Asynchrony in Kotlin

The choice of using Rx or using Coroutines

Anton Spaans
@streetsofboston

Accenture Interactive

Agenda

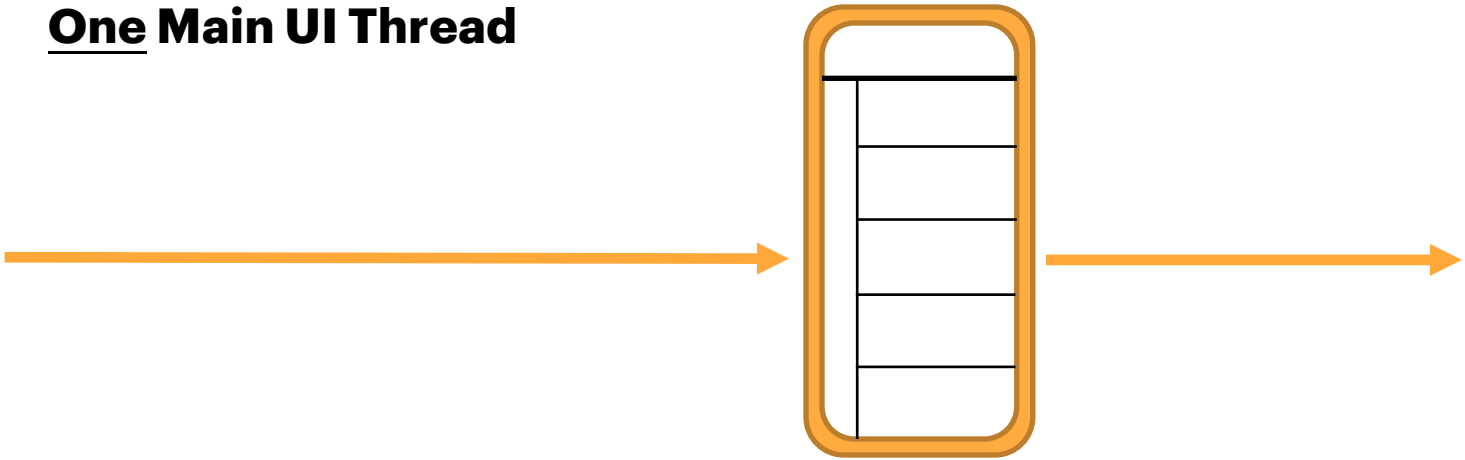
1. Asynchrony
2. No Threads
3. Rx Refresher
4. Coroutines Refresher
5. Coroutines vs Rx
6. Recap

01

Asynchrony

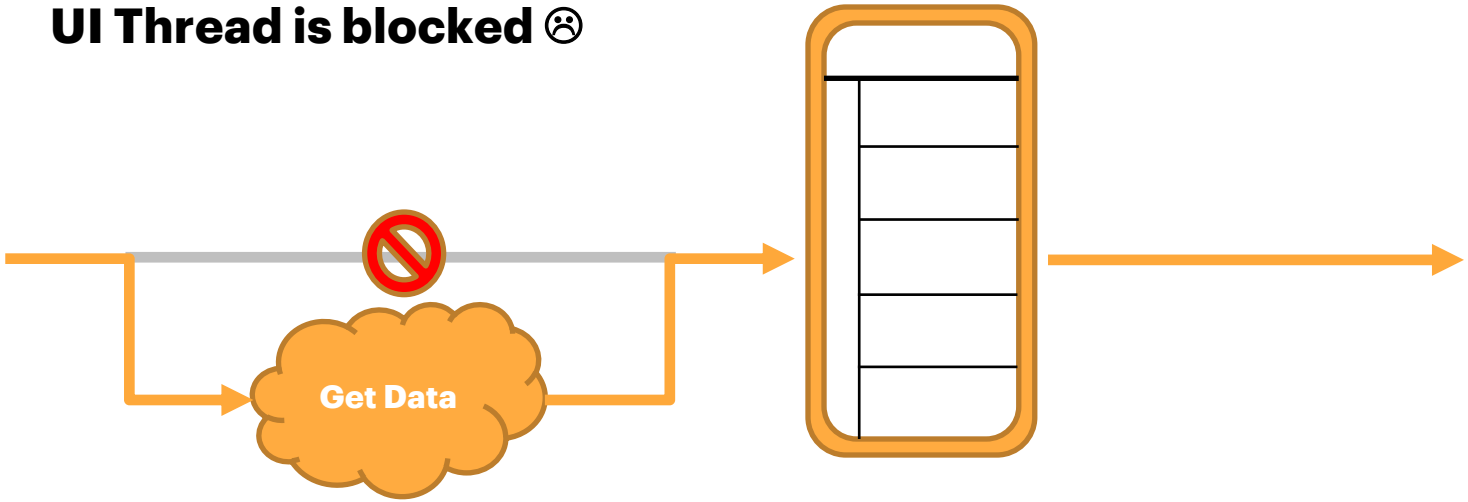
User Interface

One Main UI Thread



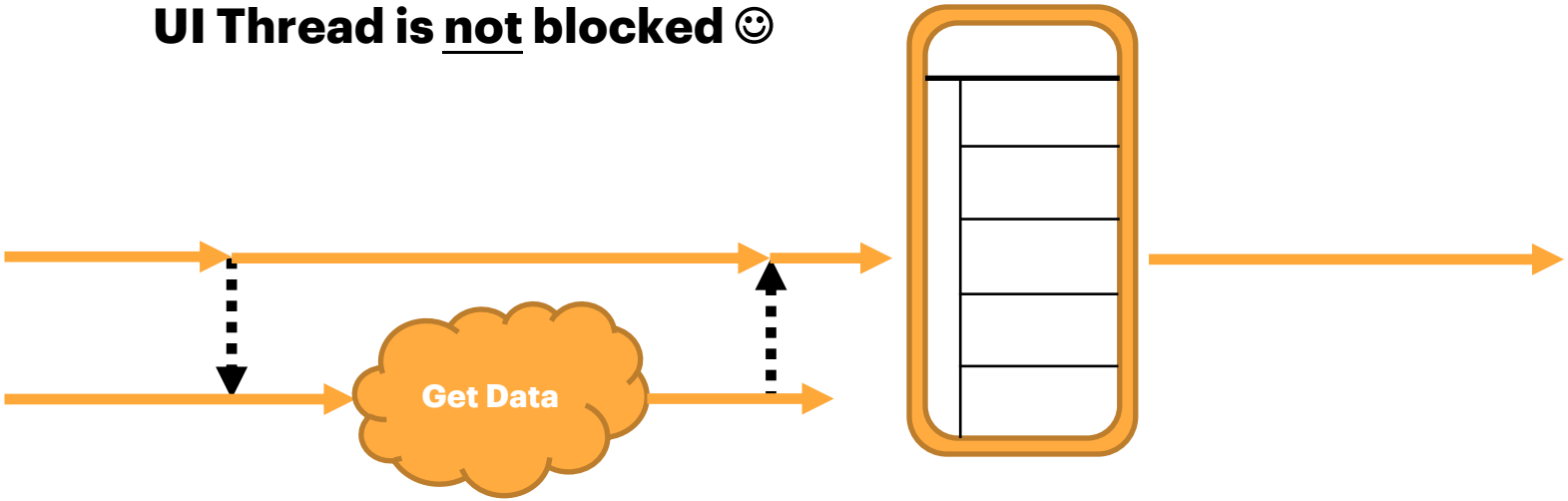
User Interface

UI Thread is blocked ☹



User Interface

UI Thread is not blocked ☺



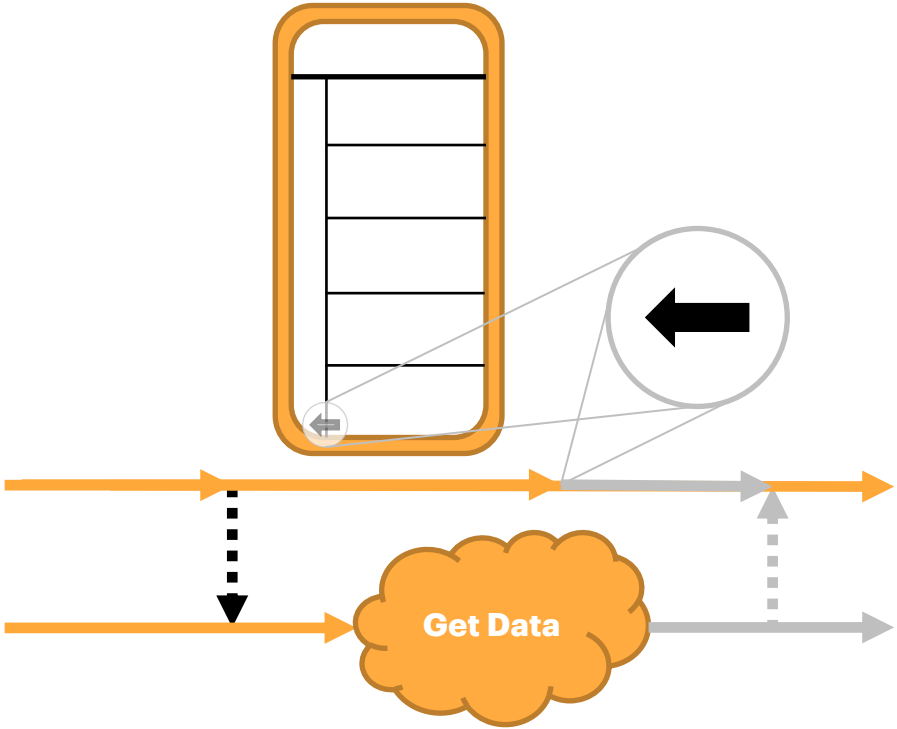
02

No Threads

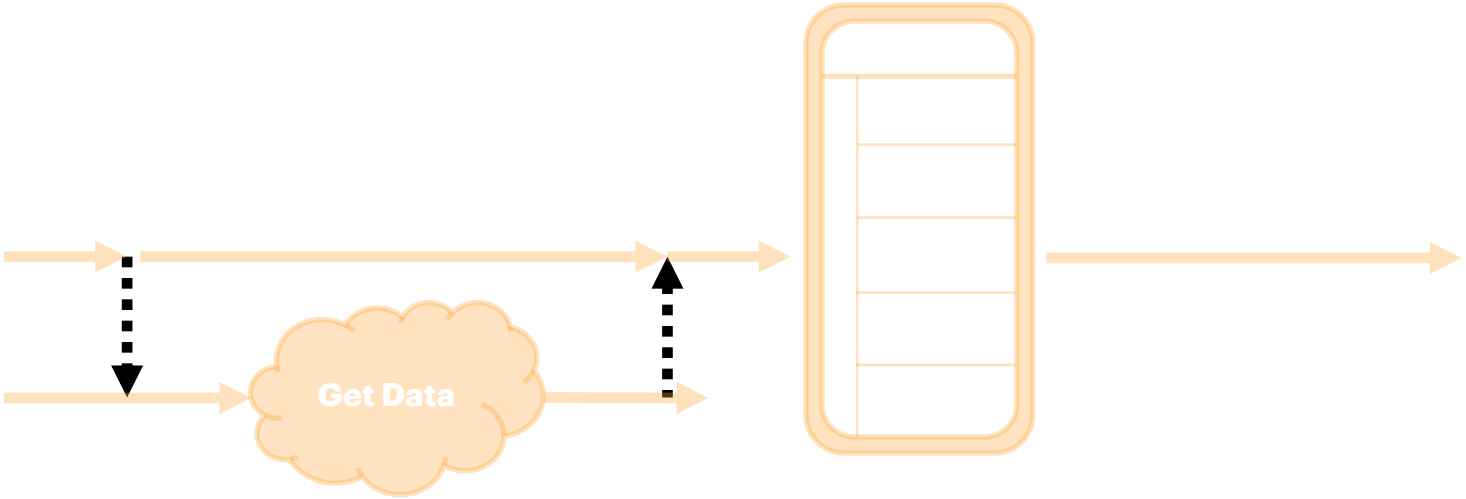
Callback Hell ™

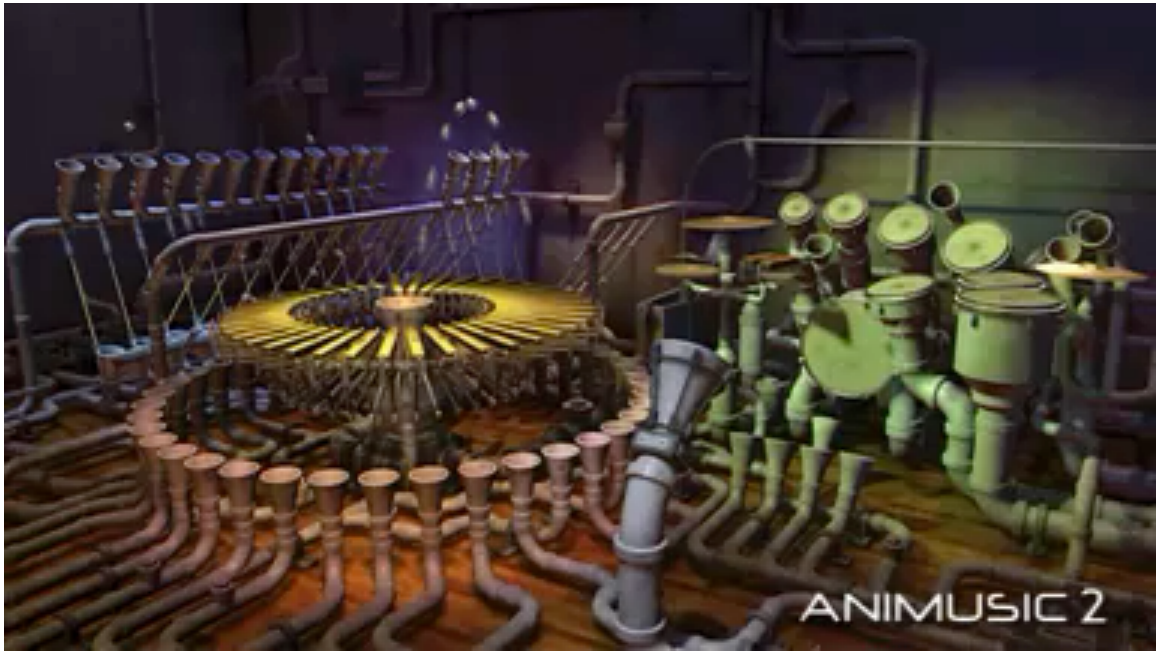
```
fun getData(input: Int, callback: (Try<String>) -> Unit) {
    getData1(input) {
        it.fold({ callback(Failure(it)) }) {
            getData2(it.toInt()) {
                it.fold({ callback(Failure(it)) }) {
                    getData3(it + 5) {
                        it.fold({ callback(Failure(it)) }) {
                            getData4(it.toInt()) {
                                it.fold({ callback(Failure(it)) }) {
                                    getData5("$it$it", callback)
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Cancellation



Move data between Threads




ANIMUSIC 2

03

Rx Refresher

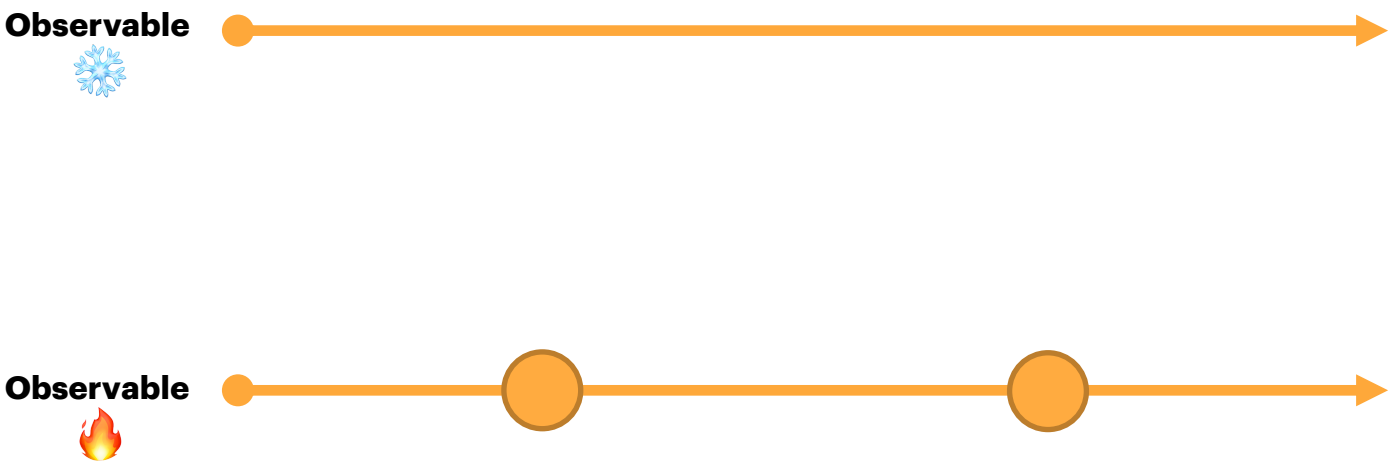
What is Rx?

An API for asynchronous programming with *observable streams*

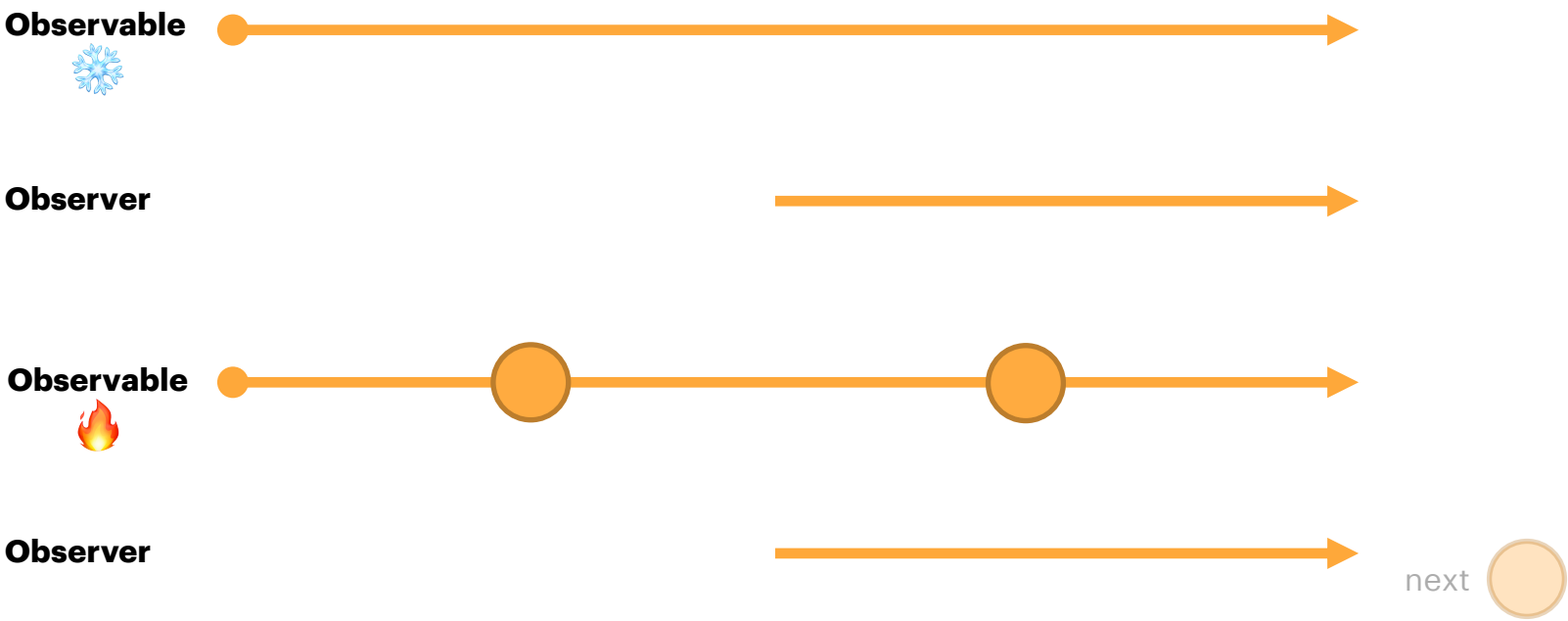
- **Collections**
  Contain objects that other code must **pull** from them.

- **Rx Observables**
  Observables stream or emit objects **over time**, **pushes** them to other code.

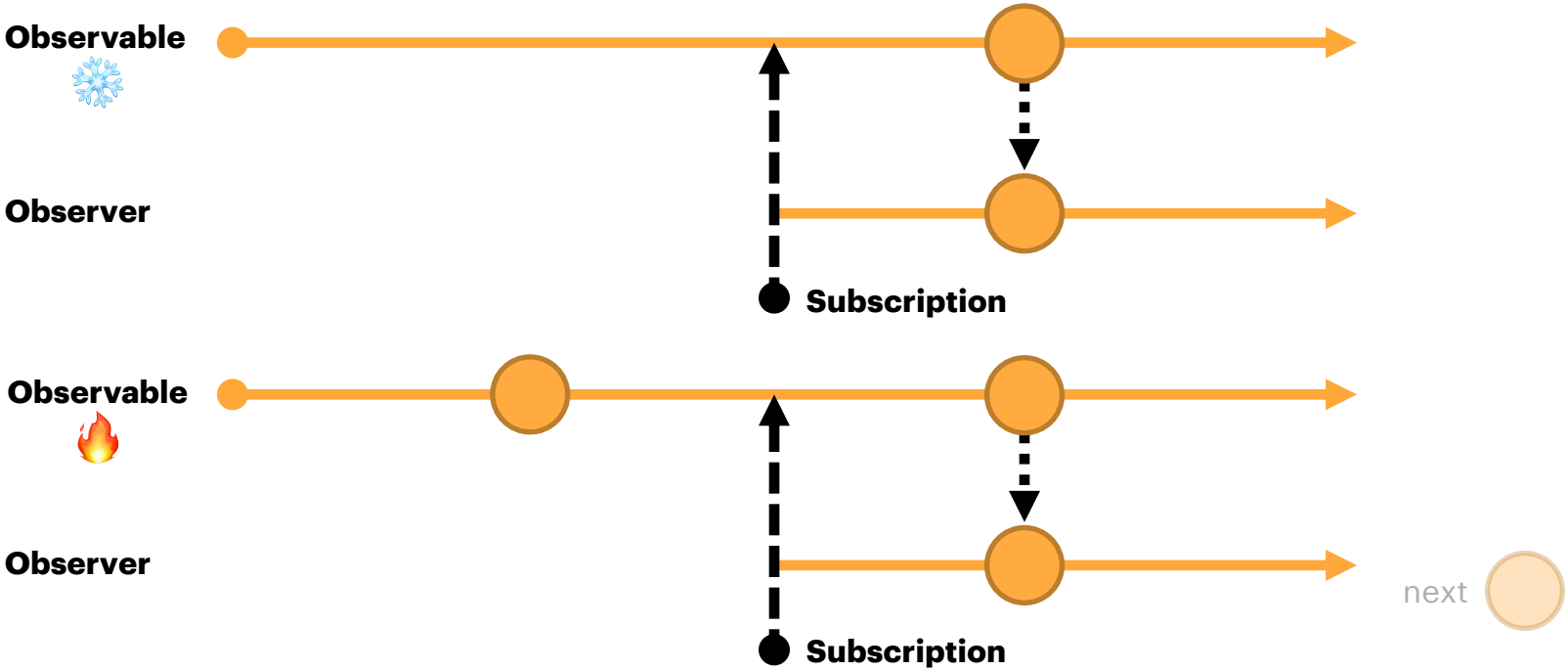
Observable



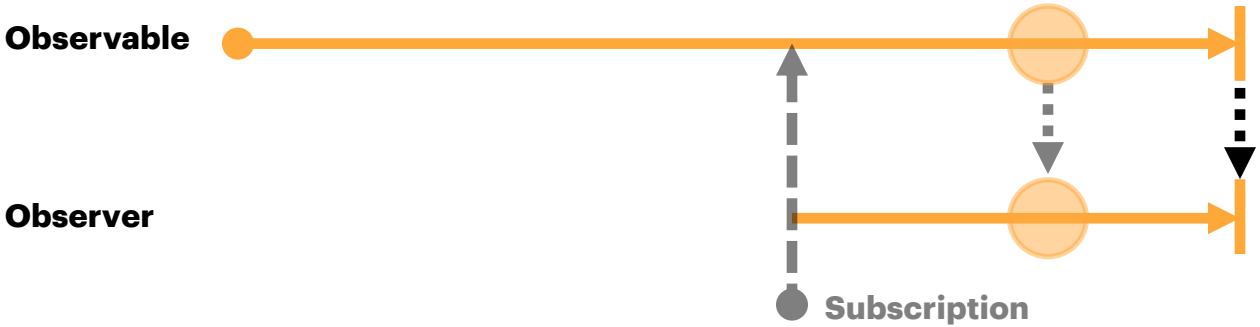
Observer



Subscription

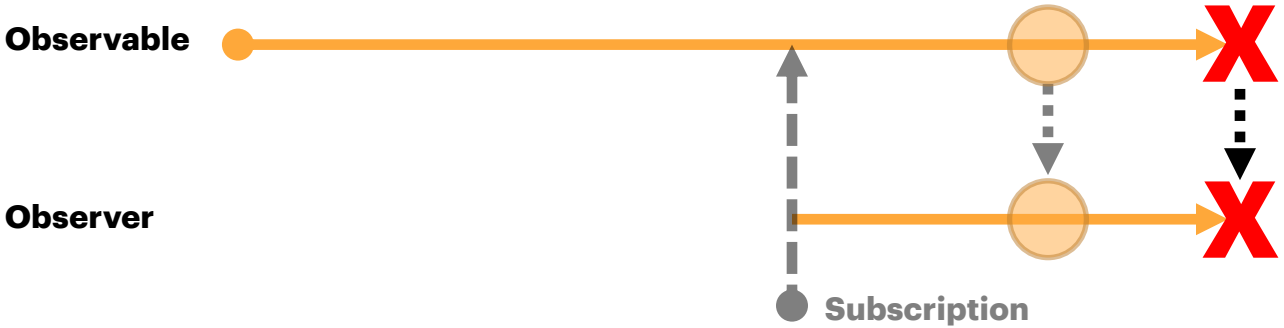


Termination: Completion



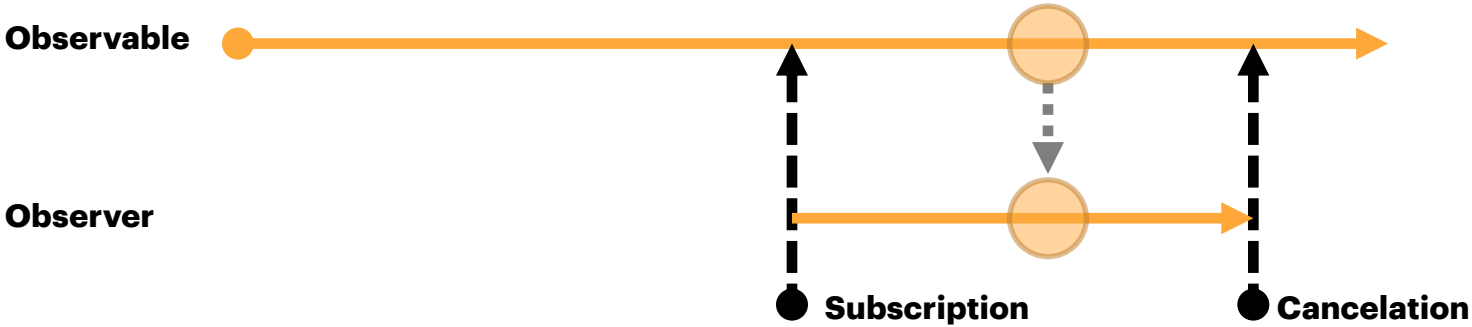
complete

Termination: Error



error

Cancelation



Observable

Observable ❄️

```
val coldObservable: Flowable<String> = Flowable.fromCallable {
    getData(4)
}
```

Observable 🔥

```
val hotObservable: FlowableProcessor<Pair<Int,Int>> = ...
fun onTouchEvent(event: MotionEvent) {
    hotObservable.onNext(event.x.toInt() to event.y.toInt())
}
```

Observer

Observable ❄️

```
val coldObservable: Flowable<String> = Flowable.fromCallable {
    getData(4)
}
```

Observer

```
...
val observerUI: (String) -> Unit = { textView.text = it }
```

Observable 🔥

```
...
val hotObservable: FlowableProcessor<Pair<Int,Int>> = ...
fun onTouchEvent(event: MotionEvent) {
    hotObservable.onNext(event.x.toInt() to event.y.toInt())
}
```

Observer

```
...
val observer: (String) -> Unit = { println(it) }
```

Subscription

Observable

```
val coldObservable: Flowable<String> = Flowable.fromCallable {
    getData(4)
}
```

Observer

```
val observerUI: (String) -> Unit = { textView.text = it  }
...

...

val subscription = coldObservable
    .subscribe(observerUI)
...
// Cancel subscription
subscription.dispose()
```

Schedulers

- **Determine on which Thread(s) stuff runs**
  Stuff here means **tasks**, usually in the form of a function or lambda.

- **Have a pool of Threads**
  Each task is scheduled to run on a Thread in the Scheduler's pool.
  - **Main Scheduler**
    One thread, the **Main UI** Thread.
  - **IO Scheduler**
    Possibly **unbounded** number of Threads on which **blocking** code may run.
  - **Computation Scheduler**
    Limited number of Threads, between 2 and the number of CPU-cores.

Scheduler for Subscribing

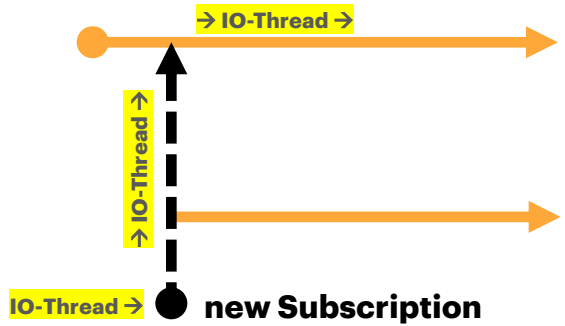
Observable

```
val coldObservable: Flowable<String> = Flowable.fromCallable {
    getData(4)
}
```

Observer

```
...
val observerUI: (String) -> Unit = { textView.text = it  }
...
```

```
val subscription = coldObservable
    .subscribeOn(Schedulers.io())

    .subscribe(observerUI)
```



Scheduler for Observing

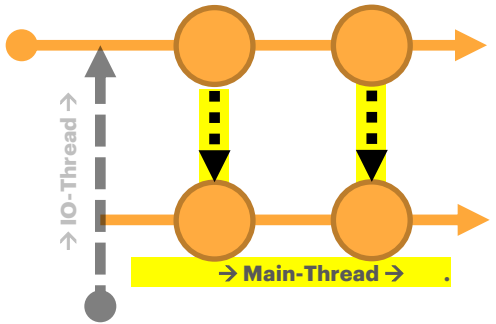
Observable

```
val coldObservable: Flowable<String> = Flowable.fromCallable {
    getData(4)
}
```

Observer

```
val observerUI: (String) -> Unit = { textView.text = it }
...
```

```
val subscription = coldObservable
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.main())
    .subscribe(observerUI)
```



Backpressure

A Flowable's backpressure determines its buffer size and behavior

- **Buffer**
  Fixed buffer size and error is emitted when buffer is full.
- **Latest**
  Buffer size of 1 and *previous* value is dropped.
- **Drop**
  Buffer size of 1 and *last* value is dropped.
- **Error**
  No buffer and *error* is emitted if observer is too slow.
- **Missing**
  No buffer and *downstream* must deal with backpressure.

Single, Maybe and Completable

- **Single** (1)
  - An Observable that emits one object and then completes
  - For single-shot results, such as a network-request or a single system callback
- **Maybe** (1 or 0)
  - An Observable that emits one object and then completes or that completes without emitting an object
  - For optional results. Often used for emitting values from a cache
- **Completable** (0)
  - An Observable that completes without emitting an object
  - For chaining operations sequentially, chaining side-effects

04

Coroutines Refresher

What are 'Coroutines'?

Coroutines allow execution to be suspended and then resumed

- **Blocking**
  Calling Thread stops at blocking point
  Thread does nothing but waits; it is idle
  Thread gets interrupted and continues after the blocking point
- **Suspending** 
  Calling Thread stops at suspension point
  Thread runs other tasks; it is never idle
  Thread resumes after the suspension point at some point in time

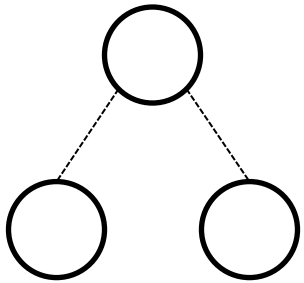
What are Suspend Functions?

```
suspend fun plus5(input: Int): Int {
    delay(100)
    return input + 5
}

suspend fun asString(input: Int): String {
    return input.toString()
}

suspend fun repeatString(input: String, times: Int): String {
    return input.repeat(times)
}

suspend fun calculate(): String {

    val value = 3
    val result = plus5(value)

    val updatedResult = result * 10
    val finalResult = asString(updatedResult) + "!"

    return repeatString(finalResult, 7)
}
```

What are Coroutines?

Coroutine is suspendable code that runs within a *Job*

- Only Coroutines can call **suspend** functions
- When Coroutine ends, Job ends
- If Job is canceled, its Coroutine is canceled
- Coroutine can have child-Coroutines, i.e. Job can have child-Jobs



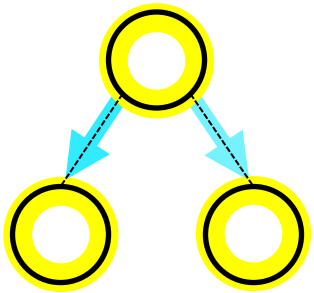
What are CoroutineContexts?

- **Coroutine needs a CoroutineContext**
  - For cancelation and synchronization (Job)
  - For dispatching tasks onto the correct Threads (Dispatcher)

What are CoroutineScopes?

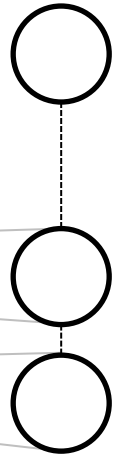
CoroutineContext is inherited through CoroutineScope

```
interface CoroutineScope {
    val coroutineContext: CoroutineContext
}
```



Structured Concurrency

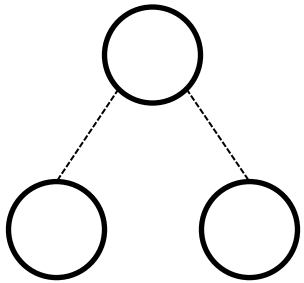
```
interface CoroutineScope {
    val coroutineContext: CoroutineContext
}

val parentJob = Job()

val scope = object: CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = parentJob + Dispatchers.IO
}

// "childJob"'s coroutine inherits "parentJob" as its parent.
// and it inherits Dispatchers.IO as its dispatcher.
val childJob: Job = scope.launch {
    // Runs on IO threads
    val times = plus5(3)
    // "grandChildJob"'s coroutine inherits "childJob" as its parent.
    // and it inherits Dispatchers.IO as its dispatcher.
    val grandChildJob: Deferred<String> = this.async {
        // Runs on IO threads
        repeatString("Hello ", times)
    }
    println(grandChildJob.await())
}
```



Structured Concurrency: Job

To have Structured Concurrency, we would like a Job to...

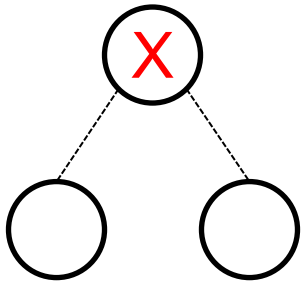
- have child-Jobs, such as ones returned by **launch** or **async** calls
-
-



Structured Concurrency: Job

To have Structured Concurrency, we would like a Job to...

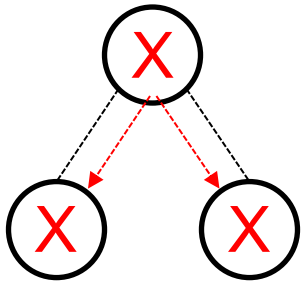
- have child-Jobs, such as ones returned by **launch** or **async** calls
- cancel all its child-Jobs, when it's cancelled
-



Structured Concurrency: Job

To have Structured Concurrency, we would like a Job to...

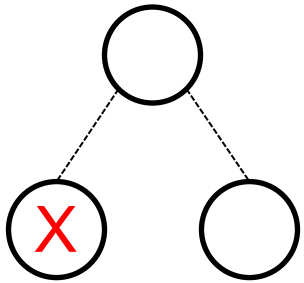
- have child-Jobs, such as ones returned by **launch** or **async** calls
- cancel all its child-Jobs, when it's cancelled
-



Structured Concurrency: Job

To have Structured Concurrency, we would like a Job to...

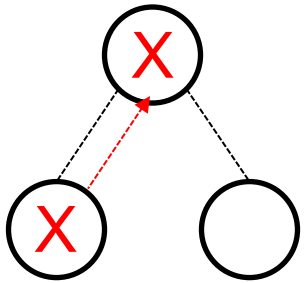
- have child-Jobs, such as ones returned by **launch** or **async** calls
- cancel all its child-Jobs, when it's cancelled
- cancel, when one of its child-Jobs is cancelled



Structured Concurrency: Job

To have Structured Concurrency, we would like a Job to...

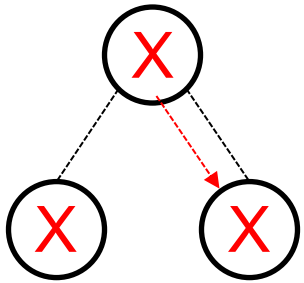
- have child-Jobs, such as ones returned by **launch** or **async** calls
- cancel all its child-Jobs, when it's cancelled
- **cancel**, when one of its child-Jobs is cancelled



Structured Concurrency: Job

To have Structured Concurrency, we would like a Job to...

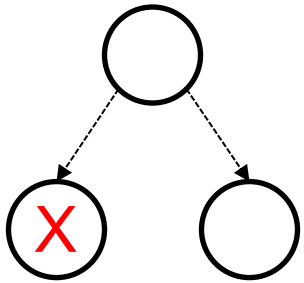
- have child-Jobs, such as ones returned by **launch** or **async** calls
- cancel all its child-Jobs, when it's cancelled
- **cancel**, when one of its child-Jobs is cancelled



Structured Concurrency: SupervisorJob

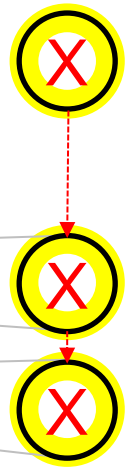
To have Structured Concurrency, we would like a **SupervisorJob** to...

- have child-Jobs, such as ones returned by **launch** or **async** calls
- cancel all its child-Jobs, when it's cancelled
- **keep running**, even when one of its child-Jobs is cancelled



Structured Concurrency

```
interface CoroutineScope {
    val coroutineContext: CoroutineContext
}

val parentJob = Job()

val scope = object: CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = parentJob + Dispatchers.IO
}

// "childJob"'s coroutine inherits "parentJob" as its parent.
// and it inherits Dispatchers.IO as its dispatcher.
val childJob: Job = scope.launch {
    // Runs on IO threads
    val times = plus5(3)
    // "grandChildJob"'s coroutine inherits "childJob" as its parent.
    // and it inherits Dispatchers.IO as its dispatcher.
    val grandChildJob: Deferred<String> = this.async {
        // Runs on IO threads
        repeatString("Hello ", times)
    }
    println(grandChildJob.await())
}

...
// Cancels all (grand)children and cancels itself.
scope.cancel()
```



Coroutine Builders

- **Regular functions cannot call suspend functions**
- **Coroutines can call suspend functions**
- **Regular functions can build and launch Coroutines**
  - CoroutineScope.**launch** 🔥
    Builds *and launches* a Coroutine and returns its Job
    For fire-and-forget
  - CoroutineScope.**async** 🔥
    Builds *and launches* a Coroutine and returns its Job as a Deferred<T>
    For getting a result asynchronously
    Deferred<T>.await() can be called multiple times to share result
  - **runBlocking**
    Runs a Coroutine and *blocks* until it has finished
    For getting a result synchronously

Lazy-Coroutine Builders

- CoroutineScope.**launch(start = LAZY)** ❄️
  Builds a Coroutine and returns its Job
  but only launches it *after* calling ***start*** or ***join*** on the Job
- CoroutineScope.**async(start = LAZY)** ❄️
  Builds a Coroutine and returns its Job as a Deferred<T>
  but only launches it *after* calling ***start***, ***join*** or ***await*** on the Deferred

'Cold' Suspend Functions

A function returning a suspend lambda

🔥

```
suspend fun ImDbCo.getFullName(actorId: String) : String {
    val actor = getActor(actorId)
    return actor.fullName
}
...
val fullName1 = getFullName("actor78hinb")
...
val fullName2 = getFullName("actor78hinb")
```

❄️

```
fun ImDbCo.getFullName(actorId: String) : suspend () -> String {
    return suspend {
        val actor = getActor(actorId)
        actor.fullName
    }
}
...
val coldFullName = getFullName("actor78hinb")
...
val fullName1 = coldFullName()
...
val fullName2 = coldFullName()
```

Conventions

When writing suspendable functions that

either **suspend for a while** and resume with a result, then write

```
suspend fun getRemoteData(…): SomeResult
```

or **return immediately** with a possible Deferred result, then write

```
fun CoroutineScope.remoteData(…): Deferred<SomeResult>
```

but *never* write this

```
suspend fun CoroutineScope.remoteData(…): …
```

Conventions

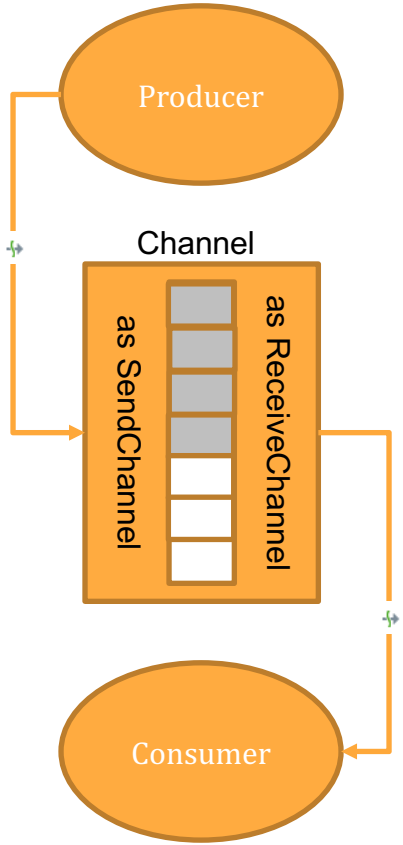
When you need to call *launch* or *async* in a suspend fun write this

```
suspend fun getImgBrightness(…) : Float = coroutineScope {
    val histogram = async { ... }
    ... takes a while ...
    parse(range, histogram.await())
}
```

Suspends for a while and resumes with a result, but allows for parallel processing of other data.

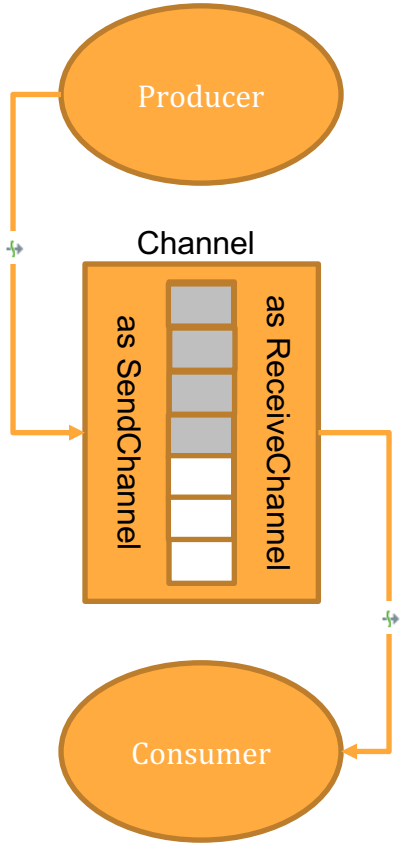
Channels

- **SendChannel**
  - For sending data through a channel
  - A call to *send* suspends as long as the buffer is full
- **ReceiveChannel**
  - For receiving data from a channel
  - A call to *receive* suspends as long as the buffer is empty
  - Is not reactive, i.e. consumers must call *receive* themselves



Channels

- **Channel**
  - Is both a *SendChannel* and a *ReceiveChannel*
  - Has a buffer
  - A call to *cancel* closes the entire *Channel* immediately
  - A call to *close* closes the *SendChannel* immediately
  - A call to *close* closes the *ReceiveChannel* after there are no more items to receive



Backpressure

A Channel's backpressure determines its buffer size and behavior

- **Rendezvous**
  - Buffer size of 0
  - A call to *send* will suspend and resume only when a *receive* call has been issued by the consumer
  - This is the default backpressure of a Channel

Backpressure

A Channel's backpressure determines its buffer size and behavior

- Rendezvous
- **Conflated**
  - Buffer size of 1
  - A call to *send* never actually suspends
  - If buffer is full, sending a value replaces the current one

Backpressure

A Channel's backpressure determines its buffer size and behavior

- Rendezvous
- Conflated
- **Unlimited**
  - Buffer can grow without limits
  - A call to *send* never actually suspends

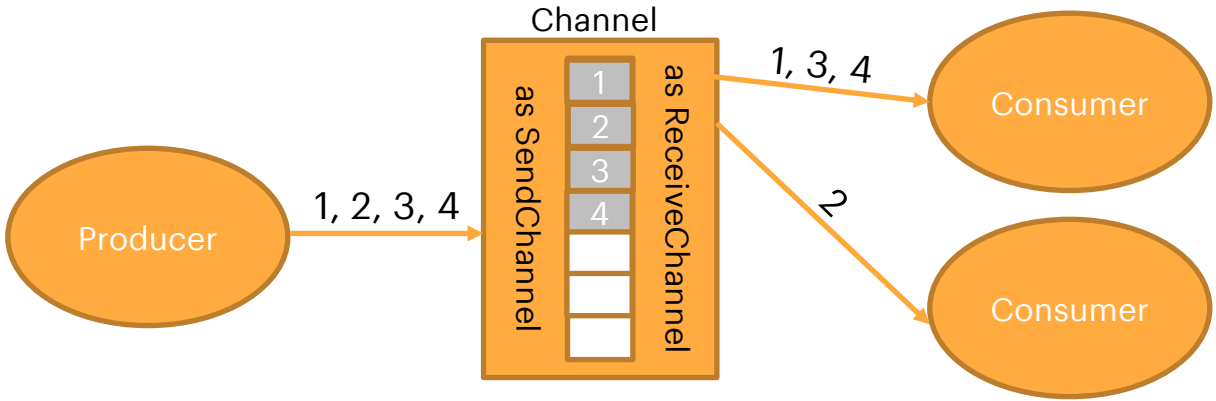
Backpressure

A Channel's backpressure determines its buffer size and behavior

- Rendezvous
- Conflated
- Unlimited
- **Some Fixed Size**
  - Buffer size is fixed
  - A call to *send* will suspend if the buffer is full and resume when a receive call has been issued by the consumer

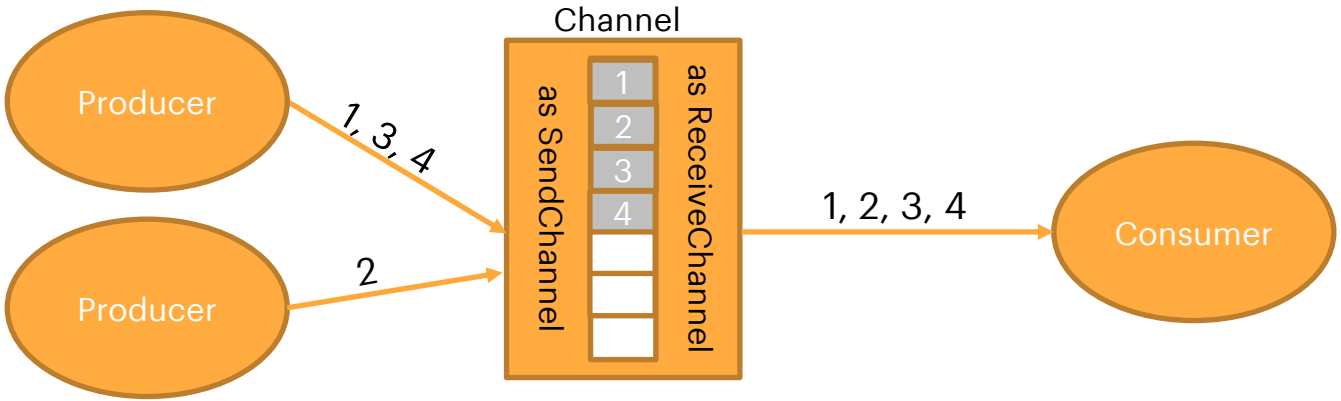
Fan-Out

Distribute a producer over multiple consumers



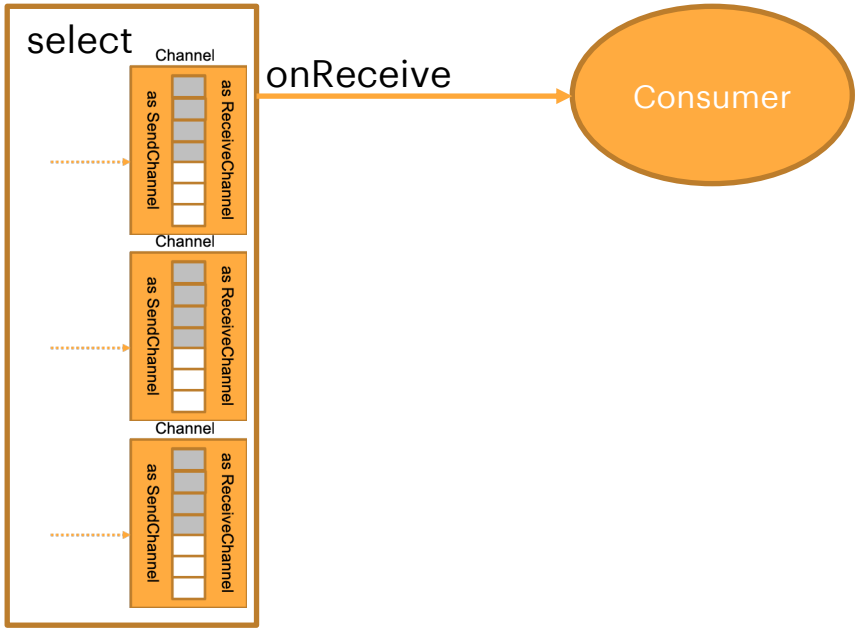
Fan-In

Merge multiple producers into one consumer



Merge

Merge multiple channels into one consumer



05

Coroutines vs Rx

Programming Style

Co

```
suspend fun getActor(actorId: String) : ActorCo { ... }

suspend fun getActorFromCache(actorId: String) : ActorCo? { ... }

suspend fun updateDb(actor: ActorCo) : Unit { ... }


...
updateDb(actor)
getActor(actor.id)
```

Rx

```
fun getActor(actorId: String) : Single<ActorRx> { ... }

fun getActorFromCache(actorId: String) : Maybe<ActorRx> { ... }

fun updateDb(actor: ActorRx) : Completable { ... }

...
updateDb(actor)
    .andThen(getActor(actor.id))
```

Programming Style

Co

```
suspend fun getActor(actorId: String) : ActorCo { ... }

suspend fun getActorFromCache(actorId: String) : ActorCo? { ... }

suspend fun updateDb(actor: ActorCo) : Unit { ... }


...
updateDb(actor)
getActor(actor.id)
```

Rx

```
fun getActor(actorId: String) : Single<ActorRx> { ... }

fun getActorFromCache(actorId: String) : Maybe<ActorRx> { ... }

fun updateDb(actor: ActorRx) : Completable { ... }

...
updateDb(actor)
    .andThen(getActor(actor.id))
```

Transformation

Co

```
suspend fun ImDbCo.getFullName(actorId: String) : String {
    val actor = getActor(actorId)
    return actor.fullName
}
```

Rx

```
fun ImDbRx.getFullName(actorId: String) : Single<String> {
    return getActor(actorId)
        .map { it.fullName }
}
```

Transformation

Co

```
suspend fun ImDbCo.getFullName(actorId: String) : String {
    val actor = getActor(actorId)
    return actor.fullName
}
```

Rx

```
fun ImDbRx.getFullName(actorId: String) : Single<String> {
    return getActor(actorId)
        .map { it.fullName }
}
```

Sequential Results

Co

```
suspend fun ImDbCo.getFellowActors(actorId: String) : List<ActorCo> {
    val actor = getActor(actorId)
    val movie = getMovie(actor.mostRecentMovieId)
    return movie.getActors()
}
```

Rx

```
fun ImDbRx.getFellowActors(actorId: String) : Single<List<ActorRx>> {
    return getActor(actorId)
        .flatMap { getMovie(it.mostRecentMovieId) }
        .flatMap { it.getActors() }
}
```

Sequential Results

Co

```
suspend fun ImDbCo.getFellowActors(actorId: String) : List<ActorCo> {
    val actor = getActor(actorId)
    val movie = getMovie(actor.mostRecentMovieId)
    return movie.getActors()
}
```

Rx

```
fun ImDbRx.getFellowActors(actorId: String) : Single<List<ActorRx>> {
    return getActor(actorId)
        .flatMap { getMovie(it.mostRecentMovieId) }
        .flatMap { it.getActors() }
}
```

Concurrent Results

Co

```
suspend fun ImDbCo.getCombinedTitle(movieId1: String, movieId2: String): String =
    coroutineScope {
        val movie1 = async { getMovie(movieId1) }
        val movie2 = async { getMovie(movieId2) }
        "${movie1.await().title} and ${movie2.await().title}"
    }
```

Rx

```
fun ImDbRx.getCombinedTitle(movieId1: String, movieId2: String): Single<String> {
    return Single.zip(
        getMovie(movieId1), getMovie(movieId2),
        BiFunction { movie1: MovieRx, movie2: MovieRx ->
            "${movie1.title} and ${movie2.title}"
        }
    )
}
```

Concurrent Results

Co

```
suspend fun ImDbCo.getCombinedTitle(movieId1: String, movieId2: String): String =
    coroutineScope {
        val movie1 = async { getMovie(movieId1) }
        val movie2 = async { getMovie(movieId2) }
        "${movie1.await().title} and ${movie2.await().title}"
    }
```

Rx

```
fun ImDbRx.getCombinedTitle(movieId1: String, movieId2: String): Single<String> {
    return Single.zip(
        getMovie(movieId1), getMovie(movieId2),
        BiFunction { movie1: MovieRx, movie2: MovieRx ->
            "${movie1.title} and ${movie2.title}"
        }
    )
}
```

Parallel Results

Co

```
suspend fun ImDbCo.getMovies(movieIds: List<String>) : List<MovieCo> = coroutineScope {
    val movies = mutableListOf<Deferred<MovieCo>>()
    for (movieId in movieIds) {
        movies += async { getMovie(movieId) }
    }
    movies.awaitAll()
}
```

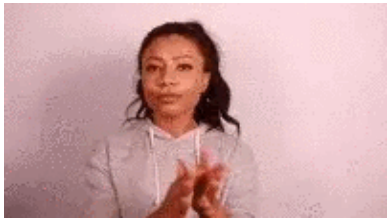
Rx

```
fun ImDbRx.getMovies(moviedIds: List<String>) : Single<List<MovieRx>> {
    return Flowable.fromIterable(moviedIds)
        .flatMapSingle { getMovie(it) }
        .toList()
}
```

Parallel Results

Co

```
suspend fun ImDbCo.getMovies(movieIds: List<String>) : List<MovieCo> = coroutineScope {
    val movies = mutableListOf<Deferred<MovieCo>>()
    for (movieId in movieIds) {
        movies += async { getMovie(movieId) }
    }
    movies.awaitAll()
}
```



Rx

```
fun ImDbRx.getMovies(moviedIds: List<String>) : Single<List<MovieRx>> {
    return Flowable.fromIterable(moviedIds)
        .flatMapSingle { getMovie(it) }
        .toList()
}
```

Shared Results

Co

```
fun ImDbCo.shareFullName(scope: CoroutineScope, actorId: String) : Deferred<String>  {
    return scope.async(start = CoroutineStart.LAZY) { getActor(actorId).fullName }
}

...
val sharedFullName = imdb.shareFullName("actor7jef")
val fullName1 = sharedFullName.await()
...
val fullName2 = sharedFullName.await()
```

Rx

```
fun ImDbRx.shareFullName(actorId: String) : Single<String> {
    return getActor(actorId)
        .map { it.fullName }
        .cache()
}

...
val sharedFullName = imdb.shareFullName("actor0jih")
sharedFullName.subscribe { val fullName1 = it }
...
sharedFullName.subscribe { val fullName2 = it }
```

Shared Results

Co

```
fun ImDbCo.shareFullName(scope: CoroutineScope, actorId: String) : Deferred<String>  {
    return scope.async(start = CoroutineStart.LAZY) { getActor(actorId).fullName }
}

...
val sharedFullName = imdb.shareFullName("actor7jef")
val fullName1 = sharedFullName.await()
...
val fullName2 = sharedFullName.await()
```

Rx

```
fun ImDbRx.shareFullName(actorId: String) : Single<String> {
    return getActor(actorId)
        .map { it.fullName }
        .cache()
}

...
val sharedFullName = imdb.shareFullName("actor0jih")
sharedFullName.subscribe { val fullName1 = it }
...
sharedFullName.subscribe { val fullName2 = it }
```

Flow Control

Co

```
suspend fun <T : Any> retry(maxRetries: Int, initialDelay: Int, data: (suspend () -> T)): T {
    var result: T? = null
    for (retryCount in 1..maxRetries) {
        try {
            result = data()
            break
        } catch (e: Throwable) {
            delay(initialDelay * 2.pow(retryCount))
        }
    }
    return result ?: throw Exception("NotFound")
}
```

Rx

Flow Control

Co

```
suspend fun <T : Any> retry(maxRetries: Int, initialDelay: Int, data: (suspend () -> T)): T {
    var result: T? = null
    for (retryCount in 1..maxRetries) {
        try {
            result = data()
            break
        } catch (e: Throwable) {
            delay(initialDelay * 2.pow(retryCount))
        }
    }
    return result ?: throw Exception("NotFound")
}
```

Rx

```
fun <T : Any> retry(maxRetries: Int, initialDelay: Int, data: Single<T>): Single<T> {
    return data.toFlowable()
        .retryWhen { error ->
            error
                .zipWith((1..maxRetries).toFlowable()) { _, retryCount -> retryCount }
                .flatMap { retryCount ->
                    Flowable.timer(initialDelay * 2.pow(retryCount))
                }
        }
        .singleElement()
        .switchIfEmpty(Single.error(Exception("NotFound")))
}
```

Flow Control

Co

```
suspend fun <T : Any> retry(maxRetries: Int, initialDelay: Int, data: (suspend () -> T)): T {
    var result: T? = null
    for (retryCount in 1..maxRetries) {
        try {
            result = data()
            break
        } catch (e: Throwable) {
            delay(initialDelay * 2.pow(retryCount))
        }
    }
    return result ?: throw Exception("NotFound")
}
```

Rx

```
fun <T : Any> retry(maxRetries: Int, initialDelay: Int, data: Single<T>): Single<T> {
    return data.toFlowable()
        .retryWhen { error ->
            error
                .zipWith((1..maxRetries).toFlowable()) { _, retryCount -> retryCount }
                .flatMap { retryCount ->
                    Flowable.timer(initialDelay * 2.pow(retryCount))
                }
        }
        .singleElement()
        .switchIfEmpty(Single.error(Exception("NotFound")))
}
```

Exception Handling

Co

```
suspend fun ImDbCo.getOptionalFullName(actorId: String) : String {
    return try {
        val actor = getActor(actorId)
        actor.fullName
    } catch (e: Throwable) {
        ""
    }
}
```

Rx

```
fun ImDbRx.getOptionalFullName(actorId: String) : Single<String> {
    return getActor(actorId)
        .map { it.fullName }
        .onErrorReturn { "" }
}
```

Exception Handling

Co

```
suspend fun ImDbCo.getOptionalFullName(actorId: String) : String {
    return try {
        val actor = getActor(actorId)
        actor.fullName
    } catch (e: Throwable) {
        ""
    }
}
```

Rx

```
fun ImDbRx.getOptionalFullName(actorId: String) : Single<String> {
    return getActor(actorId)
        .map { it.fullName }
        .onErrorReturn { "" }
}
```

Scheduling for Observation

Co

```
class MyActivity : Activity(), CoroutineScope {
    private val job = SupervisorJob()
    override val coroutineContext: CoroutineContext = job + Dispatchers.Main

    fun ImDbCo.showMovie(movieId: String) {
        launch { textView.text = getMovie(movieId).title }
    }
```

Rx

```
    fun ImDbRx.showMovie(movieId: String) {
        getMovie(movieId)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe { textView.text = it.title }
    }
}
```

Scheduling for Observation

Co

```
class MyActivity : Activity(), CoroutineScope {
    private val job = SupervisorJob()
    override val coroutineContext: CoroutineContext = job + Dispatchers.Main

    fun ImDbCo.showMovie(movieId: String) {
        launch { textView.text = getMovie(movieId).title }
    }
```

Rx

```
    fun ImDbRx.showMovie(movieId: String) {
        getMovie(movieId)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe { textView.text = it.title }
    }
}
```

Scheduling for Getting Data

Co

```
suspend fun ImDbCo.getProtagonistId(movieId: String) : String {
    return withContext(Dispatchers.IO) {
        getMovie(movieId).protagonistId
    }
}
```

Rx

```
fun ImDbRx.getProtagonistId(movieId: String): Single<String> {
    return getMovie(movieId)
        .subscribeOn(Schedulers.io())
        .map { it.protagonistId }
}
```

Scheduling for Getting Data

Co

```
suspend fun ImDbCo.getProtagonistId(movieId: String) : String {
    return withContext(Dispatchers.IO) {
        getMovie(movieId).protagonistId
    }
}
```

Rx

```
fun ImDbRx.getProtagonistId(movieId: String): Single<String> {
    return getMovie(movieId)
        .subscribeOn(Schedulers.io())
        .map { it.protagonistId }
}
```

Reading from a Stream

Co

```
suspend fun ImDbCo.printMovieTitles(actorId: String) {
    val titles = mutableListOf<String>()
    val movieStream: ReceiveChannel<MovieCo> = getActor(actorId).getMovieStream()

    movieStream.consumeEach { movie ->
        titles += movie.title
    }


    println(titles)
}
```

Rx

```
fun ImDbRx.printMovieTitles(actorId: String) {
    getActor(actorId)
        .flatMapPublisher { it.getMovieStream() }
        .map { it.title }
        .toList()
        .subscribe { titles -> println(titles) }
}
```

Reading from a Stream

Co

```
suspend fun ImDbCo.printMovieTitles(actorId: String) {
    val titles = mutableListOf<String>()
    val movieStream: ReceiveChannel<MovieCo> = getActor(actorId).getMovieStream()

    for (movie in movieStream) {
        titles += movie.title
    }


    println(titles)
}
```

Rx

```
fun ImDbRx.printMovieTitles(actorId: String) {
    getActor(actorId)
        .flatMapPublisher { it.getMovieStream() }
        .map { it.title }
        .toList()
        .subscribe { titles -> println(titles) }
}
```

Reading from a Stream

Co

```
suspend fun ImDbCo.printMovieTitles(actorId: String) {
    val titles = mutableListOf<String>()
    val movieStream: ReceiveChannel<MovieCo> = getActor(actorId).getMovieStream()

    while (true) {
        val movie = movieStream.receiveOrNull()
        if (movie == null) break
        titles += movie.title
    }
    println(titles)
}
```

Rx

```
fun ImDbRx.printMovieTitles(actorId: String) {
    getActor(actorId)
        .flatMapPublisher { it.getMovieStream() }
        .map { it.title }
        .toList()
        .subscribe { titles -> println(titles) }
}
```

Reading from a Stream

Co

```
suspend fun ImDbCo.printMovieTitles(actorId: String) {
    val titles = mutableListOf<String>()
    val movieStream: ReceiveChannel<MovieCo> = getActor(actorId).getMovieStream()

    while (true) {
        val movie = movieStream.receiveOrNull()
        if (movie == null) break
        titles += movie.title
    }
    println(titles)
}
```

Rx

```
fun ImDbRx.printMovieTitles(actorId: String) {
    getActor(actorId)
        .flatMapPublisher { it.getMovieStream() }
        .map { it.title }
        .toList()
        .subscribe { titles -> println(titles) }
}
```

Writing to a Stream

Co

```
fun CoroutineScope.produceMovieTitlesStream(movieTitles : List<String>): ReceiveChannel<String> = produce {
    movieTitles.forEach {
        send(it)
    }
    close()
}

...
val title: ReceiveChannel<String> = produceMovieTitlesStream(titles)
if (someCondition) return 0 // Possible 'hanging' of this coroutine
...
```

Rx

```
fun produceMovieTitlesStream(movieTitles : List<String>) : Flowable<String> {
    return Flowable.create<String> { observer ->
        movieTitles.forEach {
            observer.onNext(it)
        }
        observer.onComplete()
    }
}
```

Writing to a Stream

Co

```
fun CoroutineScope.produceMovieTitlesStream(movieTitles : List<String>): ReceiveChannel<String> = produce {
    movieTitles.forEach {
        send(it)
    }
    close()
}

...
val title: ReceiveChannel<String> = produceMovieTitlesStream(titles)
if (someCondition) return 0 // Possible 'hanging' of this coroutine
...
```

Rx

```
fun produceMovieTitlesStream(movieTitles : List<String>) : Flowable<String> {
    return Flowable.create<String> { observer ->
        movieTitles.forEach {
            observer.onNext(it)
        }
        observer.onComplete()
    }
}
```

Writing to a Stream

Co

```
fun ImDbCo.getMovieTitlesStream(scope: CoroutineScope, actorId: String): ReceiveChannel<String> =
    scope.produce {
        getActor(actorId).getMovieStream().consumeEach {
            send(it.title)
        }
        close()
    }

...
val title: ReceiveChannel<String> = getMovieTitlesStream(scope, titles)
if (someCondition) return 0 // Possible 'hanging' of this coroutine
...
```

Rx

```
fun ImDbRx.getMovieTitlesStream(actorId: String) : Flowable<String> {
    return getActor(actorId)
        .flatMapPublisher { it.getMovieStream() }
        .map { it.title }
}
```

Writing to a Stream

Co

```
fun ImDbCo.getMovieTitlesStream(scope: CoroutineScope, actorId: String): ReceiveChannel<String> =
    scope.produce {
        getActor(actorId).getMovieStream().consumeEach {
            send(it.title)
        }
        close()
    }

...
val title: ReceiveChannel<String> = getMovieTitlesStream(scope, titles)
if (someCondition) return 0 // Possible 'hanging' of this coroutine
...
```

Rx

```
fun ImDbRx.getMovieTitlesStream(actorId: String) : Flowable<String> {
    return getActor(actorId)
        .flatMapPublisher { it.getMovieStream() }
        .map { it.title }
}
```

Shared Streams

Co

```
fun <T : Any> ReceiveChannel<T>.shareStream(): BroadcastChannel<T> = BroadcastChannel<T>(1000).also { ch ->
    GlobalScope.launch {
        this@shareStream.consumeEach {
            ch.send(it)
        }
        ch.close()
    }
}

...
val observer: ReceiveChannel<String> = channel.shareStream().openSubscription()
// Use ConflatedBroadcastChannel for a state-full channel
```

Rx

```
fun <T : Any> Flowable<T>.shareStream(): Flowable<T> {
    return PublishProcessor.create<T>().also {
        subscribe(it)
    }
}

...
flowable.shareStream().subscribe { ... }
// Use BehaviorProcessor for a state-full observable
```

Shared Streams

Co

```
fun <T : Any> ReceiveChannel<T>.shareStream(): BroadcastChannel<T> = BroadcastChannel<T>(1000).also { ch ->
    GlobalScope.launch {
        this@shareStream.consumeEach {
            ch.send(it)
        }
        ch.close()
    }
}

...
val observer: ReceiveChannel<String> = channel.shareStream().openSubscription()
// Use ConflatedBroadcastChannel for a state-full channel
```

Rx

```
fun <T : Any> Flowable<T>.shareStream(): Flowable<T> {
    return PublishProcessor.create<T>().also {
        subscribe(it)
    }
}

...
flowable.shareStream().subscribe { ... }
// Use BehaviorProcessor for a state-full observable
```

Custom Stream Operators

Co

```
fun <T : Any, R: Any> ReceiveChannel<T>.mapIfNotNull(mapper: (T) -> R?) : ReceiveChannel<R> =
    GlobalScope.produce {
        consumeEach {
            mapper(it)?.also {
                send(it)
            }
        }
        close()
    }
```

Rx

Custom Stream Operators

Co

```
fun <T : Any, R: Any> ReceiveChannel<T>.mapIfNotNull(mapper: (T) -> R?) : ReceiveChannel<R> =
    GlobalScope.produce {
        consumeEach {
            mapper(it)?.also {
                send(it)
            }
        }
        close()
    }
```

Rx

```
fun <T : Any, R: Any> Flowable<T>.mapIfNotNull(mapper: (T) -> R?) : Flowable<R> {
    TODO() // Many lines of code (if not using existing operators)
}
```

Custom Stream Operators

Co

```
fun <T : Any, R: Any> ReceiveChannel<T>.mapIfNotNull(mapper: (T) -> R?) : ReceiveChannel<R> =
    GlobalScope.produce {
        consumeEach {
            mapper(it)?.also {
                send(it)
            }
        }
        close()
    }
```

Rx

```
fun <T : Any, R: Any> Flowable<T>.mapIfNotNull(mapper: (T) -> R?) : Flowable<R> {
    TODO() // Many lines of code (if not using existing operators)
}
```

Backpressure

Coroutines	Rx
Rendezvous	(not applicable)
Conflated	Latest
Unlimited	Buffer with *large* value
Fixed Size	Buffer
`channel.offer(data)`	Drop
`if (!channel.offer(data)) throw Error()`	Error

Backpressure

Coroutines	Rx
Rendezvous	(not applicable)
Conflated	Latest
Unlimited	Buffer
Fixed Size	Buffer
`channel.offer(data)`	Drop
`if (!channel.offer(data)) throw Error()`	Error

Rx – Coroutines Adapter

- **Add Gradle dependencies**

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-rx2:$coVersion"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-reactive:$coVersion"
```

- **From Coroutines to Rx**

```
runBlocking {
    val single = async { getData() }.asSingle(coroutineContext)
    val flowable = channel.asPublisher(coroutineContext)
}
```

- **From Rx to Coroutines**

```
val data : Int = single.await()
val channel: ReceiveChannel<Int> = flowable.openSubscription()
```

06

Recap

Advantages of Coroutines

- Familiar imperative and sequential programming paradigm
- Relatively small API
  - Just use familiar imperative programming for more complex solutions
- Uniquely tailored for single-shot results or actions
- Channels for **synced hot** streams that exist without app asking for them
- Brand new api **Flow** for **non-blocking cold** streams
  This API is in Preview Status: http://bit.ly/CoroutinesFlow

Disadvantages of Coroutines

- Suspension (vs blocking) is sometimes hard to grasp
- Only in the IDE, **suspend** functions are visible on call-site
- Channels are **hot** and can leak the resources they represent
- Large parts of Channel API are still experimental

Advantages of Rx Java

- Powerful Functional Programming paradigm and Operators
  - However, this could be less familiar to some developers
- Powerful subscription model
  - Hot vs Cold Observables
  - Postpone creation of data until moment of subscription
- Easier to share publishers (observables) of data
  - Reference counting, connectables, etc.
  - Need to create custom solutions when using Coroutines

Disadvantages of Rx

- Often overkill for code that just needs simple single-shot results
  - Storage retrieval and updates
  - Network requests
- Large API of many operators
- Hard to implement flow control

Conclusion

- No clear winner
- Imperative and sequential code is the true power of Coroutines
- Have only simple storage and network requests? Use Coroutines
- Need full control of many streams of data? Use Rx
- You can use both Rx and Coroutines
- Coroutines' new Flow API: Coroutines may become the winner!

Courtesy to Giphy for the Winner GIFs!

How to make Channels more reactive: http://bit.ly/ReactiveChannels