# RxJava

# Функциональное реактивное программирование (FRP)

1) Функциональное. Основной элемент – функции.
2) Реактивное. Программирование с асинхронными потоками данных.
3) Функциональное реактивное программирование – программирование с асинхронными потоками данных, которые манипулируются с помощью различных функций.

# Что такое поток данных?

a) Любые объекты и примитивы
b) Любая последовательность объектов и примитивов
c) Бесконечные последовательности
d) Любое событие (ввод текста, клик на кнопку и так далее)

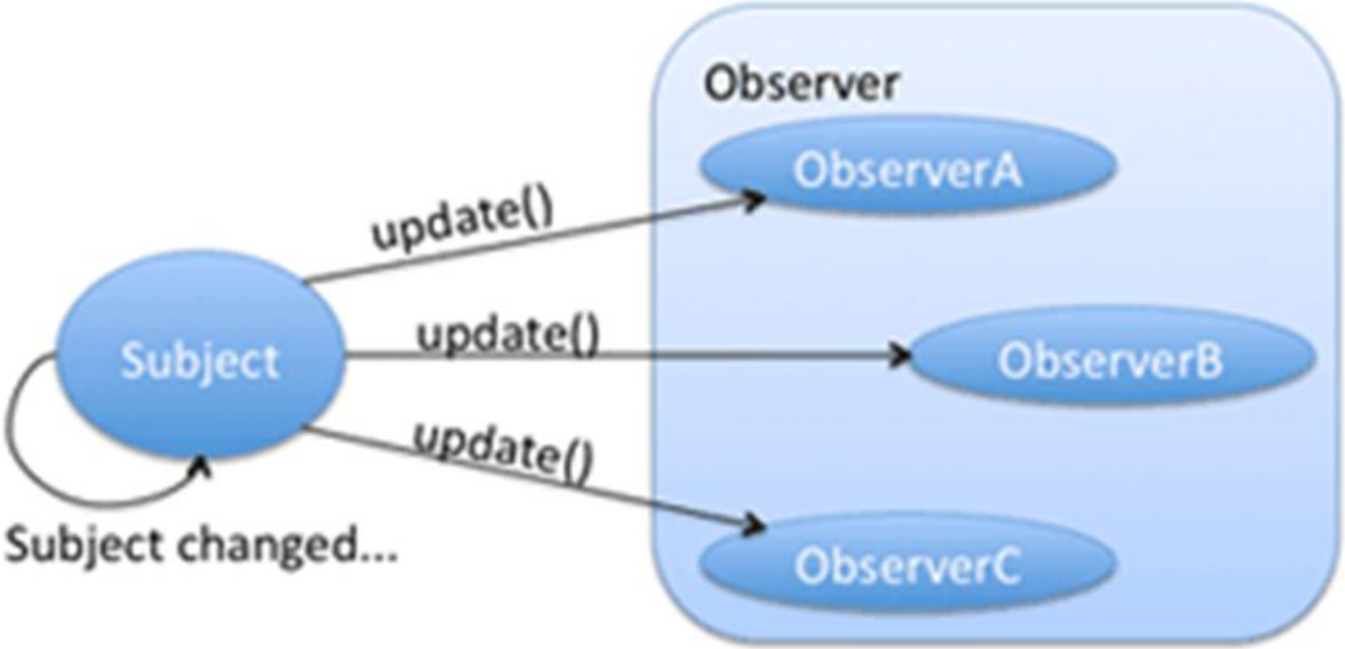
# Ввод текста как поток данных

```
editText.observeChanges()
        .debounce(500, TimeUnit.MILLISECONDS)
        .map(String::toLowerCase)
        .flatMap(this::findPerson)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(person -> {}, throwable -> {});
```

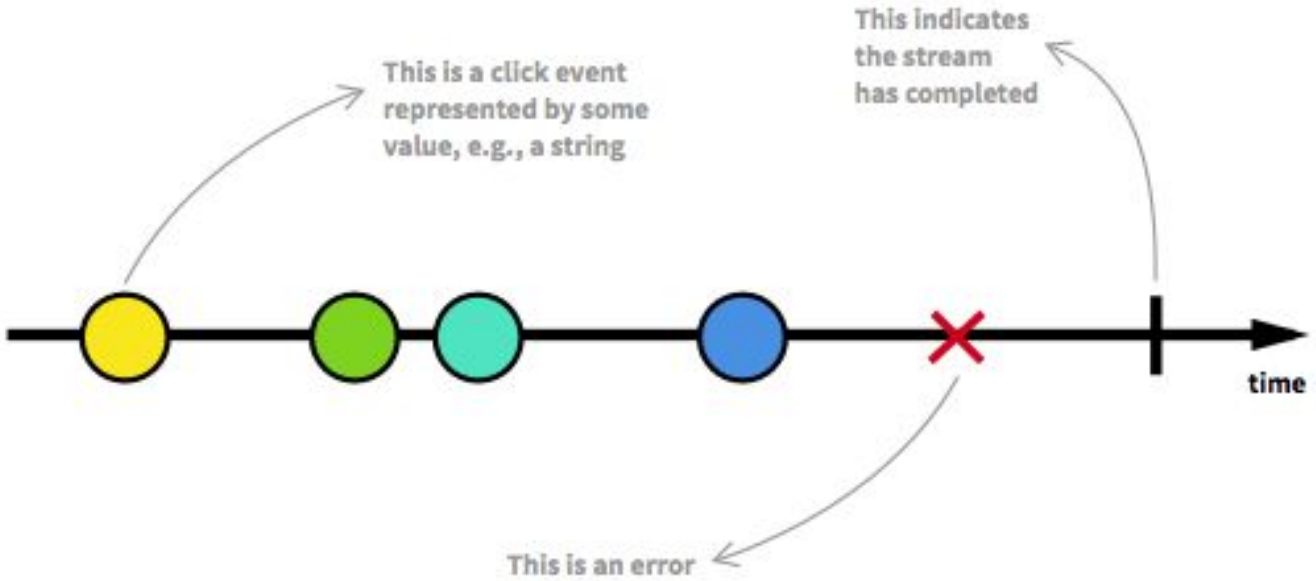
# RxJava

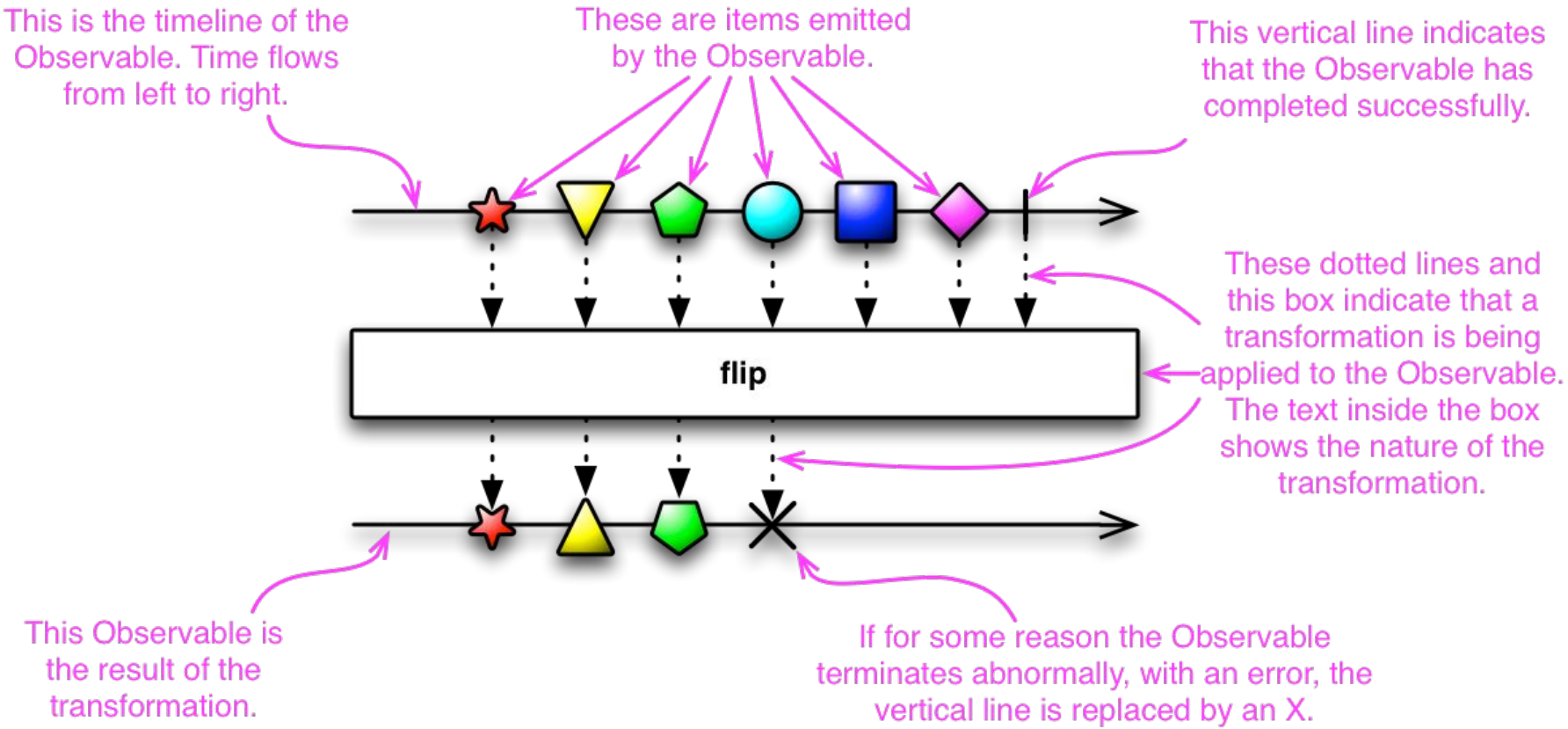
1) Обеспечение многопоточности
2) Управление потоками данных
3) Обработка ошибок
4) Красивый и компактный код (при использовании лямбда-выражений)

# Паттерн Observer



# Поток данных




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.
flip
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.

# Создаем поток данных

```
Observable<Integer> observable = Observable.just(1, 2, 4);
```

# Подписываемся на поток данных

```
Observable<Integer> observable = Observable.just(1, 2, 4);
observable.subscribe(new Observer<Integer>() {
    @Override
    public void onCompleted() {
        // do nothing
    }

    @Override
    public void onError(Throwable e) {
        // do nothing
    }

    @Override
    public void onNext(Integer integer) {
        Log.i(TAG, String.valueOf(integer));
    }
});
```

# Подписываемся на поток данных

```
Observable<Integer> observable = Observable.just(1, 2, 4);
observable.subscribe(
        integer -> Log.i(TAG, String.valueOf(integer)),
        throwable -> {/*handle error*/});
```

# Почему Observable, а не for?

```
Observable<Integer> observable = Observable.just(1, 2, 4);
observable.subscribe(integer -> Log.i(TAG, String.valueOf(integer)));

int[] items = new int[]{1, 2, 4};
for (int value : items) {
    Log.i(TAG, String.valueOf(value));
}
```

# Rx vs for

```
//RxJava
Observable.just(1, 2, 4, 8, 16, 32, 64)
        .filter(integer -> integer >= 13)
        .map(String::valueOf)
        .subscribe(value -> Log.i(TAG, value));

//For
int[] items = new int[]{1, 2, 4, 8, 16, 32, 64};
for (int value : items) {
    if (value >= 13) {
        String s = String.valueOf(value);
        Log.i(TAG, s);
    }
}
```

# Rx vs for

```
//RxJava
Observable.just(1, 2, 4, 8, 16, 32, 64)
        .filter(integer -> integer >= 13)
        .map(String::valueOf)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(value -> Log.i(TAG, value));

//For
EXECUTOR.execute(() -> {
    int[] items = new int[]{1, 2, 4, 8, 16, 32, 64};
    for (int value : items) {
        if (value >= 13) {
            String s = String.valueOf(value);
            HANDLER.post(() -> Log.i(TAG, s));
        }
    }
});
```

# Observable.from

```
@NonNull
public static Observable<Integer> from() {
    List<Integer> values = new ArrayList<>();
    values.add(5);
    values.add(10);
    values.add(15);
    values.add(20);
    return Observable.from(values);
}
```

# Observable.create

```
@NonNull
public static Observable<Integer> observableWithCreate() {
    return Observable.create(subscriber -> {
        subscriber.onNext(5);
        subscriber.onNext(10);
        try {
            //stub long-running operation
            Thread.sleep(300);
        } catch (InterruptedException e) {
            subscriber.onError(e);
            return;
        }
        subscriber.onNext(15);
        subscriber.onCompleted();
    });
}
```

# Observable.create

```
Subscription subscription = RxJavaCreate.observableWithCreate()
        .subscribeOn(Schedulers.newThread())
        .subscribe(System.out::println);
subscription.unsubscribe();
```

# Observable.create

```
@NonNull
public static Observable<Integer> observableWithCreate() {
    return Observable.create(subscriber -> {
        subscriber.onNext(5);
        subscriber.onNext(10);
        try {
            //stub long-running operation
            Thread.sleep(300);
        } catch (InterruptedException e) {
            subscriber.onError(e);
            return;
        }
        if (!subscriber.isUnsubscribed()) {
            subscriber.onNext(15);
        }
        subscriber.onCompleted();
    });
}
```

# Observable.fromCallable

```
Observable.fromCallable(new Callable<List<Integer>>() {
    @Override
    public List<Integer> call() throws Exception {
        //some long-running operation
        return getUserIdsFromDatabase();
    }
});
```

Observable.*fromCallable*(() -> getUserIdsFromDatabase());

# Observable. Создание последовательности

- Observable.empty вернет только onCompleted
- Observable.error вернет только onError
- Observable.never ничего не вернет

```
values.subscribe(
    val -> System.out.println("Received: " + val),
    error -> System.out.println("Error: " + error),
    () -> System.out.println("Completed")
);
```

# subscribeOn и observeOn

1) Код подписчика (observer) выполняется в потоке, переданном в observeOn
2) Код потока данных выполняется в потоке, переданном в subscribeOn

```
return observable
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread());
```

# Schedulers

1) Schedulers.io() – выполнение задач, которые не сильно нагружают процессор, но являются долгими: сетевые запросы, база данных
2) Schedulers.computation() – тяжелые вычислительные задачи, нагружающие CPU
3) Schedulers.newThread – новый поток для каждой новой задачи
4) Schedulers.immediate() – выполнение задачи в том же потоке
5) AndroidScheduler.mainThread() (RxAndroid) – главный поток Android-приложения

# Transformer

Одинаковый код для каждого сетевого запроса

```
return Observable.just(1)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread());
```

# Transformer

```
public class AsyncTransformer<T> implements Observable.Transformer<T, T> {

    @Override
    public Observable<T> call(Observable<T> observable) {
        return observable
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread());
    }
}
```

```
Observable.just(1)
 .compose(new AsyncTransformer<>());
```

# Transformer

```
@NonNull
public static <T> ObservableTransformer<T, T> async() {
    return observable -> observable
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread());
}

@NonNull
public static <T> ObservableTransformer<T, T> async(@NonNull final Scheduler background,
                                                    @NonNull final Scheduler main) {
    return observable -> observable
            .subscribeOn(background)
            .observeOn(main);
}
```

# Transformer

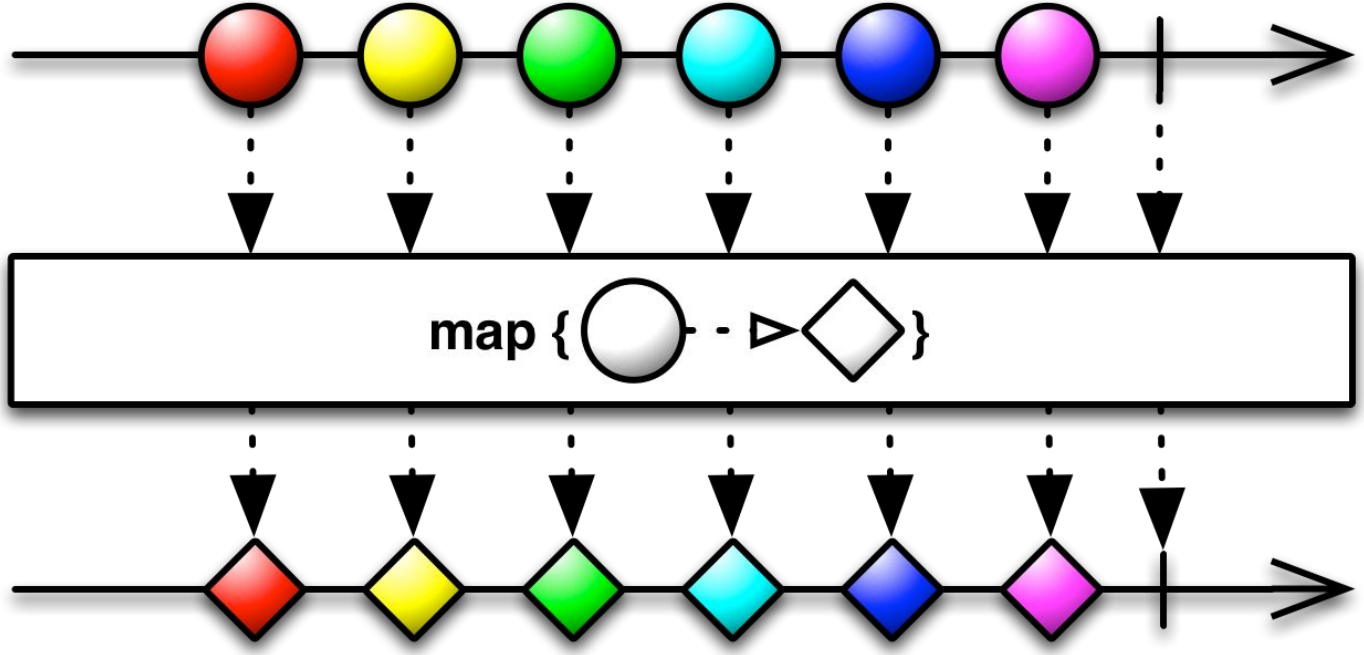
```
public static <T> ObservableTransformer<T, T> applySchedulersToObservable() {
    return upstream -> upstream
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread());
}

public static <T> SingleTransformer<T, T> applySchedulersToSingle() {
    return upstream -> upstream
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread());
}

public static CompletableTransformer applySchedulersToCompletable() {
    return upstream -> upstream
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread());
}

public static <T> MaybeTransformer<T, T> applySchedulersToMaybe() {
    return upstream -> upstream
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread());
}
```

# Observable.map

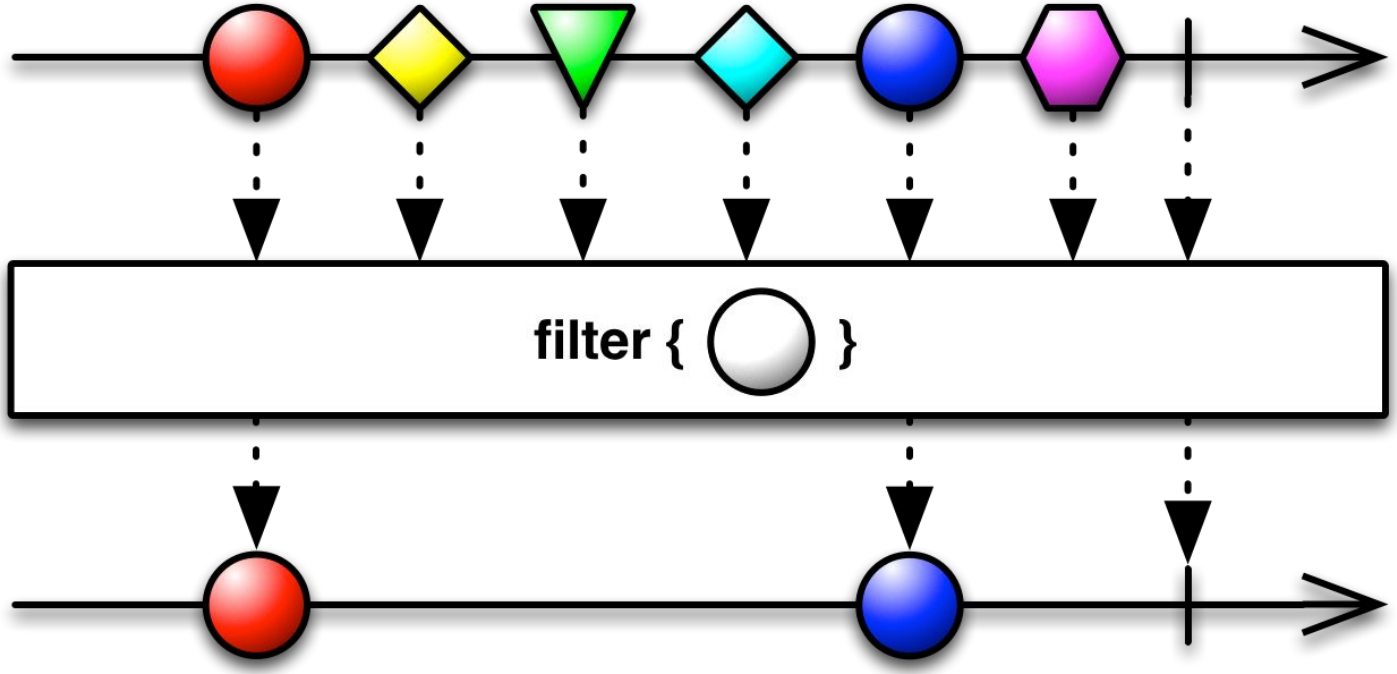


# Observable.map

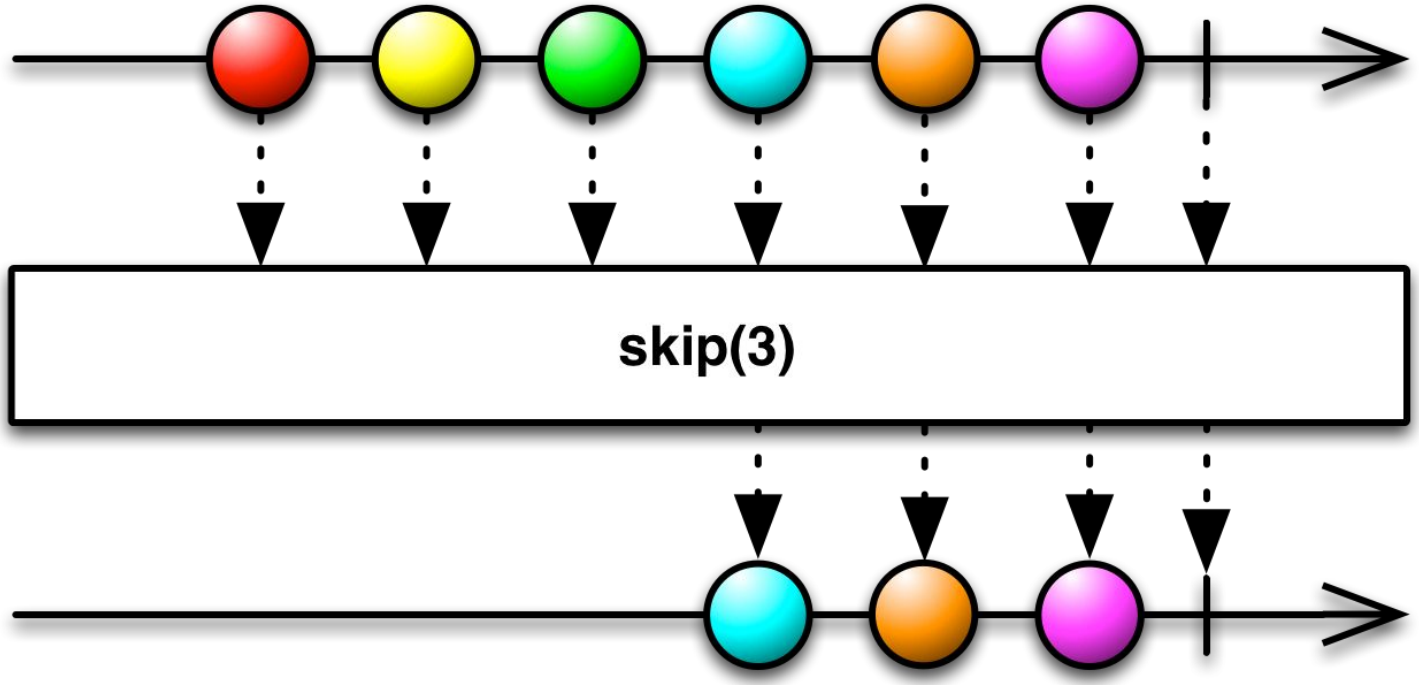
```
observable
 .map(integer -> {
     int value = integer * 2;
     String text = String.valueOf(value);
     return text.hashCode();
 });
```

```
observable
 .map(integer -> integer * 2)
 .map(String::valueOf)
 .map(String::hashCode);
```

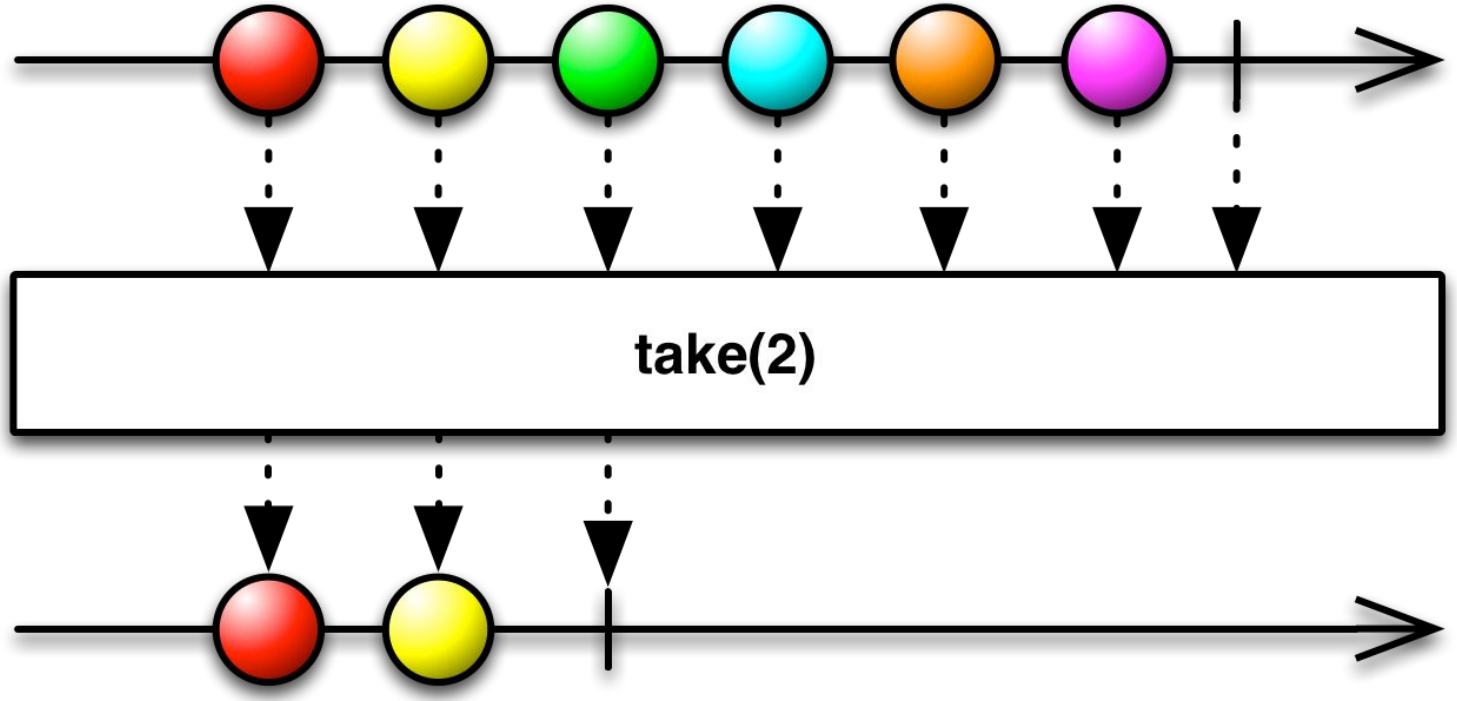
# Observable.filter



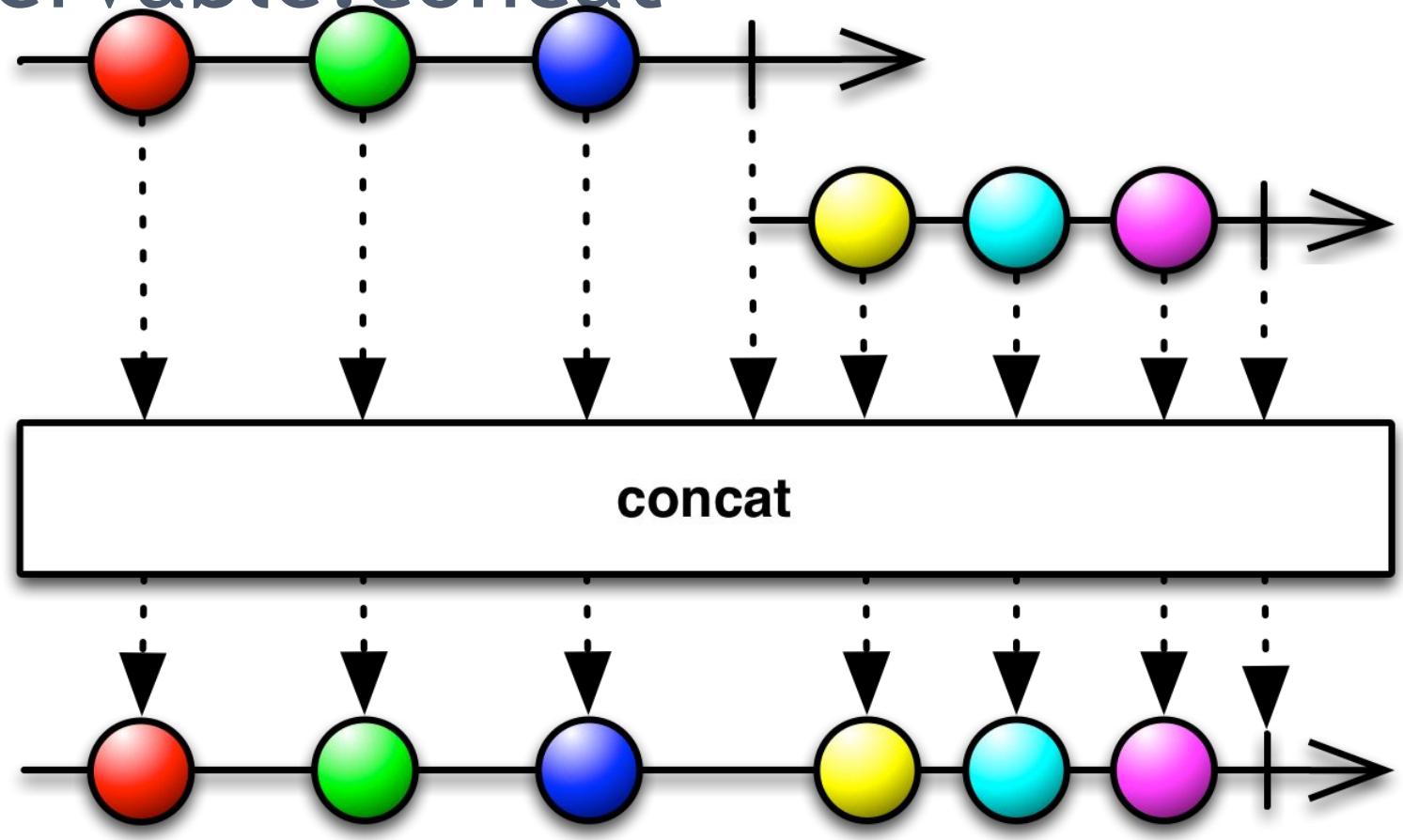
# Observable.skip



# Observable.take



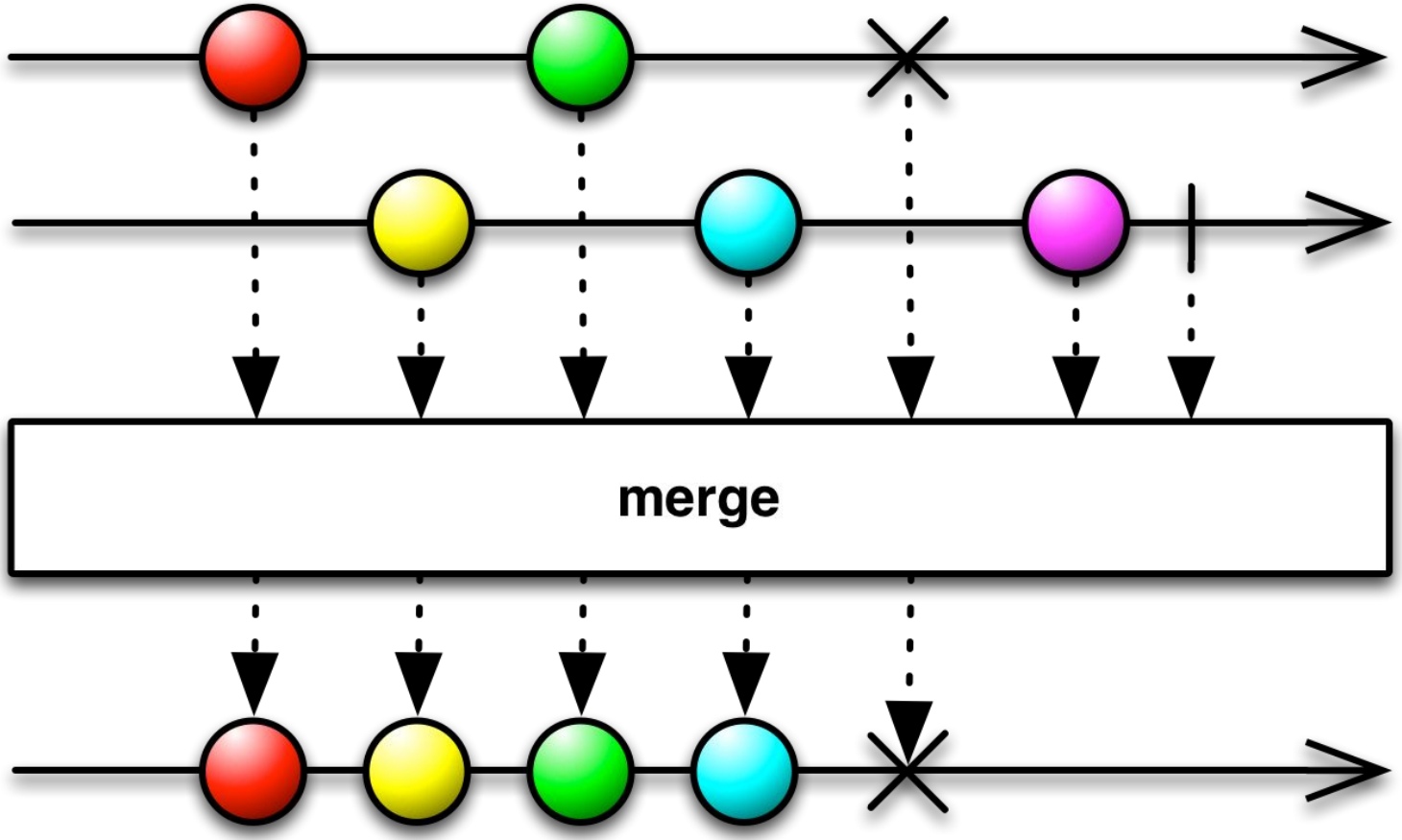
# Observable.concat



# Observable.concat

```
Observable<Integer> first = Observable.just(1, 4, 8);
Observable<Integer> second = Observable.just(2, 6, 9);
Observable<Integer> third = Observable.just("Red", "Hello").map(String::length);
return Observable.concat(first, second, third);
```

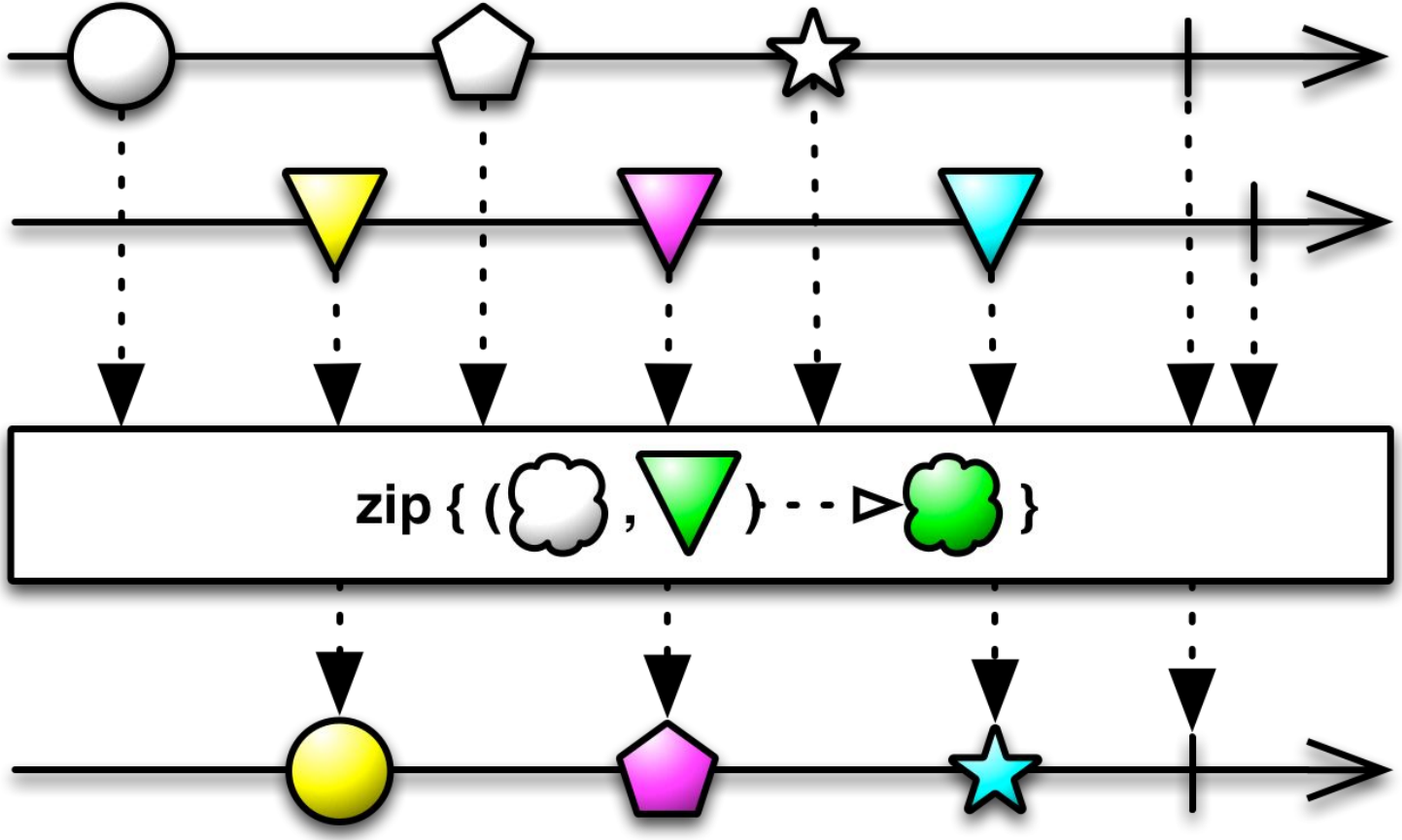
# Observable.merge



# Observable.merge

```
Observable<Integer> first = Observable.just(1, 4, 8);
Observable<Integer> second = Observable.just(2, 6, 9);
Observable<Integer> third = Observable.just("Red", "Hello").map(String::length);
return Observable.merge(first, second, third);
```

# Observable.zip



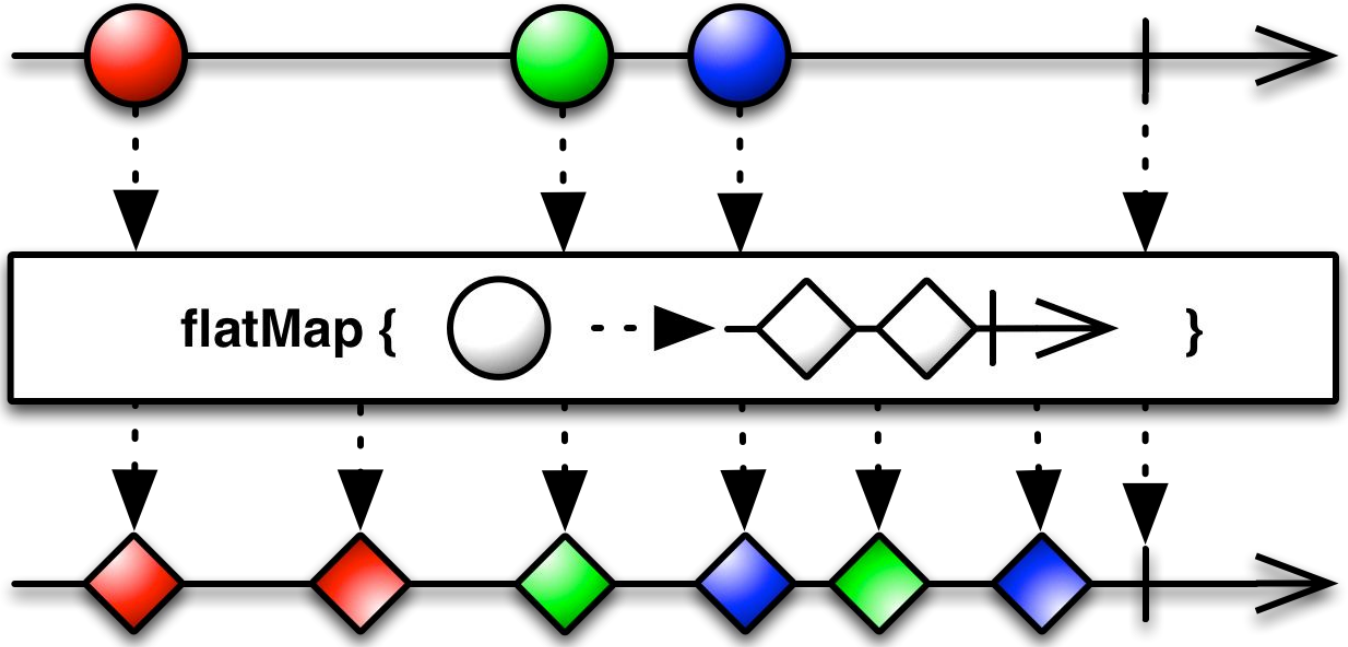
# Observable.zip

```
Observable<String> names = Observable.just("John", "Jack");
Observable<Integer> ages = Observable.just(28, 17);
return Observable.zip(names, ages, Person::new);
```

# Observable.zip

```
return Single.zip(
        api.getRates(locationId)
                .map(this::saveSelectedRates),
        api.getReoccurringRates(locationId)
                .map(ReoccurringRate.MultipleHolder::getRates)
                .map(this::getActiveReoccurringRates)
                .map(this::saveReoccurringRate),
        api.getBlackouts(locationId, skipExpired: true)
                .map(Blackout.MultipleHolder::getBlackouts)
                .map(this::saveAndMapBlackouts),
        Triple::new)
        .toCompletable()
        .compose(RxUtils.applySchedulersToCompletable());
```

# Observable.flatMap



# Observable.flatMap

```
query("Hello, world!")
    .flatMap(new Func1<List<String>, Observable<String>>() {
        @Override
        public Observable<String> call(List<String> urls) {
            return Observable.from(urls);
        }
    })
    .subscribe(url -> System.out.println(url));
```

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .subscribe(url -> System.out.println(url));
```

# RxJava в Android

1) Использование RxJava в Android не ограничено
2) Поддержка RxJava в Retrofit
3) Реализация стандартных для Android подходов в реактивном стиле: RxBindings, RxLifecycle и множество других

# RxJava в Retrofit

```
public interface ComicsService {

    @GET("comics")
    Observable<ComicsResponse> comics(@Query("offset") Long offset, @Query
                                      @Query("orderBy") String orderBy);

    @GET("comics/{comicsId}")
    Observable<ComicsResponse> comics(@Path("comicsId") Long id);

    @GET("comics/{comicsId}/characters")
    Observable<CharactersResponse> characters(@Path("comicsId") Long id);
}
```

# RxJava в Retrofit

```
@NonNull
private static Retrofit buildRetrofit() {
    return new Retrofit.Builder()
            .baseUrl(BuildConfig.API_ENDPOINT)
            .client(getClient())
            .addConverterFactory(GsonConverterFactory.create())
            .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
            .build();
}
```

# Реактивный запрос на сервер

```
ApiFactory.getComicsService()
        .comics(id)
        .map(ComicsResponse::getData)
        .map(ComicsResponseData::getResults)
        .map(list -> list.get(0))
        .compose(RxUtils.async())
        .subscribe(this::showComics, this::handleError);
```

# Отображение процесса загрузки

```
@Override
public void loadComics() {
    ApiFactory.getComicsService()
            .comics(ZERO_OFFSET, PAGE_SIZE, DEFAULT_COMICS_SORT)
            .map(ComicsResponse::getData)
            .map(ComicsResponseData::getResults)
            .doOnSubscribe(view::showLoading)
            .doOnTerminate(view::hideLoading)
            .compose(RxUtils.async())
            .subscribe(view::showItems, view::handleError);
}
```

# Other Utility Operators

- doOnEach – register an action to take whenever an Observable emits an item
- doOnCompleted – register an action to take when an Observable completes successfully
- doOnError – register an action to take when an Observable completes with an error
- doAfterTerminate – register an action to call just after an Observable terminated, either successfully or with an error
- doFinally - register an action to call when an Observable terminates or it gets disposed

# Кэшируем данные

```
return api.getLocation(locationId)
        .retry(1)
        .map(Location.formatName())
        .flatMap(location -> {
            AppDatabase.getInstance().locationDao().insertLocation(location);
            selectedLocation = location;
            return Single.just(location);
        })
        .toCompletable()
        .compose(RxUtils.applySchedulersToCompletable());
```

# Возвращаем закэшированнные данные в случае ошибки

```
.onErrorResumeNext(throwable -> {
    Realm realm = Realm.getDefaultInstance();
    RealmResults<Comics> results = realm.where(Comics.class).findAll();
    return Observable.just(realm.copyFromRealm(results));
})
```

# Кэшируем запросы

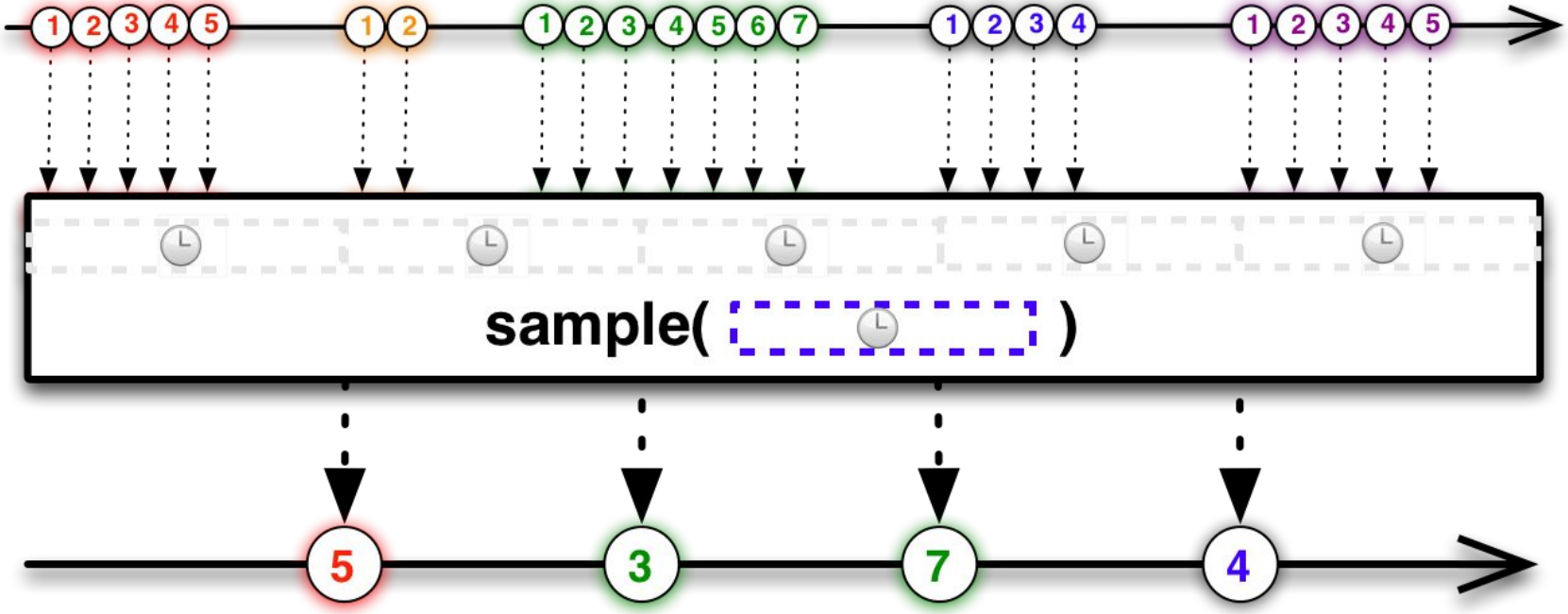
```
ApiFactory.getComicsService()
        .comics(ZERO_OFFSET, PAGE_SIZE, DEFAULT_COMICS_SORT)
        .map(ComicsResponse::getData)
        .map(ComicsResponseData::getResults)
        .cache()
        .doOnSubscribe(view::showLoading)
        .doOnTerminate(view::hideLoading)
        .compose(RxUtils.async())
        .subscribe(view::showItems, view::handleError);
```

# Проблема Backpressure

```
Observable<String> observable = Observable.create(new Observable.OnSubscribe<String>() {
    @Override
    public void call(Subscriber<? super String> subscriber) {
        for (int i = 0; i < 1000; i++) {
            subscriber.onNext(i + "");
        }
    }
});

observable.observeOn(Schedulers.computation())
        .subscribe(System.out::println, throwable -> {
            System.out.println("error: " + throwable);
        });
```

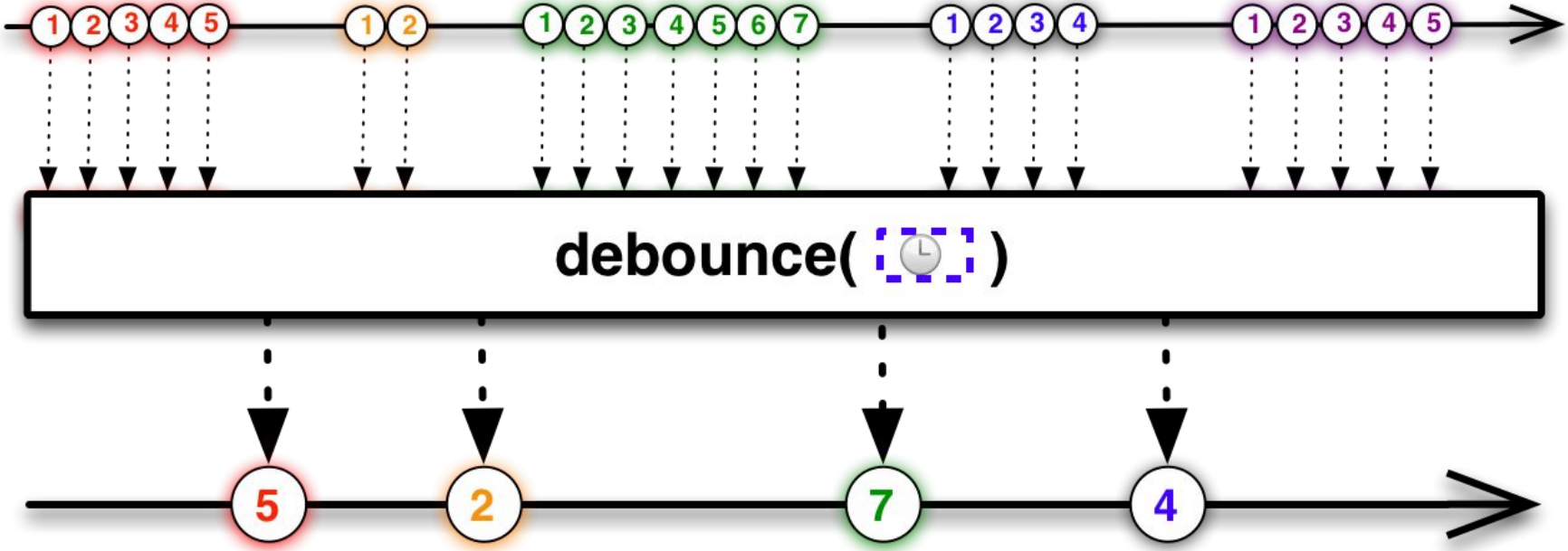
# Observable.sample



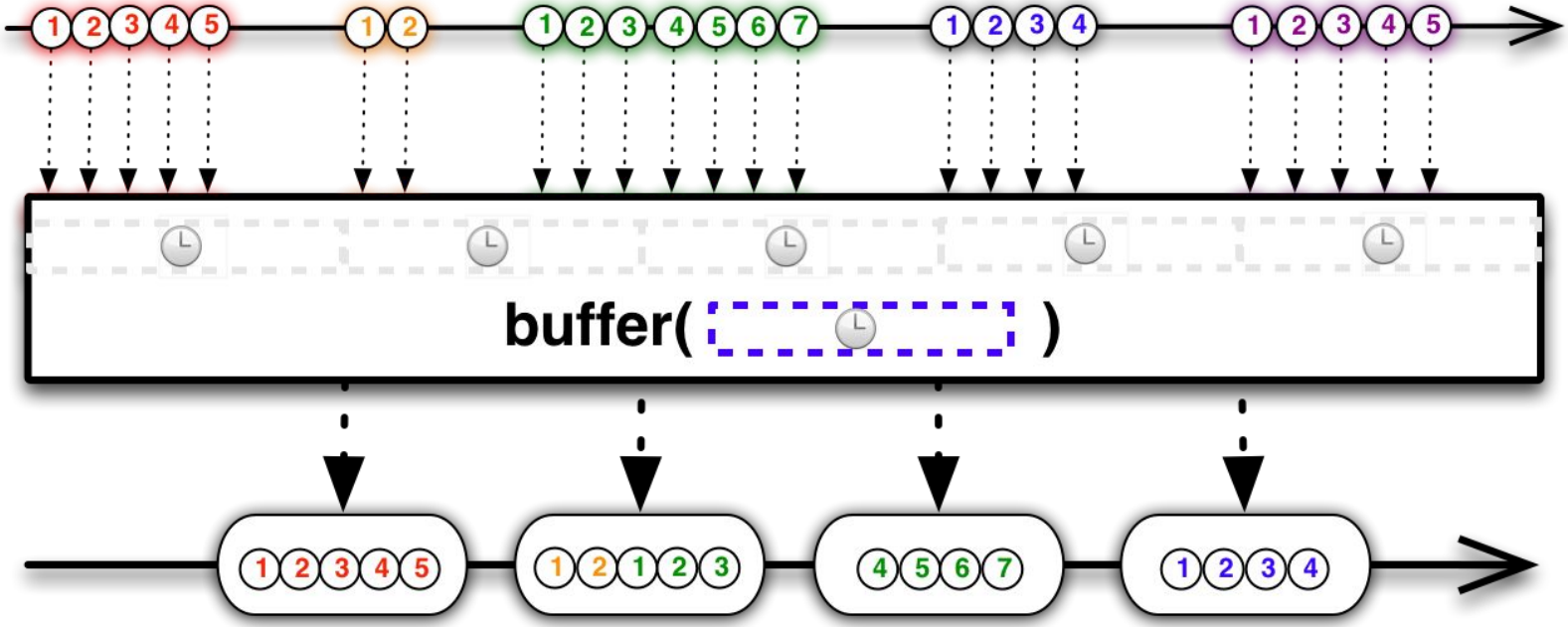
# Observable.sample

```
createBackpressureObservable()
        .sample(10, TimeUnit.MICROSECONDS)
        .observeOn(Schedulers.computation())
        .subscribe(System.out::println, throwable -> {
            System.out.println("error: " + throwable);
        });
```

# Observable.debounce



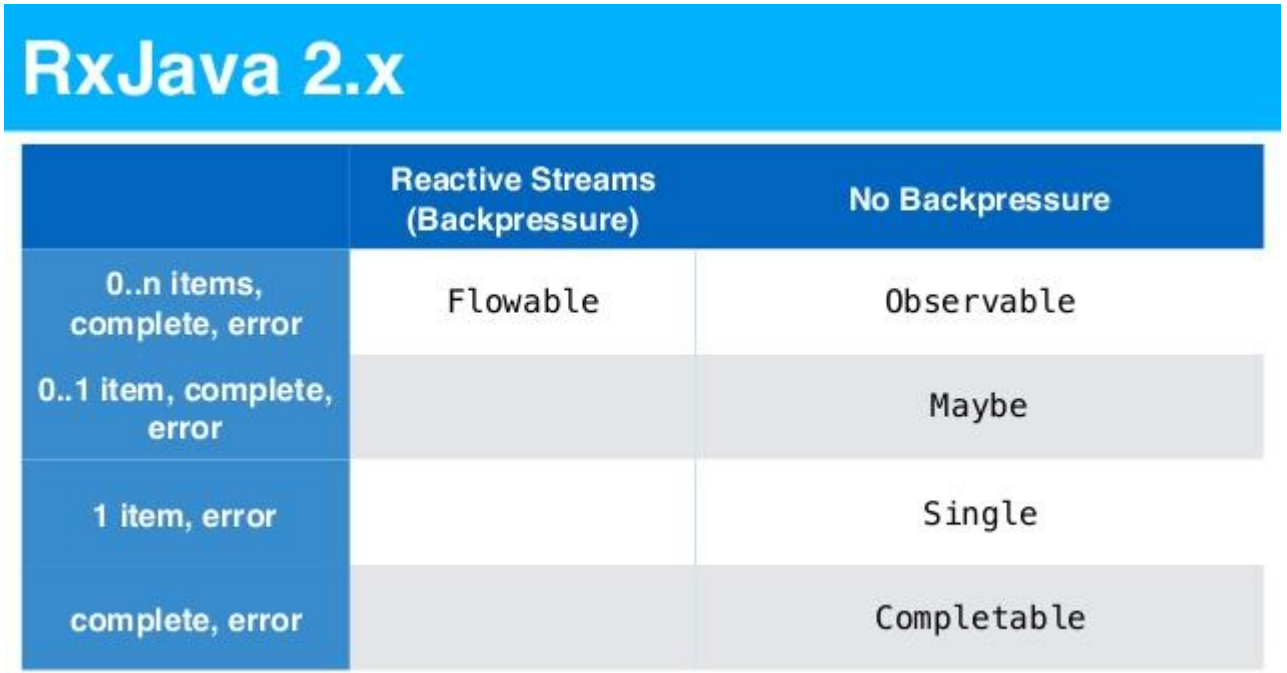
# Observable.buffer



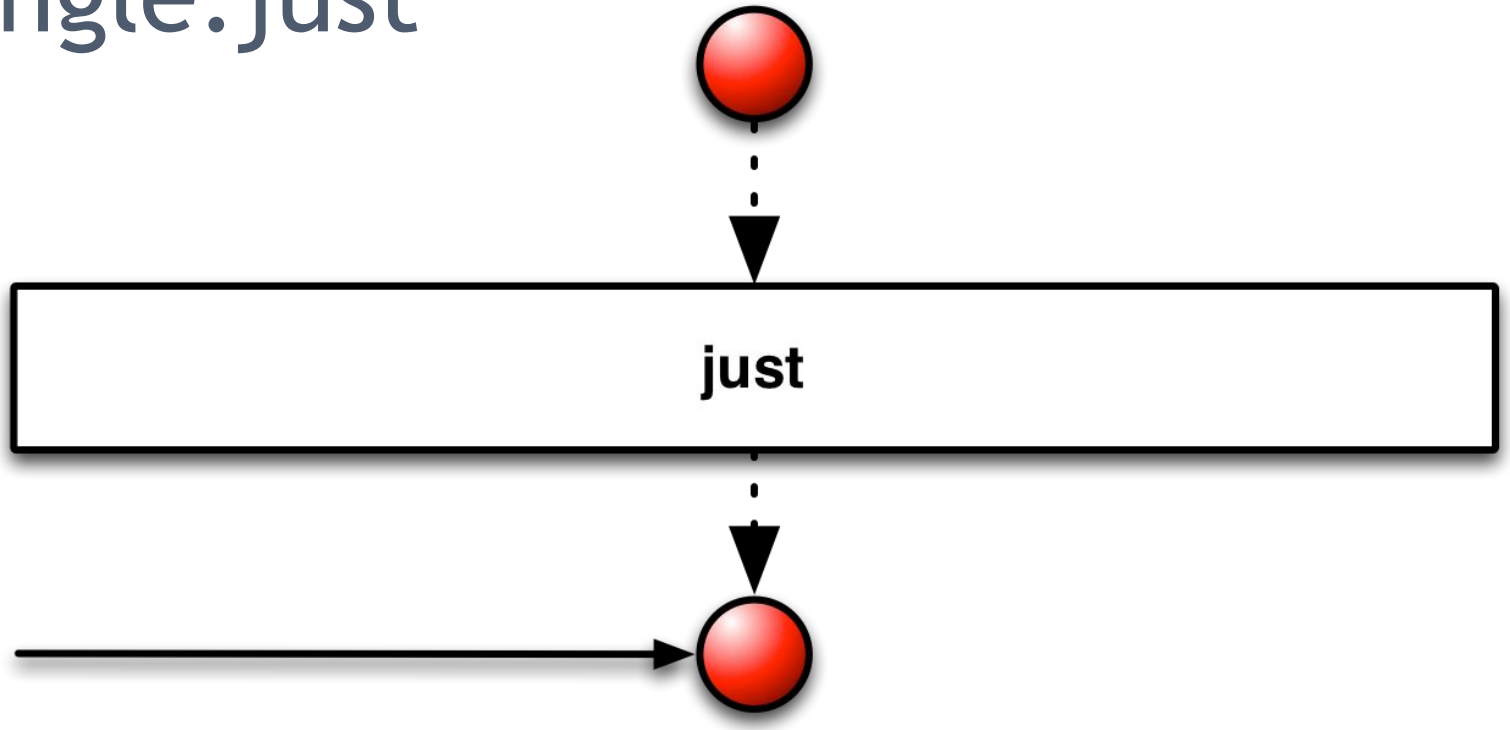
# Observable.buffer

```
createBackpressureObservable()
        .buffer(100)
        .observeOn(Schedulers.computation())
        .subscribe(System.out::println, throwable -> {
            System.out.println("error: " + throwable);
        });
```

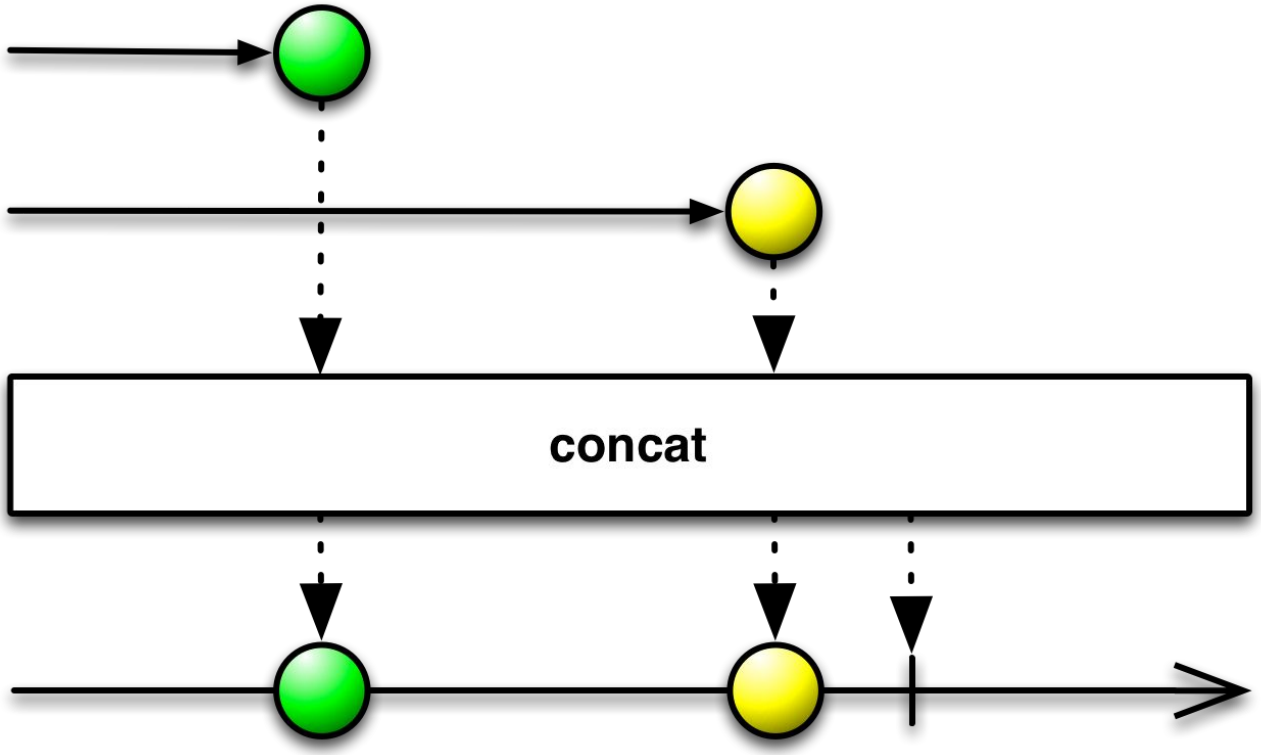
# RxJava2


RxJava 2.x
Reactive Streams (Backpressure)
No Backpressure
0..n items, complete, error
Flowable
Observable
0..1 item, complete, error
Maybe
1 item, error
Single
complete, error
Completable

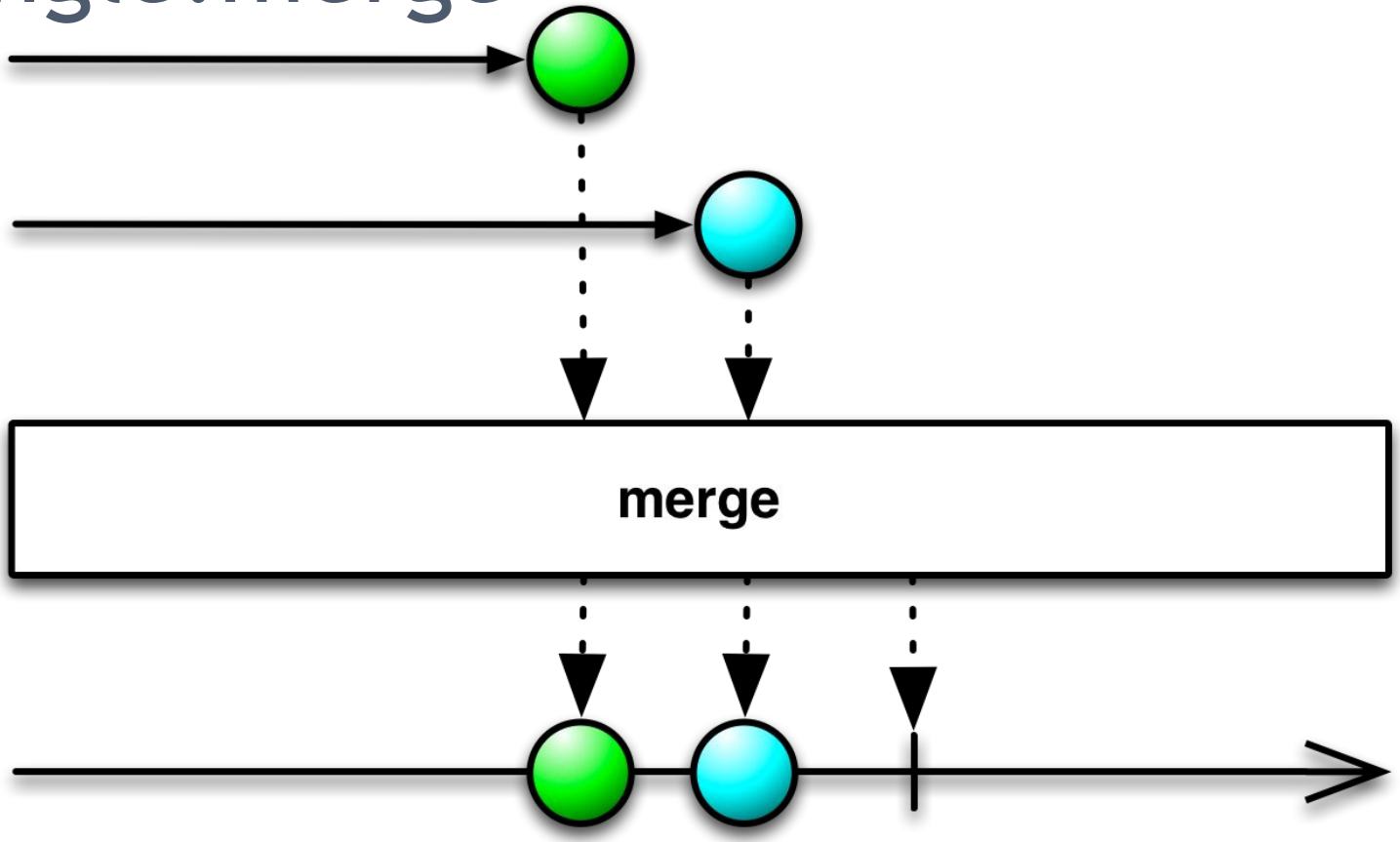
# Single.just



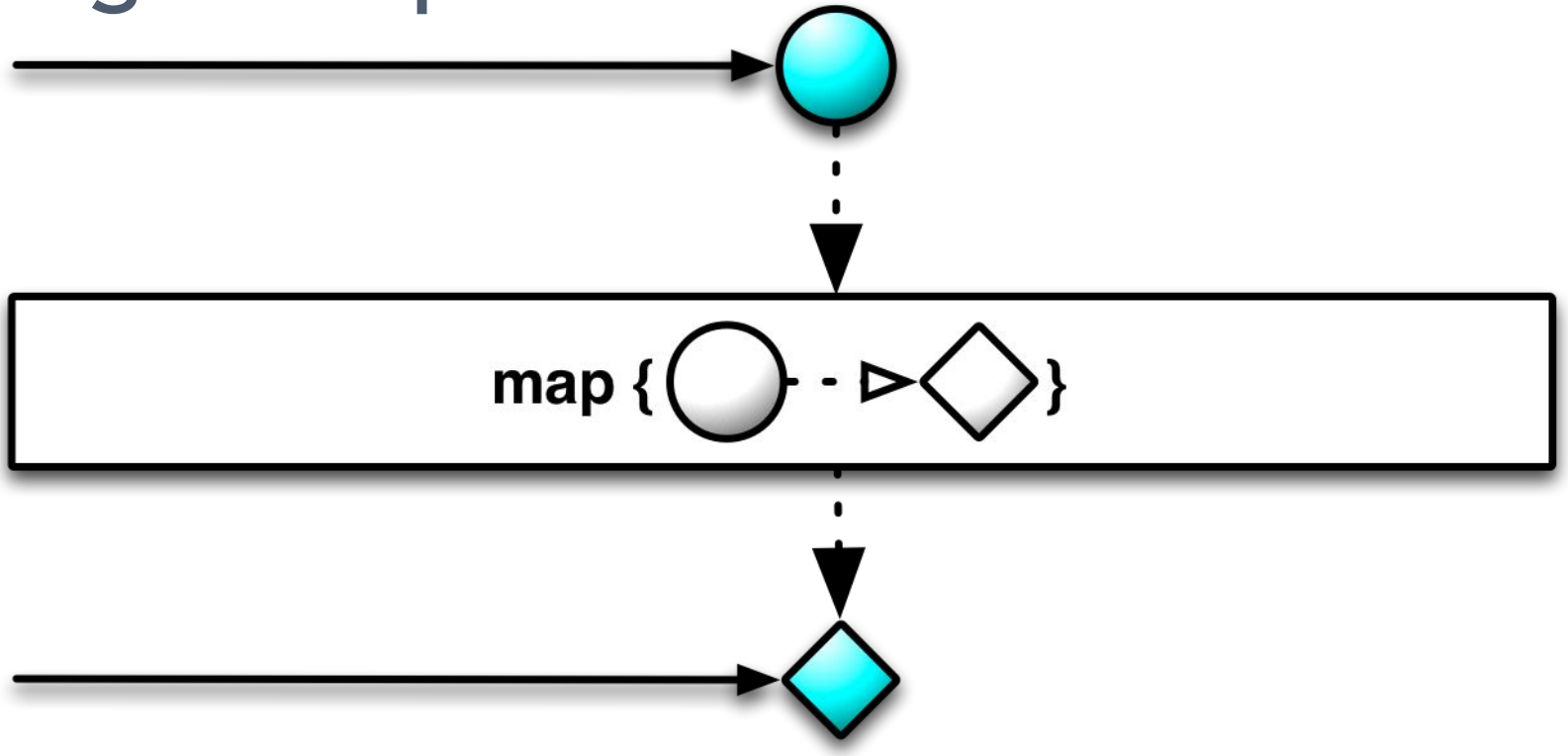
# Single.concat



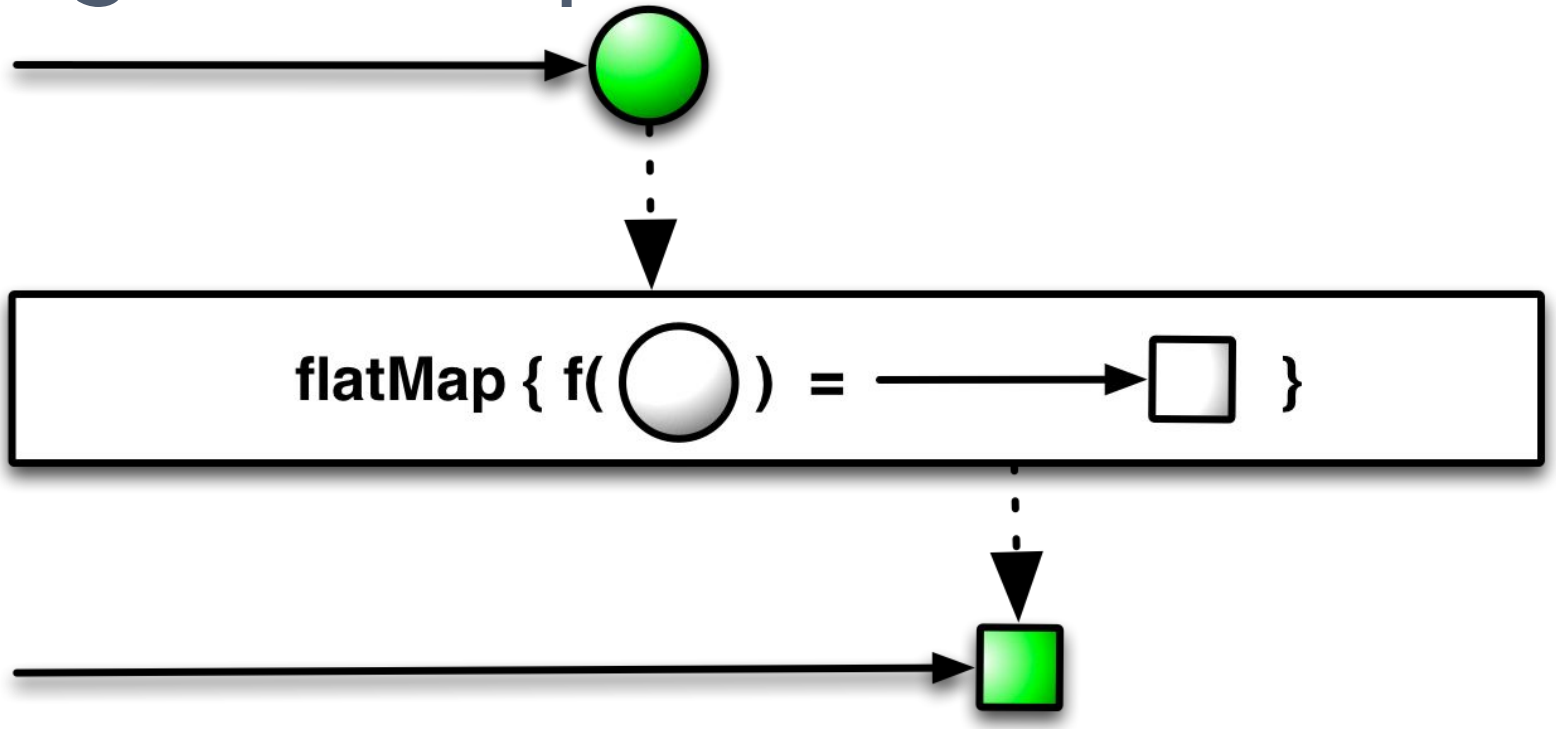
# Single.merge



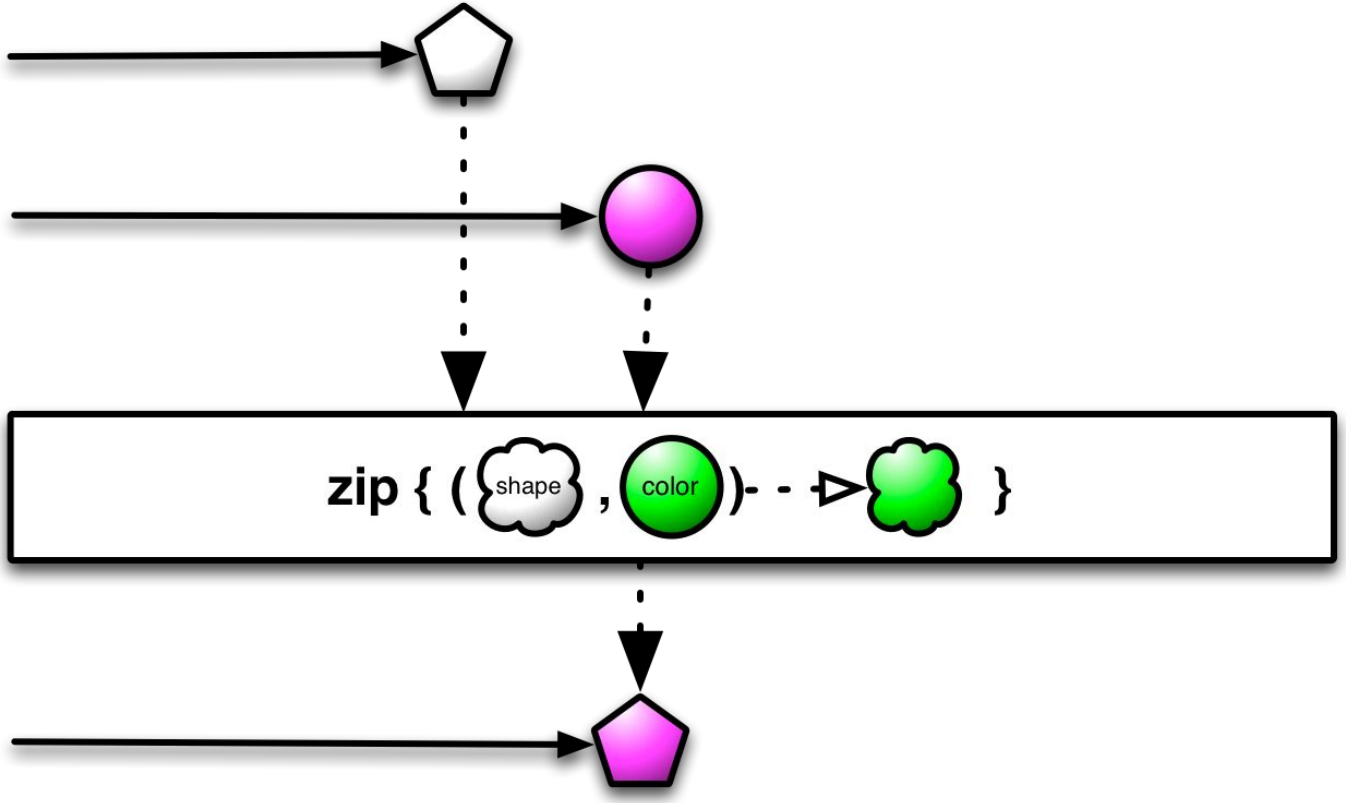
# Single.map



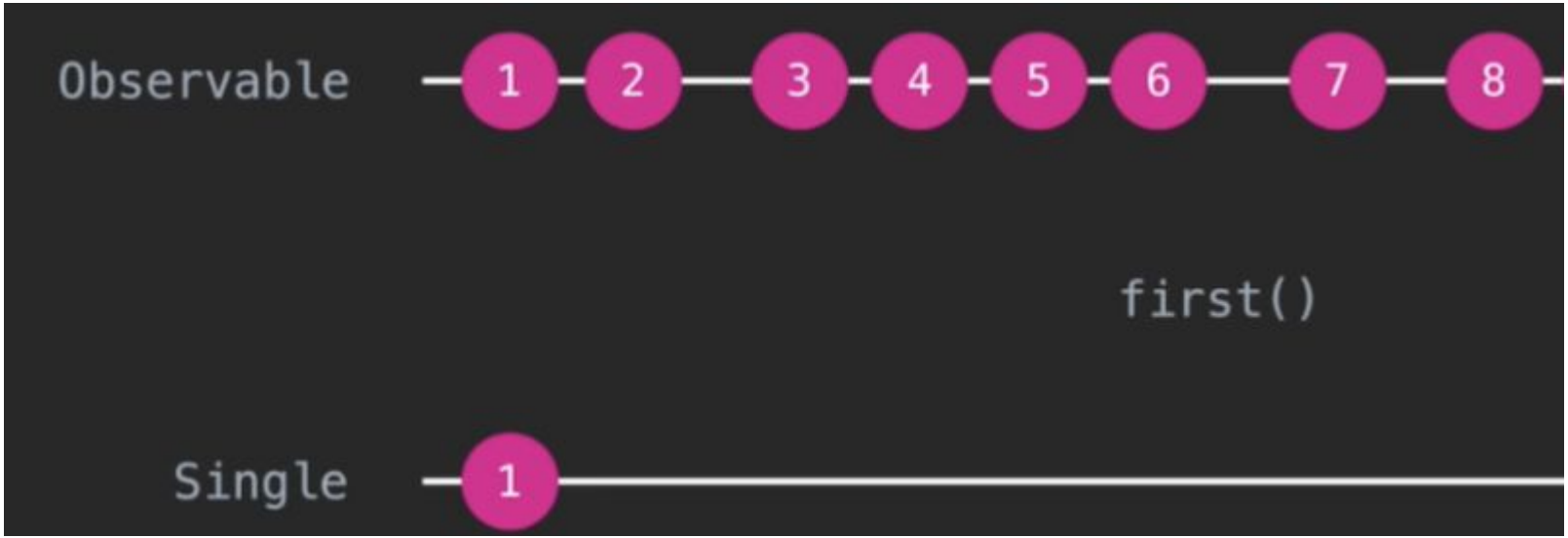
# Single.flatMap



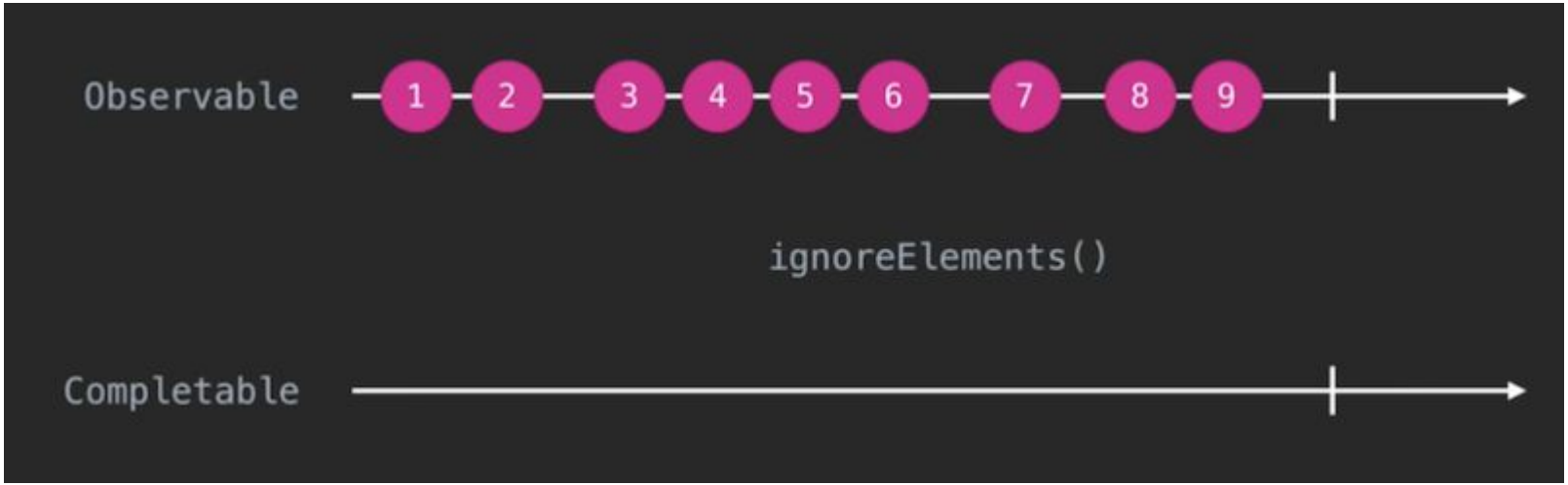
# Single.zip



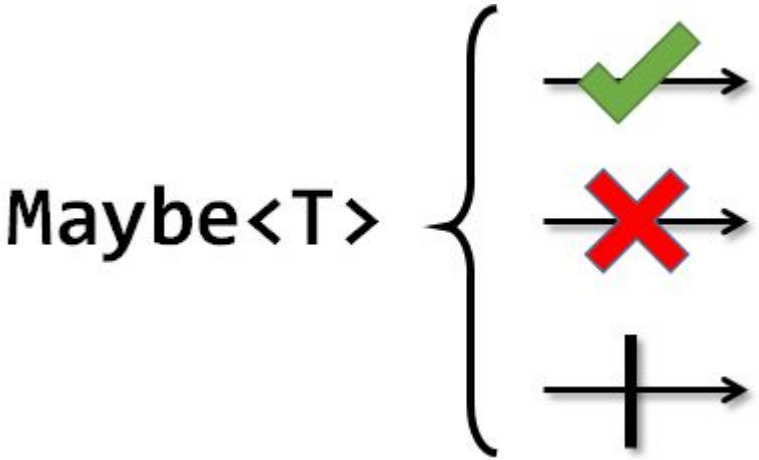
# Single



# Completable



# Maybe



# Maybe.fromAction

```
Maybe.fromAction(()-> System.out.println(“Hello”));
Maybe.fromRunnable(()-> System.out.println(“Hello”));

Completable.fromAction(()-> System.out.println(“Hello”));
Completable.fromRunnable(()-> System.out.println(“Hello”));
```

# Dispose

1) .subscribe – возвращает объект Disposable, который хранит состояние подписки на текущий момент.
2) Содержит два метода:
    1) isDisposed() – освобождены ли ресурсы
    2) dispose() – освободить ресурсы(отписаться)
3) В методе onDestroy() всегда стоит вызывать метод *dispose()*

# Dispose

```
if (nearbyLocationsDisposable?.isDisposed != false)
    nearbyLocationsDisposable = mapModel.getNearbyLocations()
            .doAfterTerminate {
                nearbyLocationsDisposable?.dispose()
                nearbyLocationsDisposable = null
            }
            .subscribeBy(onSuccess = {
                showLoadedNearbyLocations(it)
            }, onError = {
```

# Полезные ссылки

- http://reactivex.io/documentation/
- http://reactivex.io/tutorials.html
- https://github.com/ReactiveX/RxJava
- https://github.com/JakeWharton/RxBinding
- https://habrahabr.ru/post/265269/
- https://habrahabr.ru/company/badoo/blog/328434/