**40 RxJava interview questions (and answers)**

**RxJava interview questions**

I encourage you to answer these on your own. If you need help, I’ve provided sample answers below.

1. [Is RxJava following the “push” or “pull” pattern?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#push_vs_pull)
2. [What’s the difference between onNext(), onComplete() and onError()?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#observable_emissions)
3. [How many times can each of the above be called?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#observable_emissions_count)
4. [When does an Observable start emitting items?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#observable_emissions_start)
5. [What’s the difference between a COLD and HOT observables?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#cold_vs_hot)
6. [Can you transform a COLD observable to a HOT one?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#cold_to_hot)
7. [… and a HOT into a COLD one?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#hot_to_cold)
8. [What’s a Scheduler? Why does RxJava use Schedulers?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#scheduler)
9. [What’s an Observable chain?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#observable_chain)
10. [Can you have multiple operators of the same type in a single chain (e.g. map().map().map()…)?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#same_operators)
11. [Does the above work with all operators?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#same_operators_all)
12. [What’s the difference between observeOn() and subscribeOn()?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#observer_subscribe_on)
13. [What’s going to happen if you have multiple subscribeOn() operators in a chain?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#multiple_subscribe_on)
14. [What’s going to happen if you have multiple observeOn() operators in a chain?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#multiple_observer_on)
15. [What’s the difference between a map() and a flatMap()?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#map_flatmap)
16. [What’s the difference between flatMap(), concatMap() and switchMap()?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#flat_concat_switch_map)
17. [What’s the most complex operator you know?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#complex_operator)
18. [Can one create custom operators in RxJava? Anything to be aware of?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#custom_operators)
19. [What’s a Transformer?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#transformer)
20. [Describe the most complex chain you’ve implemented. What was the tricky bit there?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#complex_chain)
21. [How can you test observable chains? Mention a few classes that can help and their function.](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#test_operators)
22. [What error handling operators do you know in RxJava?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#error_handling_operators)
23. [What will happen if there’s multiple errors in a chain?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#multiple_errors)
24. [Handling multiple errors (see example code).](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#multiple_errors_practice)
25. [What are the biggest differences you know between RxJava1 and RxJava2?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#rxjava1_rxjava2)
26. [Which RxJava construct you’ll use to represent an API call that needs to be called at some point in the future?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#api_call_construct)
27. [What if the API call is a “fire-and-forget” one?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#fire_forget_call_construct)
28. [Do you know any RxJava constructs other than Observable? Describe them?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#rxjava_constructs)
29. [What’s a Subject in RxJava and what’s it used for?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#subjects)
30. [Are there different types of Subjects?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#subject_types)
31. [What’s the difference between Subject and RxRelay?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#subject_relay)
32. [What is backpressure?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#backpressure)
33. [How to deal with backpressure issues?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#backpressure_handling)
34. [How does data flow in RxJava by default?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#data_flow)
35. [Does RxJava support parallelism? If so, how to achieve it?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#parallelism)
36. [Are memory leaks an issue when using RxJava? How would you protect from such?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#memory_leaks)
37. [Is RxJava working good in combination with Kotlin?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#rxjava_kotlin)
38. [What’s a marble diagram?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#marble_diagrams)
39. [What’s a “pure” function?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#pure_functions)
40. [Do you know any other reactive libraries? What’s the biggest differences, pros, cons?](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#reactive_libraries)

**Answers**

Here are my answers to all the questions. They aren’t the most in-depth possible ones, rather that’s what I’d say in an interview environment:

**1. Is RxJava following the “push” or “pull” pattern?**

In RxJava new data is being “pushed” to observers.

**2. What’s the difference between onNext(), onComplete() and onError()?**

These are the callbacks an Observable / Flowable will receive. The first one is called for each emission of the Observable / Flowable (e.g. zero to infinity times). onComplete() and onError() are mutually exclusive – only ONE of them will be called at most once. In other words a stream cannot complete and error out at the same time.

**3. How many times can each of the above be called?**

onNext() – from zero between infinite number of times  
onComplete() – maximum once per stream  
onError() – maximum once per stream

**4. When does an observable start emitting items?**

There’s two types of Observables – *Cold* and *Hot*. Cold ones perform work (and subsequently emit items) only once someone is listening for it (e.g. someone has subscribed to them). Hot observables perform work and emit items regardless if there are any observers or not.

**5. What’s the difference between a COLD and HOT observables?**

They start emitting items differently ([see #4](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#observable_emissions_start)).

Cold observables are created multiple times and each instance can be triggered on it’s own. Hot observables are like a “stream” of ongoing events – observers can come and go, but the stream is created ones and just goes on.

**6. Can you transform a COLD observable to a HOT one and vice-versa?**

One way to make a Cold observable Hot is by using publish().connect(). publish() converts the Cold observable to a ConnectableObservable, which pretty much behaves like a Hot one. Once triggered with the .connect() operator, it’ll publish events regardless if there are any subscribers.

Another way to transform a Cold observable to a Hot one is by wrapping it with a Subject. The Subject subscribes to the Cold observable immediately and exposes itself as an Observable to future subscribers. Again, the work is performed regardless whether there are any subscribers … and on the other hand multiple subscribers to the Subject won’t trigger the initial work multiple times.

**7. … and a HOT into to a COLD one?**

The first way of transforming (or rather “masking”) a Hot observable to Cold is by using the defer() operator. It defers the creation of the Hot observable altogether, so each new subscriber will trigger the work again (feature of a Cold observable).

Depending on the use-case the pattern mentioned above might be quite wasteful, so another strategy is using the replay().autoConnect(0) paradigm. The replay() operator will cache the values emitted by the Hot observable and re-emit them to future subscribers. autoConnect(0) returns an observable that can be triggered even when there are no subscribers to the underlaying Hot observable. The combination of both just replays cached values from the Hot observable as a Cold one.

**8. What’s a Scheduler? Why does RxJava use Schedulers?**

By default RxJava is single-threaded – all operations are executed on a single thread. Schedulers are the means to switch the execution to a different thread. They’re also an abstraction over the concept of “time”, which is needed for time-sensitive operations (delay(), timeout(), buffer(), window(), etc).

**9. What’s an Observable chain?**

A list of operations / transformations performed between the source and the end subscriber. A simple example is emitting a User object, filtering out admin users (filter()), checking whether they’re authenticated (filter()) and finally emitting they’re full name (map()).

**10. Can you have multiple operators of the same type in a single chain (e.g. map().map().map()…)?**

Yes, and it’s actually a good practice to have these. Ideally every operator is a “pure” function ([see #39](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#pure_functions)) that performs a single filtering / transformation / collection from an input -> output. Splitting complex functions into multiple simple ones makes them more composable and easy to understand. A small downside is performance (since every operator in a chain requires a bit of execution time and perhaps memory), so as always – balance is needed.

**11. Does the above work with all operators?**

Works as expected for pretty much all operators from the standard library except subscribeOn(). [See #13](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#multiple_subscribe_on) for details why multiple subscribeOn() operators don’t make sense.

<https://veskoiliev.com/simple-rx-mistakes-and-how-to-avoid-them/#subscribeon_observeon>

**12. What’s the difference between observeOn() and subscribeOn()?**

[**https://proandroiddev.com/understanding-rxjava-subscribeon-and-observeon-744b0c6a41ea**](https://proandroiddev.com/understanding-rxjava-subscribeon-and-observeon-744b0c6a41ea)

subscribeOn() denotes the Scheduler on which the source work will be performed on. Since there’s only one initial source of an Observable chain, it makes sense to only have one subscribeOn() operator.

observeOn() denotes the Scheduler on which all downstream operations will be performed. In other words it changes the Scheduler for all operators **after** it. Since there can be many such operators, having multiple observeOn() operators in a single chain makes sense and works as expected.

We can specify a thread to execute any operator by using subscribeOn and/or observeOn.

* subscribeOn affects **upstream** operators (operators above the subscribeOn)
* observeOn affects **downstream** operators (operators below the observeOn)
* If only subscribeOn is specified, all operators will be be executed on that thread
* If only observeOn is specified, all operators will be executed on the current thread and only operators below the observeOn will be switched to thread specified by the observeOn

**13. What’s going to happen if you have multiple subscribeOn() operators in a chain?**

Only the very first subscribeOn() in the chain has the desired effect, all subsequent ones would not have any effect, apart from potentially wasting system resources (depending on the Scheduler specified). For a detailed explanation read [THIS](https://veskoiliev.com/simple-rx-mistakes-and-how-to-avoid-them/#subscribeon_observeon) article.

**14. What’s going to happen if you have multiple observeOn() operators in a chain?**

Each observeOn() switches the Scheduler (thread) on which all subsequent operators will be executed on. Complex RxJava streams can absolutely benefit from multiple observeOn() operators. For a detailed explanation read [THIS](https://veskoiliev.com/simple-rx-mistakes-and-how-to-avoid-them/#subscribeon_observeon) article.

**15. What’s the difference between a map() and a flatMap()?**

The map() operator maps from a concrete ValueA to a concrete ValueB (e.g. from an Int -> String, or User -> String).

 flatMap() maps from a concrete ValueA to a Stream<ValueB>. If Stream<ValueB> emits multiple items, all of these will be eventually served to the original observer (e.g. they are being “flattened” to a single Observer). Since there’s no restrictions on the Stream<ValueB>, flatMap() is useful to introduce parallelism in executing tasks ([see #35](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#parallelism)).

**16. What’s the difference between flatMap(), concatMap() and switchMap()?**

The difference is best illustrated using marble diagrams ([see #38](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#marble_diagrams)), but …

flatMap() can split a chain to multiple intermediate streams (let’s call them [A, B, C]), the results of which are all “flattened” to a single stream again. The emissions of these intermediate streams are propagated directly to the main stream without any guarantees about the order in which they’ll appear. It’s entirely possible the end result to look like this: *[B1, A1, B2, C1, B3, A2, C2, C3, A3]*

concatMap() works similarly to flatMap() with the exception that the intermediate streams are “activated” in the order they appear. The end result will look like this: *[A1, A2, A3, B1, B2, B3, C1, C2, C3]*

switchMap() is again similar to flatMap() with the exception that whenever a new item is emitted from the source Observable, all previously generated intermediate streams are terminated (e.g. disposed()). At this moment only the latest intermediate Observable remains active. With switchMap() it’s entirely possible for events to be dismissed altogether, so a possible output from the example above is: *[A1, B1, B2, C1, C2, C3]*. In this example the source Observable emitted item B when the A intermediate observable has emitted only A1. Similarly, the source emitted C before the intermediate stream B could finish (thus B3 event is missing).

**17. What’s the most complex operator you know?**

It’s really hard to single out one here. The operations of ConnectableObservable and their correct usage took a while to grasp (publish().connect(), refCount(), etc). Also getting familiar with the most complete overload of [flatMap()](https://github.com/ReactiveX/RxJava/blob/2.x/src/main/java/io/reactivex/Observable.java" \l "L8386) is quite beneficial, as it helps you understand how the operator works, but also unlocks other paradigms like parallelism ([see #35](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#parallelism)).

**18. Can one create custom operators in RxJava? Anything to be aware of?**

Although you can create your own operator, it’s highly recommended to reuse existing ones or any combination of them. Implementing a new operator correctly is very tricky and there’s too much room for errors (backpressure, breaking an API contract, thread safety, etc). Still, if required, [THIS](https://github.com/ReactiveX/RxJava/wiki/Implementing-Your-Own-Operators) is the best read to get started.

**19. What’s a Transformer?**

A Transformer is a convenient way to encapsulate common operations in a reusable way. This encapsulated logic can be tested in isolation, which is easier, and further simplifies the tests of all chains that use it. For example if you find yourself repeating a sequence of operators multiple times in your code (e.g. *.map(user -> user.age).filter(age -> age > 18)*), this logic can be factored out in a *UserLegalAgeTransformer*. It can be reused in the chains like this .compose(new UserLegalAgeTransformer()).

**20. Describe the most complex chain you’ve implemented? What was the tricky bit there?**

Can’t recall a specific example, but it must be a long one that combines multiple data sources. Generally the tricky bit is decomposing the business requirements to multiple pure functions ([see #39](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#pure_functions)). Once that’s done one can find really good and descriptive name for each operation in the chain, thus making it easier to reason about.

Here’s a paradigm I really like for implementing a very simple *Repository* pattern using the concat().take(1) operators:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | fun fetchData(val id: Long): Single<Data> {  **return** Observable.concat(          getFromLocalCache(id),          getFromPersistentCache(id),          getFromNetwork(id)             )          .take(1)          .singleOrError()  } |

All complex logic is hidden away in the respective *getFrom()* methods, thus making the chain easily understandable. Concat() will exhaust each getFrom() method until it finishes. If a value is received, the take(1) operator “stops” the whole chain, so subsequent getFrom() methods won’t be called. For example, if there’s no data in the local cache, *getFromLocalCache()* will just complete without emitting any values. If *getFromPersistentCache()* returns a value, the take(1) operator terminates the rest of the chain, so *getFromNetwork()* will not be executed at all.

**21. How can you test observable chains? Mention a few classes that can help and their function.**

TestObserver and TestSubscriber are common classes used to test Observable / Flowable operations. With these you can wait for, and inspect all received events and their exact values.

The TestScheduler class is very useful when testing time-based operators (e.g. timeout(), buffer(), window(), etc) – it allows to manually control the “time” to you can test all possible code paths in your chain.

**22. What error handling operators do you know in RxJava?**

There’s two main categories of such operators – one for performing *side-effects only*, after which the error is passed downstream (doOnError(...)). The other category is the onErrorReturn(...), onErrorResumeNext(...) ones, which can *handle an error and continue* the stream successfully.

**23. What will happen if there’s multiple errors in a chain?**

All unhandled errors (via any of the error handling operators, [see #22](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#error_handling_operators)) are propagated downstream. A chain can have only one terminal error event (e.g. one call to onError(throwable)), so the first unhandled error will terminate the stream. In case there’s other undelivered exceptions, please [see #24](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#multiple_errors_practice).

**24. Handling multiple errors (see example code).**

What will happen (in terms of error handling) if *getAllUsers()* emits a sequence of 10 userIds and *getUserProfile(userId)* emits error for every userId? Is the behaviour different between RxJava1 and RxJava2?

|  |  |
| --- | --- |
| 1  2  3  4 | getAllUsers()      .flatMap(userId -> getUserProfile(userId).subscribeOn(Schedulers.io()))      .onErrorResumeNext(...)      .subscribe(...) |

In RxJava1 the first propagated error will terminate the stream. All other started parallel streams (that’ll error out as well) are the so-called “undelivered exceptions”, which are just “swallowed” (printed in console by default). Difference with RxJava2 comes from the handling of these undeliverable exceptions – they’ll be sent to a global error handler (set via RxJavaPlugins.setErrorHandler()), where further handing can occur. If such error handler isn’t set, the exceptions are propagated upstream to the calling thread (e.g. will cause a crash of the app).

**25. What are the biggest differences you know between RxJava1 and RxJava2?**

Here’s the 3 differences one surely needs to take into account when updating to RxJava2:

*Null not supported anymore* – previously Observable.just(user) would work even if user == null. In RxJava2 the same will throw a NullPointerException. Not supporting null values makes streams a bit easier to work with (as there’s no null checks everywhere), but one must be more cautious what data flows through them. Wrapping emissions with classes like [Java Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html) is handy for places where you’re not sure if the data is null-safe.

RxJava2 *Observable type doesn’t support backpressure anymore*, so places where that might be an issue had to be changed to use Flowable instead. It’s a great change as it makes pretty clear where Backpressure care is needed. On the other hand – it requires careful inspection of the flows that were using RxJava1 Observable. Common places where Backpressure might occur is when fetching data from a database or mapping user-actions to events. On a related note – in RxJava2 Subjects no longer support Backpressure. The new Processors classes do.

Global error handling – in RxJava2 *no error can be swallowed* (as it could in RxJava1). All such undelivered errors are passed to a global error handler, which can be set using the RxJavaPlugins.setErrorHandler(). Such exceptions usually occur on streams that have parallel execution. When updating, one must always set such handler and at least log the errors. Ideally this global error handler will never receives anything.

**26. Which RxJava construct you’ll use to represent an API call that needs to be called at some point in the future?**

Usually API calls either return a response (*onSuccess()*) or error out (*onError()*), so Single is a great fit here.

**27. What if the API call is a “fire-and-forget” one?**

Since we’re not interested in the response of the API call, a Completable is a good it. If we’re not interested in errors either, adding onErrorComplete() operator will achieve the “fire-and-forget” nature.

**28. Do you know any RxJava constructs other than Observable? Describe them?**

The main constructs are:

Completable – maps an operation that either completes without returning a value (*onComplete()*) or errors out (*onError(throwable)*).

Single – either returns a value (*onSuccess(value)*) or errors out (*onError(throwable)*).

Maybe – has 3 options – returns a value successfully (*onSuccess(value)*), completes successfully without any value (*onComplete()*) or errors out (*onError(throwable)*).

Observable – represents a stream of events that emits zero to many events (*onNext(value)*), then either completes (*onComplete()*) or errors out (*onError(throwable)*). It does **NOT** support backpressure ([see #32](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#backpressure)).

Flowable – like an Observable, however it **DOES** support backpressure.

**29. What’s a Subject in RxJava and what’s it used for?**

A Subject is both a Subscriber and an Observer at the same time. My favourite analogy is a pipe – it can receive events from one end (because it’s a Subscriber) and let them through (“emit them”) via the other end because it’s an Observable. With Subjects one can transform Cold observables to Hot ones ([see #5](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#cold_vs_hot)). Subjects also are one of the easiest ways to introduce some type of local, temporary caching of a stream. Last but not least – Subjects can help you transform your non-reactive code to reactive if you don’t find any of the Observable.create() operators useful for your use-case.

**30. Are there different types of Subjects?**

There are 4 common Subject types:

PublishSubject – just passes incoming events to all it’s subscribers. New Subscribers will receive only the events emitted from the point of Subscription onwards.

BehaviourSubject – like PublishSubject, however each new Subscriber also receives the latest value of the stream (or a default value). The default value often provides nice user experience, as consumers of the Subject don’t need to worry about receiving empty streams.

AsyncSubject – emits only the **last** value emitted by the source Observable and only after that source Observables completes. Common use-case is for operations that need to finish before all clients (Subscribers) can proceed – e.g. the initial loading of an application.

ReplaySubject – every Subscriber will receive **all** events emitted by the source Observable, regardless at which point they subscribe. In other words all events are “cached” via the Subject. That’s one of the easiest ways to cache a stream, but one needs to be careful of memory issues (if the source Observable emits too many items that need to stay in the memory cache).

**31. What’s the difference between Subject and [RxRelay](https://github.com/JakeWharton/RxRelay)?**

A Relay is a Subject that cannot be terminated (cannot call onError() or onComplete() on it). That’s sometimes useful, as terminating a regular Subject makes it unusable in the future.

**32. What is backpressure?**

Backpressure is the inability of a Subscriber to handle all incoming events in time. In other words Backpressure can occur when the Producer of events is faster than the Consumer. If not handled correctly it can error out a stream.

**33. How to deal with backpressure issues?**

The first important thing is to choose the correct [RxJava construct](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/" \l "rxjava_constructs) for your stream. If you thing Backpressure might occur, then Flowable with a correct BackpressureStrategy is the safest choice. You can also try to manually “slow-down” the Producer by adding buffer-type (buffer(), window(), etc) operators before your event handling. Finally you can try to speed up your Consumer – ideally it should be doing small and fast operations. If you require more computation-intensive ones, perhaps some of that logic can be moved to- and parallelised by the Rx stream ifself.

**34. How does data flow in RxJava by default?**

Pick all that apply: sequential, parallel, overlapping, non-overlapping, single-threaded, multi-threaded?

RxJava is *single-threaded* by default. One can use Schedulers ([see #8](https://veskoiliev.com/40-rxjava-interview-questions-and-answers/#scheduler)) to change this if needed. Data flows *sequentially* by default, and events *don’t overlap* each other unless an operation is specifically parallelised.

**35. Does RxJava support parallelism? If so, how to achieve it?**

The two most common ways to execute a set of tasks in parallel are:

a) Using the flatMap() operator. Each inner stream inside the flatMap() operation should subscribeOn() a background thread (preferably Schedulers.io()). An example to note the difference:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // This version is sequential  getUserIds()      .flatMap(userId -> getUserProfile(userId))      .subscribe(...)    // This version is parallel  getUserIds()      .flatMap(userId -> getUserProfile(userId).subscribeOn(Scheudlers.io()))      .subscribe(...) |

The first example is synchronous and there’s no parallel execution – the User profiles are fetched one after the other. That’s done on the Scheduler inherited from the *getUserIds()* Observable. In the second example however user profiles are fetched in parallel. The number of parallel executions can be controlled if an overload of flatMap() is used, where you specify the *maxConcurrency* parameter.

b) Although the above pattern works correctly to achieve parallelism, in recent versions of RxJava a better construct was intro

duced – the ParallelFlowable. It provides an easier and more explicit API to achieve parallelism. Let’s rewrite the example from above:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | getUserIds()      .parallel(16)      .runOn(Schedulers.io())      .map(userId -> getUserProfile(userId))      .sequential()      .subscribe(...) |

The *maxConcurrency* is specified explicitly in the parallel() method. The scheduler (or thread pool) where the parallel streams should be executed is specified by the runOn() operator. The sequential() operator is how you switch back to the “sequential” world. All operations between .parallel() and .sequential() are executed in parallel streams.

**36. Are memory leaks an issue when using RxJava? How would you protect from such?**

As a general good practice in programming, one must clean-up the used resources after they’re no longer needed. In the case of RxJava this means disposing your *Disposables* correctly. A common pattern is to keep adding all long-running operations from a screen in a CompositeDisposable and ensuring that’s clean-up when the screen is gone.

One thing to watch out for is creating new Disposables in the middle of a chain.

|  |  |
| --- | --- |
| 1  2  3  4  5 | compositeDisposable.add(          getUserFromNetwork(userId)         .doOnNext(user -> fireAndForgetOperation(user).subscribeOn(Schedulers.io()).subscribe())         .subscribe(...)  ) |

In the example above *fireAndForgetOperation(user).subscribeOn(Schedulers.io()).subscribe()* creates a new Disposable that won’t be automatically cleaned up if the *compositeDisposable* is disposed. A memory leak can occur for the duration of the *fireAndForgetOperation()* operation.

**37. Is RxJava working good in combination with Kotlin?**

Yes, they’re fully compatible. The built-in support for Lambdas and single abstract methods (SAM) makes RxJava streams even more concise and easy to read in Kotlin. There is also the [RxKotlin](https://github.com/ReactiveX/RxKotlin) library, which is a set of useful extension functions on top of RxJava.

**38. What’s a marble diagram?**

It’s a graphical representation of how RxJava operators work. In most cases it has a source stream, an operator and a resulting stream. Each stream is represented by a timeline with all emissions (“marbles”) and terminating events (completion / error).

**39. What’s a “pure” function?**

A pure function is one that doesn’t have any side effects and has stable output – e.g. the same input will always produce the same output. Working with pure functions makes code easier to reason, as there’s no hidden side effects and implicit dependencies between functions. Given the composable nature of RxJava operators, a very good combination is keeping each operation a highly isolated pure function – this way alterations of the stream are easier.

**40. Do you know any other reactive libraries? What’s the biggest differences, pros, cons?**

Even the [Java8](http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html) language supports a few “reactive” constructs ([CompletableFuture](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html), [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html), etc). Other popular libraries are [Project Reactor](http://projectreactor.io/) and [Akka](https://doc.akka.io/docs/akka/2.5.12/guide/index.html). Project Reactor is quite similar to RxJava2 in terms of constructs and API, but it targets Java8 which makes it unusable for some Android projects. A good high-level comparison between reactive libraries can be found [HERE](http://alexsderkach.io/comparing-java-8-rxjava-reactor/).

**10 RxJava Interview Questions With Sample Answers**

Here are some RxJava [interview questions](https://in.indeed.com/career-advice/interviewing/top-interview-questions-and-answers) you are likely to encounter in your upcoming interview for an Android developer position:

**1. What is RxJava?**

An employer might ask this question to assess your knowledge about the language and its purpose. When answering this question, it is beneficial for you to provide details about the language, such as how you can use it to write asynchronous applications.**Example:** *RxJava is the Java extension of Reactive X. The primary objective of RxJava is to provide functional reactive development. It also focuses on providing reactive programming to different languages. Reactive programming involves scenarios where computer programs react to data. The building blocks of RxJava are observers and observables. Observables emit items, whereas observers consume those items. Although the application programming interface (API) of RxJava might resemble Java8 Stream, it is more flexible and fluent, making it a powerful programming language.***Related:** [**What Is A Java Project? Explanation And Project Ideas**](https://in.indeed.com/career-advice/career-development/what-is-java-project)

**2. Name the different observables in RxJava.**

*RxJava uses two different observables, blocking and non-blocking. The non-blocking observables support execution and allow subscription during an event. Whereas the blocking observables focus on synchronous calls and do not allow a subscriber to unsubscribe in the middle of an event stream.***Related:** [**50 Essential Java Interview Programs**](https://in.indeed.com/career-advice/interviewing/java-interview-programs)

**3. What is the subject in RxJava?**

*A subject is a proxy available in ReactiveX that serves both as an observer and observable simultaneously. When the subject is an observer, it can subscribe to one or more observables, and when it is an observable, it passes through items and observes and emits new items. RxJava has four subjects, including publish, replay, behaviour and async subject. The published subject passes all events to observables, and new subscribers receive events only from the point of subscription.The objective of the async subject is to emit the last value of the observables, which occurs when the observables get completed. The behaviour subject emits the most recent items at the time of their subscription and all events after that. Conversely, the relay subject focuses on emitting every item of the observables irrespective of when users subscribe.***Related:** [**30 Java Multithreading Interview Questions And Answers**](https://in.indeed.com/career-advice/interviewing/java-multithreading-interview-questions)

**4. What are hot and cold observables?**

Observables represent different sources of data and are an important concept in RxJava. When answering, briefly outline the differences between the two observables.**Example:** *When the observable produces data itself, it is the cold observable. Conversely, when the observable produces data outside, it is the hot observable. While the cold observables start to run upon subscription and push values, hot observables produce values even before a subscription occurs.***Related:** [**15 Important Java Developer Skills (And How To Improve)**](https://in.indeed.com/career-advice/career-development/java-developer-skills)

**5. How can you transform a cold observable into a hot one?**

*There are two ways of converting a cold observable into a hot one. The first method uses publish() .connect(). The publish() converts cold observable to ConnectableObservable, which behaves like a hot one. Once triggered with the .connect() function, the observable publishes an event irrespective of whether there are subscribers. Another way to convert a cold observable into a hot one is by wrapping it using a subject. The subject first subscribes to a cold observable and exposes itself as a cold observable to other subscribers. The work gets performed irrespective of whether there are any subscribers.***Related:** [**15 Java 8 Interview Questions (With Example Answers)**](https://in.indeed.com/career-advice/interviewing/java-8-interview-questions)

**6. What is the purpose of the scheduler in RxJava?**

*A scheduler in RxJava controls the execution of observables on different threads. It can do this when the subscription starts. It also controls how the chain of operators might apply to different methods. Schedulers allow developers to specify when and where to execute different tasks related to operations of an observable chain. RxJava offers various schedulers, such as schedulers.io(), schedulers.computation(), schedulers.newThread() and schedulers.trampoline().The purpose of each of these schedulers is different. For instance, schedulers.io() creates and returns a scheduler intended for IO-bound work, whereas schedulers.newThread() creates and returns schedulers that create a new thread for each unit of work.*

**7. Outline the difference between subscribeOn() and observeOn().**

*The SubscribeOn() operator conveys the scheduler the source to perform the required work. As the observable chain has only one initial source, every observable has only one subscribeOn() operator. Conversely, ObserveOn() conveys the scheduler as the source to perform all downstream operations. As there can be many such operators, having multiple observeOn() operators in a single chain observable is ideal. SubscribeOn() impacts all upstream operators, whereas observeOn() impacts all downstream operators.If the developer specifies only subscribeOn(), all operators get executed on that thread. If the developer specifies only observeOn() operator, all operators on the current thread get executed. Only operators below the observeOn() get switched to threads specified by the observeOn().*

**8. Explain the difference between OnNext(), OnError() and OnCompleted().**

*OnNext() gets called when the observable emits a new item and you can perform some action on each item, whereas the OnError() interface method gets called when an error occurs and the data does not get completed. The OnComplete() interface method gets called when the observable emits all items.*

**9. What is the difference between switchMap(), flatMap() and concatMap()?**

*SwitchMap(), flatMap() and concatMap() are operators that help in modifying data emitted by an observable. The primary difference between concatMap() and flatMap() is in the order in which they emit items. While flatMap() can interleave items while emitting, meaning that it does not maintain the order of the emitted items, concatMap() can preserve the order. ConcatMap() waits for each observable to complete the work. SwitchMap() is another Rx transformation operator that unsubscribes from previous observables after emitting new ones, which means only the most recently emitted observables get emitted.*

**10. What is the difference between imperative programming and reactive programming?**

**Example:** *In reactive programming, observables emit data and send it back to the subscribers, which ensures data streams get pushed.*

*Conversely, in imperative programming, data gets pulled and users explicitly request data from the database. In most non-*[*JavaScript*](https://in.indeed.com/career-advice/career-development/what-is-javascript)*languages, imperative programming is the def-facto standard.*

**RxJava Interview Tips**

Here are a few tips you can follow when preparing for your RxJava interview:

* **Review your RxJava knowledge.** Ensure that you have the basic and advanced knowledge to handle various situations involving RxJava. Reviewing the basic RxJava concepts can help you answer confidently.
* **Research additional interview questions.** Prepare yourself by researching and practising with additional interview questions. This might be useful if you lack experience with RxJava.
* **Participate in a**[**mock interview**](https://in.indeed.com/career-advice/interviewing/mock-interview)**.** Ask your colleagues, friends and family members to help you conduct [mock interviews](https://in.indeed.com/career-advice/interviewing/mock-interview). Choose professionals who have some experience with RxJava to conduct your interview.

**10 RxJava Interview Questions and Answers in 2023**



As the world of software development continues to evolve, so too does the technology used to create it. RxJava is a popular library for reactive programming that has been gaining traction in recent years. In this blog, we will explore 10 of the most common RxJava interview questions and answers for 2023. We will provide a comprehensive overview of the topics, so that you can be well-prepared for any RxJava-related interview.

**1. What is the difference between Observable and Flowable in RxJava?**

The main difference between Observable and Flowable in RxJava is the way they handle backpressure.  
  
Observable is a stream of data that emits items to its subscribers. It does not support backpressure, meaning that it will emit items as fast as it can, regardless of the subscriber's ability to process them. This can lead to an OutOfMemoryError if the subscriber is not able to keep up with the rate of emission.  
  
Flowable is a stream of data that emits items to its subscribers. It supports backpressure, meaning that it will emit items at a rate that is determined by the subscriber's ability to process them. This helps to prevent OutOfMemoryErrors by ensuring that the subscriber is not overwhelmed with data.  
  
In summary, the main difference between Observable and Flowable in RxJava is that Flowable supports backpressure, while Observable does not.

**2. How do you handle errors in RxJava?**

When dealing with errors in RxJava, it is important to understand the different types of errors that can occur and how to handle them.  
  
The first type of error is an onError event. This is an event that is emitted when an exception is thrown within an Observable. When this occurs, the onError event will be emitted and the subscription will be terminated. To handle this type of error, you can use the onErrorReturn() operator to return a default value or the onErrorResumeNext() operator to return an alternate Observable.  
  
The second type of error is an onErrorResumeNext event. This is an event that is emitted when an exception is thrown within an Observable and the onErrorResumeNext operator is used. This operator will return an alternate Observable that will be used instead of the original Observable. To handle this type of error, you can use the onErrorResumeNext() operator to return an alternate Observable.  
  
The third type of error is an onExceptionResumeNext event. This is an event that is emitted when an exception is thrown within an Observable and the onExceptionResumeNext operator is used. This operator will return an alternate Observable that will be used instead of the original Observable. To handle this type of error, you can use the onExceptionResumeNext() operator to return an alternate Observable.  
  
**Finally, you can also use the catch() operator to catch any exceptions that are thrown within an Observable. This operator will catch any exceptions that are thrown and will return an alternate Observable**. To handle this type of error, you can use the catch() operator to return an alternate Observable.  
  
By understanding the different types of errors that can occur in RxJava and how to handle them, you can ensure that your code is robust and reliable.

**3. What is the purpose of the Schedulers in RxJava?**

The purpose of Schedulers in RxJava is to provide a way to control the concurrency of an Observable. Schedulers allow you to specify which thread an Observable should run on, and how many threads it should use. This is important for ensuring that your application runs efficiently and that it does not overwhelm the system with too many concurrent tasks. Schedulers also provide a way to control the order in which tasks are executed, which is important for ensuring that tasks are executed in the correct order. Finally, Schedulers provide a way to control the backpressure of an Observable, which is important for ensuring that an Observable does not emit too many items too quickly.

**4. How do you handle backpressure in RxJava?**

Backpressure is an important concept to understand when working with RxJava. It is a mechanism that allows the producer of data to control the rate at which the consumer of data can process it.  
  
In RxJava, backpressure is handled by using the Flowable class. This class is designed to handle backpressure by allowing the producer to control the rate at which the consumer can process the data. The Flowable class provides operators such as onBackpressureBuffer(), onBackpressureDrop(), and onBackpressureLatest() which allow the producer to control the rate at which the consumer can process the data.  
  
The onBackpressureBuffer() operator allows the producer to buffer the data until the consumer is ready to process it. The onBackpressureDrop() operator allows the producer to drop the data if the consumer is not ready to process it. The onBackpressureLatest() operator allows the producer to keep the latest data and discard the older data if the consumer is not ready to process it.  
  
By using these operators, the producer can control the rate at which the consumer can process the data and handle backpressure in RxJava.

**5. What is the difference between a Subject and an Observer in RxJava?**

The Subject in RxJava is a special type of Observable that allows values to be multicasted to many Observers. It is typically used to bridge the gap between a cold Observable and multiple Observers.  
  
An Observer is an object that implements the Observer interface and is used to receive notifications from an Observable. It is typically used to subscribe to an Observable and receive notifications when the Observable emits new values.  
  
In summary, a Subject is an Observable that can multicast values to multiple Observers, while an Observer is an object that subscribes to an Observable and receives notifications when the Observable emits new values.

**6. How do you create an Observable from a list of items in RxJava?**

Creating an Observable from a list of items in RxJava is a fairly straightforward process. First, you need to create an Observable object using the Observable.from() method. This method takes a list of items as its parameter and returns an Observable object.  
  
Once you have the Observable object, you can then use the subscribe() method to subscribe to the Observable. This method takes an Observer object as its parameter, which is used to receive notifications from the Observable.  
  
Finally, you can use the onNext() method of the Observer object to receive the items from the list one by one. This method takes an item from the list as its parameter and passes it to the Observer.  
  
In summary, creating an Observable from a list of items in RxJava involves creating an Observable object using the Observable.from() method, subscribing to the Observable using the subscribe() method, and receiving the items from the list one by one using the onNext() method of the Observer object.

**7. What is the difference between a Hot and Cold Observable in RxJava?**

The main difference between a Hot and Cold Observable in RxJava is that a Hot Observable can begin emitting items as soon as it is created, while a Cold Observable will only begin emitting items when it is subscribed to.  
  
A Hot Observable is useful when you want to share a single source of data among multiple subscribers. For example, a Hot Observable could be used to share a stream of stock prices among multiple subscribers.  
  
A Cold Observable is useful when you want to ensure that each subscriber receives the same data. For example, a Cold Observable could be used to ensure that each subscriber receives the same stream of stock prices.  
  
In addition, Hot Observables can emit items even if there are no subscribers, while Cold Observables will not emit any items until they are subscribed to. This means that Hot Observables can potentially lose data if there are no subscribers to receive the data.

**8. How do you create an Observable from a single item in RxJava?**

To create an Observable from a single item in RxJava, you can use the just() operator. This operator takes a single item and emits it as an Observable. For example, if you wanted to create an Observable from a single String item, you could do the following:  
  
Observable singleItemObservable = Observable.just("My String Item");  
  
The just() operator is a convenient way to create an Observable from a single item. It is also possible to create an Observable from a single item using the create() operator. This operator takes an OnSubscribe object as a parameter, which is responsible for emitting the item. For example, if you wanted to create an Observable from a single String item, you could do the following:  
  
Observable singleItemObservable = Observable.create(subscriber -> { subscriber.onNext("My String Item"); subscriber.onCompleted(); });  
  
The create() operator is more verbose than the just() operator, but it gives you more control over the emission of the item.

**9. What is the purpose of the RxJava operators?**

The purpose of RxJava operators is to manipulate and transform the data emitted by Observables. Operators are used to filter, combine, and transform the data emitted by Observables into the desired form. They are also used to create new Observables from existing ones.  
  
RxJava operators can be divided into two categories: intermediate operators and terminal operators. Intermediate operators are used to transform the data emitted by an Observable and can be chained together to create complex data transformations. Terminal operators are used to consume the data emitted by an Observable and can be used to trigger side effects or to terminate the Observable.  
  
RxJava operators are essential for creating reactive applications as they allow developers to manipulate and transform data in a declarative manner. They are also used to create complex data flows and to handle errors and exceptions.

**10. How do you debug an RxJava application?**

Debugging an RxJava application requires a few steps.  
  
First, you should identify the source of the issue. This can be done by examining the stack trace and looking for any errors or exceptions that may have been thrown. You can also use logging statements to help pinpoint the source of the issue.  
  
Once you have identified the source of the issue, you can start to debug the application. This can be done by using the RxJava debugging tools such as RxJavaPlugins. These tools allow you to observe the flow of data through the application and identify any issues.  
  
You can also use the RxJava debugger to step through the code and identify any issues. This can be done by setting breakpoints and stepping through the code line by line.  
  
Finally, you can use the RxJava TestObserver to test the application. This allows you to observe the data that is being emitted from the application and identify any issues.  
  
By following these steps, you should be able to debug your RxJava application and identify any issues.