Conquering Concurrency

Bringing the Reactive Extensions to the Android platform

Matthias Käppler @mttkay

October 2013

“To enrich our lives through the shared love of sound.”

“That place where Snoop ~~Dog~~ Lion, indie musicians, podcasters and dubstep drop junkies share a space.”

~ Me (ca. 2013)

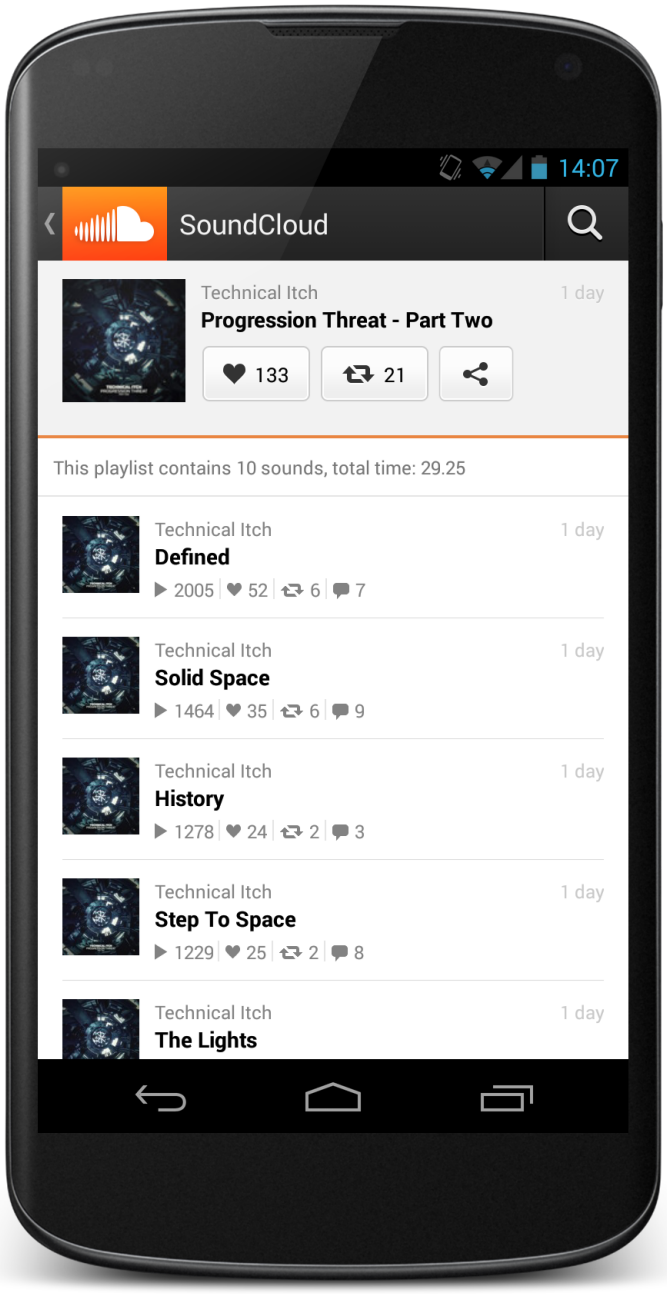
12 hours of audio added every minute

200M users reached every month

14M Android downloads

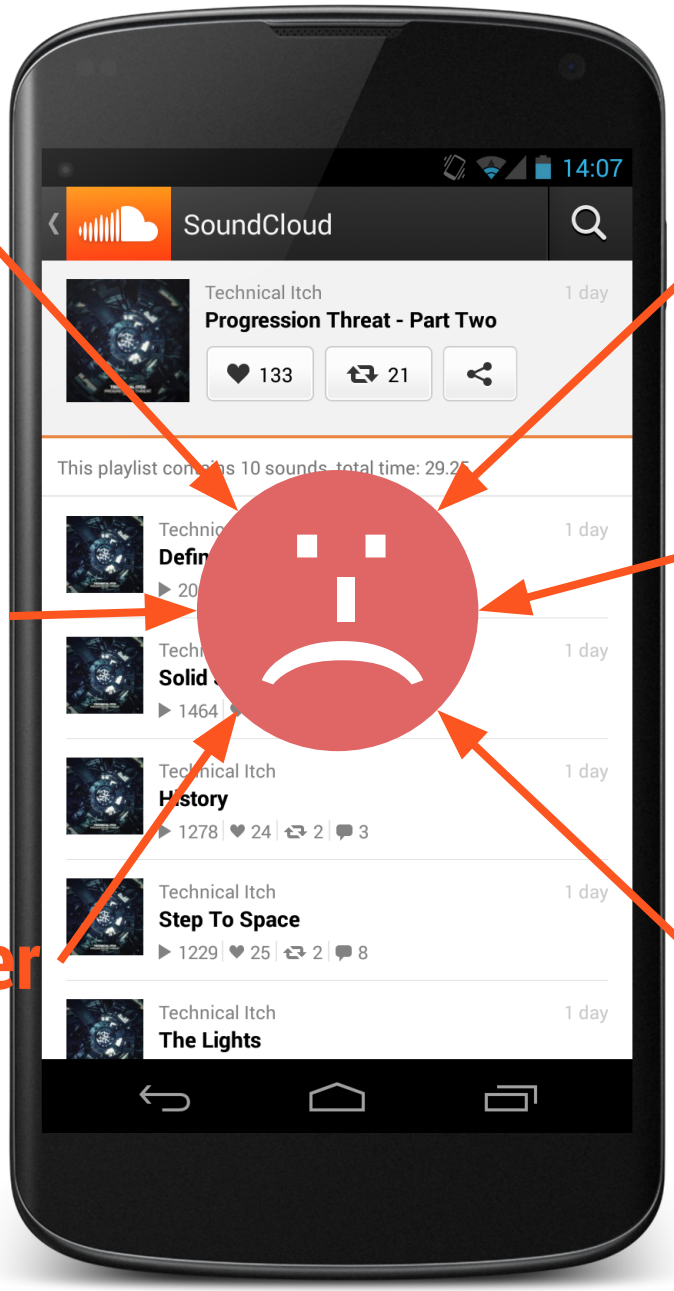
Essentially

This talk is about functional reactive programming.


14:07
SoundCloud
Technical Itch
1 day
Progression Threat - Part Two
133
21
This playlist contains 10 sounds, total time: 29.25
Technical Itch
1 day
Defined
2005 52 6 7
Technical Itch
1 day
Solid Space
1464 35 6 9
Technical Itch
1 day
History
1278 24 2 3
Technical Itch
1 day
Step To Space
1229 25 2 8
Technical Itch
1 day
The Lights

ResultReceiver

ContentObserver



Custom listeners

Loader callbacks

BroadcastReceiver

Pull-to-refresh callbacks


I TRIED
Sketching
The event flow


More broken windows
Nested AsyncTasks
Inline threads (or none)
Custom callbacks galore
http://somrat.deviantart.com/gallery/
WALLPAPERSWIDE.COM

Radical changes required to

→ Streamline event handling

→ Embrace concurrency

→ **Unified event model**

RxJava

github.com/netflix/rxjava

“Functional Reactive Programming on the JVM”

Imperative programming

```
int x = 1

int y = x + 1

x = 2
```

→ y: 2

Imperative programming

```
int x = 1

int y = x + 1

x = 2
```

→ y: 2

Reactive programming

```
int x = 1

Func<int> y = () → { x + 1 }

x = 2
```

→ y: 3

Reactive programming

```
int x = 1

Func<int> y = () → { x + 1 }

x = 2
```

→ y: 3

Imperative programming

+ declarative, lazy evaluation

= Reactive programming

+ higher order functions, composition

= Functional reactive programming

What does this mean for mobile applications?

Fact: UI driven applications are event based and reactive by nature.

Fact: Today's data comes from the web.

Our programming style should reflect that!

→ Asynchronous, declarative APIs

aka "Ask to construct" (M. Odersky)

→ Events as observable sequences

→ Embrace failure

What about...

AsyncTask

→ Single threaded, uses Futures + Handlers

→ Very prone to leaking **Context**

→ No error-handling

→ Not composable

Event buses

Go a long way to improve this, however:

→ No built in error-handling model

→ Events are not composable

→ Designed around global, shared state

BACK TO

RxJava

Observables

⟶ Events as **observable** sequences

```
Observable.create((observer) -> {
    for (int i = 1; i <= 3; i++) {
      observer.onNext(i);
    }
    observer.onCompleted();
  }).subscribe(intObserver);

// Emits values: 1, 2, 3
```

Observers

```
Observer<Integer> intObserver = new Observer<Integer> {

  public void onNext(Integer value) {
    System.out.println(value);
  }

  public void onCompleted() {
    System.out.println("Done!");
  }

  public void onError(Throwable t) { … }
}
```

Composition

→ Transformed/composed with operators

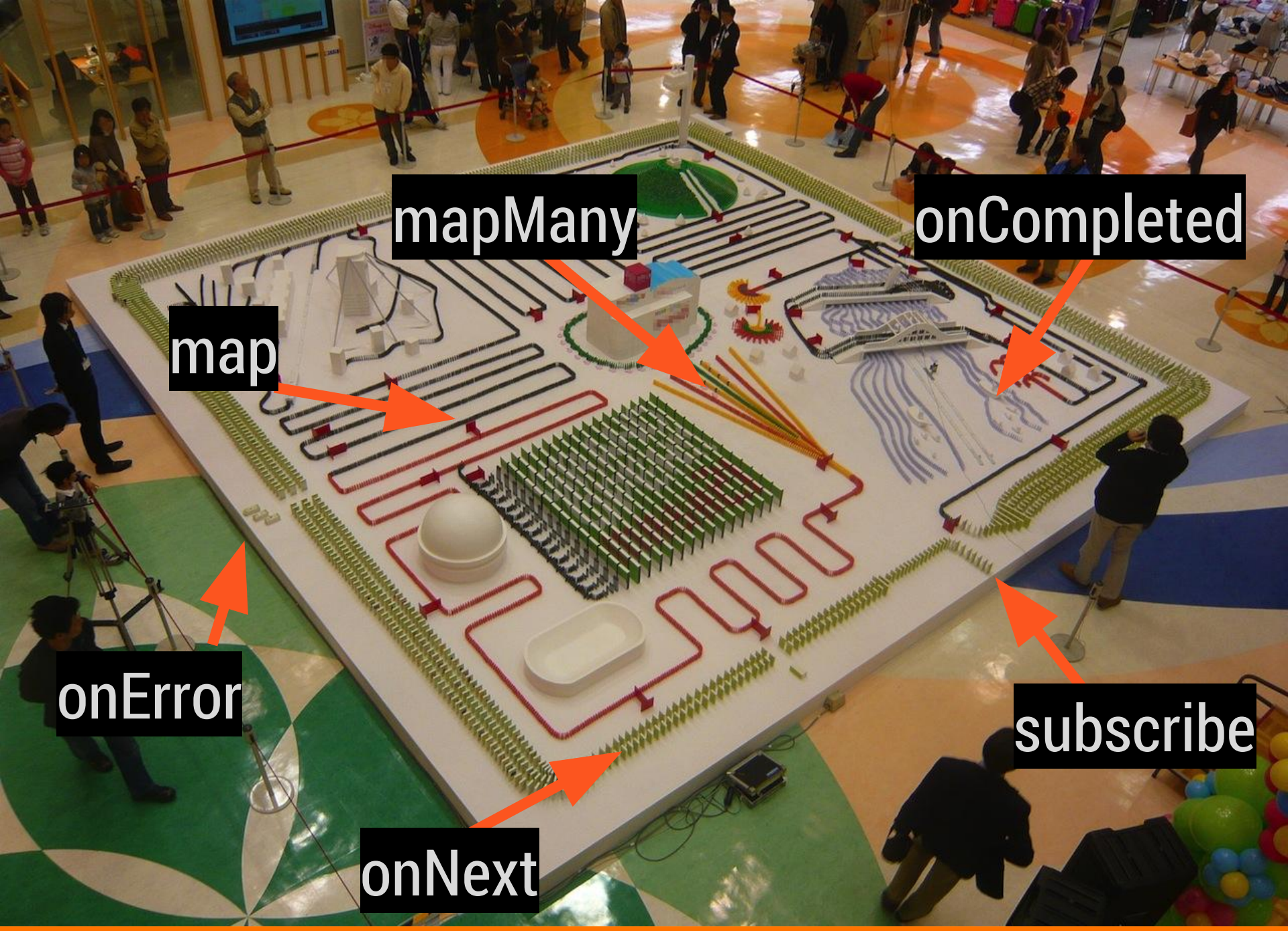
```
// Observable from previous example
observable.map((i) -> { return i * 2; }).subscribe(...)

// Standard out now prints:
2
4
6
Done!
```

Schedulers

→ Parameterized concurrency via **schedulers**

```
// observable from previous example
observable
    .subscribeOn(Schedulers.newThread())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(intObserver);
```


mapMany
onCompleted
map
onError
subscribe
onNext

How do we do it.

Dumb fragments (observe and update)

+ Service calls exposed as Observable<T>

+ Custom operators (e.g. for paging)

+ Reactive components (e.g. adapters)

Example: Fragment

Interacts with service object to get results pushed into observer

```
//e.g. in onCreate
Observable<Track> observable =
  AndroidObservables.fromFragment(
    this, service.loadTracks())
  .subscribe(this)
```

Example: Service object

Interacts with service API to fetch, map, and emit result data

```
public Observable<Track> loadTracks() {
    APIRequest<Track> request = /* build request */
    return mRxHttpClient.fetchModels(request);
}
```

Example: HTTP client

Sends HTTP request + maps response data

```
public Observable<T> fetchModels(APIRequest request) {
  return fetchResponse(request).mapMany((response) -> {
     return mapResponseToModels(request, response);
  });
}
```

Some observations

1. **Simple, uniform** event model
   onNext* ⟶ onCompleted | onError

2. **Reusable:** declarative definition of asynchronous task compositions

3. **Simple to test:** concurrency is parameterized

Could it possibly…?

:-(

→ Java 6 anonymous classes

→ Deep call stacks

→ Slight increase in GC activity

→ Learning curve

soundcloud.com/jobs

References

1) https://github.com/soundcloud/rxjava
2) http://rx.codeplex.com/
3) http://www.reactivemanifesto.org
4) mttkay.github.io/blog/2013/08/25/functional-reactive-programming-on-android-with-rxjava
5) http://laser.inf.ethz.ch/2012/slides/Odersky/odersky-laser-1.pdf
6) http://blog.maybeapps.com/post/42894317939/input-and-output
7) http://paulstovell.com/blog/reactive-programming

Image attributions

1) Sad guy: http://www.empireclaims.co.uk/blog/wp-content/uploads/2012/07/Desperate-man-neyvendotcom.jpg
2) Broken window: http://4.bp.blogspot.com/_yNbpKuMvpDI/TUuzAiAI8yI/AAAAAAAAATU/RNqPwwcllxc/s1600/3D-broken_window-1(www.CoolWallpapers.org).jpg
3) Android: http://vmatechs.com/wp-content/uploads/2013/07/android.jpg
4) Domino Days: http://upload.wikimedia.org/wikipedia/commons/6/6d/Domino_01.jpg
5) Holy grail: http://powet.tv/powetblog/wp-content/uploads/2010/02/holy_grail_post_banner1.jpg