|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | RxJava or Reactive Programming | <chare> | 1 | <pext> | When there is streams of data is emmiting out from the data source  and to handle this kind of scenario’s we have RxJava library.  RxJava is reactive programming library for dealing with asynchronous  and event based programming by using Observable sequences.  It’s main building blocks are Observables, Operators, Observer  that also called as triple O’s. | </end> |
| <hitle> | RxAndroid | <chare> | 1 | <pext> | Rx Android is an support of RxJava for Android development  and it introduced the Main Thread require for Android UI. | </end> |
| <hitle> | Observable RxJava | <chare> | 1 | <pext> | <starting>Observable RxJava</heading> Observable represents any object that can get data from a data source  and whose state may be of interest in a way  that other objects may register an interest. In Observable, there are two types: Cold and Hot Observables. Cold Observables will perform work and subsequently emit items only once  someone has subscribed, whereas Hot Observables will perform work  and emit items regardless of observers or not. Also, there are two types: 1. Non-Blocking – asynchronous execution is supported and is allowed  to unsubscribe at any point in the event stream. 2. Blocking – all onNext observer calls will be synchronous, and it is not possible to unsubscribe in the middle of an event stream. We can always convert an Observable into a Blocking Observable, using the method toBlocking: </ending> | </end> |
| <hitle> | Observable types | <chare> | 1 | <pext> | <starting>Observable types</heading> In Observable, there are two types: Cold and Hot Observables. Cold Observables will perform work and subsequently emit items only once  someone has subscribed, whereas Hot Observables will perform work  and emit items regardless of observers or not. Also, there are two types: 1. Non-Blocking – asynchronous execution is supported and is allowed  to unsubscribe at any point in the event stream. 2. Blocking – all onNext observer calls will be synchronous, and it is not possible to unsubscribe in the middle of an event stream. We can always convert an Observable into a Blocking Observable, using the method toBlocking: </ending> | </end> |
| <hitle> | COLD, HOT observable type | <chare> | 1 | <pext> | ------------COLD observable---------------  Cold Observables only emit values on subscription.  The data is produced inside the Observable.  Cold observables are created multiple times  It looks like Youtube which plays movie only when there’s visitor.  Using Cold Observable we can implement Watch solution  to get what time now.  -----------Hot Observable----------------  Hot Observables push events or values  even when no one is subscribed to them.  The data is produced outside the Observable.  In case of Hot Observable, Observers can come and go,  but the stream is created once and just goes on.  It looks like Movie Theater which plays movie at exact showtime  no matter when visitors come.  Using Hot Observable we can implement Timer solution  to measure how long it will take for doing something. | </end> |
| <hitle> | COLD vs HOT observables | <chare> | 1 | <pext> | They start emitting items differently. 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. | </end> |
| <hitle> | transform COLD/HOT to another | <chare> | 1 | <pext> | ------------COLD to HOT----------------  1. COLD to HOT One way to make a Cold observable to 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. -------------HOT to COLD-------------------  2. HOT to COLD 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. 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. | </end> |
| <hitle> | observer RxJava | <chare> | 1 | <pext> | <starting>observer RxJava</heading> An observer subscribes to an Observable sequence. The sequence sends items to the observer one at a time.  onNext() -> the next upcoming data will be received here.  onError() -> if got error in observing data  onComplete() -> if all data received from Observable  onSubscribe() -> Once observer got subscribed to the observable  that the stream of data flow starts.  </ending> | </end> |
| <hitle> | Scheduler RxJava | <chare> | 1 | <pext> | <starting>Scheduler RxJava</heading> Schedulers are used to switch execution to a different thread. RxJava is single-threaded by default, i.e., all operations are executed on a single thread. Also used as an abstraction overtime concept for time-sensitive operations  such as delay(), buffer(), timeout(), window(), etc. </ending> | </end> |
| <hitle> | flatMap(), switchMap(), concatMap() | <chare> | 1 | <pext> | <starting>flatMap(), switchMap(), concatMap()</heading> 1. flatMap() is used to split chain to multiple intermediary streams,  and results of which are flattened to a single stream. Emissions of these intermediary streams are directly propagated  to the main stream in any order. 2. switchMap() is similar to flatMap(),  but whenever a new item is being emitted from a source,  all the intermediary streams generated are terminated,  and hence only the latest intermediary stream remains active. 3. concatMap() also works similar to flatMap(),  with an exception such that intermediary streams are activated  based on the order appearance. </ending> | </end> |
| <hitle> | onNext(), onComplete(), onError() | <chare> | 1 | <pext> | <starting>onNext(), onComplete(), onError()</heading> 1. OnNext is called on our observer each time a new event is published  to the attached Observable. This is the method where we'll perform some action on each event. 2. OnCompleted is called when the sequence of events  associated with an Observable is complete, indicating that  we should not expect any more onNext calls on our observer. 3. OnError is called when an unhandled exception is thrown  during the RxJava framework code or our event handling code. These are the callbacks an Observable / Flowable will receive. onNext 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.  onNext() – from zero between infinite number of times onComplete() – maximum once per stream onError() – maximum once per stream </ending> | </end> |
| <hitle> | Observable chain | <chare> | 1 | <pext> | <starting>Observable chain</heading> List of operations or transformations performed in  between the source and end subscriber. One of the examples, it to emit User objects, filtering out the admin users  and checking for authentication of users, and finally map full name. </ending> | </end> |
| <hitle> | custom operators in RxJava | <chare> | 1 | <pext> | <starting>custom operators in RxJava</heading> Users can create custom operators in RxJava, and this is highly recommended  to reuse existing or any combination. It is tricky to implement a new operator as it would lead to too many of errors,  such as thread safety, API breaking, etc. </ending> | </end> |
| <hitle> | error handling operators in RxJava | <chare> | 1 | <pext> | <starting>error handling operators in RxJava</heading> We have two categories of such operators, one for side effects only  and the other for handle error and continue. doOnError(…), onErrorReturn(…), onErrorResumeNext(…) are  some of the error handling operators in RxJava. </ending> | </end> |
| <hitle> | Marble Diagram | <chare> | 1 | <pext> | <starting>Marble Diagram</heading> It is a graphical representation of how the RxJava operators work. In most of the cases, RxJava has a source stream, operator,  and resulting stream. Each of the streams is represented by a timeline with all emissions,  known as marbles and terminating events. </ending> | </end> |
| <hitle> | Backpressure | <chare> | 1 | <pext> | <starting>Backpressure</heading> Backpressure is the inability of a subscriber to handle incoming events in time. Backpressure can occur when the producer of events  is much faster than consumers; if not will error stream.  The first important thing is  to choose the correct RxJava construct 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. </ending> | </end> |
| <hitle> | Subject RxJava | <chare> | 1 | <pext> | <starting>Subject RxJava</heading> Subject means both subscriber and observer at the same time. With subjects in RxJava, users can transform cold observables to hot ones. They are also used to introduce some type of local and temporary caching  of the stream. Also, help in transforming non-reactive code to reactive  if the user does not find any operator for use case creation. Types of Subjects in RxJava: 1. PublishSubject: It passes incoming events to all subscribers. New subscribers will receive events only from the point of subscription. 2. BehaviourSubject: Similar to publish subject, but each new subscriber will receive the latest value of the stream, i.e., the default value. Here, the default value of the stream provides a good user experience. 3. AsyncSubject: It emits only the last value of Observable, and that too only after the source observable completes emitting. 4. ReplaySubject: Each subscriber will receive all the events emitted by the source, regardless of at which point it is subscribed. If Observable emits too many items, they need to be in-memory cache. </ending> | </end> |
| <hitle> | Subject types RxJava | <chare> | 1 | <pext> | <starting>Subject types RxJava</heading> 1. PublishSubject: It passes incoming events to all subscribers. New subscribers will receive events only from the point of subscription. 2. BehaviourSubject: Similar to publish subject, but each new subscriber  will receive the latest value of the stream, i.e., the default value. Here, the default value of the stream provides a good user experience. 3. AsyncSubject: It emits only the last value of Observable,  and that too only after the source observable completes emitting. 4. ReplaySubject: Each subscriber will receive all the events  emitted by the source, regardless of at which point it is subscribed. If Observable emits too many items, they need to be in-memory cache. </ending> | </end> |
| <hitle> | Parallelism RxJava | <chare> | 1 | <pext> | <starting>Parallelism RxJava</heading> RxJava will support parallelism, and this is achieved in two ways;  using flatMap() operator, each stream inside flatMap() should subscribe  background thread. Using ParallelFlowable, it provides easier and explicit API  to achieve parallelism. a) Using the flatMap() operator. Each inner stream inside the flatMap() operation should subscribeOn()  a background thread (preferably Schedulers.io()). b) Although the above pattern works correctly to achieve parallelism,  in recent versions of RxJava a better construct was introduced  – the ParallelFlowable. It provides an easier and more explicit API to achieve parallelism. </ending> | </end> |
| <hitle> | Transformer | <chare> | 1 | <pext> | <starting>Transformer</heading> 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()). </ending> | </end> |
| <hitle> | Non-Blocking | <chare> | 1 | <pext> | <starting>Non-Blocking</heading> An algorithm is considered to be non-blocking  if threads competing for resources do not have execution postponed  due to mutual exclusion of protecting the resource. </ending> | </end> |
| <hitle> | Elasticity in RxJava | <chare> | 1 | <pext> | <starting>Elasticity in RxJava</heading> It means that the throughput of the system scales up or down automatically  to meet demand as a resource is proportionally added/ removed. Elasticity, therefore, builds up Scalability  and expands by adding the notion of automatic resource management. </ending> | </end> |
| <hitle> | Operators RxJava | <chare> | 1 | <pext> | <starting>Operators RxJava</heading> An operator is a function  that takes one Observable (the source) as its first argument  and returns another Observable (the destination). Then for every item that the source observable emits,  it will apply a function to that item,  and then emit the result on the destination Observable. Operators can be chained together  to create complex data flows that filter event based on certain criteria. Multiple operators can be applied to the same observable. It is not difficult to get into a situation  in which an Observable is emitting items faster than an operator  or observer can consume them. You can read more about back-pressure here. </ending> | </end> |
| <hitle> | Map RxJava | <chare> | 1 | <pext> | <starting>Map RxJava</heading> The map operator transforms items emitted by an Observable by applying a function to each item. </ending> | </end> |
| <hitle> | Scan RxJava | <chare> | 1 | <pext> | <starting>Scan RxJava</heading> The scan operator applies a function to each item emitted by an Observable sequentially and emits each successive value. </ending> | </end> |
| <hitle> | GroupBy RxJava | <chare> | 1 | <pext> | <starting>GroupBy RxJava</heading> GroupBy operator allows us to classify the events in the input Observable into output categories. </ending> | </end> |
| <hitle> | Filter RxJava | <chare> | 1 | <pext> | <starting>Filter RxJava</heading> The operator filter emits only those items from an observable that pass a predicate test. </ending> | </end> |
| <hitle> | Conditional Operators RxJava | <chare> | 1 | <pext> | <starting>Conditional Operators RxJava</heading> 1. DefaultIfEmpty emits item from the source Observable, or a default item if the source Observable is empty. 2. TakeWhile operator discards items emitted by an Observable after a specified condition becomes false. 3. Of course, there more others operators that could cover our needs like Contain, SkipWhile, SkipUntil, TakeUntil, etc. </ending> | </end> |
| <hitle> | Connectable Observables RxJava | <chare> | 1 | <pext> | <starting>Connectable Observables RxJava</heading> A ConnectableObservable resembles an ordinary Observable, except that it doesn't begin emitting items when it is subscribed to, but only when the connect operator is applied to it. In this way, we can wait for all intended observers to subscribe to the Observable before the Observable begins emitting items. </ending> | </end> |
| <hitle> | Single RxJava | <chare> | 1 | <pext> | <starting>Single RxJava</heading>  Single type of Observable is used when you have to return a single item.  Item will be returned when onSuccess will be called,  If there is any error occurs then it will call onError. Single is like an Observable who, instead of emitting a series of values,  emits one value or an error notification. With this source of data, we can only use two methods to subscribe: OnSuccess returns a Single that also calls a method we specify. OnError also returns a Single that immediately notifies subscribers of an error. </ending> | </end> |
| <hitle> | observeOn() vs subscribeOn() | <chare> | 1 | <pext> | <starting>observeOn() vs subscribeOn()</heading> 1. 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. 2. 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. </ending> | </end> |
| <hitle> | multiple subscribeOn() | <chare> | 1 | <pext> | <starting>multiple subscribeOn()</heading> 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) </ending> | </end> |
| <hitle> | multiple observeOn() | <chare> | 1 | <pext> | <starting>multiple observeOn()</heading> 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. </ending> | </end> |
| <hitle> | map() vs flatMap() RxJava | <chare> | 1 | <pext> | <starting>map() vs flatMap() RxJava</heading> 1. The map() operator maps from a concrete ValueA to a concrete ValueB  (e.g. from an Int -> String, or User -> String). 2. 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. </ending> | </end> |
| <hitle> | test *observable chains RxJava* | <chare> | 1 | <pext> | <starting>test observable chains RxJava</heading> 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. </ending> | </end> |
| <hitle> | multiple errors RxJava | <chare> | 1 | <pext> | <starting>multiple errors RxJava</heading> All unhandled errors (via any of the 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. </ending> | </end> |
| <hitle> | RxJava1 vs RxJava2 | <chare> | 1 | <pext> | <starting>RxJava1 vs RxJava2</heading> 1. 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 is handy for places  where you’re not sure if the data is null-safe. 2. 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. 3. 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. </ending> | </end> |
| <hitle> | Completable RxJava | <chare> | 1 | <pext> | <starting>Completable RxJava</heading> maps an operation that either completes without returning a value  (onComplete()) or errors out (onError(throwable)). </ending> | </end> |
| <hitle> | Maybe RxJava | <chare> | 1 | <pext> | Maybe type of Observable is used when you have to return  a single optional value as this is mutually exclusive with error,  it means it will call either success or error and then onComplete. Maybe – has 3 options – returns a value successfully (onSuccess(value)),  completes successfully without any value (onComplete()) or errors out (onError(throwable)). | </end> |
| <hitle> | Observable(keyword) RxJava | <chare> | 1 | <pext> | <starting>Observable(keyword) RxJava</heading> 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 </ending> | </end> |
| <hitle> | Flowable RxJava | <chare> | 1 | <pext> | <starting>Flowable RxJava</heading> Flowable – represents a stream of events that emits zero  to many events (onNext(value)), then either completes (onComplete())  or errors out (onError(throwable)). However it DOES support backpressure. </ending> | </end> |
| <hitle> | RxJava constructs | <chare> | 1 | <pext> | <starting>RxJava constructs</heading> 1. 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)). 2. Maybe – has 3 options – returns a value successfully (onSuccess(value)),  completes successfully without any value (onComplete()) or errors out  (onError(throwable)). 3. 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. 4. Flowable – like an Observable, however it DOES support backpressure. </ending> | </end> |
| <hitle> | Subject vs RxRelay | <chare> | 1 | <pext> | <starting>Subject vs RxRelay</heading> 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. </ending> | </end> |
| <hitle> | pure function RxJava | <chare> | 1 | <pext> | <starting>pure function RxJava</heading> 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. </ending> | </end> |
| <hitle> | memory leak RxJava | <chare> | 1 | <pext> | <starting>memory leak RxJava</heading> 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. </ending> | </end> |
| <hitle> | Types of Observables in RxJava | <chare> | 1 | <pext> | 1. Observable  This is the simplest Observable which can emit more than one value.  Example use-case: Let’s say downloading a file and we have to push  the current status of download percentage.  Here, we will have to emit more than one value.  2. Single  Single is used when the Observable has to emit only one value  like a response from a network call.  3. Maybe  Maybe type of Observable is used when we have to return  a single optional value as this is mutually exclusive with error,  it means it will call either success or error and then onComplete.  4. Flowable  Flowable comes into picture when there is a case  that the Observable is emitting huge numbers of values  that can’t be consumed by the Observer.  In this case, the Observable needs to skip some values  on the basis of some strategy else it will throw an exception.  The Flowable Observable handles the exception with a strategy.  The strategy is called BackPressureStrategy  and the exception is called MissingBackPressureException.  5. Completable  Completable is used when the Observable has to do some task  without emitting a value. | </end> |
| <hitle> | Types of Operators in RxJava | <chare> | 1 | <pext> | Mostly used operators are:  1. Just  convert an object or a set of objects into an Observable  that emits that or those objects.  2. Map  transform the items emitted by an Observable  by applying a function to each item  3. FlatMap  transform the items emitted by an Observable into Observables,  then flatten the emissions from those into a single Observable.  4. take  emit only the first n items emitted by an Observable.  5. debounce  only emit an item from an Observable if a particular timespan  has passed without it emitting another item.  6. buffer  periodically gather items from an Observable into bundles  and emit these bundles rather than emitting the items one at a time.  7. repeat  create an Observable that emits a particular item  or sequence of items repeatedly.  8. zip  combine the emissions of multiple Observables together  via a specified function and emit single items for each combination  based on the results of this function.  9. concat  emit the emissions from two or more Observables  without interleaving them.  10. merge  combine multiple Observables into one by merging their emissions.  11. filter  emit only the items from an Observable that pass a predicate test.  ----------Chaining Operators-------------------  Most operators operate on an Observable and return an Observable.  This allows to apply these operators one after the other, in a chain.  Each operator in the chain modifies the Observable  that results from the operation of the previous operator.  A chain of Observable operators do not operate independently  on the original Observable that originates the chain,  but they operate in turn, each one operating on the Observable  generated by the operator immediately previous in the chain.  --------------Creating Observables-----------------  Operators that originate new Observables.  1. Create — create an Observable from scratch by calling observer methods programmatically  2. Defer — do not create the Observable until the observer subscribes, and create a fresh Observable for each observer  3. Empty/Never/Throw — create Observables that have very precise and limited behavior  4. From — convert some other object or data structure into an Observable  5. Interval — create an Observable that emits a sequence of integers spaced by a particular time interval  6. Just — convert an object or a set of objects into an Observable that emits that or those objects  7. Range — create an Observable that emits a range of sequential integers  8. Repeat — create an Observable that emits a particular item or sequence of items repeatedly  9. Start — create an Observable that emits the return value of a function  10. Timer — create an Observable that emits a single item after a given delay  -------------Transforming Observables-------------------  Operators that transform items that are emitted by an Observable.  1. Buffer — periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time  2. FlatMap — transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable  3. GroupBy — divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key  4. Map — transform the items emitted by an Observable by applying a function to each item  5. Scan — apply a function to each item emitted by an Observable, sequentially, and emit each successive value  6. Window — periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time  Filtering Observables  Operators that selectively emit items from a source Observable.  Debounce — only emit an item from an Observable if a particular timespan has passed without it emitting another item  Distinct — suppress duplicate items emitted by an Observable  ElementAt — emit only item n emitted by an Observable  Filter — emit only those items from an Observable that pass a predicate test  First — emit only the first item, or the first item that meets a condition, from an Observable  IgnoreElements — do not emit any items from an Observable but mirror its termination notification  Last — emit only the last item emitted by an Observable  Sample — emit the most recent item emitted by an Observable within periodic time intervals  Skip — suppress the first n items emitted by an Observable  SkipLast — suppress the last n items emitted by an Observable  Take — emit only the first n items emitted by an Observable  TakeLast — emit only the last n items emitted by an Observable  ----------------Combining Observables----------------  Operators that work with multiple source Observables to create a single Observable  1. And/Then/When — combine sets of items emitted by two or more Observables by means of Pattern and Plan intermediaries  2. CombineLatest — when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function  3. Join — combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable  4. Merge — combine multiple Observables into one by merging their emissions  5. StartWith — emit a specified sequence of items before beginning to emit the items from the source Observable  6. Switch — convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables  7. Zip — combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function  ----------------Error Handling Operators----------------  Operators that help to recover from error notifications from an Observable  1. Catch — recover from an onError notification by continuing the sequence without error  2. Retry — if a source Observable sends an onError notification, resubscribe to it in the hopes that it will complete without error  ----------------Observable Utility Operators----------------  A toolbox of useful Operators for working with Observables  1. Delay — shift the emissions from an Observable forward in time by a particular amount  2. Do — register an action to take upon a variety of Observable lifecycle events  3. Materialize/Dematerialize — represent both the items emitted and the notifications sent as emitted items, or reverse this process  4. ObserveOn — specify the scheduler on which an observer will observe this Observable  5. Serialize — force an Observable to make serialized calls and to be well-behaved  6. Subscribe — operate upon the emissions and notifications from an Observable  7. SubscribeOn — specify the scheduler an Observable should use when it is subscribed to  8. TimeInterval — convert an Observable that emits items into one that emits indications of the amount of time elapsed between those emissions  9. Timeout — mirror the source Observable, but issue an error notification if a particular period of time elapses without any emitted items  10. Timestamp — attach a timestamp to each item emitted by an Observable  11. Using — create a disposable resource that has the same lifespan as the Observable  ----------------Conditional and Boolean Operators----------------  Operators that evaluate one or more Observables or items emitted by Observables  1. All — determine whether all items emitted by an Observable meet some criteria  2. Amb — given two or more source Observables, emit all of the items from only the first of these Observables to emit an item  3. Contains — determine whether an Observable emits a particular item or not  4. DefaultIfEmpty — emit items from the source Observable, or a default item if the source Observable emits nothing  5. SequenceEqual — determine whether two Observables emit the same sequence of items  6. SkipUntil — discard items emitted by an Observable until a second Observable emits an item  7. SkipWhile — discard items emitted by an Observable until a specified condition becomes false  8. TakeUntil — discard items emitted by an Observable after a second Observable emits an item or terminates  9. TakeWhile — discard items emitted by an Observable after a specified condition becomes false  ----------------Mathematical and Aggregate Operators----------------  Operators that operate on the entire sequence of items emitted by an Observable  1. Average — calculates the average of numbers emitted by an Observable and emits this average  2. Concat — emit the emissions from two or more Observables without interleaving them  3. Count — count the number of items emitted by the source Observable and emit only this value  4. Max — determine, and emit, the maximum-valued item emitted by an Observable  5. Min — determine, and emit, the minimum-valued item emitted by an Observable  6. Reduce — apply a function to each item emitted by an Observable, sequentially, and emit the final value  7. Sum — calculate the sum of numbers emitted by an Observable and emit this sum  ----------------Backpressure Operators----------------  backpressure operators — strategies for coping with Observables that produce items more rapidly than their observers consume them  ----------------Connectable Observable Operators----------------  Specialty Observables that have more precisely-controlled subscription dynamics  1. Connect — instruct a connectable Observable to begin emitting items to its subscribers  2. Publish — convert an ordinary Observable into a connectable Observable  3. RefCount — make a Connectable Observable behave like an ordinary Observable  4. Replay — ensure that all observers see the same sequence of emitted items, even if they subscribe after the Observable has begun emitting items  ----------------Operators to Convert Observables----------------  To — convert an Observable into another object or data structure | </end> |
| <hitle> | map() vs flatMap() in RxJava | <chare> | 1 | <pext> | 1. Map transforms the items emitted by an Observable  by applying a function to each item.  2. FlatMap transforms the items emitted  by an Observable into Observables. | </end> |
| <hitle> | Whentouse operators RxJava | <chare> | 1 | <pext> | 1. I want to create a new Observable  1-1. that emits a particular item: \*Just\*  1-1-1. that was returned from a function called at subscribe-time \*Start\*  1-1-2. that was returned from an Action, Callable, Runnable, or something of that sort, called at subscribe-time \*From\*  1-1-3. after a specified delay \*Timer\*  1-2. that pulls its emissions from a particular Array, Iterable, or something like that \*From\*  1-3. by retrieving it from a Future \*Start\*  1-4. that obtains its sequence from a Future \*From\*  1-5. that emits a sequence of items repeatedly \*Repeat\*  1-6. from scratch, with custom logic \*Create\*  1-7. for each observer that subscribes \*Defer\*  1-8. that emits a sequence of integers \*Range\*  1-8-1. at particular intervals of time \*Interval\*  1-8-1-1. after a specified delay \*Timer\*  1-9. that completes without emitting items \*Empty\*  1-10. that does nothing at all \*Never\*  2. I want to create an Observable by combining other Observables  2-1. and emitting all of the items from all of the Observables in whatever order they are received \*Merge\*  2-2. and emitting all of the items from all of the Observables, one Observable at a time \*Concat\*  2-3. by combining the items from two or more Observables sequentially to come up with new items to emit  2-3-1. whenever each of the Observables has emitted a new item \*Zip\*  2-3-2. whenever any of the Observables has emitted a new item \*CombineLatest\*  2-3-3. whenever an item is emitted by one Observable in a window defined by an item emitted by another \*Join\*  2-3-4. by means of Pattern and Plan intermediaries \*And/Then/When\*  2-4. and emitting the items from only the most-recently emitted of those Observables \*Switch\*  3. I want to emit the items from an Observable after transforming them  3-1. one at a time with a function \*Map\*  3-2. by emitting all of the items emitted by corresponding Observables \*FlatMap\*  3-2-1. one Observable at a time, in the order they are emitted \*ConcatMap\*  3-3. based on all of the items that preceded them \*Scan\*  3-4. by attaching a timestamp to them \*Timestamp\*  3-5. into an indicator of the amount of time that lapsed before the emission of the item \*TimeInterval\*  4. I want to shift the items emitted by an Observable forward in time before reemitting them \*Delay\*  5. I want to transform items and notifications from an Observable into items and reemit them  5-1. by wrapping them in Notification objects \*Materialize\*  5-2. which I can then unwrap again with \*Dematerialize\*  6. I want to ignore all items emitted by an Observable and only pass along its completed/error notification \*IgnoreElements\*  7. I want to mirror an Observable but prefix items to its sequence \*StartWith\*  7-1. only if its sequence is empty \*DefaultIfEmpty\*  8. I want to collect items from an Observable and reemit them as buffers of items \*Buffer\*  8-1. containing only the last items emitted \*TakeLastBuffer\*  9. I want to split one Observable into multiple Observables \*Window\*  9-1. so that similar items end up on the same Observable \*GroupBy\*  10. I want to retrieve a particular item emitted by an Observable:  10-1. the last item emitted before it completed \*Last\*  10-2. the sole item it emitted \*Single\*  10-3. the first item it emitted \*First\*  11. I want to reemit only certain items from an Observable  11-1. by filtering out those that do not match some predicate \*Filter\*  11-2. that is, only the first item \*First\*  11-3. that is, only the first items \*Take\*  11-4. that is, only the last item \*Last\*  11-5. that is, only item n \*ElementAt\*  11-6. that is, only those items after the first items  11-6-1. that is, after the first n items \*Skip\*  11-6-2. that is, until one of those items matches a predicate \*SkipWhile\*  11-6-3. that is, after an initial period of time \*Skip\*  11-6-4. that is, after a second Observable emits an item \*SkipUntil\*  11-7. that is, those items except the last items  11-7-1. that is, except the last n items \*SkipLast\*  11-7-2. that is, until one of those items matches a predicate \*TakeWhile\*  11-7-3. that is, except items emitted during a period of time before the source completes \*SkipLast\*  11-7-4. that is, except items emitted after a second Observable emits an item \*TakeUntil\*  11-8. by sampling the Observable periodically \*Sample\*  11-9. by only emitting items that are not followed by other items within some duration \*Debounce\*  11-10. by suppressing items that are duplicates of already-emitted items \*Distinct\*  11-10-1. if they immediately follow the item they are duplicates of \*DistinctUntilChanged\*  11-11. by delaying my subscription to it for some time after it begins emitting items \*DelaySubscription\*  12. I want to reemit items from an Observable only on condition that it was the first of a collection of Observables to emit an item \*Amb\*  13. I want to evaluate the entire sequence of items emitted by an Observable  13-1. and emit a single boolean indicating if all of the items pass some test \*All\*  13-2. and emit a single boolean indicating if the Observable emitted any item (that passes some test) \*Contains\*  13-3. and emit a single boolean indicating if the Observable emitted no items \*IsEmpty\*  13-4. and emit a single boolean indicating if the sequence is identical to one emitted by a second Observable \*SequenceEqual\*  13-5. and emit the average of all of their values \*Average\*  13-6. and emit the sum of all of their values \*Sum\*  13-7. and emit a number indicating how many items were in the sequence \*Count\*  13-8. and emit the item with the maximum value \*Max\*  13-9. and emit the item with the minimum value \*Min\*  13-10. by applying an aggregation function to each item in turn and emitting the result \*Scan\*  14. I want to convert the entire sequence of items emitted by an Observable into some other data structure \*To\*  15. I want an operator to operate on a particular Scheduler \*SubscribeOn\*  15-1. when it notifies observers \*ObserveOn\*  16. I want an Observable to invoke a particular action when certain events occur \*Do\*  17. I want an Observable that will notify observers of an error \*Throw\*  17-1. if a specified period of time elapses without it emitting an item \*Timeout\*  18. I want an Observable to recover gracefully  18-1. from a timeout by switching to a backup Observable \*Timeout\*  18-2. from an upstream error notification \*Catch\*  18-2-1. by attempting to resubscribe to the upstream Observable \*Retry\*  19. I want to create a resource that has the same lifespan as the Observable \*Using\*  20. I want to subscribe to an Observable and receive a Future that blocks until the Observable completes \*Start\*  21. I want an Observable that does not start emitting items to subscribers until asked \*Publish\*  21-1. and then only emits the last item in its sequence \*PublishLast\*  21-2. and then emits the complete sequence, even to those who subscribe after the sequence has begun \*Replay\*  21-3. but I want it to go away once all of its subscribers unsubscribe \*RefCount\*  21-4. and then I want to ask it to start \*Connect\* | </end> |