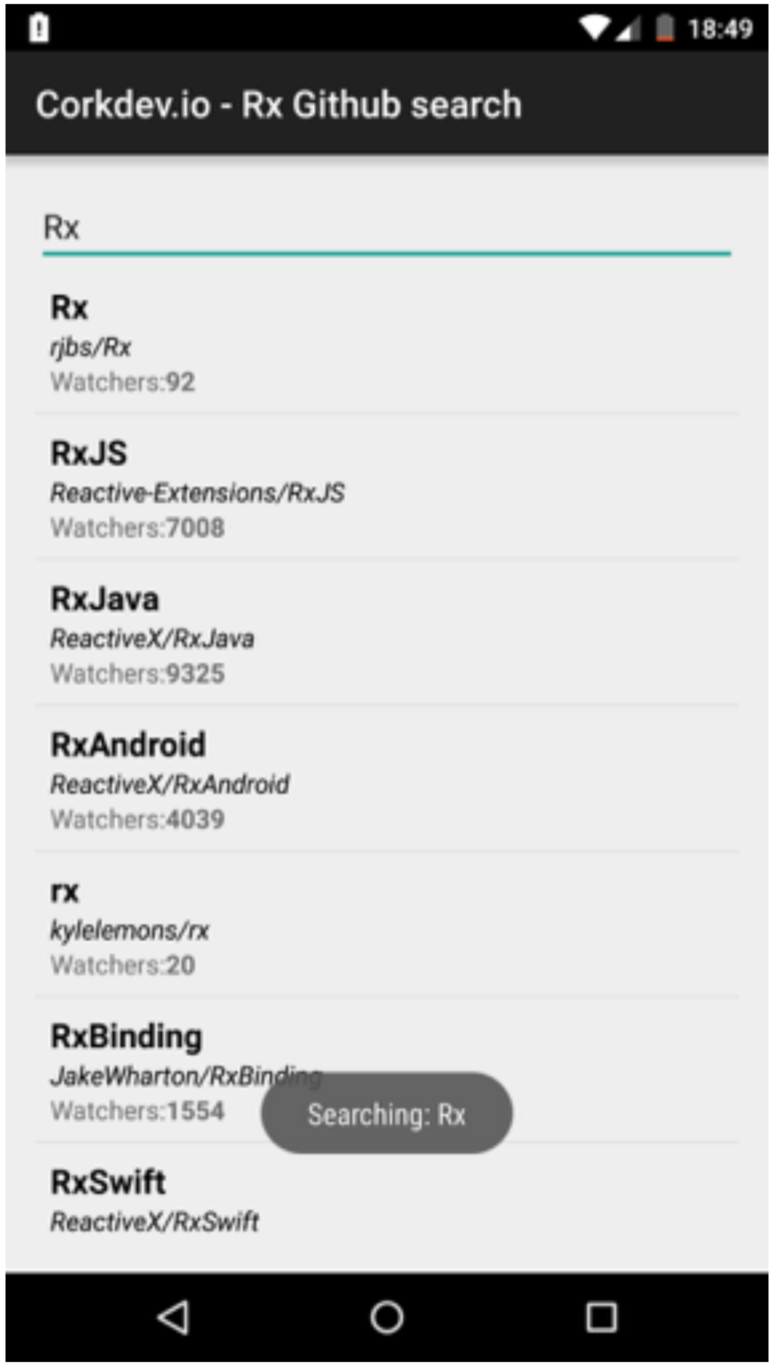
Rx in example

3

2

1

Throttled text search:



18:49

Corkdev.io - Rx Github search

Rx

Rx
rjbs/Rx
Watchers:92

RxJS
Reactive-Extensions/RxJS
Watchers:7008

RxJava
ReactiveX/RxJava
Watchers:9325

RxAndroid
ReactiveX/RxAndroid
Watchers:4039

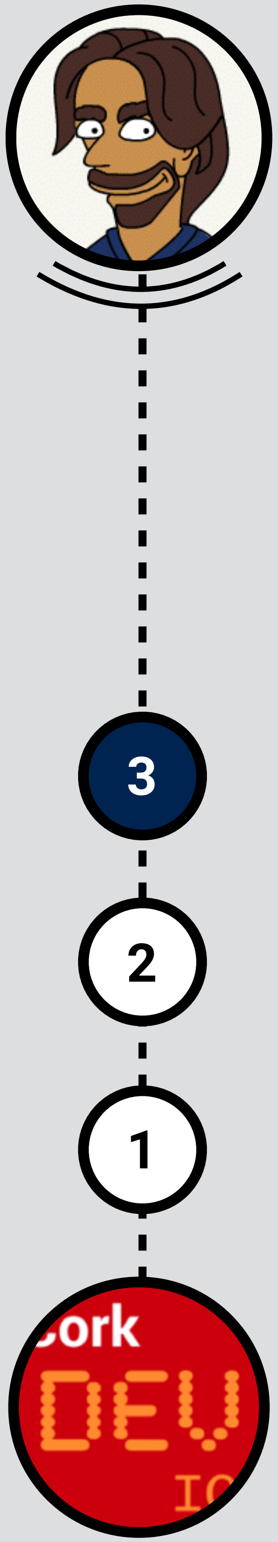
rx
kylelemons/rx
Watchers:20

RxBinding
JakeWharton/RxBinding
Watchers:1554

Searching: Rx

RxSwift
ReactiveX/RxSwift

Rx in example



Throttled text search:

- Search Github repositories using a simple app
- Throttle the text input so it doesn’t send the requests each time you type a character
- Don’t perform the search if the input is empty
- Perform network request for search query
- Filter the list of repositories to leave only those with more than 10 watchers
- Show the results in the list

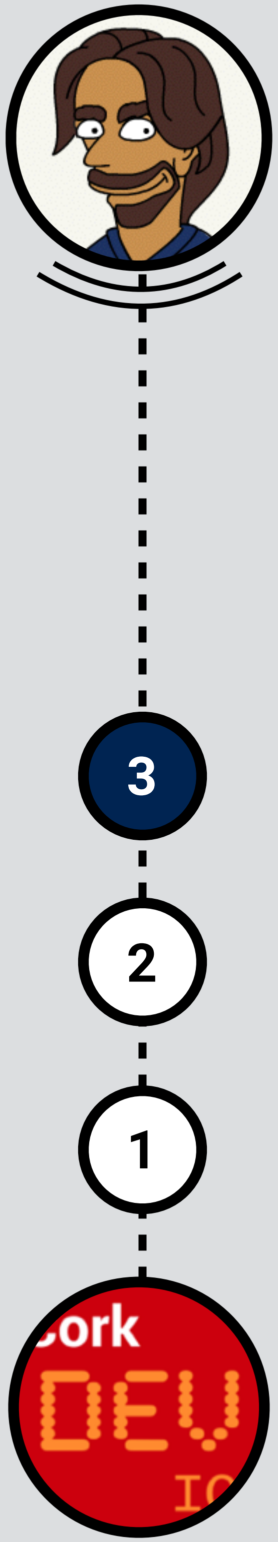
Rx in examples

3

2

1

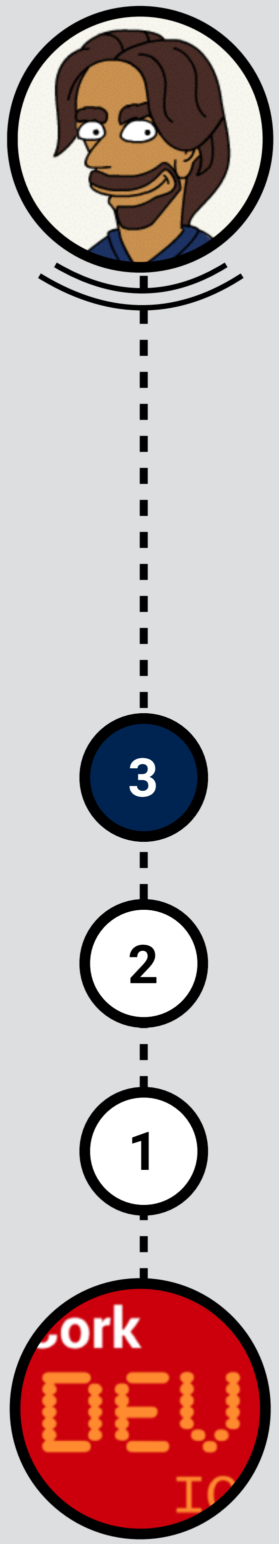
Rx in example



Simplified version - walkthrough:

```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Rx in example

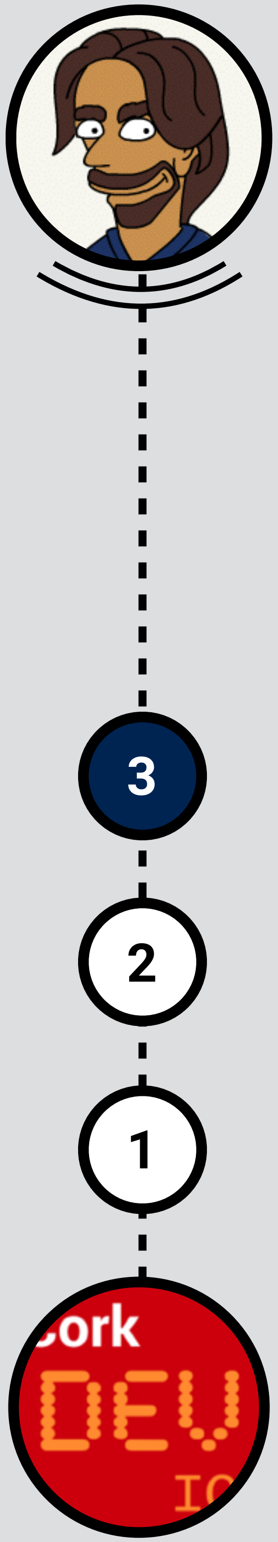


Simplified version - walkthrough:

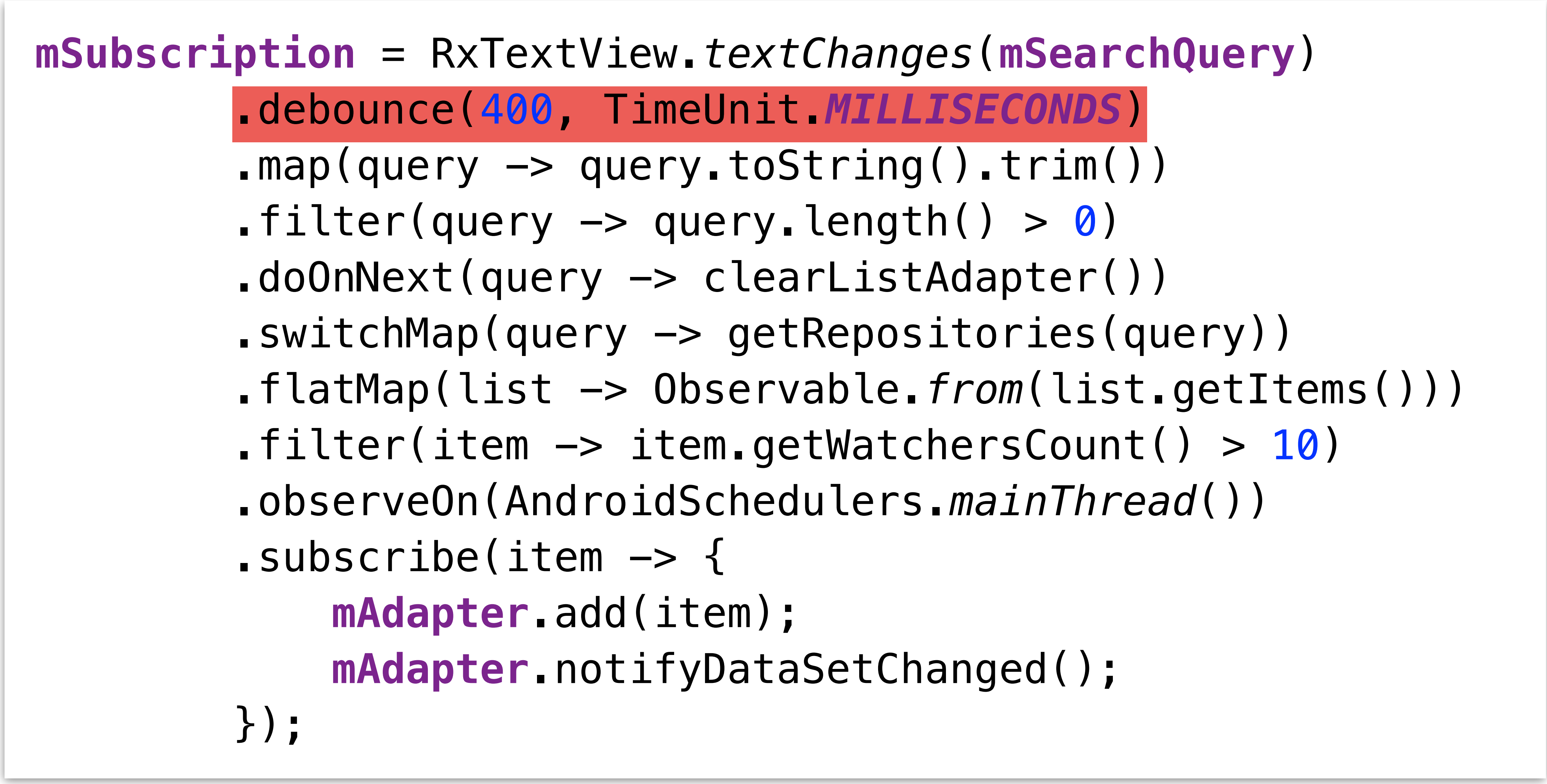
```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Android specific way of creating Observable from EditText

Rx in example



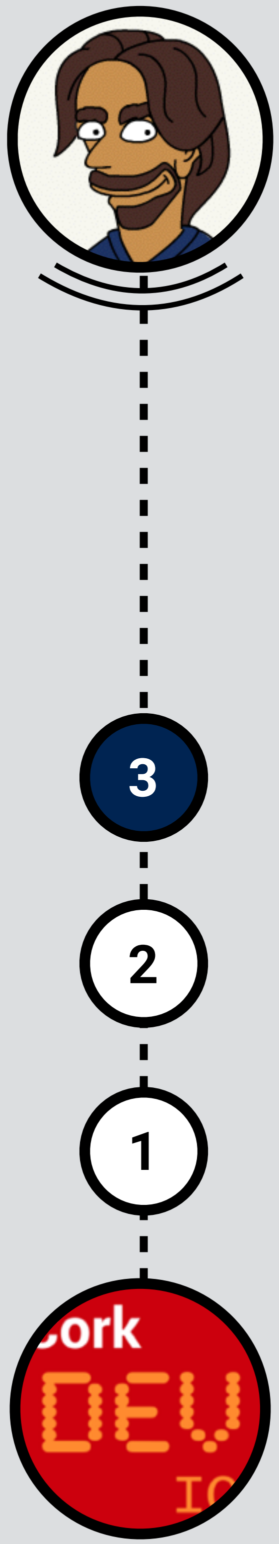
Simplified version - walkthrough:



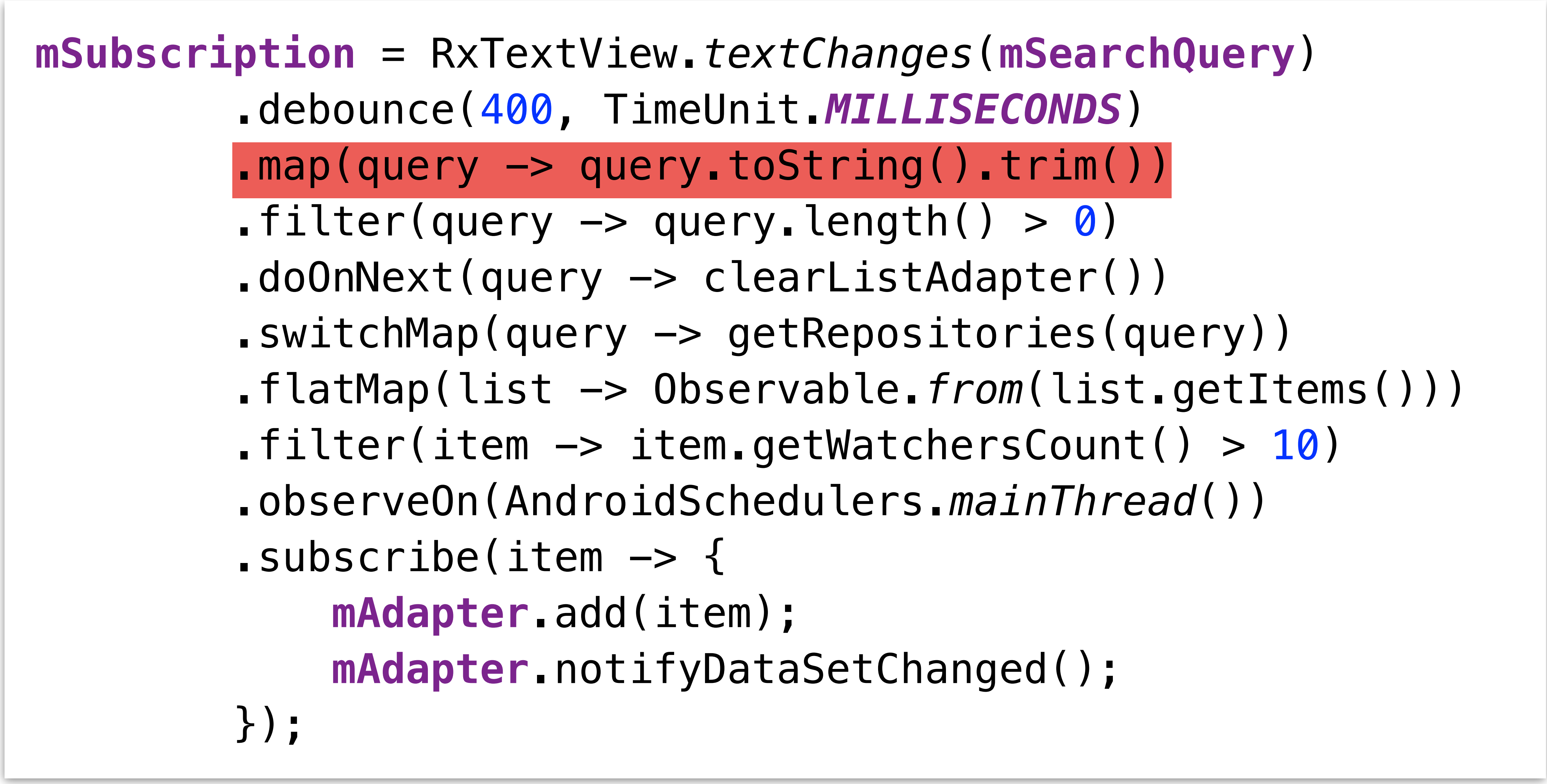
```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Throttle the emission of the events with 400 ms window

Rx in example



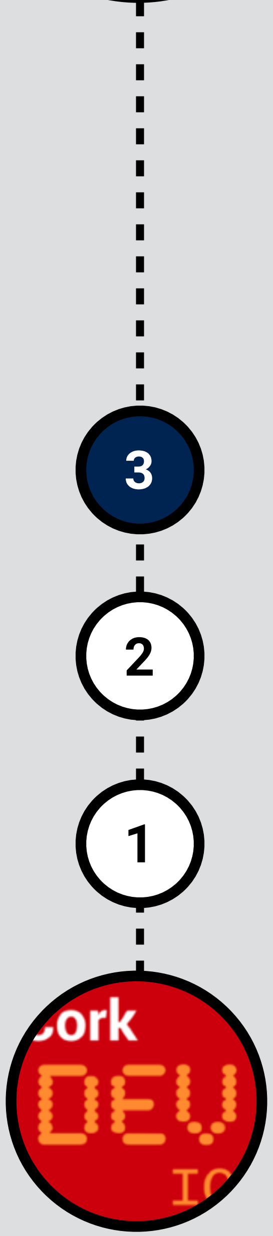
Simplified version - walkthrough:



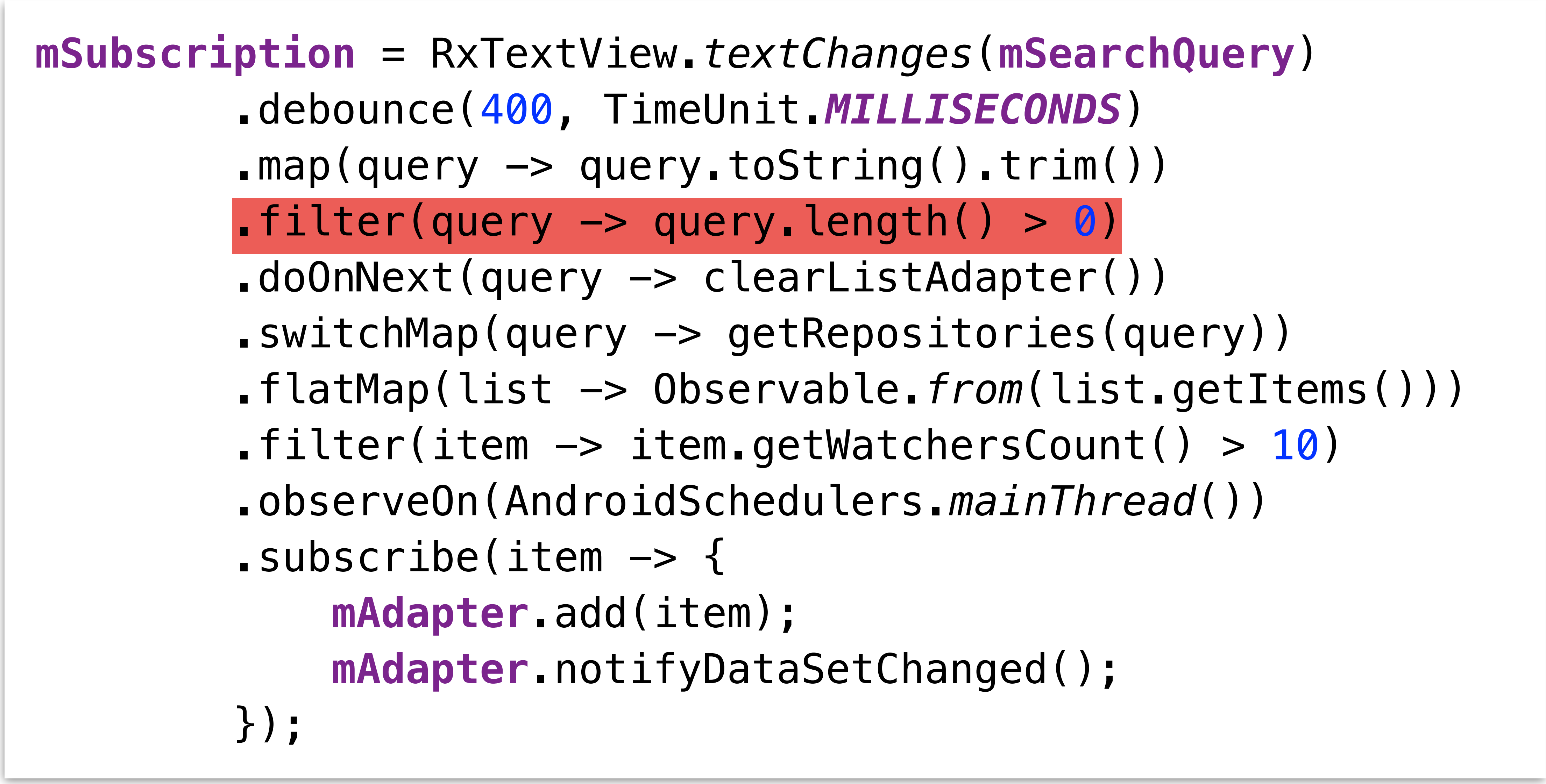
```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Trim the leading and trailing spaces

Rx in example



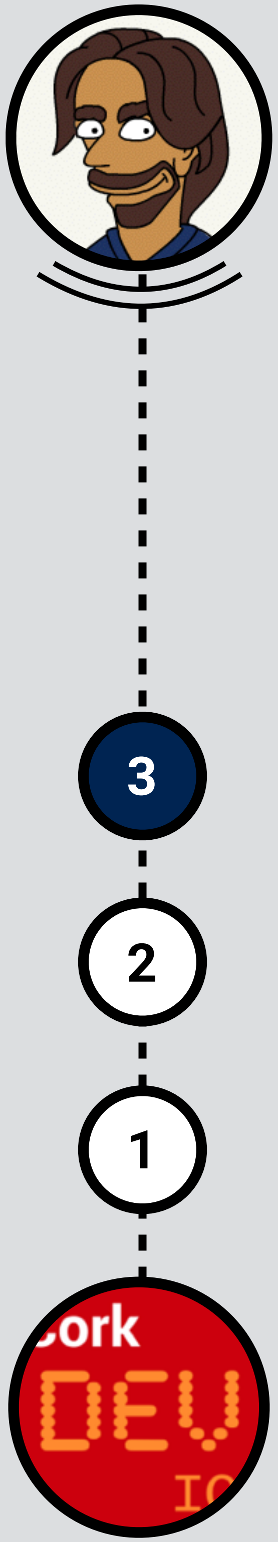
Simplified version - walkthrough:



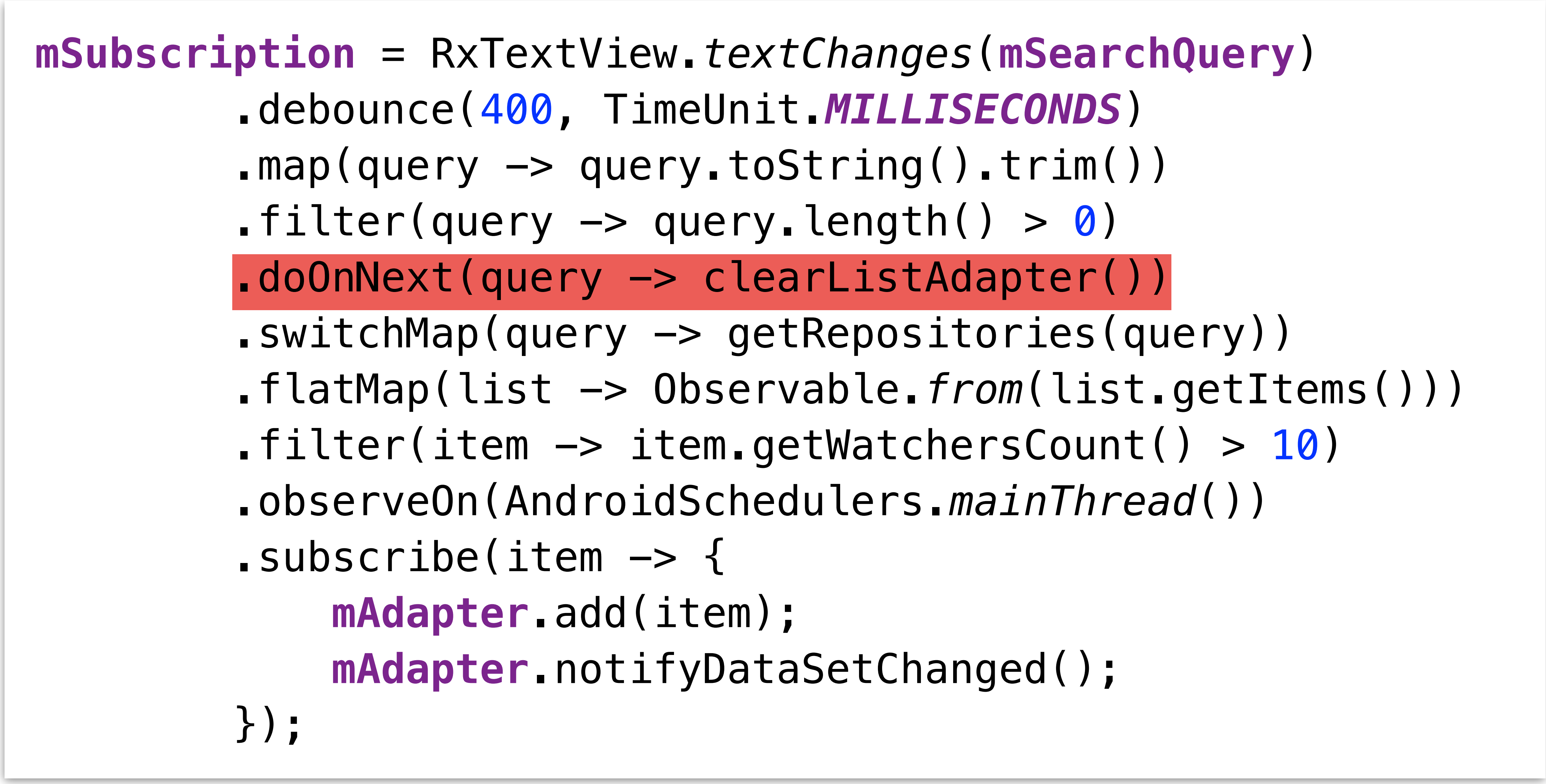
```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Emit the item only if the trimmed query is not empty

Rx in example



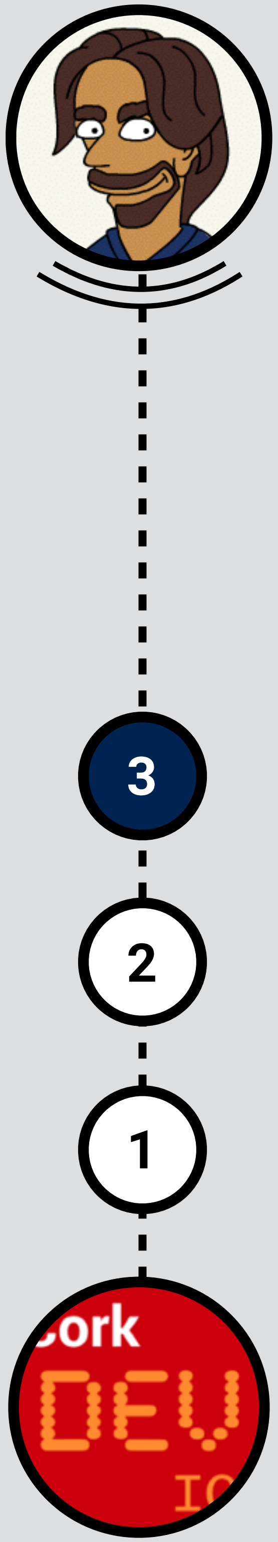
Simplified version - walkthrough:



```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Clear the list view each time a query changes

Rx in example



Simplified version - walkthrough:

```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

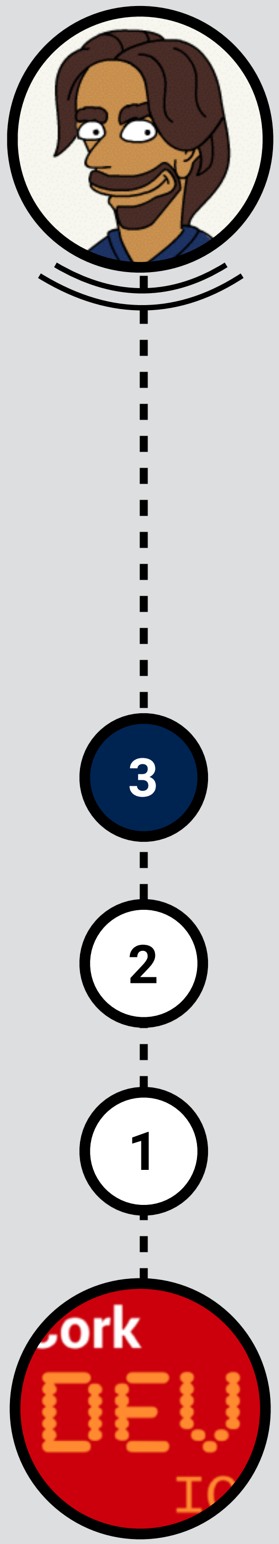
Perform network request based on the query

Simplified version - walkthrough:

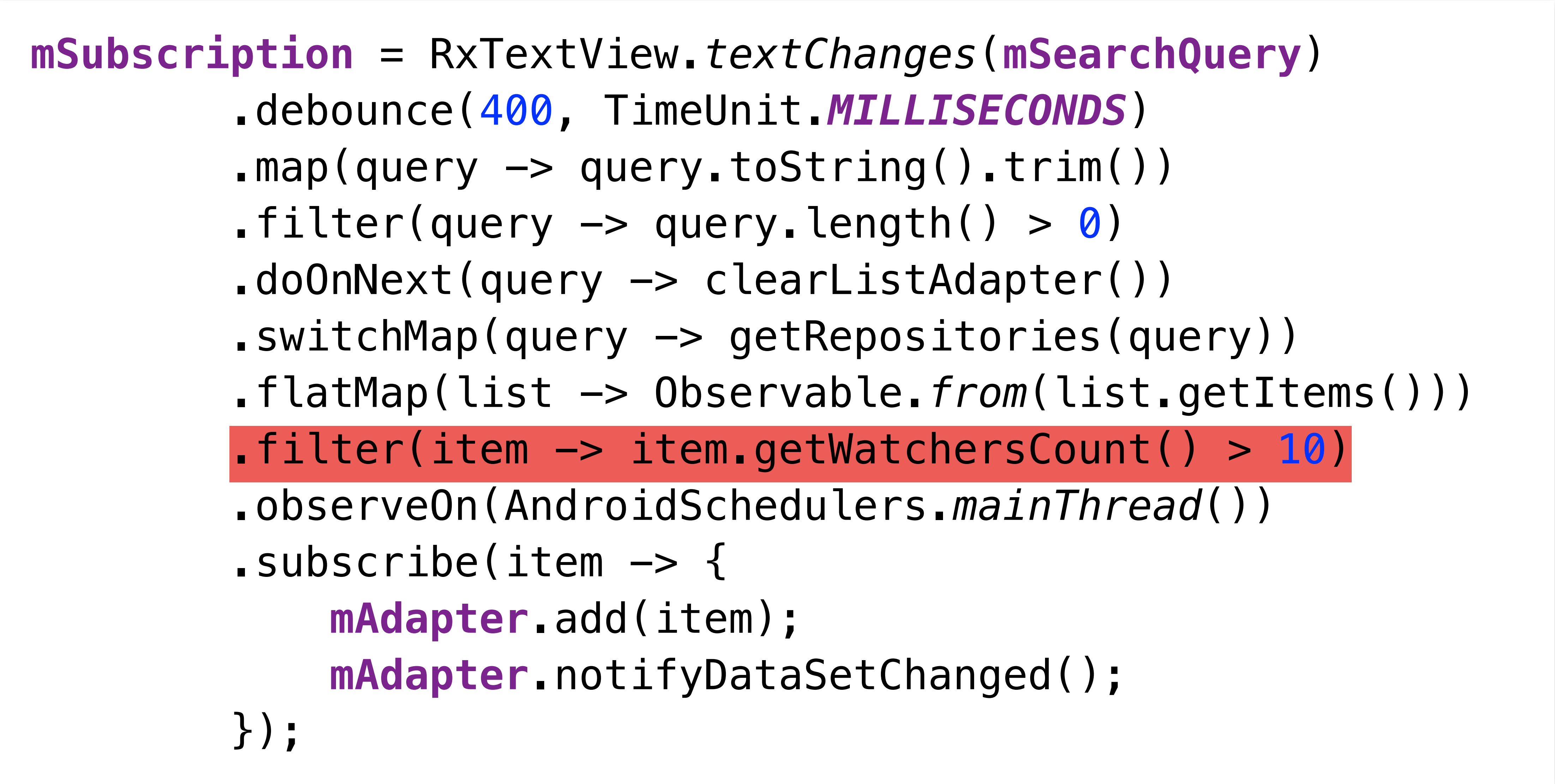
```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Emit single items from the list received in the previous call

Rx in example



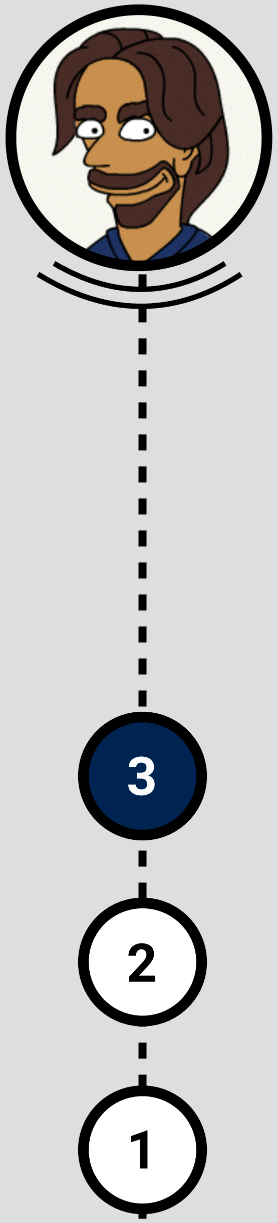
Simplified version - walkthrough:



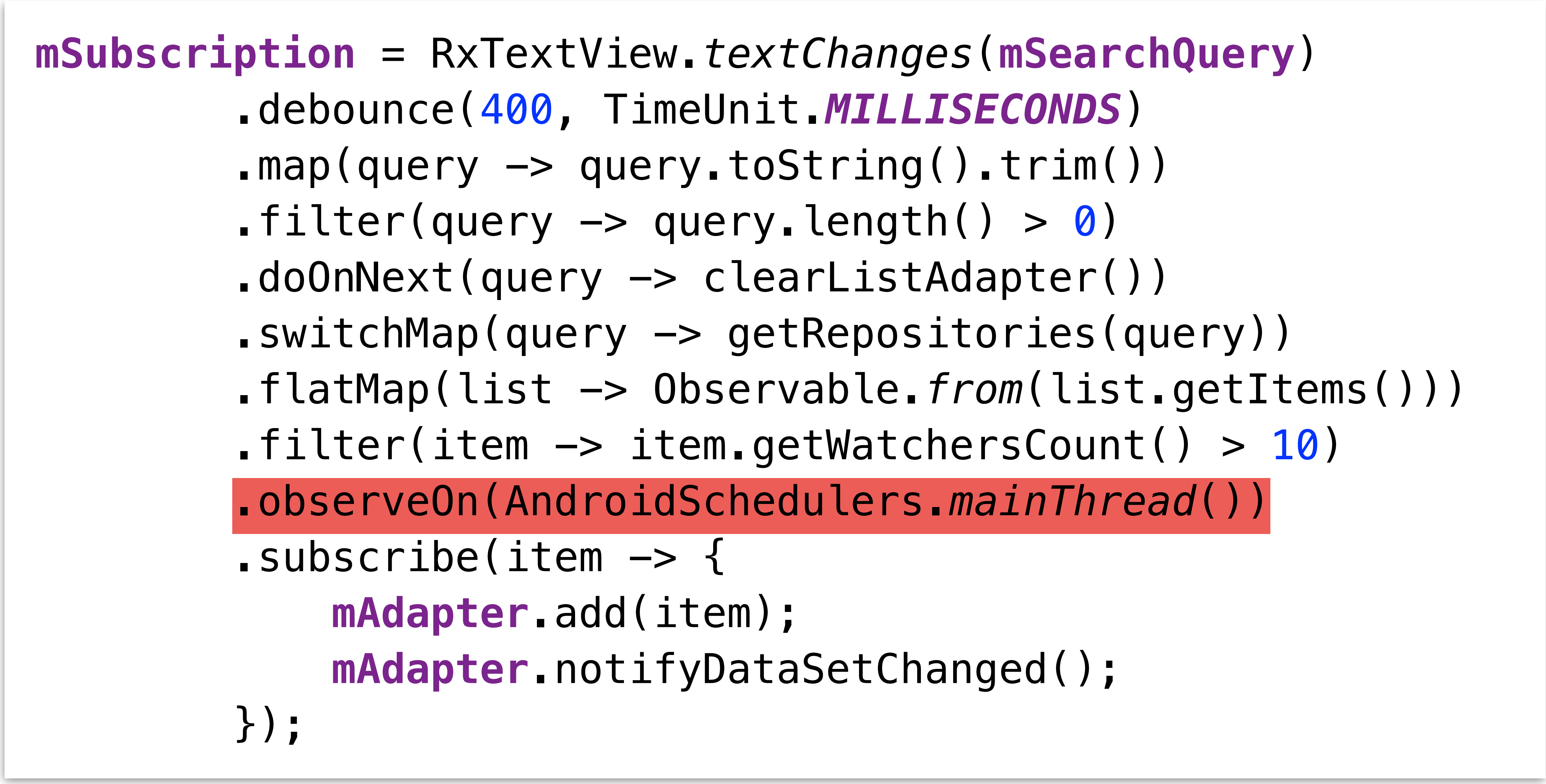
```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Filter the items with more than 10 watchers

Rx in example



Simplified version - walkthrough:

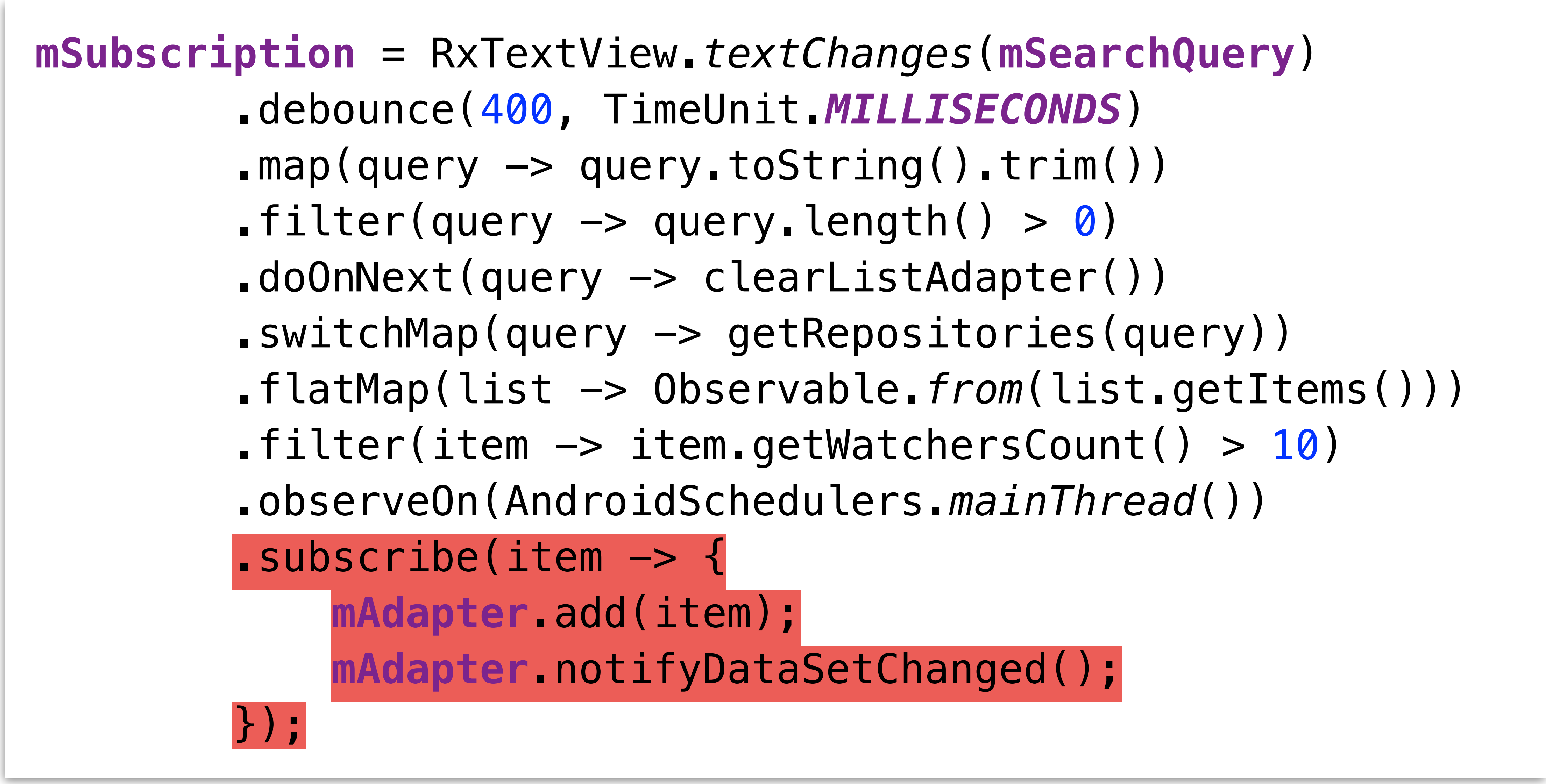


```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Handle the results on Android’s UI Thread

Rx in example

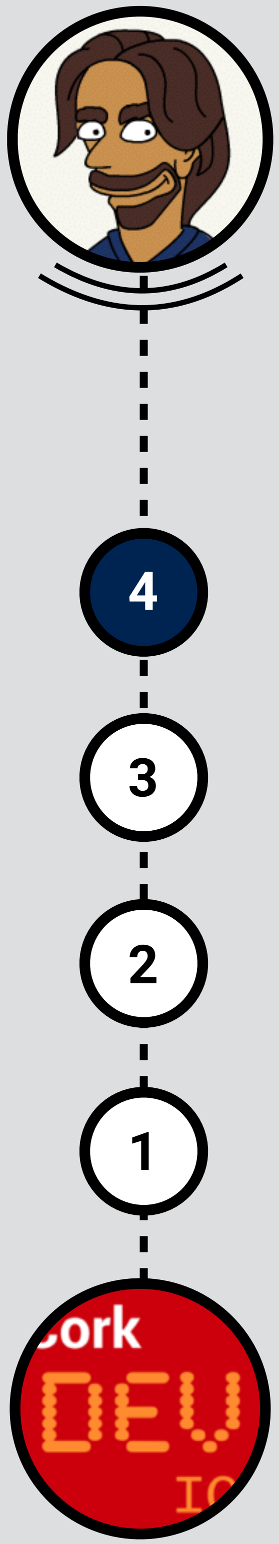
Simplified version - walkthrough:



```
mSubscription = RxTextView.textChanges(mSearchQuery)
        .debounce(400, TimeUnit.MILLISECONDS)
        .map(query -> query.toString().trim())
        .filter(query -> query.length() > 0)
        .doOnNext(query -> clearListAdapter())
        .switchMap(query -> getRepositories(query))
        .flatMap(list -> Observable.from(list.getItems()))
        .filter(item -> item.getWatchersCount() > 10)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(item -> {
            mAdapter.add(item);
            mAdapter.notifyDataSetChanged();
        });
```

Create the Subscription and add item by item to the list

Tips & resources



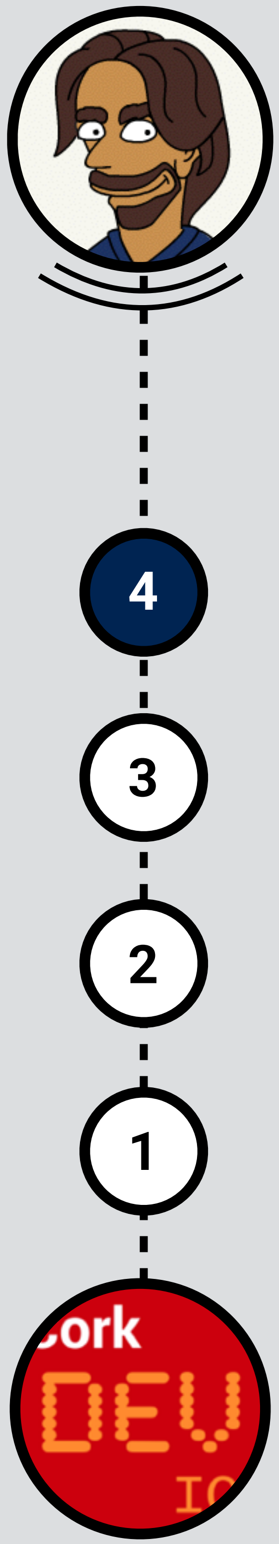
Useful tips:

- First, master the rules and then go beyond them!
- Rx is single-threaded by default!
- Use it only when it makes sense - it’s very tempting to use Rx everywhere..
- Side effect methods doOnNext, doOnError are very useful for debugging.
- Lots of operators are too hard to learn by heart - explore them with docs and marble diagrams.

Tips & resources

And the most important one:

Tips & resources



Useful resources:

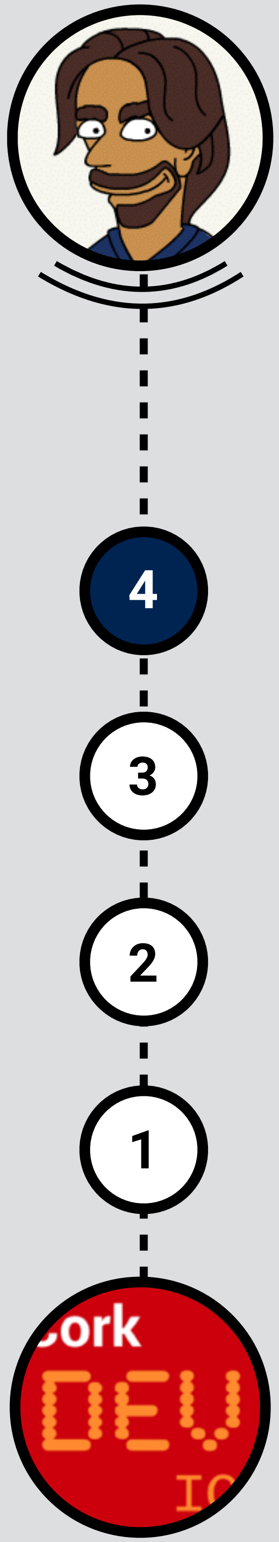
- ReactiveX:
  http://reactivex.io/

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

- The intro to Rx you've been missing:
  https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

- RxMarbles:
  http://rxmarbles.com/

Tips & resources



Useful resources:

- RxJava plugins:
  https://github.com/ReactiveX/RxJava/wiki/Plugins
- Async JavaScript at Netflix:
  https://www.youtube.com/watch?v=XRYN2xt11Ek
- Intro to Rx (website):
  http://introtorx.com/
- A Playful Introduction to Rx:
  https://www.youtube.com/watch?v=WKore-AkisY

Q&A

onComplete((s) -> Log.i(“Thank you!”));

Thanks for your attention!
I’d really appreciate your feedback!

Stay in touch!

@andrzej_sitek

+AndrzejSitek