

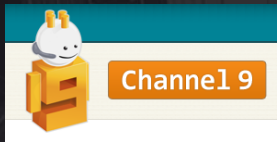
RxJava:

Functional Reactive Programming on Android

Alexey Chernin, Software Developer



Based on:



“Rx in 15 Minutes” - Erik Meijer

REACT 2014
London, April 7th - 9th

“What does it mean to be Reactive” - Erik Meijer

LambdaJam – July 2013

“Functional Reactive Programming in the Netflix API”
- Ben Christensen



“Rx-Fy all the things!” - Benjamin Augustin

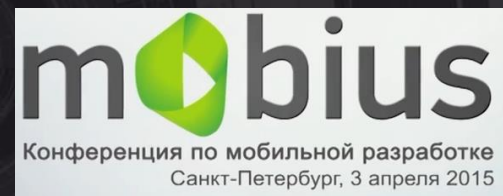
Droidcon NYC
2014

“RxJava Easy Wins” - Ron Shapiro



“Android reactive programming with Rxjava”
- Ivan Morgillo

Based on:



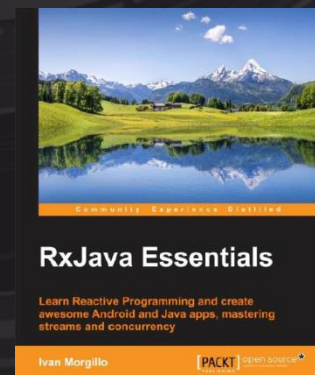
“Реактивный двигатель для вашего Android приложения”
- Матвей Мальков



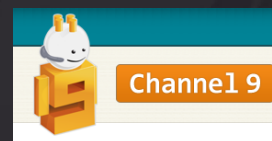
“Learning RxJava (for Android) by example” - Kaushik Gopal



“Reactive Programming in Java 8 With RxJava”
- Russell Elledge



"RxJava Essentials" by
Ivan Morgillo, May 2015



“Rx Workshop”



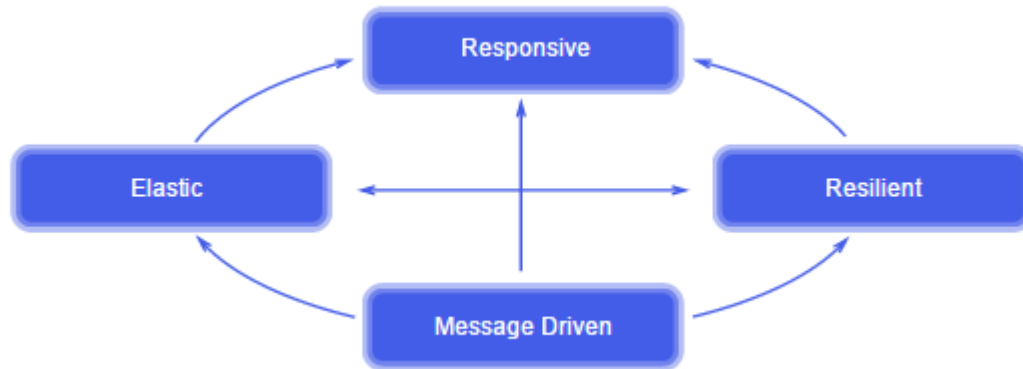
<https://github.com/ReactiveX/RxJava/wiki>

The Reactive Manifesto

Reactive Systems are:

- ✓ **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 to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages.

Published on September 16 2014. (v2.0)



www.reactivemanifesto.org

From .NET to RxJava



Functional reactive programming is an idea from the late 90s that inspired **Erik Meijer**, a computer scientist at Microsoft, to design and develop the Microsoft Rx library.

“Reactive programming is a programming paradigm based on the concept of an asynchronous data flow. A data flow is like a river: it can be observed, filtered, manipulated, or merged with a second flow to create a new flow for a new consumer”. Ivan Morgillo

In 2012, at Netflix, they started to port .NET Rx to the JVM. With a post on the Netflix tech blog in February 2013, **Ben Christensen** and Jafar Husain showed RxJava to the world for the first time.



Pure functions

Pure functions always returns the same result for a given set of parameter values.

- ✓ No side effects caused by Class or Instance state.
- ✓ No side effects caused by I/O devices.
- ✓ No time related side effects.

```
function1(5) == 10 for all calls
function2(5) == 11
function2(5) == 12
function2(5) == 10
function2(5) == 11
function2(5) == 12
function2(5) == 10
function2(5) == 11
function2(5) == 12
function2(5) == 10
function2(5) == 11
function2(5) == 12
```

```
1 public class SideEffectDemo {
2
3     private int state;
4
5     public int function1(int value) {
6         return value * 2;
7     }
8
9     public int function2(int value) {
10        state = (++state % 3);
11        return value * 2 + state;
12    }
13
14    public static void main(String[] args) {
15        SideEffectDemo demo = new SideEffectDemo();
16
17        for (int i = 0; i < 100500; i++) {
18            if (demo.function1(5) != 10) {
19                throw new IllegalStateException();
20            }
21        }
22        System.out.println("function1(5) = 10 for all calls");
23
24        for (int i = 0; i < 100500; i++) {
25            System.out.println("function2(5) = " + demo.function2(5));
26        }
27    }
28 }
```

Pure functions: example 1

```
6 public static void main(String[] args) {
7     BiFunction<String, String, String> concatFunction = (s, t) -> { return s + t; };
8     System.out.println(concatFunction.apply("Hello ", "World!"));
9
10    concatFunction = FunctionsDemo::ConcatFunctionStatic;
11    System.out.println(concatFunction.apply("Hello ", "World!"));
12
13    FunctionsDemo demo = new FunctionsDemo();
14    concatFunction = demo::concatFunctionInstance;
15    System.out.println(concatFunction.apply("Hello ", "World!"));
16
17    System.out.println(concatAndTransform("Hello ", "World!", (s) -> { return s.toUpperCase(); }));
18
19    Function<String, String> transformToLower = (s) -> { return s.toLowerCase(); };
20    System.out.println(concatAndTransform("Hello ", "World!", transformToLower));
21 }
22
23 public static String ConcatFunctionStatic(String a, String b) { return a + b; }
24
25 public String concatFunctionInstance(String a, String b) { return a + b; }
26
27 public static String concatAndTransform(String a, String b, Function<String, String> stringTransform) {
28     if (stringTransform != null) {
29         a = stringTransform.apply(a);
30         b = stringTransform.apply(b);
31     }
32     return a + b;
33 }
34 }
```

Hello World!
Hello World!
Hello World!
HELLO WORLD!
hello world!

Pure functions: example 2

```
4 public class FunctionsDemo2 {
5
6     public static void main(String[] args) {
7         Supplier<String> stringFromFunction =
8             createCombineAndTransform("Hello ", "World!", (s) -> {
9             return s.toUpperCase();
10         });
11         System.out.println(stringFromFunction.get());
12     }
13
14     public static Supplier<String> createCombineAndTransform(
15         final String a,
16         final String b,
17         final Function<String, String> transformer) {
18         return () -> {
19             String aa = a;
20             String bb = b;
21             if (transformer != null) {
22                 aa = transformer.apply(a);
23                 bb = transformer.apply(b);
24             }
25             return aa + bb;
26         };
27     }
28 }
```

HELLO WORLD!

- Store function as a variable
 - Pass a function as a parameter
 - Function can return a function
-
- ✓ Composition
 - ✓ Lazy execution

Butter Knife by Jake Wharton

```
17 private TextView textView;
18 @Bind(R.id.activity_main_title)
19 TextView title;
20
21 @Override
22 protected void onCreate(Bundle savedInstanceState) {
23     super.onCreate(savedInstanceState);
24     setContentView(R.layout.activity_main);
25     ButterKnife.bind(this);
26
27     //Instead of this slow reflection.
28     textView = (TextView) findViewById(R.id.activity_main_title);
29     Button buttonView = (Button) findViewById(R.id.activity_main_button);
30     buttonView.setOnClickListener(new View.OnClickListener() {
31         @Override
32         public void onClick(View v) {
33             textView.setText("Hello World!");
34         }
35     });
36 }
37
38 @OnClick(R.id.activity_main_button)
39 public void onClickButton(View view) {
40     title.setText("Hello World!");
41 }
```

Retrofit by Square

```
8 public interface RetrofitService {
9     @GET("/auth")
10     String login1(@Query("login") String login, @Query("password") String password);
11     @GET("/auth")
12     void login2(@Query("login") String login, @Query("password") String password,
13                 Callback<String> callback);
14     @GET("/auth")
15     Observable<String> login3(@Query("login") String login,
16                               @Query("password") String password);
```

```
52 RestAdapter restAdapter = new RestAdapter.Builder()
53     .setEndpoint("https://api.source.com")
54     .setLogLevel(RestAdapter.LogLevel.FULL)
55     .build();
56 RetrofitService service = restAdapter.create(RetrofitService.class);
57 String token = service.login1("login", "password");
58 service.login2("login", "password", new Callback<String>() {
59     @Override
60     public void success(String token, Response response) {
61         Log.d("TOKEN", token);
62     }
63     @Override
64     public void failure(RetrofitError error) {
65         Log.d("ERROR", error.toString());
66     }
67 });
```

Why Retrofit?

Fast:

	One Discussion	Dashboard (7 requests)	25 Discussions
AsyncTask	941 ms	4,539 ms	13,957 ms
Volley	560 ms	2,202 ms	4,275 ms
Retrofit	312 ms	889 ms	1,059 ms

<https://instructure.github.io/blog/2013/12/09/volley-vs-retrofit>

Supports:

- Gson - JSON serialization.
- Simple - XML serialization.
- OkHttp - HTTP client.
- **Robospice** - asynchronous network requests.

Droidcon NYC
2014

“Retrofit” - Jacob Tabak



“A Few Ok Libraries”
- **Jake Wharton**

The old way

```
1 package com.tusharp.droidcon2014retrofitexample.data.foursquare.legacy;
2 import java.io.*;
3
4 public class VenueSearchTask extends AsyncTask<NameValuePair, Void, List<Venue>> {
5     public static final String TAG = "VenueSearchTask";
6
7     @Override
8     public List<Venue> doInBackground(NameValuePair... params) {
9         try {
10             String endpoint = "https://api.foursquare.com/v2/venues/search";
11             String query = buildQuery(params);
12             URL url = new URL(endpoint + query);
13             HttpURLConnection con = (HttpURLConnection) url.openConnection();
14             String response = readStream(con.getInputStream());
15             List<Venue> venues = parseJson(response);
16
17             } catch (IOException e) {
18                 Log.e(TAG, "Error retrieving venues", e);
19             } catch (JSONException e) {
20                 Log.e(TAG, "Error parsing venues", e);
21             }
22             return null;
23         }
24
25         private String buildQuery(NameValuePair[] params) throws UnsupportedEncodingException {
26             StringBuilder queryBuilder = new StringBuilder();
27             for (NameValuePair param : params) {
28                 String value = param.getValue();
29                 value = Uri.encode(value, "UTF-8");
30                 queryBuilder.append(queryBuilder.length() > 0 ? "&" : "?");
31                 queryBuilder.append(param.getName()).append("=").append(value);
32             }
33             return queryBuilder.toString();
34         }
35
36         private String readStream(InputStream in) throws IOException {
37             BufferedReader reader = null;
38             StringBuilder responseBuilder = new StringBuilder();
39             try {
40                 reader = new BufferedReader(new InputStreamReader(in));
41                 String line;
42                 while ((line = reader.readLine()) != null) {
43                     responseBuilder.append(line);
44                 }
45             } finally {
46                 if (reader != null) {
47                     reader.close();
48                 }
49             }
50             return responseBuilder.toString();
51         }
52
53         private List<Venue> parseJson(String response) throws JSONException {
54             List<Venue> venues = new ArrayList<>();
55             JSONObject responseJson = new JSONObject(response);
56             JSONArray venuesJson = responseJson.getJSONArray("venues");
57             for (int i = 0; i < venuesJson.length(); i++) {
58                 JSONObject venueJson = venuesJson.getJSONObject(i);
59                 Venue venue = new Venue();
60                 venue.setId(venueJson.getString(Venue.FIELD_ID));
61                 venue.setName(venueJson.getString(Venue.FIELD_NAME));
62                 venue.setLocation(venueJson.getJSONObject(Venue.FIELD_LOCATION));
63                 venue.setContact(venueJson.getJSONObject(Venue.FIELD_CONTACT));
64                 venue.setPhone(venueJson.getString(Venue.FIELD_PHONE));
65                 venue.setWebsite(venueJson.getString(Venue.FIELD_WEBSITE));
66                 venue.setHours(venueJson.getJSONObject(Venue.FIELD_HOURS));
67                 venue.setCrossStreet(venueJson.getString(Venue.FIELD_CROSS_STREET));
68                 venue.setAddress(venueJson.getString(Venue.FIELD_ADDRESS));
69                 venue.setCity(venueJson.getString(Venue.FIELD_CITY));
70                 venue.setState(venueJson.getString(Venue.FIELD_STATE));
71                 venue.setPostalCode(venueJson.getString(Venue.FIELD_POSTAL_CODE));
72                 venue.setCountry(venueJson.getString(Venue.FIELD_COUNTRY));
73                 venue.setLat(venueJson.getDouble(Venue.FIELD_LAT));
74                 venue.setLong(venueJson.getDouble(Venue.FIELD_LONG));
75                 venue.setDistance(venueJson.getDouble(Venue.FIELD_DISTANCE));
76                 venue.setLocation(venueJson.getJSONObject(Venue.FIELD_LOCATION));
77
78                 JSONArray categoriesJson = venueJson.getJSONArray(Venue.FIELD_CATEGORIES);
79                 List<Category> categories = new ArrayList<>(categoriesJson.length());
80                 for (int j = 0; j < categoriesJson.length(); j++) {
81                     JSONObject categoryJson = categoriesJson.getJSONObject(j);
82                     Category category = new Category();
83                     category.setId(categoryJson.getString(Category.FIELD_ID));
84                     category.setName(categoryJson.getString(Category.FIELD_NAME));
85                     category.setPrimaryName(categoryJson.getString(Category.FIELD_PRIMARY_NAME));
86                     category.setShortName(categoryJson.getString(Category.FIELD_SHORT_NAME));
87
88                     JSONArray iconJson = categoryJson.getJSONArray(Category.FIELD_ICON);
89                     Icon icon = new Icon();
90                     icon.setPrefix(iconJson.getString(Icon.FIELD_PREFIX));
91                     icon.setSuffix(iconJson.getString(Icon.FIELD_SUFFIX));
92                     icon.setUrl(iconJson.getString(Icon.FIELD_URL));
93
94                     category.setIcon(icon);
95                 }
96                 categories.add(category);
97             }
98             venue.setCategories(categories);
99
100             venue.setVerified(venueJson.getBoolean(Venue.FIELD_VERIFIED));
101
102             JSONObject stationJson = venueJson.getJSONObject(Venue.FIELD_STATION);
103             Station station = new Station();
104             station.setCheckinCount(stationJson.getInt(Station.FIELD_CHECKIN_COUNT));
105             station.setOwnerCount(stationJson.getInt(Station.FIELD_OWNER_COUNT));
106             station.setFollowerCount(stationJson.getInt(Station.FIELD_Follower_COUNT));
107             venue.setStation(station);
108
109             venue.setUrl(venueJson.getString(Venue.FIELD_URL));
110
111             JSONObject menuJson = venueJson.getJSONObject(Venue.FIELD_MENU);
112             if (menuJson != null) {
113                 Menu menu = new Menu();
114                 menu.setMenuId(menuJson.getString(Menu.FIELD_ID));
115                 menu.setPhotoUrl(menuJson.getString(Menu.FIELD_PHOTO_URL));
116                 menu.setMenu(menuJson.getJSONObject(Menu.FIELD_MENU));
117             }
118             venues.add(venue);
119         }
120         return venues;
121     }
122 }
```

```
30
31 public class VenueSearchTask extends AsyncTask<NameValuePair, Void, List<Venue>> {
32     public static final String TAG = "VenueSearchTask";
33
34     @Override
35     public List<Venue> doInBackground(NameValuePair... params) {...}
36
37     private String buildQuery(NameValuePair[] params) throws UnsupportedEncodingException {...}
38
39     private String readStream(InputStream in) throws IOException {...}
40
41     private List<Venue> parseJson(String response) throws JSONException {...}
42 }
```

- Extend AsyncTask to make HTTP request in background
- Build the query string from array of NameValuePairs (12 lines)
- Read InputStream into String (20 lines)
- Parse JSON (75 lines)



The RxJava Android Module

The **rxjava-android** module contains Android-specific bindings for **RxJava**. It adds a number of classes to RxJava to assist in writing reactive components in Android applications.

- It provides a Scheduler that schedules an Observable on a given Android Handler thread, particularly the main UI thread.
- It provides operators that make it easier to deal with Fragment and Activity life-cycle callbacks.
- It provides wrappers for various Android messaging and notification components so that they can be lifted into an Rx call chain.
- It provides reusable, self-contained, reactive components for common Android use cases and UI concerns. (coming soon).

Gradle and Maven Retrolambda Plugin

```
apply plugin: 'com.android.application'
apply plugin: 'me.tatarka.retrolambda'

android {
    compileSdkVersion 22
    buildToolsVersion "23.0.0 rc2"

    defaultConfig {
        applicationId "com.alkche.android.rxsample"
        minSdkVersion 10
        targetSdkVersion 22
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            ...
        }
    }

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
}
```

```
<plugin>
  <groupId>net.orfjackal.retrolambda</groupId>
  <artifactId>retrolambda-maven-plugin</artifactId>
  <version>2.0.3</version>
  <executions>
    <execution>
      <goals>
        <goal>process-main</goal>
        <goal>process-test</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Project dependencies

```
dependencies {  
    compile fileTree(dir: 'libs', include: ['*.jar'])  
    compile 'com.android.support:appcompat-v7:22.2.0'  
    compile 'com.jakewharton:butterknife:7.0.1'  
    compile 'com.squareup.retrofit:retrofit:1.9.0'  
    compile 'io.reactivex:rxandroid:0.25.0'  
}  
  
buildscript {  
    repositories {  
        mavenCentral()  
    }  
  
    dependencies {  
        classpath 'me.tatarka:gradle-retrolambda:3.2.0'  
    }  
}
```

Iterable and Observable

Pattern	Single return value	Multiple return values
Synchronous	<code>T getData()</code>	<code>Iterable<T> getData()</code>
Asynchronous	<code>Future<T> getData()</code>	<code>Observable<T> getData()</code>

- Observables and Iterables share a similar API.
- Observable is the push equivalent of Iterable, which is pull.

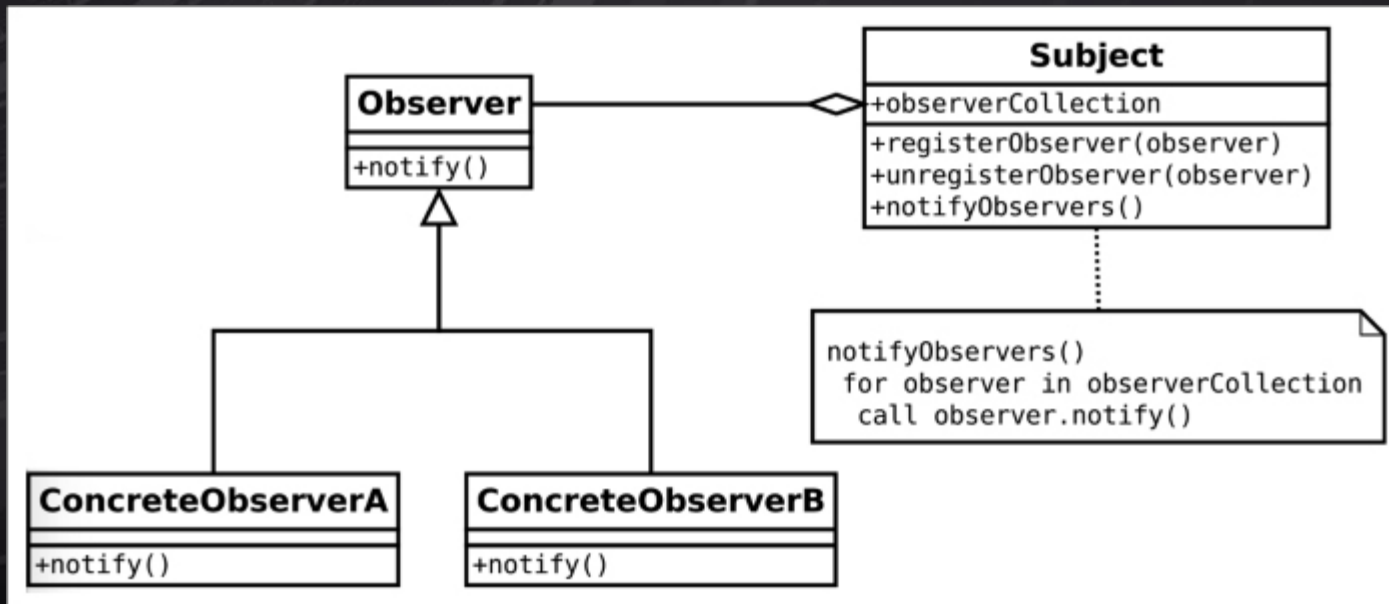
With Iterable, the consumer synchronously pulls values from the producer and the thread is blocked until these values arrive. By contrast, with Observable, the producer asynchronously pushes values to the Observer whenever values are available.

Event	Iterable (pull)	Observable (push)
retrieve data	<code>T next()</code>	<code>onNext(T)</code>
discover error	throw exception	<code>onError(Throwable)</code>
complete	returns	<code>onCompleted()</code>

Observer pattern

The Observer pattern is the perfect fit for any of these scenarios:

- When your architecture has two entities, one depending on the other, and you want to keep them separated to change them or reuse them independently.
- When a changing object has to notify an unknown amount of related objects about its own change.
- When a changing object has to notify other objects without making assumptions about who these objects are.



Observer and Subscription

Observer

`onNext(T)`

`onCompleted(T)`

`onError(Throwable t)`

`<interface>`
`Observer`

Subscription

`unsubscribe()`

`<interface>`
`Subscription`

Types of Observables

Non-Blocking Observables

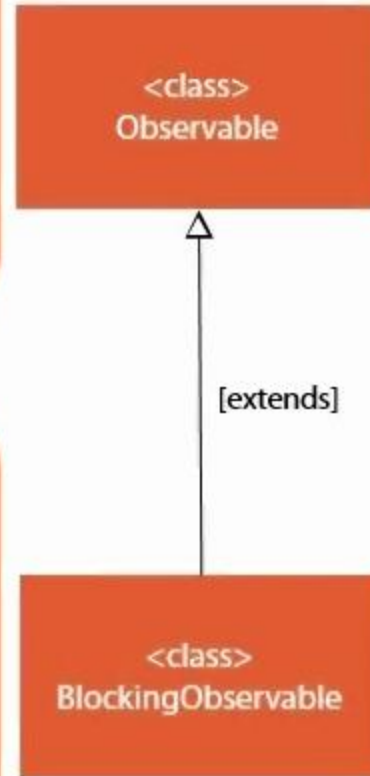
Asynchronous execution supported

Unsubscribe at any point in the event stream

Blocking Observables

BlockingObservable subclass

Events are synchronous



Subject

A **subject** is a object that can be an Observable and an Observer at the same time. A subject can subscribe to an Observable, acting like an Observer, and it can emit new items or even pass through the item it received, acting like an Observable.

RxJava provides four different types of subjects:

- **PublishSubject** - emits all subsequently observed items to the subscriber, once an Observer has subscribed.
- **BehaviorSubject** - emits the most recent item it has observed and all subsequent observed items to each subscribed Observer.
- **ReplaySubject** - buffers all items it observes and replays them to any Observer that subscribes.
- **AsyncSubject** - publishes only the last item observed to each Observer that has subscribed, when the source Observable completes.

Schedulers

<class>
Schedulers

computation()

currentThread()

immediate()

io()

newThread()

executor(Executor)

executor(ScheduledExecutor)

<class>
Observable

subscribeOn(Scheduler)

observeOn(Scheduler)

Observable creation

<Static Factory Method>
Observable.from(...)

<T> Observable<T> Observable.from(T object [, Scheduler s])

<T> Observable<T> Observable.from(Iterable<T> list [, Scheduler s])

<T> Observable<T> Observable.from(T[] array [, Scheduler s])

<T> Observable<T> Observable.from(Future<T> future [, Scheduler s])

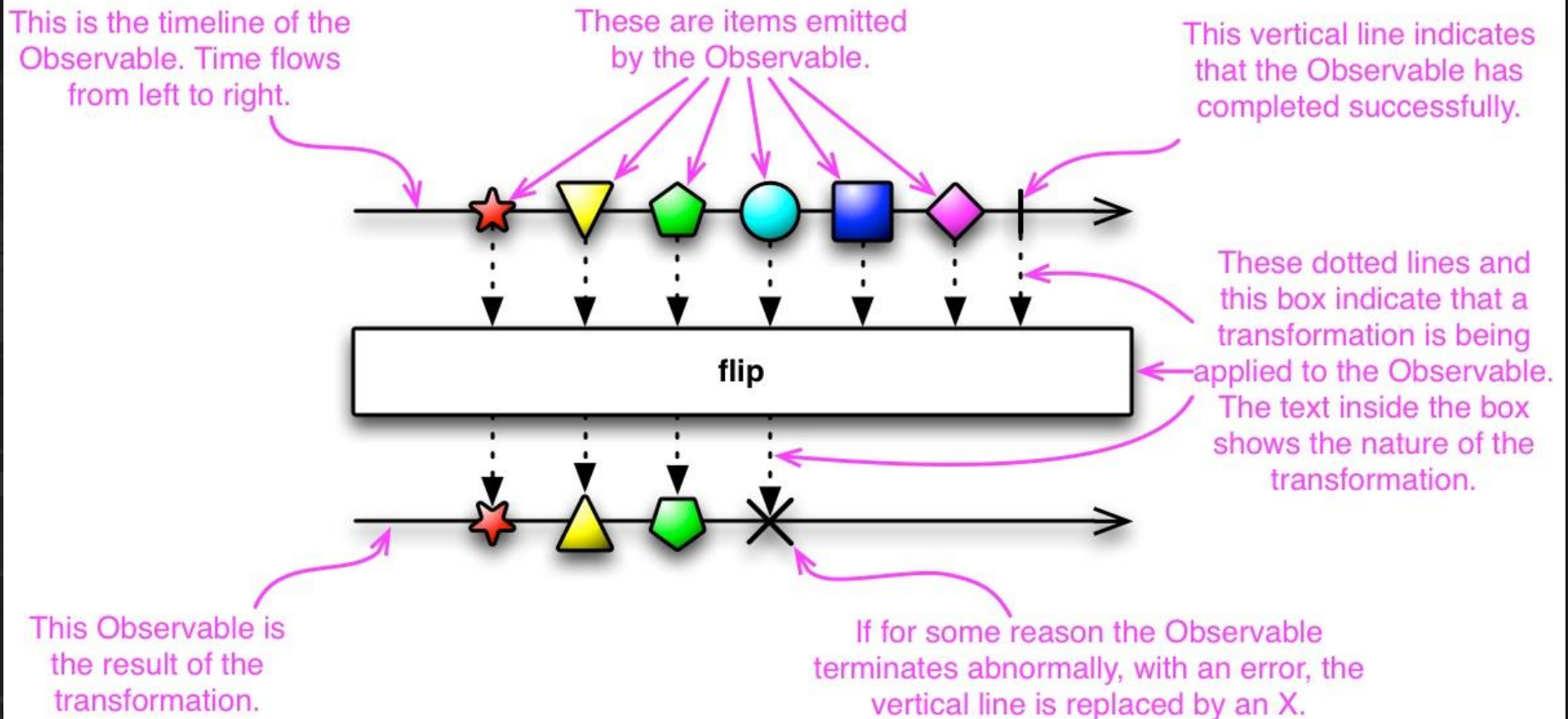
<Static Factory Method>
BlockingObservable.from(...)

<T> Observable<T> BlockingObservable.from(Observable o)

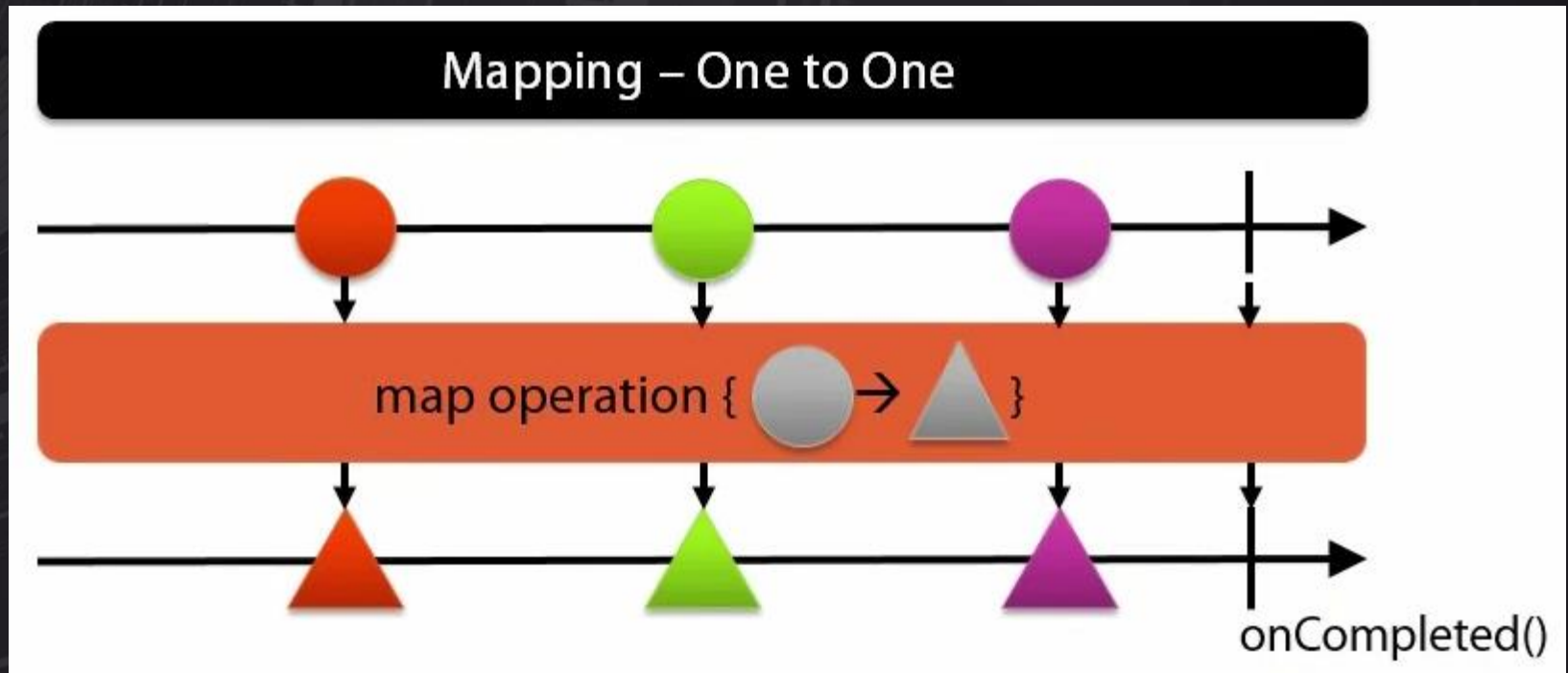
Observable creation: examples

- `Observable.just("one", "two", "three");`
- `Observable.just("one", "two", "three").repeat(3);`
- `Observable.range(10, 3);` - Takes two numbers as parameters: the first one is the starting point, and the second one is the amount of numbers we want to emit.
- `Observable.interval(3, TimeUnit.SECONDS);` - Takes two parameters: a number that specifies the amount of time between two emissions, and the unit of time to be used.
- `Observable.timer(3, 3, TimeUnit.SECONDS);` - Starts with an initial delay (3 seconds in the example) and then keeps on emitting a new number every N seconds (3 in the example).

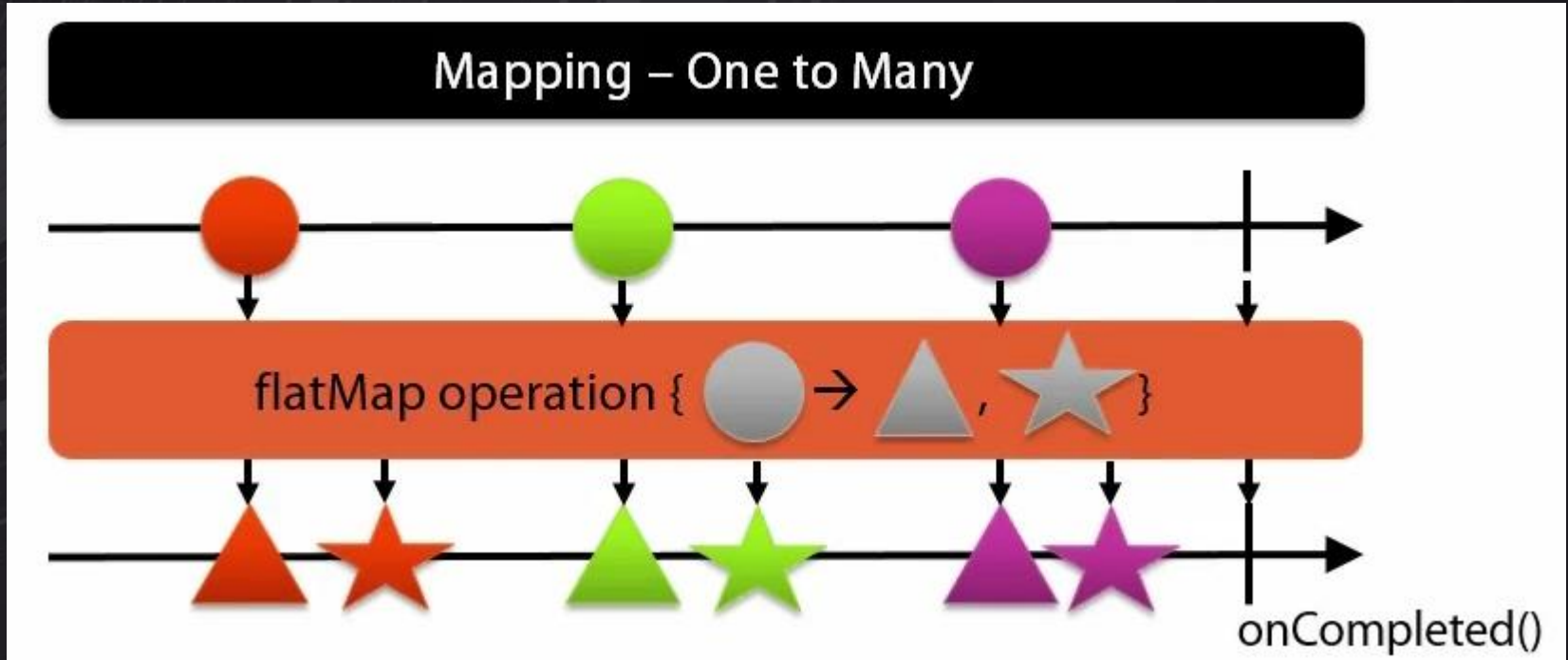
Marble diagram



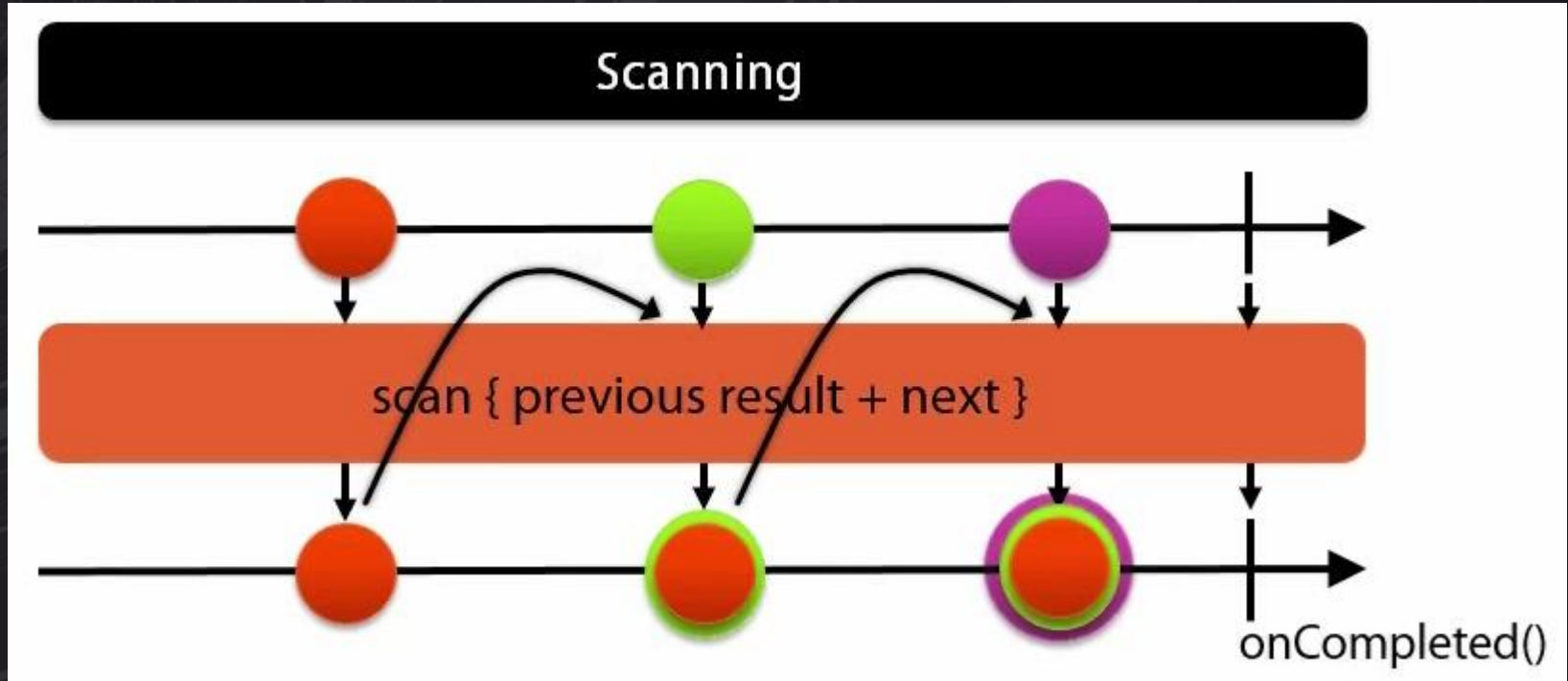
Transformations: one to one



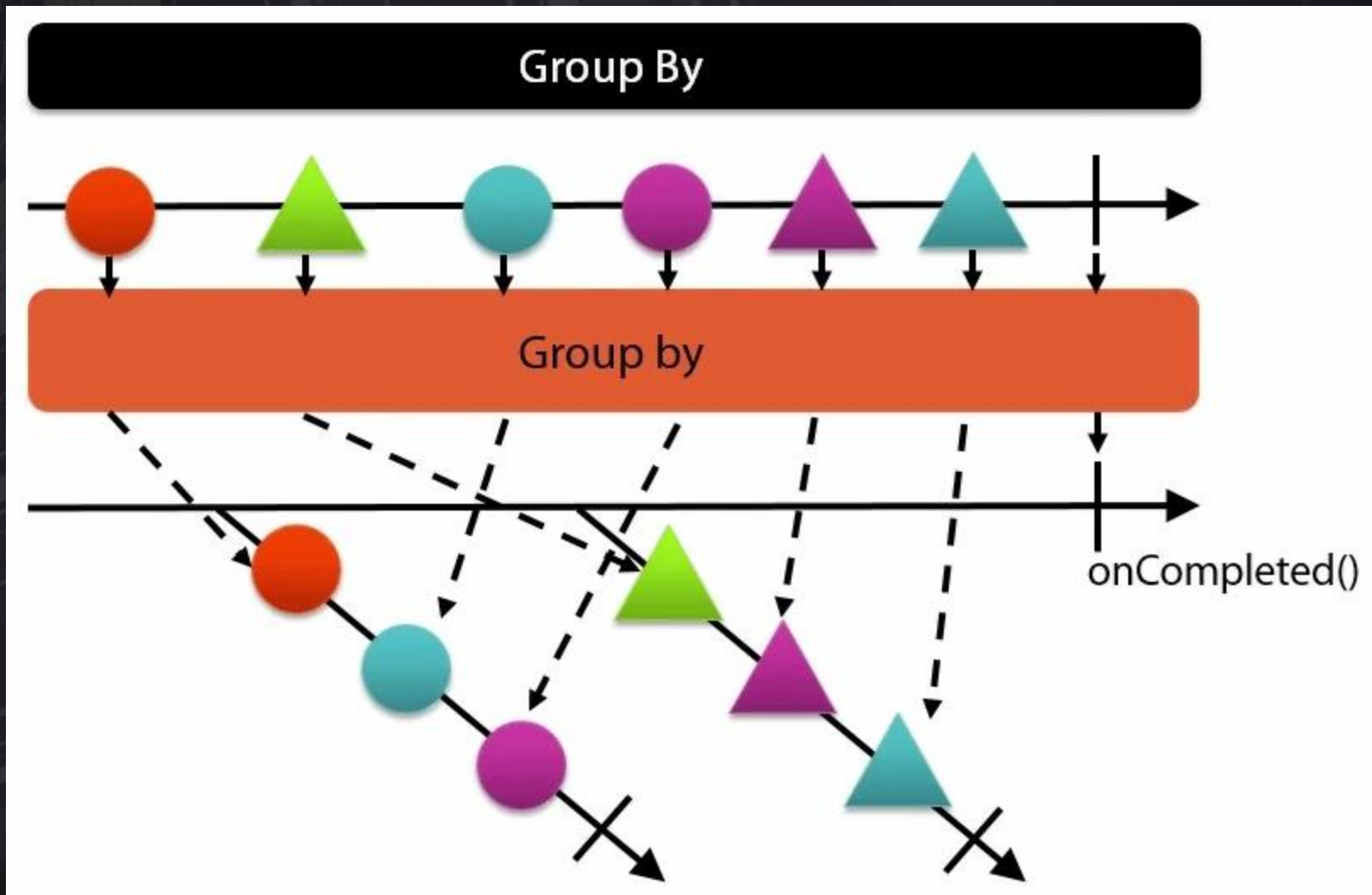
Transformations: one to many



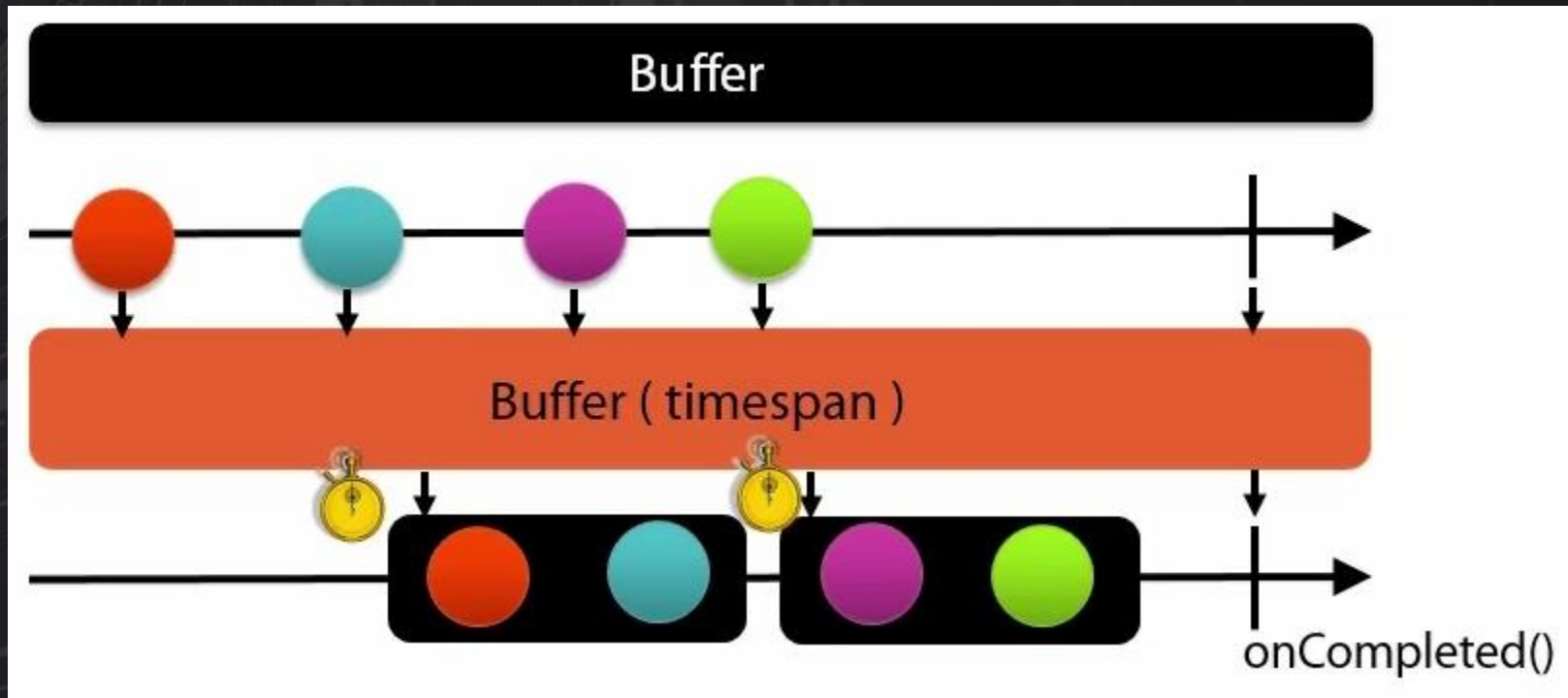
Transformations: scanning



Transformations: group by

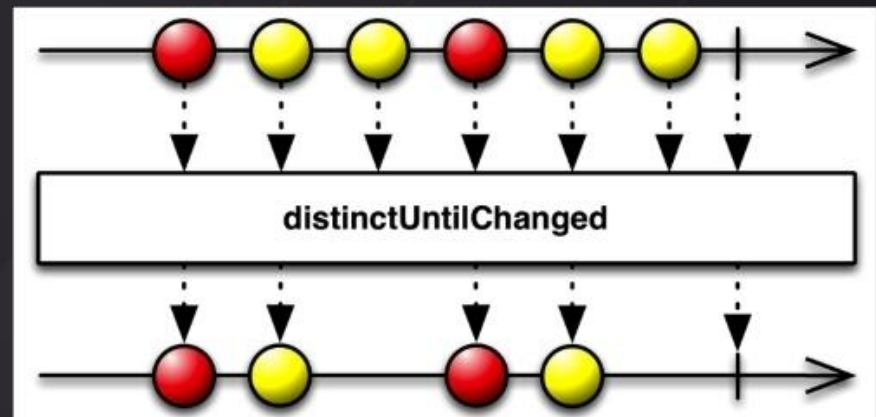
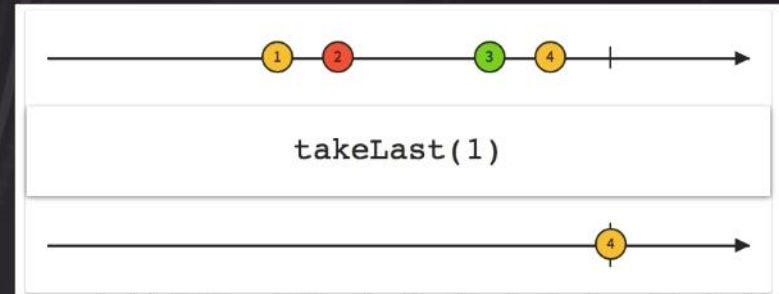
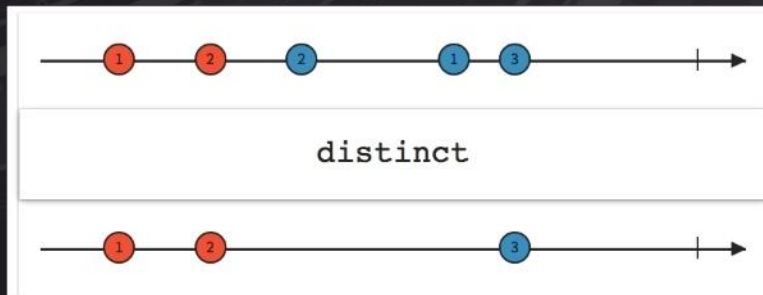
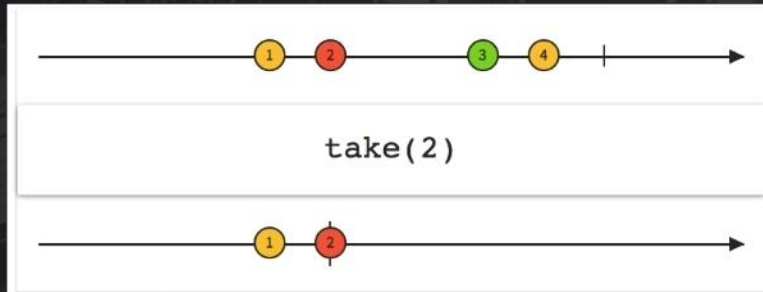


Transformations: buffer

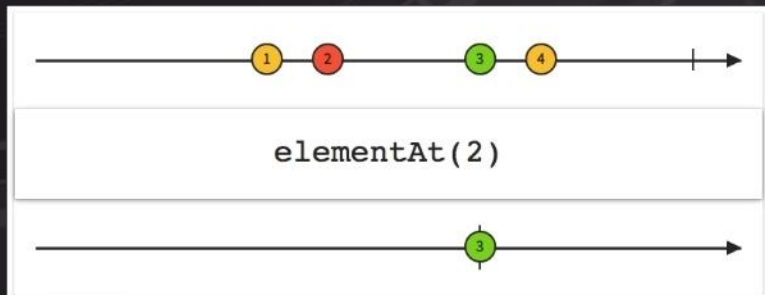
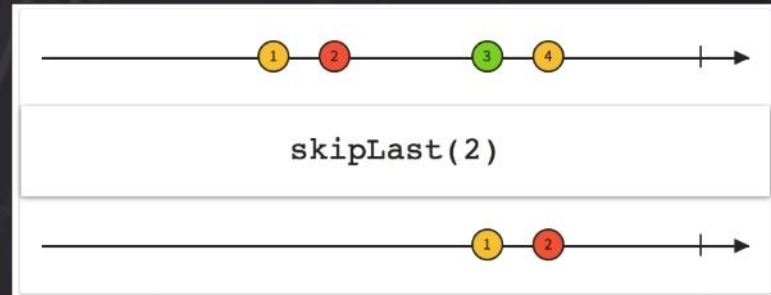
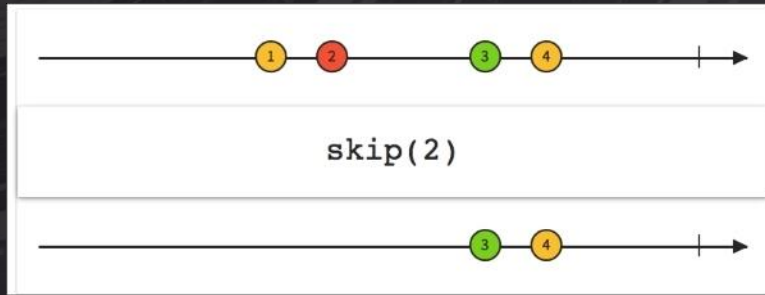
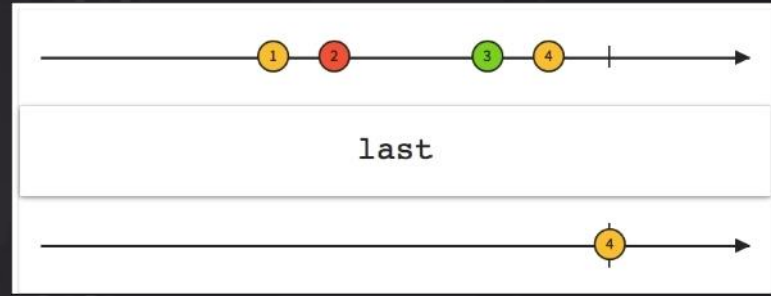
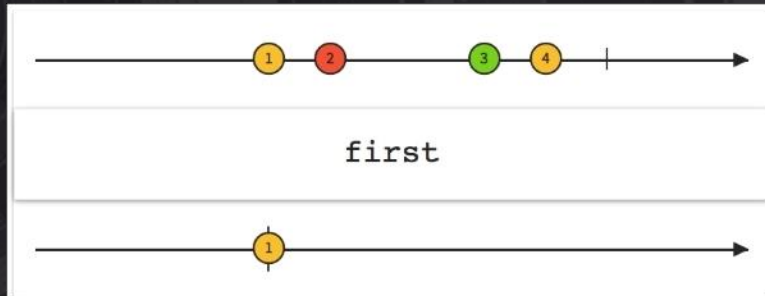


Filtering Observables

```
Observable.just("one", "two", "three").filter((s)->{  
    return !s.equals("two");  
});
```

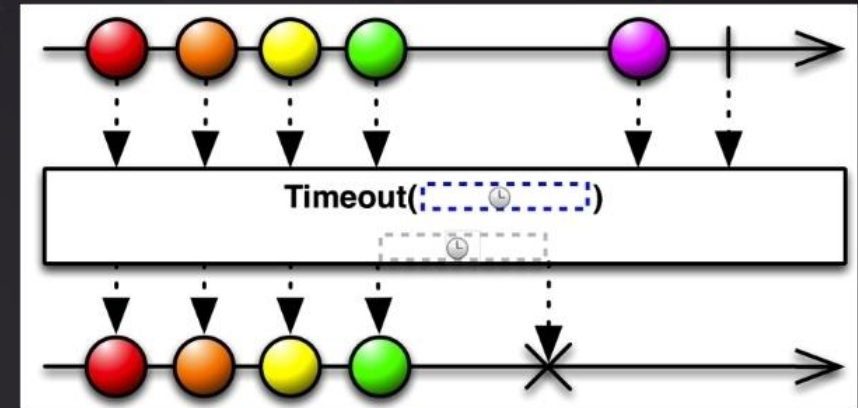
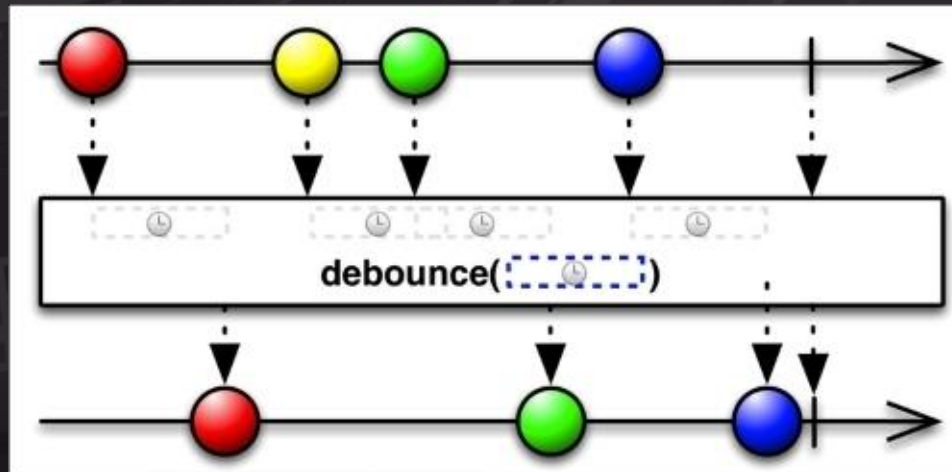
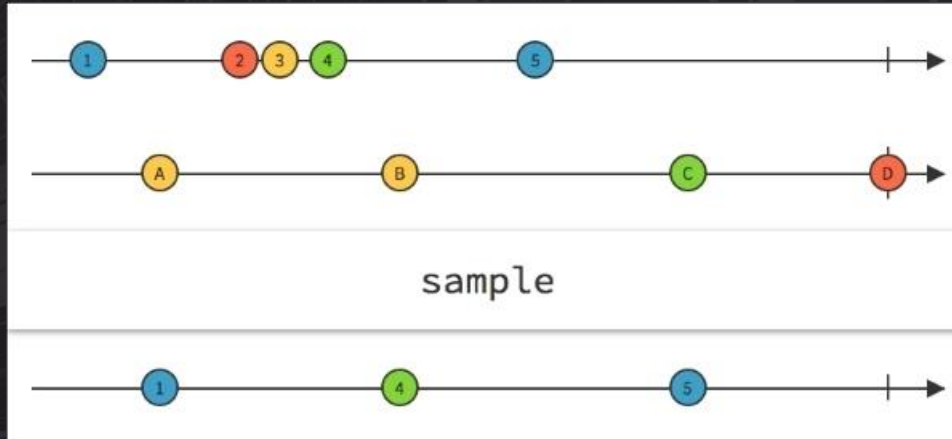


Filtering Observables



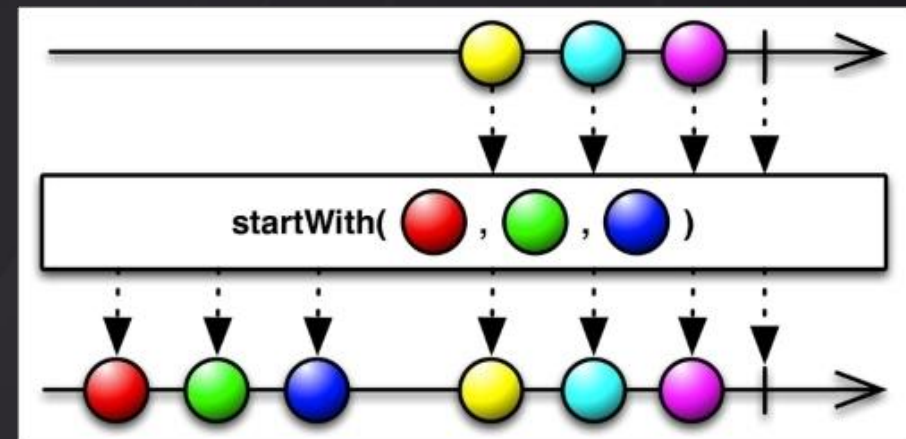
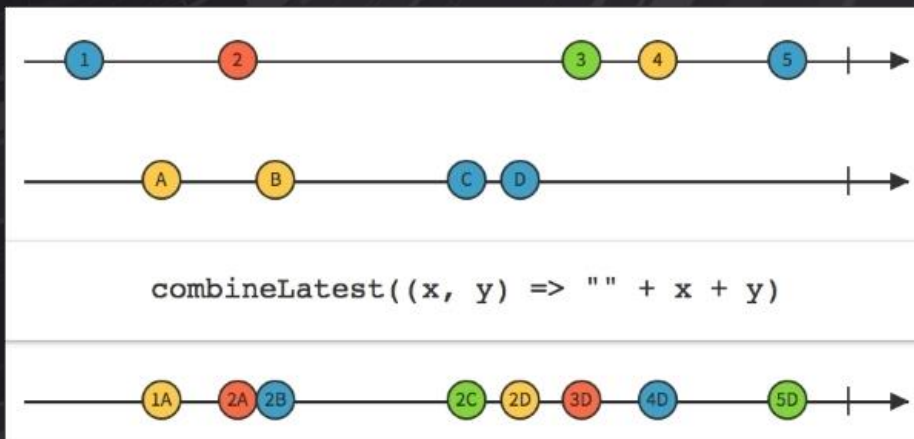
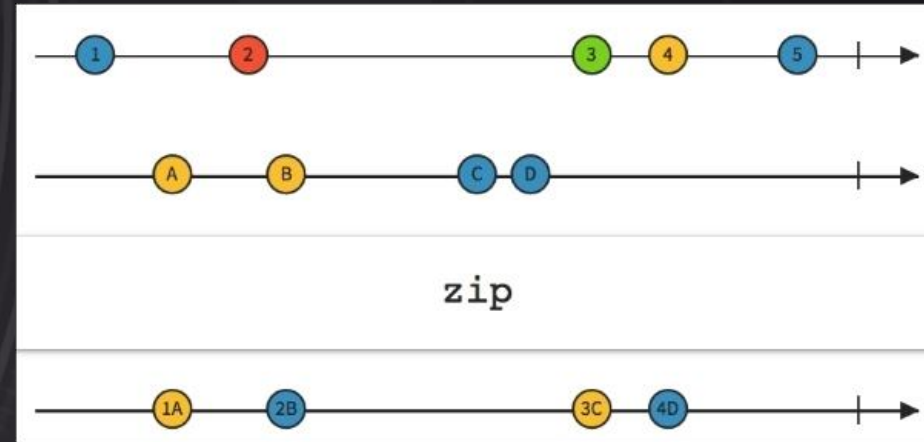
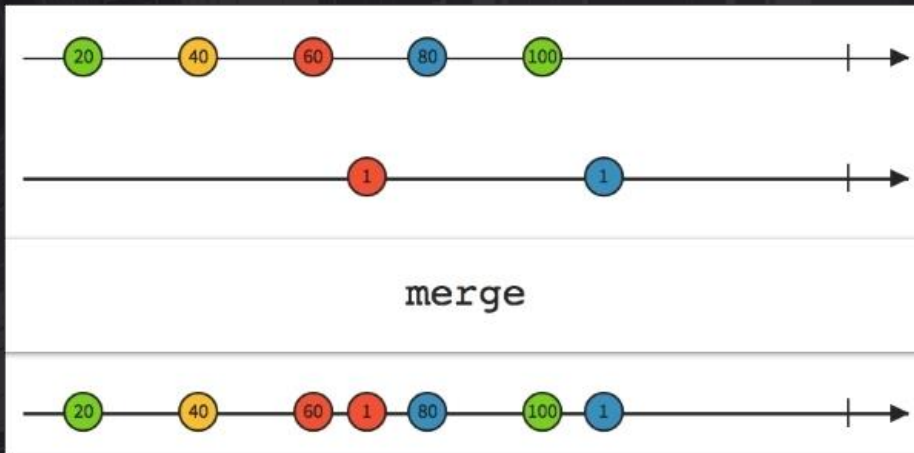
Filtering Observables

sample() creates a new Observable sequence that will emit the most recent item emitted by the Observable source in a decided time interval.



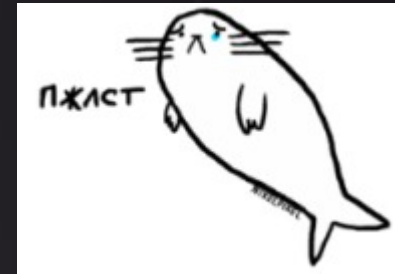
debounce() starts its internal timer, and if no new item is emitted during this timespan, the last item is emitted by the new Observable.

Combining Observables



Death to AsyncTask

```
80 new AsyncTask<Void, Void, String>() {
81     @Override
82     protected String doInBackground(Void... params) {
83         return service.login1("login", "password");
84     }
85
86     @Override
87     protected void onPostExecute(String token) {
88         title.setText(token);
89     }
90 }.execute();
```



```
92 Observable<String> tokenObservable = service.login3("login", "password");
93 tokenObservable
94     .subscribeOn(Schedulers.newThread())
95     .observeOn(AndroidSchedulers.mainThread())
96     .subscribe(title::setText, (error) -> {
97         title.setText(error.getMessage());
98     });
```


Example

Sign up

Username

New password

Confirm password

CANCEL

CONFIRM

Sign up, Жорик :

Жорик

New password

Confirm password

CANCEL

CONFIRM

```
nameEditText.addTextChangedListener(new Watcher() {  
  
    @Override public void afterTextChanged(Editable editable) {  
        lastIsNameEmpty = editable.toString().length() == 0;  
        requestDuplicates(editable.toString(), new MyCallback() {  
  
            @Override public void onResponse(Response response){  
                checkResponseForDuplicates(response, new MyCallback() {  
  
                    @Override public void onResponse(Response response) {  
                        lastIsHasDuplicates = calcDuplicates(response);  
                        setHasDuplicates(lastIsHasDuplicates);  
                        checkEqualityToEnableButton();  
                    }  
                });  
            }  
        });  
    }  
});
```

Example

```
Observable<String> nameObs = EditTextObservable
    .from(nameEditText);

nameObs.subscribe(name -> requestDuplicates(name));

Observable<String> rightTitleObservable = nameObs
    .map(name -> {
        if (TextUtils.isEmpty(name)) return name;
        else return ", " + name;
    });

rightTitleObservable.subscribe(name ->
    signUpTitle.setText("Sign up" + name));
```

```
Observable<String> nameObs = EditTextObservable
    .from(nameEditText);

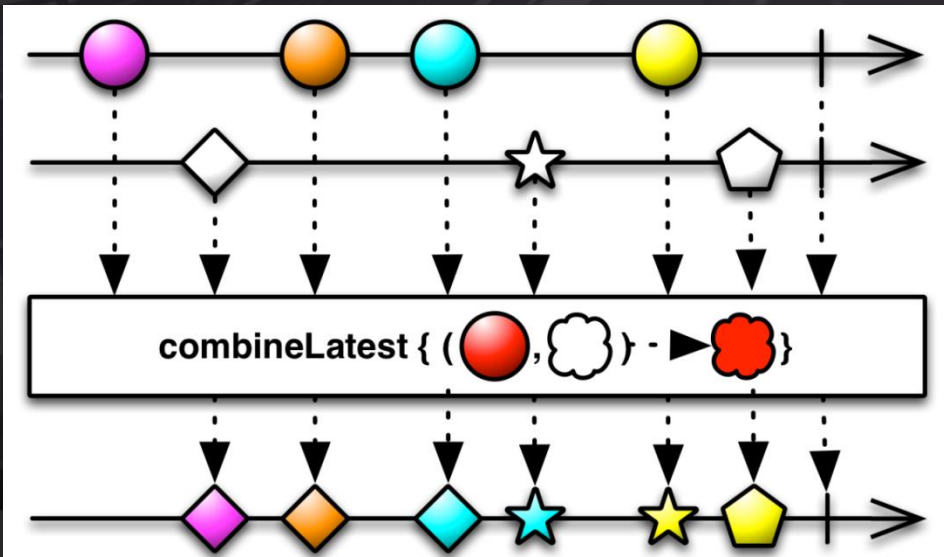
Observable<Boolean> duplicatesCheckObservable = nameObs
    .debounce(500, TimeUnit.MILLISECONDS)
    .observeOn(Schedulers.newThread())
    .map(name -> requestDuplicates(name))
    .map(response -> checkResponseForDuplicates(response))
    .map(checkedException -> calcDuplicates(checkedException))
    .subscribeOn(mainThread)
    .observeOn(mainThread)
    .onErrorReturn(throwable -> false);

duplicatesCheckObservable.subscribe(isUnique -> { /*process*/ });
```

Example

```
Observable<Boolean> shouldEnableButtonObs = Observable
    .combineLatest(
        passObs.map(pass -> pass.trim()),
        confirmObs.map(confirm -> confirm.trim()),
        duplicatesCheckObservable,
        (pass, confirmPass, isNameUnique) ->
            TextUtils.equals(pass, confirmPass)
                && isNameUnique
    );

shouldEnableButtonObs
    .subscribe(enabled -> button.setEnabled(enabled));
```



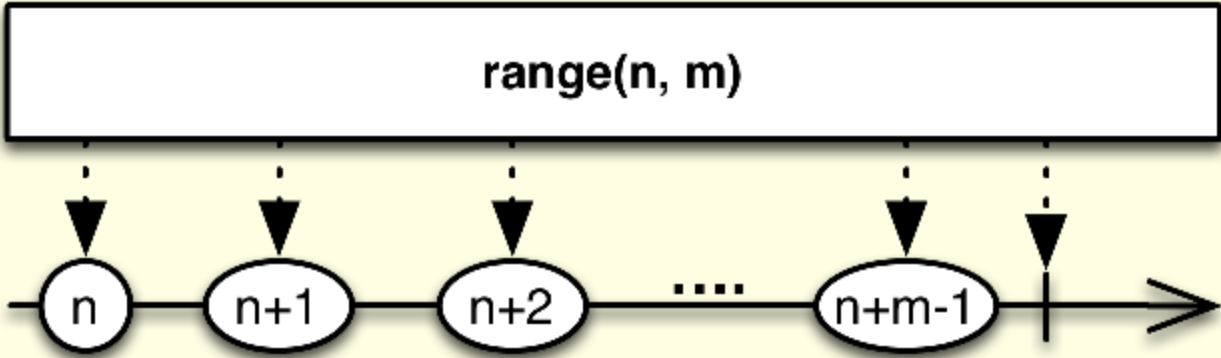
Well documented

Documentation for `range(int, int)`

[rx.Observable](#)

```
public static final Observable<java.lang.Integer> range(int start,
                                                         int count)
```

Returns an Observable that emits a sequence of Integers within a specified range.



The diagram illustrates the sequence of integers emitted by the `range(n, m)` method. A rectangular box labeled `range(n, m)` has five dashed arrows pointing down to a sequence of nodes. The nodes are circles containing the values `n`, `n+1`, `n+2`, an ellipsis (`...`), and `n+m-1`. These nodes are connected by solid horizontal lines. A vertical line is placed after the final node, and a solid arrow points to the right from this line, indicating the sequence continues.

Scheduler:
range does not operate by default on a particular [Scheduler](#).

```
//initRe... range (int start, int count)  Observable<Integer>
... range (int start, int co... Observable<Integer>
title.se... Pressing Ctrl+Space twice without a class qualifier would show all accessible static methods F00) );
```

Thank you for your attention

Questions?



S P E R  S O F T