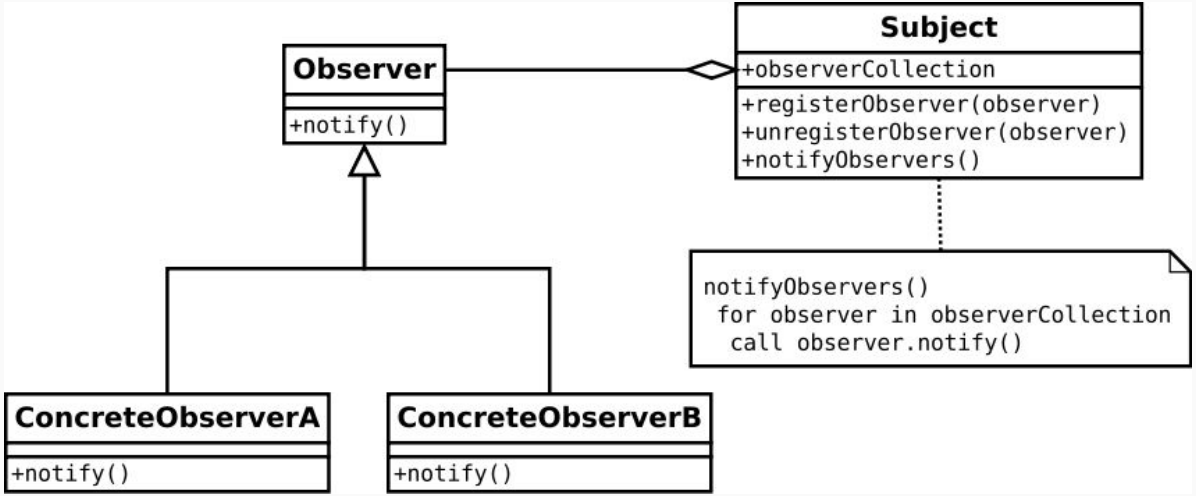
Brief Overview of Rx

What is Rx

- Stands for Reactive Extensions
- Is a family of libraries across many programming languages
  - Conforms to a similar API across the board, with some changes based on language conventions and threading models
- Allows for a functional approach to concurrency and parallelism
  - “An API for asynchronous programming with observable streams”
- The project is backed by some big companies
  - Netflix maintains RxJava
  - Microsoft maintains RxJs

The Observable or Stream



- Essentially just the Observer pattern
  - On steroids
- An Observable is a stream of values, which an Observer can react on
- Focus on immutability, functional programming paradigms and composition

What Makes Rx Work?

Observable

Scheduling

Observer

```
Observable.create<ApiResponse> { emitter: ObservableEmitter<T> ->
        val response: Response<T> = getApiResponse()
        if (response.isSuccessful) {
            emitter.onNext(response.body())
        } else {
            val httpStatusCode = HTTPStatusCode.init(response.code())
            emitter.onError(HttpError(httpStatusCode,response.errorBody().string()))
        }

        emitter.onComplete()
}
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe({ response: ApiResponse ->
        // onNext
        val viewModel = ViewModel(response)
        view.update(viewModel)
}, { error: Throwable ->
        // onError
        view.showError()
})
```

PublishSubject

- A subclass of Observable that allows for new events to be emitted
- Common use case
  - One class creates a PublishSubject
  - Gives the PublishSubject to some other object as an Observable (read only)
  - When some event happens, emit a value: subject.onNext(item)
- Very useful shortcut for creating your own Observables

```
val subject = PublishSubject.create<String>
subject.onNext("Hello World")
subject.subscribe { toast.show(it) }
```

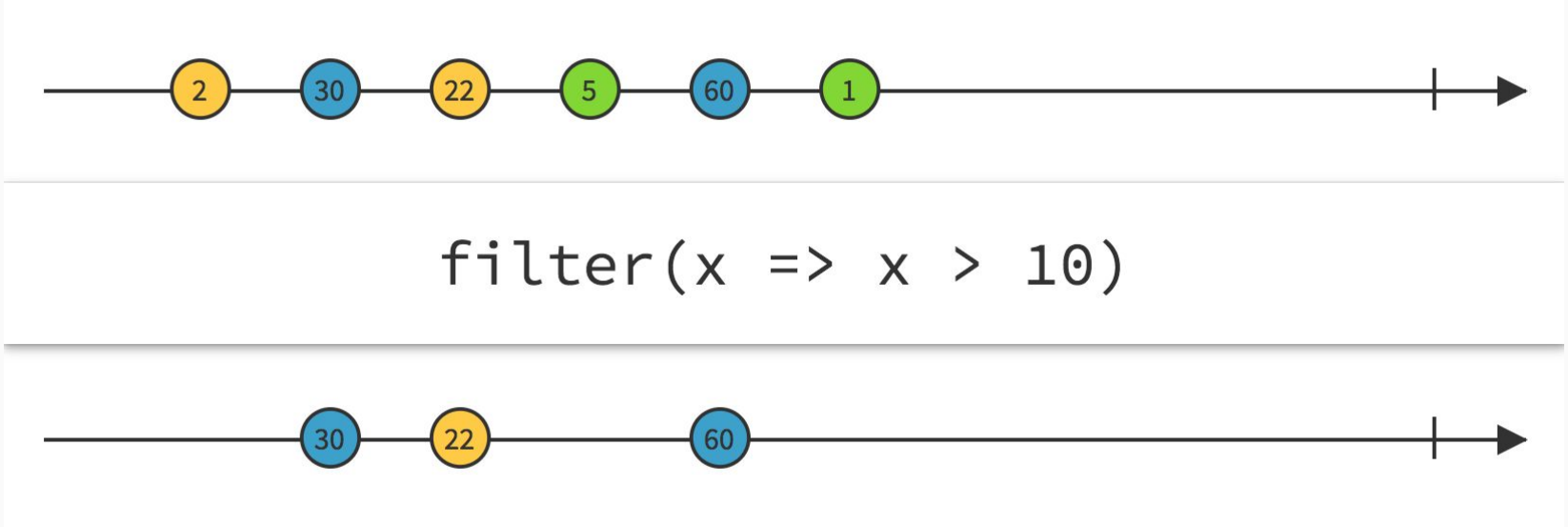
Basic Observable Operators

- Filter
- Map
- Buffer
- Merge
- Zip
- FlatMap

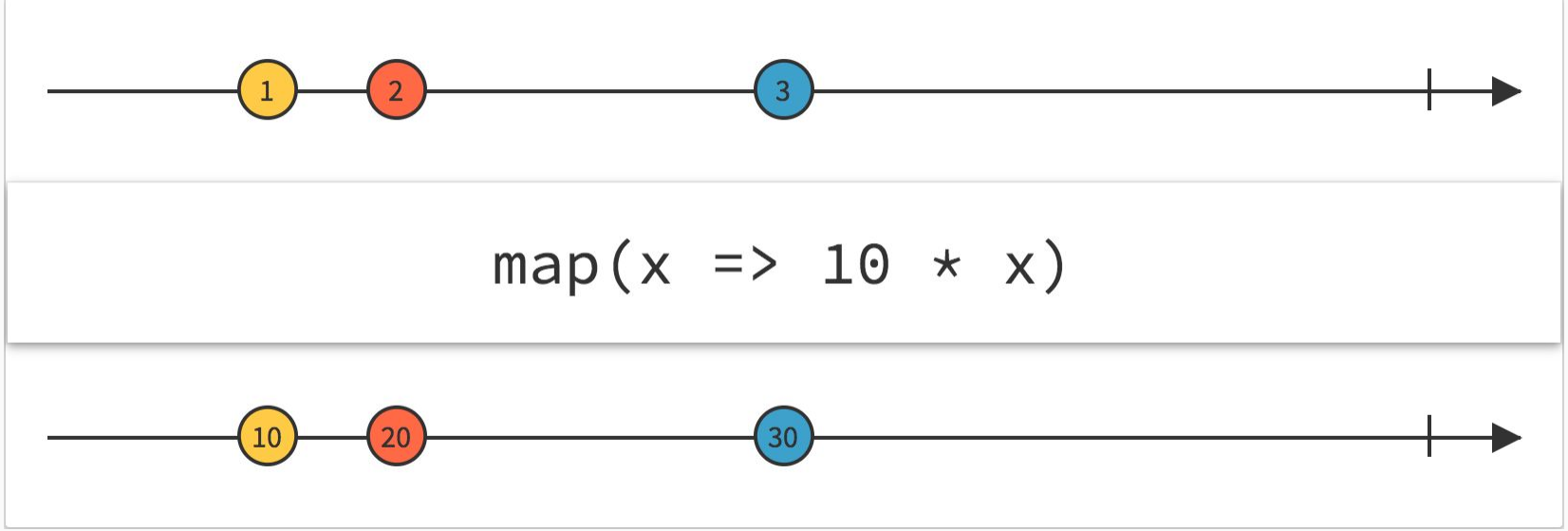
```
val stringObservable : Observable<String>

stringObservable.filter { it.length() > 2 }.map(String::toUpperCase)
```

Immutable! Every operation creates a new observable

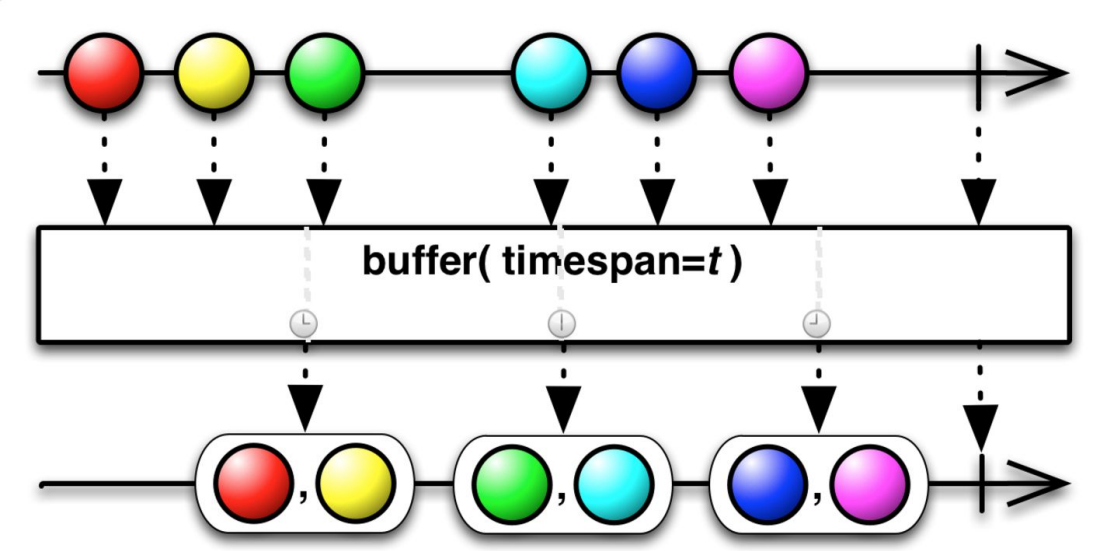
Filter



Map

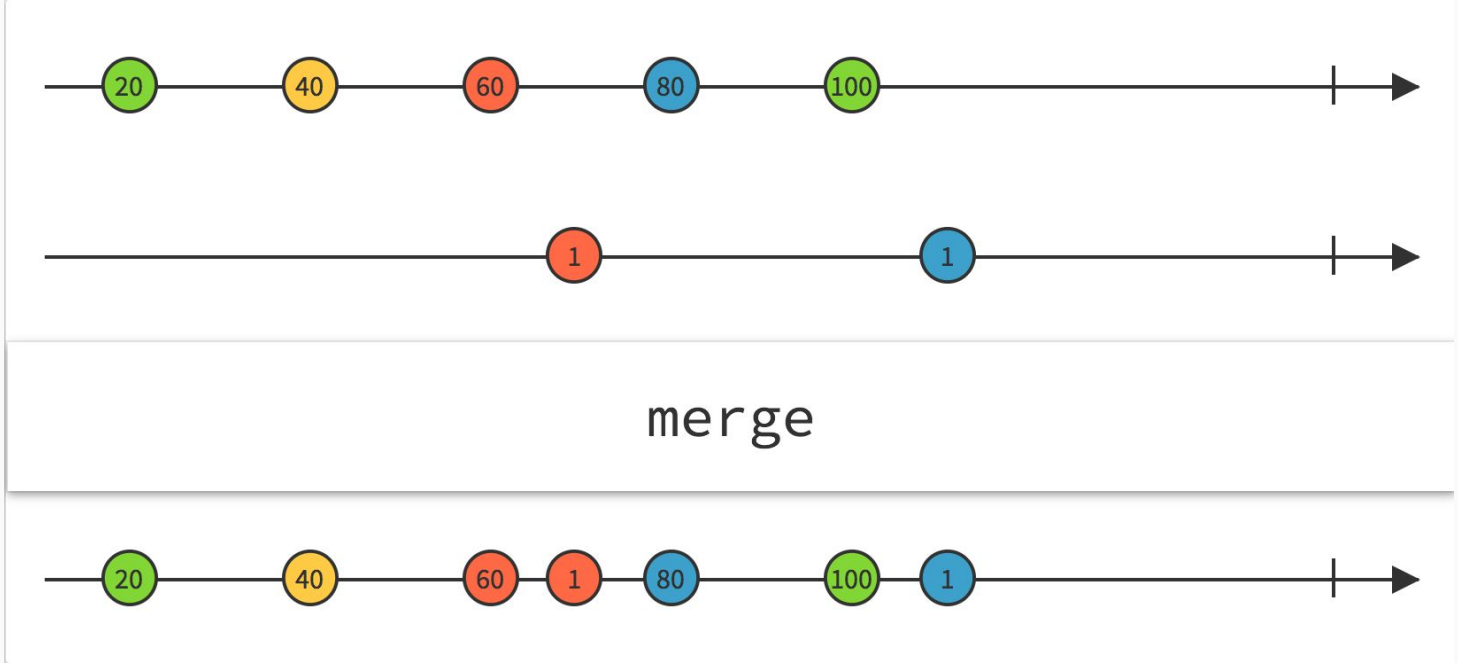


Buffer

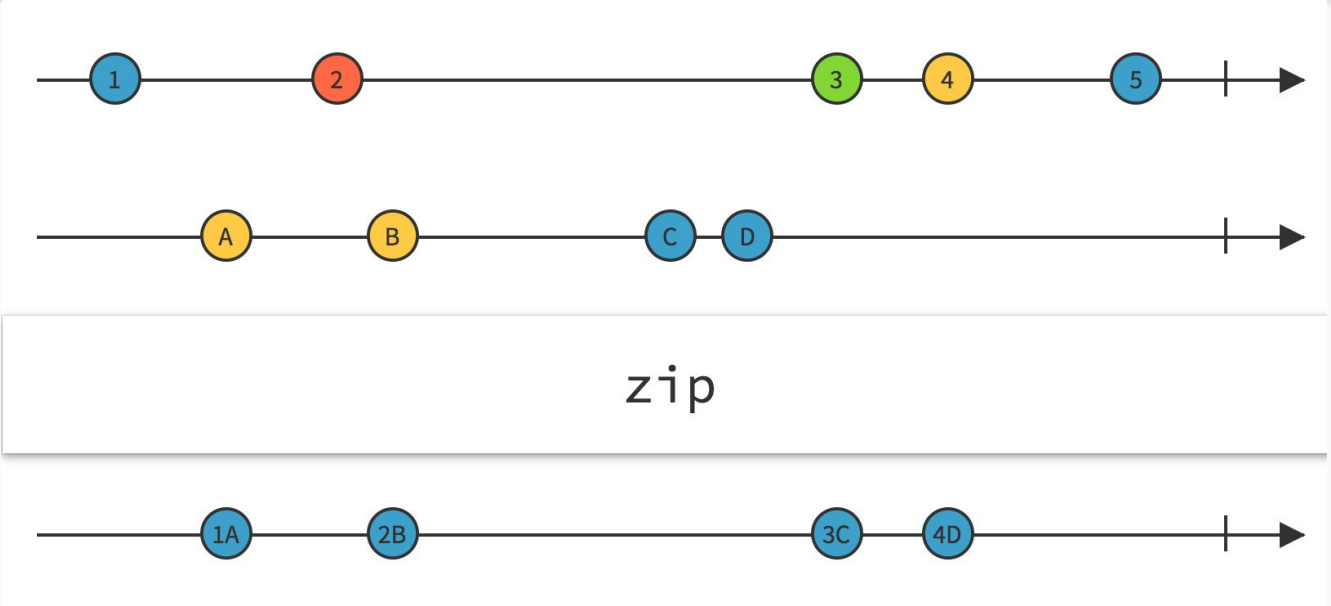


Observable<T> -> Observable<List<T>>

Merge

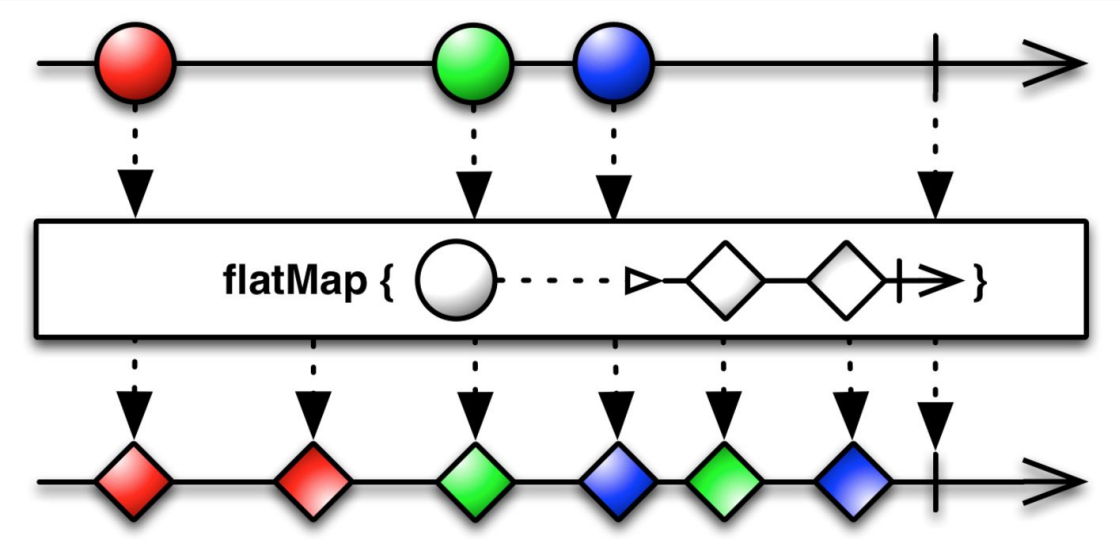


Zip



Only emits a new value when both source observables have emitted

FlatMap



- Useful when you want to perform another asynchronous operation for each value emitted by the source observable
- flatMap “flattens” the “steam-of-streams” or Metastream

Meta Streams?

Example

```
RxView.clicks(button)
    .flatMap {
        val api = Api()
        api.call().asObservable()   // Observable<ApiResponse>
    }
    .subscribe({
        view.show(it)               // it is ApiResponse
    }, {
        view.error()
    })
```

Example

```
RxView.clicks(button)
    .flatMap {
        val api = Api()
        api.call().asObservable()
    }
    .subscribeOn(Schedulers.IO())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe({
        view.show(it)
    }, {
        view.error()
    })
```

Rx as an Interface (RAAI?)

Do some work in the future

```
val handler = Handler()
handler.postDelayed({
    // Do the work
}, 2000)


handler.removeCallbacks()
```

```
val observer = Observable
     .timer(2, TimeUnit.Seconds)
     .observeOn(AndroidSchedulers.mainThread())
     .subscribe {
          // Do the work
     }

observer.dispose()
```

What if we want “Do the work” to happen in a background thread?

Do some work regularly

```
val handler = Handler()
val timer = Timer()
val timerTask = object : TimerTask {
    override fun run() {
        // Do the work
        handler.post {
            view.update()
        }
    }
}
// every 5s, with a delay of 1s
timer.scheduleAtFixedRate(timerTask, 1000, 5000)
timer.cancel()
```

```
val observer = Observable.interval(5, TimeUnit.Seconds)
        .delay(1, TimeUnit.Seconds)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe {
            // Do the work
            view.update()
        }
observer.dispose()
```

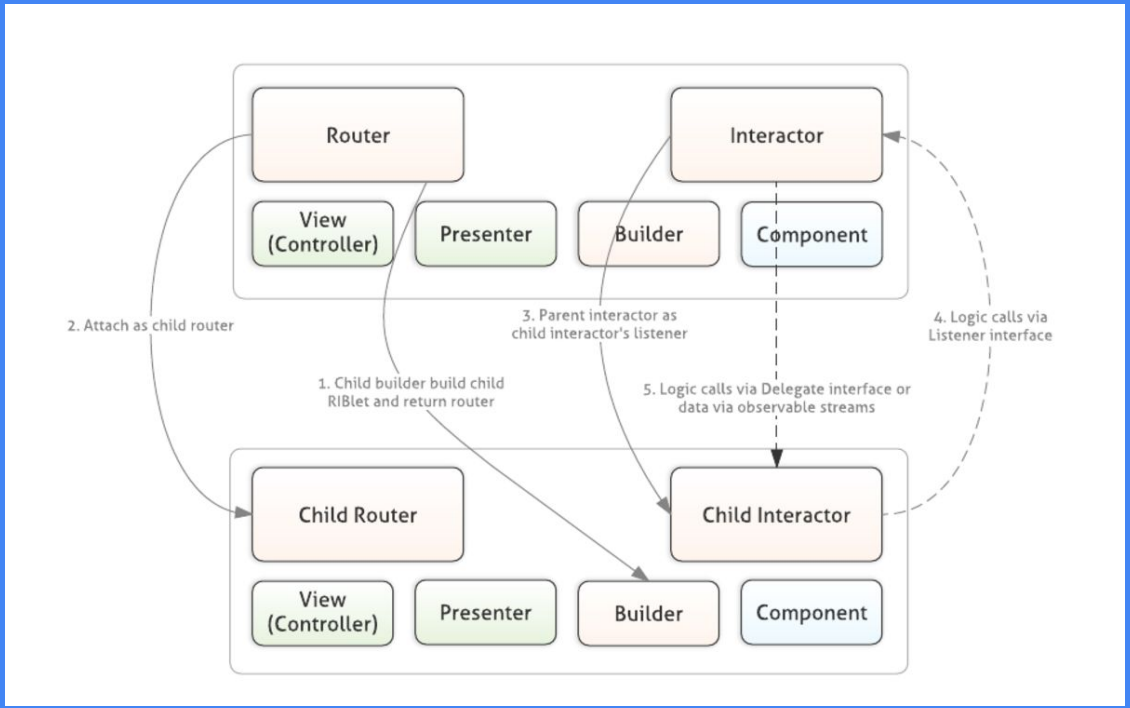
Timer operates on background thread, which is why the handler is needed

CompositeDisposable

```
val observer = Observable.just(Unit)
        .delay(2, TimeUnit.Seconds)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe {
                // Do the work
        }

val observer2 = Observable.interval(5, TimeUnit.Seconds)
        .delay(2, TimeUnit.Seconds)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe {
                // Do the work
                view.update()
        }

val observers = CompositeDisposable()
observers.add(observer)
observers.add(observer2)
```

```
override fun onDestroy() {
    // dispose of all observers
    observers.dispose()
}
```

Uber RIBs

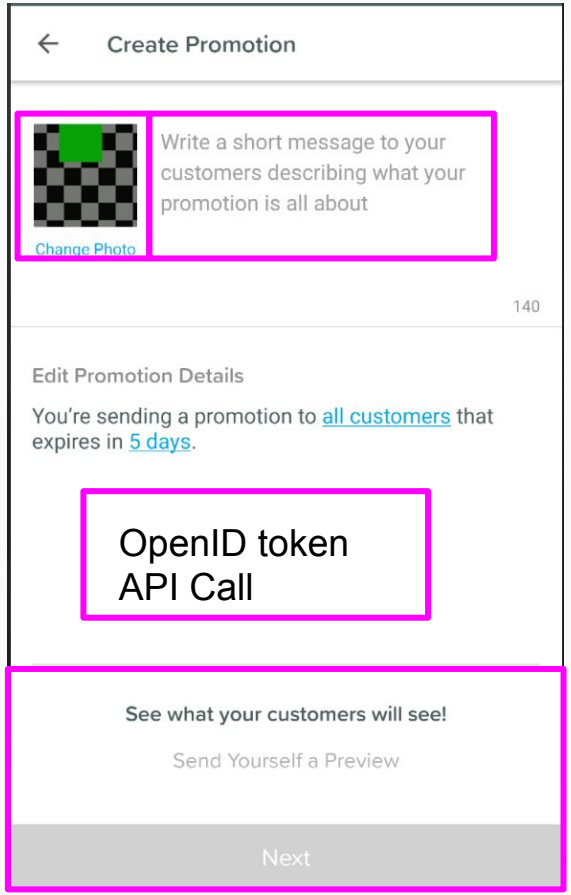

Router
Interactor
View (Controller)
Presenter
Builder
Component
2. Attach as child router
3. Parent interactor as child interactor's listener
4. Logic calls via Listener interface
1. Child builder build child RIBlet and return router
5. Logic calls via Delegate interface or data via observable streams
Child Router
Child Interactor
View (Controller)
Presenter
Builder
Component

RxJava in MMA

Presenters/UseCase

- We are using an MVP architecture
- Presenters responsible for coordinating business logic
- UseCases perform actions, such as calling an API or storing some value in the local db
  - Executing a UseCase always return an Observable
  - API calls can be acted on with standard Observable operations

UI Example

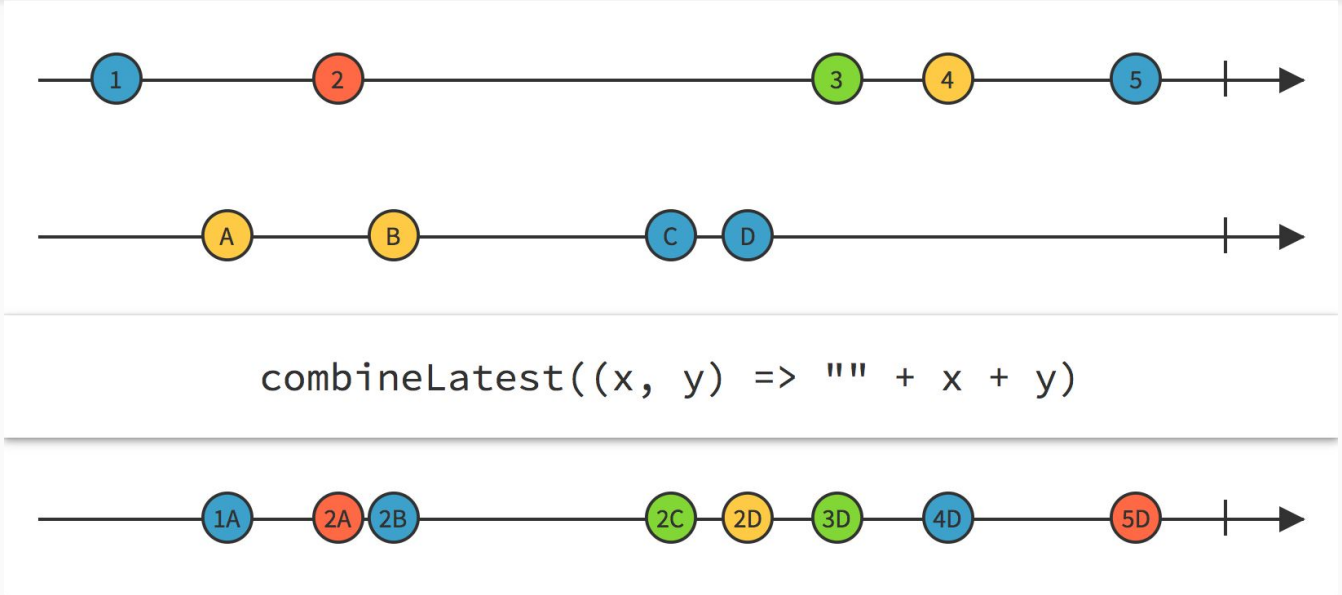


1. Photo file path
2. OpenID token API Call
3. EditText Updates
4. Button enabled state

We want to use the same OpenID token with consecutive uploads

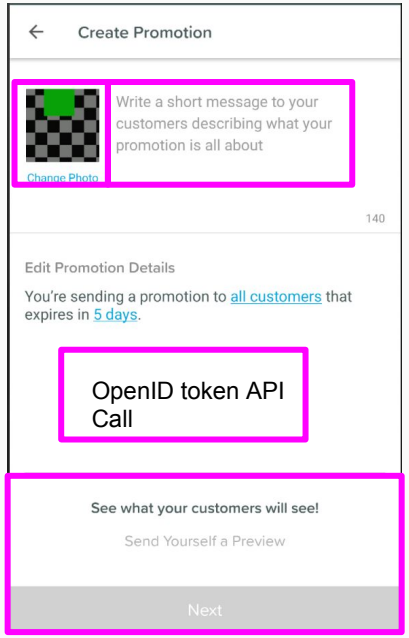
Must handle error cases: OpenID token expires, and photo upload fails

CombineLatest



Just like Zip, but emits if any of the source observables updates

Continued (Presenter)

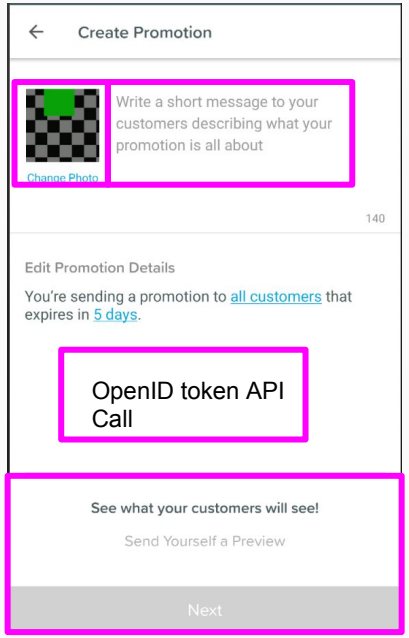



```
val openId: PublishSubject<OpenTokenIdResponse> = PublishSubject.create<OpenTokenIdResponse>()
val filePath: PublishSubject<String> = PublishSubject.create<String>()
```

```
// when new photo is selected
filePath.onNext(newFilePath)

// get new open Id token and update subject
fun getOpenIdToken() {
    OpenIdTokenUseCase().execute()
        .subscribe {
            openId.onNext(it)
        }
}
```

```
// photo upload
Observable.combineLatest(
    openId,
    filePath,
    { tokenResponse, photoFilePath ->
        AuthProvider(tokenResponse) to photoFilePath
        // Pair<AuthProvider, String>
    }
)
.doOnNext { view.uploadFinished(false) }
.flatMap { (auth, filePath) ->
    uploadPhoto(auth, filePath).onError {
        if (error is token expired) {
            getOpenIdToken()
        }
    }
}.subscribe {
    view.uploadFinished()
}
```

Continued (View)




```
val uploadFinished : PublishSubject<Boolean>
fun uploadFinished(fin: Boolean) {
    setLoadingState(!fin)
    uploadFinished.onNext(fin)
}

Observable.combineLatest(
    uploadFinished,
    promoMessage,   //RxTextView.textChanges(promotion_message)
    { fin, string ->
        string.length() > MIN_MESSAGE && fin
    }
)
.distinctUntilChanged()
.subscribe { enable ->
    enableButtons(enable)
}
```

Rx @ FiveStars

We Already Use It!

- Definitely some Rx in CTS/Mtab code
  - Websocket “library”
  - Api/Internet connection status
- Websocket Server has a little with the Pub-Sub system
- MMA using it extensively

Hystrix Anybody?

- Has already been mentioned and experimented by Devo
- Essentially a wrapper around Rx
- Not completely necessary, since Rx provides a timeout operator and error handling already
  - Hystrix provides circuit breaking, which Rx does not give you automatically
- Useful for Microservice communication
  - Call 3 services in parallel
  - Combine Observables from Hystrix commands with Zip
  - ???
  - Profit

Conclusion

- Rx is pretty nifty
- Concepts can be hard to grasp at first
  - Seems like overkill for small tasks
  - When requirements change, just need to apply different operators on your observables
- State == bad, Rx == good
- When doing Rx coding I usually start by drawing out my Observable streams, like the marble diagrams
  - May struggle to get it working exactly right, but once it works...

Questions?

Links

- http://reactivex.io/
- http://rxmarbles.com/
- https://github.com/ReactiveX/RxJava
- https://github.com/Netflix/Hystrix