Switch from RxJava to Kotlin Coroutines and Flow

Atul Kumar

@iamatul_k

in/iamatulk

Prerequisite:

Basic knowledge of RxJava event-based programming observers and subscribers

Section Overview:

- Overview of RxJava, RxKotlin, Kotlin Coroutine, and Flow
- Explanation of key concepts based on categories with examples to help for switching from RxJava to kotlin

  RxJava vs Kotlin

  - [CompA] Comparison based on Reactive Type
    - RxJava : - e.g Observable, Flowable etc
    - Kotlin : - suspend function, Flow
  - [CompB] Comparison based on Operators
    - RxJava : - e.g flatMap etc
    - Kotlin : - suspend function, Flow
  - [CompC] Comparison based on error handling
    - RxJava : - e.g onErrorResumeWith, onErrorResumeNext etc
    - Kotlin : - try/catch, catch builder block
  - [CompD] Comparison based on Backpressure handling
    - RxJava : - Back Pressure strategies(e.g drop, buffer etc)
    - Kotlin : - Back Pressure using flow buffer

RxJava (Rxkotlin), Kotlin coroutine, Flow

RxJava: is a Java library that enables Functional Reactive Programming

RxKotlin: is basically the same as RxJava, It just adds some syntactic sugar to make it more comfortable Idiomatic writing RxJava code in kotlin

Kotlin Coroutine: kotlin is a programing language and coroutine is a feature of kotlin, they are lightweight Threads that help to write simplified asynchronous code

Suspend function: Suspend function is a function that could be started, paused, and resume.
suspend functions are only allowed to be called from a coroutine or another suspend function
Suspend function asynchronously returns a single value

Flow: Flow returns multiple asynchronously computed values.

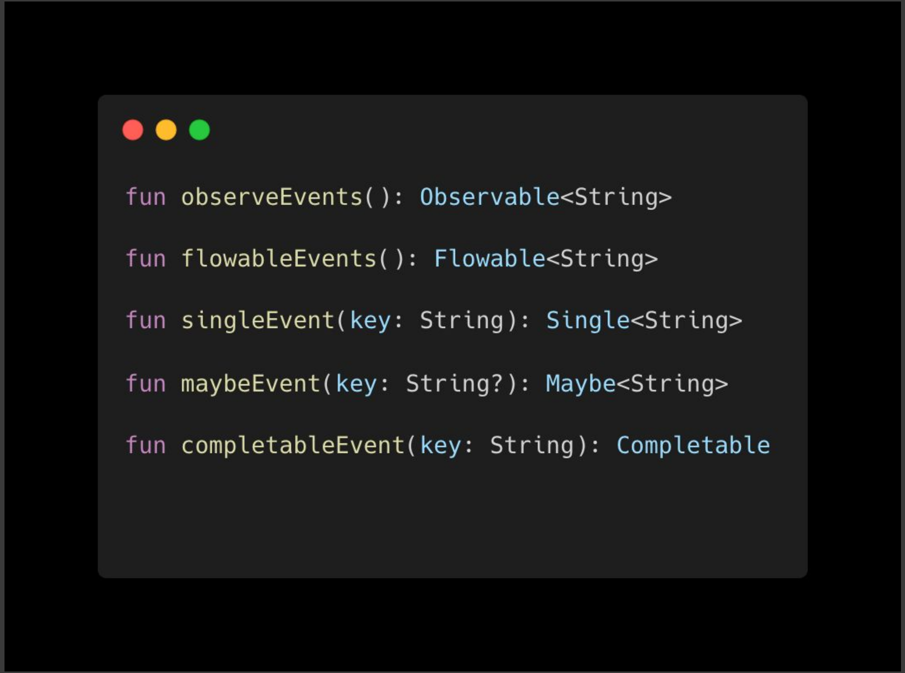
[CompA]

Comparison based on Reactive type

Up Next

[CompA] RxJava

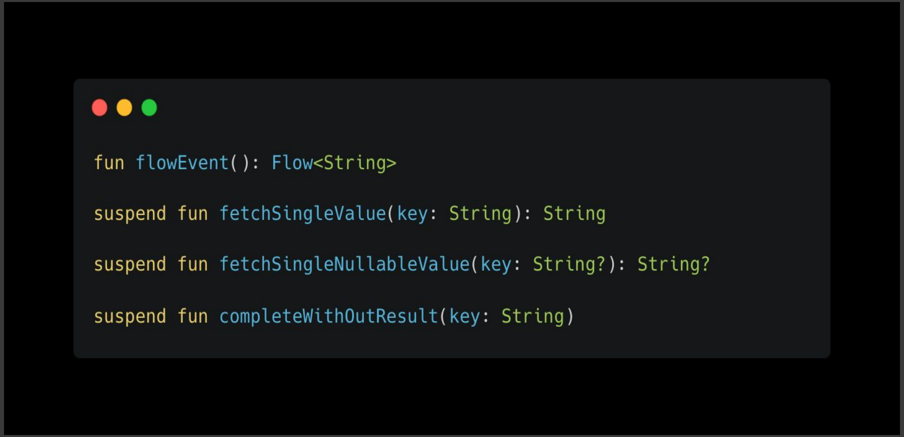
- Observable
- Flowable
- Single
- Maybe
- Completable



```
fun observeEvents(): Observable<String>

fun flowableEvents(): Flowable<String>

fun singleEvent(key: String): Single<String>

fun maybeEvent(key: String?): Maybe<String>

fun completableEvent(key: String): Completable
```

[CompA]Kotlin Coroutine and Flow

- Coroutine(Suspend function)
- Flow



```kotlin
fun flowEvent(): Flow<String>

suspend fun fetchSingleValue(key: String): String

suspend fun fetchSingleNullableValue(key: String?): String?

suspend fun completeWithOutResult(key: String)
```

[CompB]

Comparison based on Operators

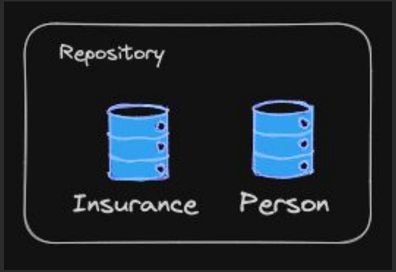
Up Next

[CompB] RxJava

- flatMap
  - Generate new stream for each event
  - flatMap might not preserves the order of output event
- ConcatMap
  - Same as flat map but preserves the order of item
  - concateMap waits for each observable to finish all the work until next one processed
- SwitchMap
  - Similar to flatMap, except that switchMap retains the result of only the latest observable

Example:

Insurance company and Tom is a support agent



Tom UseCase

Case1. Fetch single Insurance details

Case2. Fetch person details and then fetch insurance details

Case3. Fetch all person list and then fetch their corresponding insurance details

[CompB]RxJava Example

Case 1 :

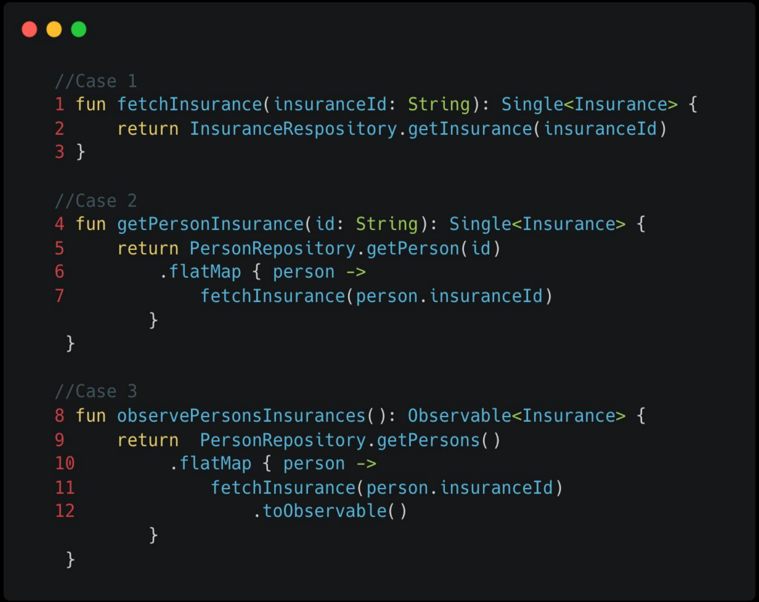
- Call fetchInsurance fun
- Return single insurance detail

Case 2:

- Call getPersonInsurance fun
- Fetch single person detail
- Fetch person insurance details async
- Return the single insurance detail

Case 3:

- Call observePersonsInsurance Fun
- Fetch all persons list
- Fetch each person insurance async
- Return insurance details observable



```kotlin
//Case 1
fun fetchInsurance(insuranceId: String): Single<Insurance> {
    return InsuranceRespository.getInsurance(insuranceId)
}

//Case 2
fun getPersonInsurance(id: String): Single<Insurance> {
    return PersonRepository.getPerson(id)
        .flatMap { person ->
            fetchInsurance(person.insuranceId)
        }
 }

//Case 3
fun observePersonsInsurances(): Observable<Insurance> {
    return  PersonRepository.getPersons()
         .flatMap { person ->
             fetchInsurance(person.insuranceId)
                 .toObservable()
        }
 }
```

[CompB]Kotlin coroutine, Flow

Case 1 :

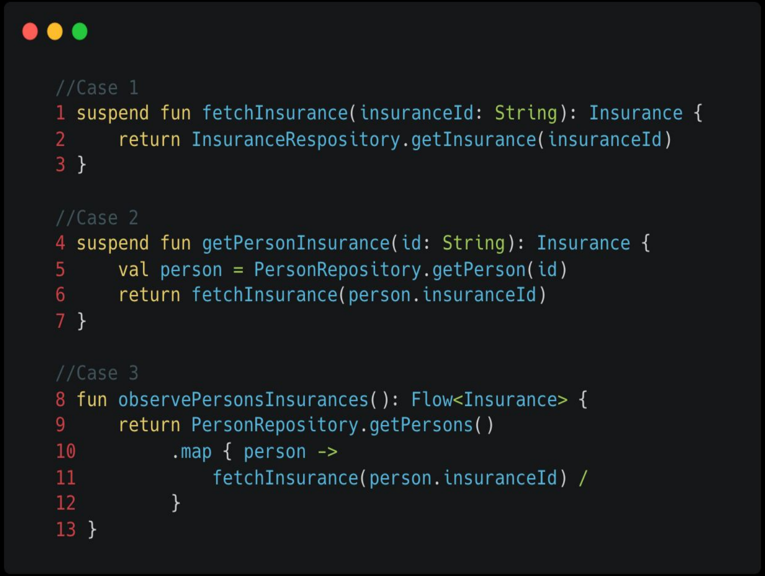
- Call fetchInsurance suspend fun
- Return insurance detail

Case 2:

- Call getPersonInsurance suspend fun
- Fetch insurance details

Case 3:

- Call observePersonsInsurance Fun That return a insurance flow
- Fetch all person list
- Map over the list and call fetchInsurance suspend function
- Return insurance details flow



```kotlin
//Case 1
suspend fun fetchInsurance(insuranceId: String): Insurance {
    return InsuranceRespository.getInsurance(insuranceId)
}

//Case 2
suspend fun getPersonInsurance(id: String): Insurance {
    val person = PersonRepository.getPerson(id)
    return fetchInsurance(person.insuranceId)
}

//Case 3
fun observePersonsInsurances(): Flow<Insurance> {
    return PersonRepository.getPersons()
         .map { person ->
             fetchInsurance(person.insuranceId) /
         }
}
```

[CompC]

Comparison based on Error Handling

Up Next

[CompC]RxJava

- onErrorResumeWith()
- onErrorResumeNext()
- onErrorComplete()
- onErrorReturn()
- onErrorReturnItem()

[CompC]Kotlin coroutine and Flow

- Error handling in kotlin coroutine
    - Try/catch
    - CoroutineExceptionHandler

- Error handling in Flow
    - Try/catch
    - Flow catch builder block

[CompD]

Comparison based on Back Pressure Handling

Up Next

[CompD]RxJava

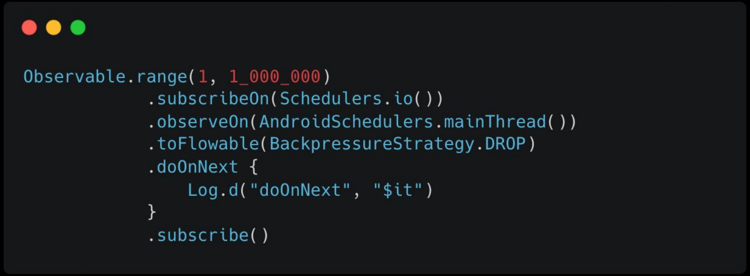
Drop: Discards the unrequested item if It exceeds the buffer size

Buffer: Buffers all the items from the producer, watch for outofMemory

Latest: Keeps only the most recent item

Error: Throws a MissingBackpressureException in case of over emission

Missing: No, strategy, it would throw a MissingBackpressureException sooner or later somewhere on the downstream



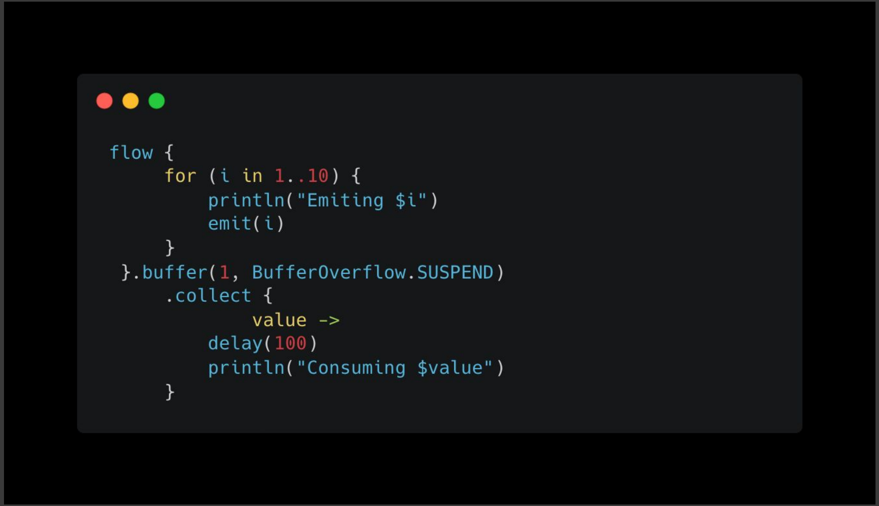
```
Observable.range(1, 1_000_000)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .toFlowable(BackpressureStrategy.DROP)
            .doOnNext {
               Log.d("doOnNext", "$it")
            }
            .subscribe()
```

[CompD]Kotlin coroutine, Flow

SUSPEND: Upstream that is sending or is emitting a value is suspended while the buffer is full.

DROP_OLDEST: Drop the oldest value in the buffer on overflow, add the new value to the buffer, do not suspend.

DROP_LATEST: Drop the latest value that is being added to the buffer right now on buffer overflow (so that buffer contents stay the same), do not suspend.



```
flow {
        for (i in 1..10) {
            println("Emiting $i")
            emit(i)
        }
 }.buffer(1, BufferOverflow.SUSPEND)
        .collect {
                value ->
            delay(100)
            println("Consuming $value")
        }
```

Thank you!

droidcon academy