GDG Rzeszów

Taras Bazyshyn

Reaktywny Android z RxJava

GDG Rzeszów #2

13 kwietnia 2016

Kim jestem?

- Dłubię w Androidzie > 3 lata
- Team Leader/Android Software Developer @ PGS-Software
- fan Kotlina, RxJava'y

@tbazyshyn

PGS SOFTWARE

ReactiveX

Reactive Extensions
albo po prostu
Rx

Framework wzorowany na Observable pattern który

- wspiera sekwencje danych i/lub wydarzeń(events),
- pozwala przekształcać i przetwarzać te sekwencje w dowolny sposób nawet na Java6(Android 2.3)
- dodaje poziom abstrakcji dla sterowania wątkami.

CREATE

Easily create event streams or data streams.

COMBINE

Compose and transform streams with query-like operators.

LISTEN

Subscribe to any observable stream to perform side effects.

Functional

Avoid intricate stateful programs, using clean input/output functions over observable streams.

Less is more

ReactiveX's operators often reduce what was once an elaborate challenge into a few lines of code.

Async error handling

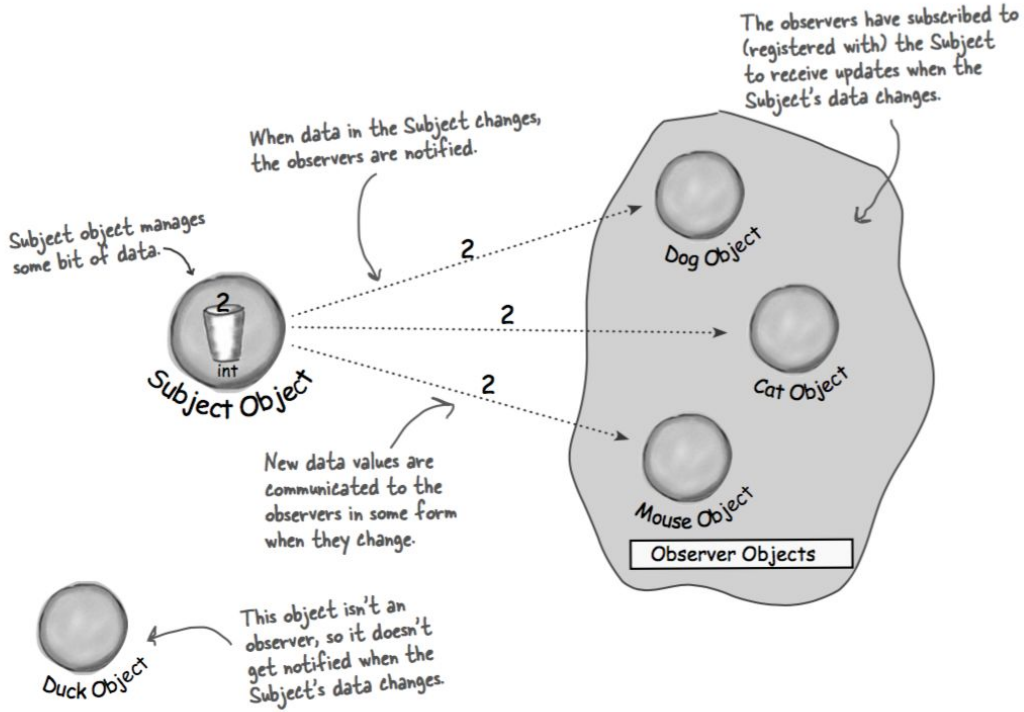
Traditional try/catch is powerless for errors in asynchronous computations, but ReactiveX is equipped with proper mechanisms for handling errors.

Concurrency made easy

Observables and Schedulers in ReactiveX allow the programmer to abstract away low-level threading, synchronization, and concurrency issues.

Observable pattern

- Observer nasłuchuje zdarzenia(events)
- Subject albo Observable generuje zdarzenia



źródło “Head First Design Patterns”, O'Reilly

Observer = Listener


```
btn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Timber.d("click");
    }
});
```

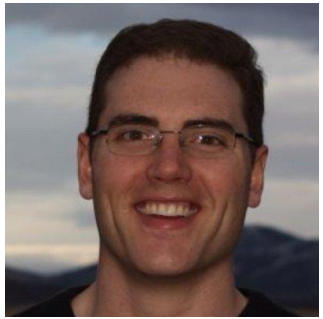
```
btn.setOnClickListener(v -> Timber.d("click"));
```

Dla czego kolejny framework

- Piekło z AsyncTask’ów, Handler’ów itd
- Trudne przerzucanie się między wątkami.
- Brak funkcjonalności Stream API w Java 1.7(filter(), map() itd)
- Żeby nadążyć za światem technologii w którym Rx coraz bardziej rośnie

Twórcy Rx’a

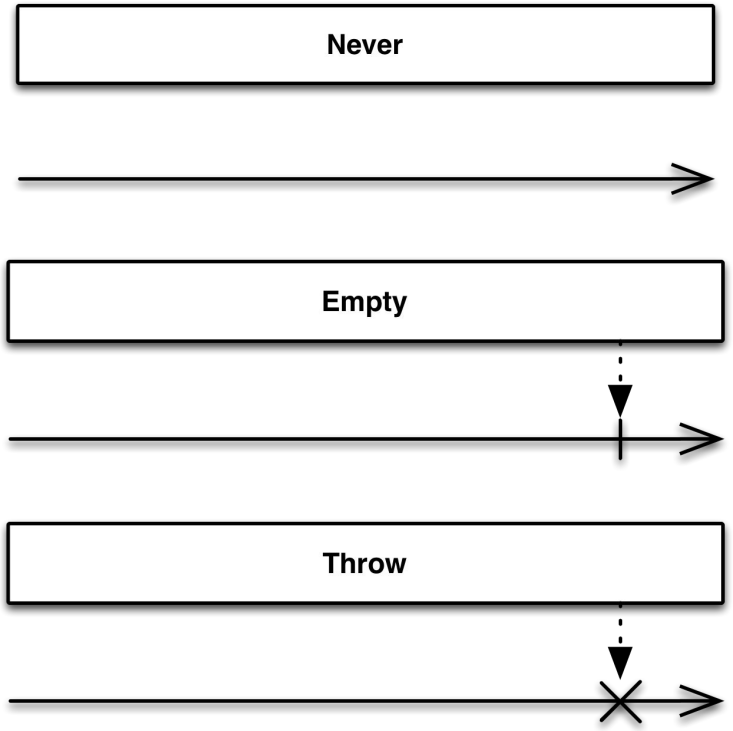
Erik Meijer, Applied Duality - Rx.Net



Ben Christensen, Netflix(teraz FB) - RxJava

Jake Wharton, Square - RxAndroid, RxBinding

Podstawy Rx’a



Observer/Subscriber

```
Subscriber<T> implements Observer
rx.Observer<String> observer = new Observer<String>() {
    @Override
    public void onCompleted() {
        Timber.d("completed");  ==  () -> Timber.d("completed")
    }

    @Override
    public void onError(Throwable e) {
        Timber.e(e, "onError:");  ==  e -> Timber.e(e, "onError:")
    }

    @Override
    public void onNext(String s) {
        Timber.d("onNext: %s", s);  ==  s -> Timber.d("onNext: %s", s)
    }
};
```

Observable

```
final List<String> LIST = Arrays.asList("Hello", "GDG", "Rzeszow", null);
//creates stream that is iterating on list of strings
Observable<String> stringObservable =
        Observable.create(new Observable.OnSubscribe<String>() {
            @Override
            public void call(Subscriber<? super String> subscriber) {
                for (String s : LIST) {
                    if (s == null) {
                        subscriber.onError(new NullPointerException("NPE!!!"));
                        break;
                    }
                    subscriber.onNext(s);
                }
                subscriber.onCompleted();
            }
        });
```

Subscription

Odpalamy magie

```
Subscription subscription = stringObservable
        .subscribe(
                s -> Timber.d("onNext: %s", s),
                e -> Timber.e(e, "onError:"),
                () -> Timber.d("completed")
        );
```

wynik:

```
onNext: Hello
onNext: GDG
onNext: Rzeszow
onError:
java.lang.NullPointerException: NPE!!!
```

Unsubsribe

```
Subscription subscription = stringObservable
        .subscribe(
                s -> Timber.d("onNext: %s", s),
                e -> Timber.e(e, "onError:"),
                () -> Timber.d("completed")
        );
```

trzyma referencje **$this**

Unsubsribe

```
Subscription subscription = stringObservable
        .subscribe(
                s -> Timber.d("onNext: %s", s),
                e -> Timber.e(e, "onError:"),
                () -> Timber.d("completed")
        );
```

```
@Override
protected void onDestroy() {
    super.onDestroy();
    if (!subscription.isUnsubscribed()) {
        subscription.unsubscribe();
    }
}
```

Scheduler


```
Subscription subscription = stringObservable
        .subscribeOn(Schedulers.io()) // IO-bound work.
        .subscribeOn(Schedulers.computation()) // event-loops,
                                processing callbacks and other computational work.
        .subscribeOn(Schedulers.immediate()) // immediately on the current thread.
    .subscribeOn(Schedulers.newThread())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
            s -> Timber.d("onNext: %s", s),
            e -> Timber.e(e, "onError:"),
            () -> Timber.d("completed")
    );
```

observeOn vs subscribeOn

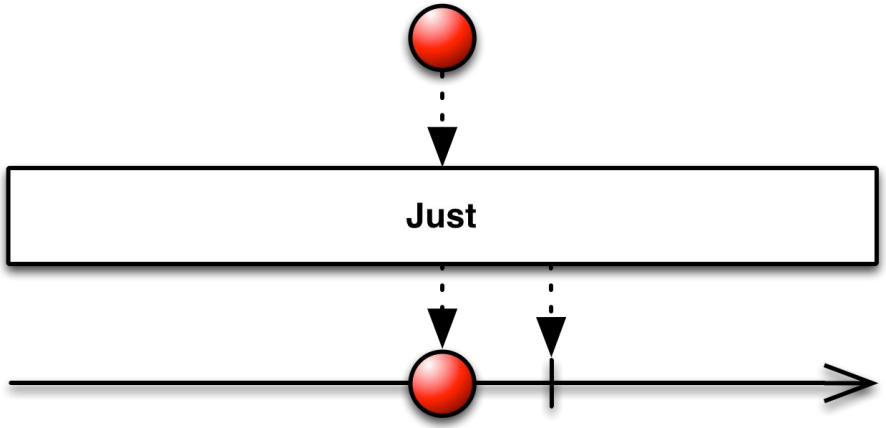
- pierwszy **subscribeOn** jest najważniejszy. Kolejne dodane będą ignorowane.
- **observeOn** odpala wszystkie akcje które idą po nim. **observeOn** możemy używać wiele razy w sekwencji
- operatory (interval, range, never) mogą odpalać się na specyficznym dla każdego wątku. Przed użyciem operatora przeczytać @javadoc część o thread'ingu

Praca z wyjątkami

- .doOnError()
- .onErrorResumeNext()
- .onErrorReturn()
- .onError() we wlasnej implementacji

Operatory tworzenia Observable

```
//just creates stream with one string element
Observable.just("Hello GDG Rzeszow")
        .subscribe(s -> Timber.d("onNext: %s", s),
                e -> Timber.e(e, "onError:"),
                () -> Timber.d("completed"));
```

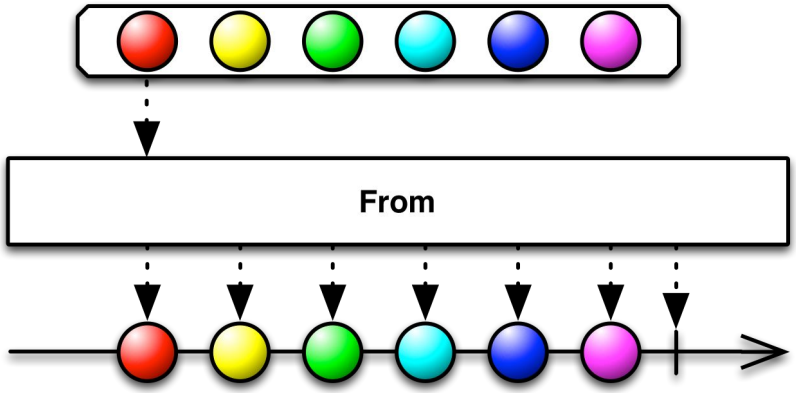


wynik:

```
onNext: Hello GDG Rzeszow
completed
```

Operatory tworzenia Observable#2

```
Subscription subscription =
        Observable.from(Arrays.asList("Hello", "GDG", "Rzeszow", null))
                .subscribe(
                        s -> Timber.d("onNext: %s", s),
                        e -> Timber.e(e, "onError:"),
                        () -> Timber.d("completed")
                );
```



wynik:

```
onNext: Hello
onNext: GDG
onNext: Rzeszow
onNext: null
completed
```

Operatory tworzenia Observable#3

- create()
- deffer()
- timer()
- range()
- interval()
- fromCallable()

WARNING!!!

Operatory

map()

```
Subscription subscription = Observable.from(
        Arrays.asList("Hello", "GDG", "Rzeszow"))
        .map(s -> String.format("%s (%d)", s, s.length()))
        .subscribeOn(Schedulers.computation())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
                s -> Timber.d("onNext: %s", s),
                e -> Timber.e(e, "onError:"),
                () -> Timber.d("completed")
        );
```

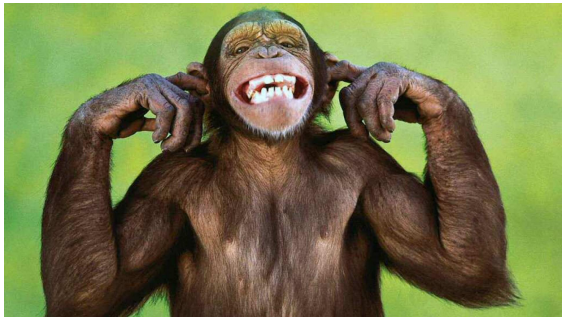
wynik:

```
onNext: Hello (5)
onNext: GDG (3)
onNext: Rzeszow (7)
completed
```

flatMap()

```
Subscription subscription = Observable.from(
        Arrays.asList("Cat", "Dog", "Monkey"))
        // request to search service
        .flatMap(s -> imageSearchService.search(s))
        .subscribeOn(Schedulers.computation())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
                s -> Timber.d("onNext: %s", s),
                e -> Timber.e(e, "onError:"),
                () -> Timber.d("completed")
        );
```

wynik:



Inne piękne operatory

- zip()
- merge()
- distinct()
- filter()
- take()
- takeLast()
- skip()

Przykłady użycia w Androidzie

Async Task

```
private void runAsync() {
    new HeavyAsyncTask().execute();
}

class HeavyAsyncTask
        extends AsyncTask<String, Void, List<String>> {

    @Override
    protected void onPreExecute() {
        super.onPreExecute();
    }

    @Override
    protected List<String> doInBackground(String... params) {
        SystemClock.sleep(1000);
        //doing long operation
        LIST.addAll(Arrays.asList(params));
        return LIST;
    }

    @Override
    protected void onPostExecute(List<String> strings) {
        super.onPostExecute(strings);
        adapter.addAll(strings);
    }
}
```

```
Subscription subscription = stringObservable
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(s -> adapter.add(s),
                e -> Timber.e(e, "onError:"));
```

Prosta migracja z AsyncTask'ów


```
Observable.fromCallable(() -> runHeavyOperation())
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
                s -> Timber.d("onNext: %s", s),
                e -> Timber.e(e, "onError:"),
                () -> Timber.d("completed")
        );
```

TimerTask


```
//run a task constantly every 5s,
// but start only after 2s,
// after 10th event ,
// terminate automatically
Observable
        .interval(2, 5, TimeUnit.SECONDS)
        .take(10)
        .subscribe(
                number -> Timber.d("next %d", number),
                e -> Timber.e(e, "error:"),
                () -> Timber.d("completed")
        );
```

Bus


```
public class RxBus {

    private final Subject<Object, Object> _bus =
            new SerializedSubject<>(PublishSubject.create());

    public void send(Object o) { _bus.onNext(o); }

    public Observable<Object> toObserverable() { return _bus; }

    public boolean hasObservers() { return _bus.hasObservers(); }
}
```

używamy https://github.com/JakeWharton/RxBinding

Scroll Events


```
RxAbsListView.scrollEvents(listView)
        .subscribe(
                sE -> {
                    int visibleItemCount = sE.visibleItemCount();
                    int pastVisibleItems = sE.firstVisibleItem();
                    int totalItemCount = sE.totalItemCount();
                    if ((visibleItemCount + pastVisibleItems) >= totalItemCount)
                        loadNextPage();
                    }
                },
                e -> Timber.e(e, "error:")

        );
```

Inteligent search calls

```
subscription =
        //listen onTextChanged
        RxTextView.textChangeEvents(inputSearchText)
                //waiting for 400 to get last event
                // returns us text written after 400ms
                .debounce(400, TimeUnit.MILLISECONDS) //computation scheduler
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(
                        textChangeEvent ->
                                Timber.d(textChangeEvent.text().toString()),
                        e -> Timber.e(e, "error:")
                );
```

UWAGI

- Android Studio pod czas import'u podpowiada dwie klasy Observable
  - **import** rx.Observable; < ten uzywamy w calej prezentacji
  - **import** java.util.Observable;
- W Java6 tworzy dużo anonimowych klas
- Biblioteka jest dość wielka : ~5.1k metod, 770KB(v1.1.3)
- Bardzo łatwo złapać Memory Leak
- Nie zawsze łatwe do debugowania
- Nadużycie **flatMap()** może doprowadzić do końca miejscu na stack'u a Dalvik/ART nic nam nie powie.
- RxJava domyślnie jest synchroniczną

Cudowne biblioteki

- https://github.com/mcharmas/Android-ReactiveLocation - nakładka na Play Services
- https://github.com/Polidea/RxAndroidBle - obsługa Bluetooth Low Energy
- https://github.com/patloew/RxFit - obsługa Google Fit
- https://github.com/patloew/RxWear
- https://github.com/VictorAlbertos/RxGcm - obsługa Google Cloud Messaging
- https://github.com/trello/RxLifecycle - bindowanie do cyklu życia

PYTANIA?

Taras Bazyshyn

@tbazyshyn

Materiały:

https://goo.gl/mvr1st