Reactive Programming :

* Subset / Special Case of event-driven / asynchronous programming
* Data flow Programming

Reactive Programming Pillars :

* Asynchronous Data processing
* Non-Blokcing
* Functional Style / Declarative

Reactive Streams Implementations :

* Akka
* rxJava2
* Reactor \*

Publisher / Subscriber Communication :

1. Subscriber wants to Connect to publisher ( subscriber wants update )

Publisher

Subscriber

1. Publisher Calls onSubscribe method

Publisher creates a subscribtion object and pass to subscribe

Publisher

Subscriber

1. Subscription

In this point the relationship stablished ( through the subscription object )

Now the subscriber can request for data or cancel if it wants

Subscriber

Publisher

1. Publisher pushesdata via onNext ( backpressure concept )

Subscriber

Publisher

1. When the publisher has no more item to emmit or if it has already emmited all the

Items request the publisher can call onComplete method to notify the subscriber that his job is done

Publisher

Subscriber

Done

1. onError -> when publisher proccessing the request there might be some unexpected happens – so the publisher can pass the error details to the subscriber via onError call

so in this case will not be emitting any more items to the subscriber

Publisher

Subscriber

onError

**Project Reactor :**

Reactive Streams is a specification and Project Reactor is a library – (Something like hibernate) – an implementaion for implementing async/non-blocking

Reactor Publisher : publisher one of the interfaces for which reactor provides two different implementations : Mono and Flux

**Mono :** mono can emmit 0 or 1 item followd by an onComlete or onError signal

Remember that emmiting items is not mandatory for publisher !

They can simply call onComlete without emmitting an item .

Subscribe also can call 100 items but the publisher only emmits 1 item

**Flux :** can emmit 0 or n items followed by onComplete or onError signal

Q. Once we have flux , why do we need mono ?

A. when we know for sure we are expecting only one item we should use mono , for example we can ask the database how many record are on the table ?

We just want a count – but when we want the records we could use flux , cause we don’t know there is one record, 0 record or a thousand records .

Another thing is in reactor we can use mono just like we used to use optional in java Streams

Optional<Address> can be Mono<Address> -> so now we can have 0 or 1 address just like null or one address in optional.

**Java Concepts :**

Streams :

* Stream is bascially one-time use. Once we completed our operation. We can not reuse the stream.
* streams are lazy -> it will not do anything unless you connect the terminal operator.

Q.What is terminal operators ?

A.These operation are used to produce results. They can’t be used for chaining the other methods.

Pipeline Build vs Execution :

* Imagine a method which returns a Mono<String> which can sleep thread for 3 seconds, when calling this method the output will be printed immiediatly because it takes only a few moments to create the pipline, but since somone starts to subscribe to this method, it’ll take 3 seconds to complete because now we are Executing the pipeline (not just building it) .
* ( look at Mono.L7\_PipelineBuildVsExecution)

CompletableFuture : It represents a promise that a result will be available at some point in the future. It allows you to work with asynchronous computations, enabling you to perform tasks concurrently and compose them in a more readable and maintainable way.

* CompletableFeature Example : ( Java.CompletableFeatureExplained)
* While both reactive programming and CompletableFuture can handle asynchronous operations, reactive programming is more suited for applications dealing with continuous streams of data and events, whereas CompletableFuture is more suitable for tasks that have a clear starting point and ending point, with intermediate computations in between.

Runnable:

* The concept of a "runnable" is a fundamental abstraction in concurrent programming. It allows you to represent a task or a unit of work that can be executed independently and concurrently in a separate execution context, typically in a separate thread

Callable :

* In Java, a Callable is another interface from the Java Concurrency API that represents a task or computation that can be executed asynchronously and return a result. Unlike the Runnable interface, which doesn't return a result and is used with void methods, a Callable represents a task that returns a value upon completion.

**Mono :**

* just like streams, nothing happens unless you subscribe to mono.
* only use just() method when you have data already. Otherwise use fromSupplier() method.

While using fromSupplier method, publisher only do works when there is a request for it

Our works should be lazy, we don’t want to do anything unless someone want it to do ( someone subscribed to publisher)

* Block() -> this method internally Subscribes and takes the result as a return value.
* Mono from Runnable : as we know – runnable will not take any parameter and will not return any value. So the Question is why we need to create mono from a runnable ?

The answer is sometimes there will be some time consuming operation ( operation that can take a while to complete )

And we want to be notified when the operation is done . this is when this method will be very usefull.

Now when the operation is completed we will be notified and we can do something that we want in onComplete method.

* Summary :

|  |  |  |
| --- | --- | --- |
| Type | Condition | What to use |
| Create Mono | Data already present | Mono.just(data) |
| Create Mono | Data to be calculated | Mono.fromSuplier(() -> getData())  Mono.fromCallable(() -> getData()) |
| Create Mono | Data is coming from completableFeature | Mono.fromFeature(feature) |
| Create Mono | Emit empty once a given runanable is complete | Mono.fromRunnable(runnable) |
| Pass Mono as argument | Function that needs to accept a mono as argument but we don’t have data | Mono.empty() |
| Return Mono | Function needs to return a Mono | Mono.error()  Mono.empty()  ... |

* Mono.Just() will do things eagerly
* In reactive Programming we don’t have nulls so we pass Mono.empty() when the method needs a Mono as a parameter.

**Flux :**

* Emits 0,1 ... n items.
* Assume subscriber ask for 50 items but publisher only has 3, in this situation publisher emits 3 and then pass onComplete signal.
* Just like Mono, we can use Flux.just() when we have data already.
* When we create flux from streams, remember that we can only have one terminal operator on stream then it will be closed.

In situations that we want to have multiple subscribers for one stream, we can use supplier of stream. (Look at L4\_FluxFromStream)

But remember we now should create the stream as a part of supplier and we can not reference to an already created supplier.

* Flux vs List : let’s assume we have a service call which takes 1 second to complete and retrieve data. Now if we want 5 data we need to wait for 5 seconds to list being completed and sent to us as a package. But if we use flux, whenever one data is ready, it will send it to us and we don’t need to wait anymore.

`

List publisher

Flux publisher

Subscriber

Subscriber

ubscriber

|  |  |  |
| --- | --- | --- |
| Type | Condition | What to use |
| Create Flux | Data already present | Flux.just(data)  Flux.fromIterable()  Flux.fromArray()  Flux.fromStream() |
| Create Flux | Range/Count | Flux.range(start, count) |
| Create Flux | Periodic | Flux.interval(duration) |
| Create Flux | Mono -> Flux | Flux.from(mono) |

* Above methods only good but in some cases they will not do the job.

For example if we want to keep emiiting data untill some condition met we can not do that with the methods we learn by now.

Or we want to have control on what we want to emmit and when to emmit.

* Flux.create():

Using this method we can have access to next, error, complete methods which can be used to emmit new item, throw the error if some condition were meet or pass complete signal.

This method takes the consumer of fluxSink

* Flux.generate():

Using flux.create() we can emmit whatever we want then we can use fluxSink and it methods.

Using create we have only one flux sink and we should use it wisely .

But we have only one instance of fluxsink and when it completed we can not use it anymore.

This method takes the consumer of synchronizedSink

Run example on L16 – we face an error – because using synchronizedSink we can emmit only one item ( example 1 )

But when we remove the second item , we face an infinite loop of item emmitting

What is happenning ?

In fluxSink we can have loops to emmit items but in synchronizedSink it is handled by generate method and we no longer need any loop

We can use take method to control number of items

Or even call complete method

So if we do not use take or complete or error , sink we continue emmiting items (example 4)

* Flux.Push() :

Create basically is thread safe – but push is not thread safe and it used for single thread producer.

|  |  |
| --- | --- |
| Create | Generate |
| Accepts a Consumer<FluxSink<T>> | Accepts a consumer<SynchronousSink<T>> |
| Consumer is invoked only once | Consumer can emit only one element |
| Publisher might not be aware of downstream processing speed. So we need to provide overflow sstrategy as an additional parameter | Publisher produces elements based on the downstream demand |
| Thread-safe | N/A |
| fluxSink.requestedFromDownstrem()  fluxSink.isCanceled | N/A |

**Subscribe :** Subscribing to a Mono or Flux will make the publisher to emmit the items to the subscriber.

when we subscribe to a publisher there are many parameters that accpets :

* onNext() -> consumer<T> : the first parameter is onNext call . when we provide the consumer our publisher invoke the onNext call.
* onError() -> consumer<Throwable> : we can provide the behaviour on errors happening

if we not provide the behaviour onErrors it will throws an exception .

* onComplete() – runnable : we can also provide the behaviour on stream completed.

onComplete is just a signal and not returning any data.

**Subscription Object :** we can create our custom implementation of subscription method ( all this methods will generate internally by reactor but we can also craete them manually)

When we create this Subscription we should override all the methods . (onSubscribe, onNext, onError, onComplete)

( Check L7\_subscription)

Sinks :

* what is sink in java ?

In Java, a "sink" generally refers to a component or entity that receives data, typically as part of a data processing pipeline. It is used to consume or process data emitted by a source or producer. The concept of a "sink" is commonly used in various programming paradigms, including reactive programming.

* What is sink in project reacotr ?

In the context of Project Reactor, which is a library for reactive programming in Java, a sink is an important construct. Project Reactor provides support for creating and processing reactive streams, and it follows the Reactive Streams specification. In a reactive stream, a sink is the final subscriber that consumes the data produced by a publisher.

Project Reactor defines several types of sinks:

1. Consumer: A simple sink that takes the emitted data and performs some action on it, such as logging, writing to a file, or updating a database.
2. Subscriber: A more advanced sink that subscribes to a stream of data and defines how to handle the elements (onNext), errors (onError), and completion signal (onComplete) received from the publisher.
3. MonoSink: A sink for processing elements from a Mono (a publisher that emits at most one item) in Project Reactor. It allows manually triggering the various signals like success, error, and completion.
4. FluxSink: A sink for processing elements from a Flux (a publisher that emits multiple items). It provides similar functionalities as MonoSink, but for multi-valued streams.

* Using sinks in Project Reactor allows you to control the flow of data and handle various scenarios like backpressure (a mechanism to control data flow when the subscriber is slower than the producer) efficiently. Sinks are a powerful tool for managing data processing in a reactive and non-blocking manner, enabling developers to build scalable and responsive applications

Operators :

Operators can add additional behaviours to the publisher .

* Take : using this operator we can specify how much item we want to have in upstream pipeline. ( L14 and L15 )
* Handle : filter + map operator – this operator accepts a biConsumer of given data type and synchronousSink
* Do Hooks Callbacks :
  + onCompleted -> accepts a runnable and excutes after excution completed
  + doFirst -> accepts a runnable and excute before any execution
  + doOnNext -> accepts the real object emitted by publisher
  + doOnSubscribe -> accepts the subscription object and Add behavior (side-effect) triggered when the Flux is being subscried
  + doOnRequest -> accepts a LongConsumer and add behaviour when flux receives any request .
  + doOnEach -> accepts the error object and add behaviour
  + doOnTerminate -> accetps a runnable and excutes after terminating
  + doOnCancel -> accepts a runnable and excutes after cancel
  + doFinally -> accepts a signal and add behaviour on finally like try-catch-finally block.
  + doOnDiscard ->
* Tip : in L03 we add 2 doOnFirst hook in the pipeline, what we expect is that the first one will be invoked sooner but we see is that the second one will be called before the first one , the reason is in the pipeline the subscriber starts to request the data from the publisher from bottom to top and second doOnFirst is earlier in the pipeline so everything in the pipeline will be called from bottom to top.
* LimitRate : think about google search page, google as a publisher will publish 10 items to the subscriber, but it contains more items, when the subscriber clicks on the next page, it will emitt the next 10 items , this is what limitRate do in reactive.

So assume we request for 1000 item with a limitRate of 100, in this case the publisher only receives the request for 100 , and when the subscriber reaches 75% of the items , it will request for the next 100 items.

Q.Why we need limitRate when we gonna have all itmes in the end ?

A. the publisher and the subscriber will do their job in different threads so the publisher has no information about what subscriber is going to do with data so it will keep publishing the data till the end. In this scenarios it will be better to no taking the data completly .

We can pass another argument to limitRate hook to change the 75% to whatever we want.

* delayElements : this hook will add a delay between subscribing each item ( NOT PUBLISHING ITEMS !!!! )

look at L05 and comment the sleep line and look at the log. We have 32 onNext calls without any subscribtion methods invoked because we did’nt wait for the subscribe to do it’s job.

Now after 60 seconds we have the next 32 onNext call ( 32 next items) and the subscriber do the job with this ( it’s like delayElements have an internal limitRate in it)

* onErrorReturn : we can provide a fallbackValue when the error happens with this operator.
* onErrorResume: we can provide a fallback subscriber for this hook.
* onErrorContinue: this will give us the object that cause the error and also the error will not thrown and nothin happens when error happens.
* timeOut() : assume we have a publisher which produce values every 5 seconds, in this situation we can use a timeout function for the subscriber and tell it you should wait for given time ( for example 2 seconds ). Also we can pass a fallback so if it can not receive any value within 2 seconds, it will invoke the fallback method.
* defaultIfEmpty() : we can have a default value if the publisher did’nt pass any value to the subscriber ( or subscriber did’t receive because of a filter for I.E ) but we don’t have any exception.
* switchIfEmpty() : switch to another publisher if we did’nt receive any value

for example if redis cache does not have the value then go to sql

* transform() : we can use this hook to provide a function to do some change in the middle of the pipeline on the data we got from publisher using filter, map or other hooks.
* swithcOnFirst() : an operator which takes the first item (request only one item) and then based on that first item it will make the decision to take more or not.
* flatMap() : assume we are receiving the userName of our users , then we want to have additional information about that users which came from another publisher as a flux .

now we have a flux (userNames) and another flux for each user ( additional info )

we can not handle that with map, map is suitalbe for one to one changes not one to many . in this cases we need to use flat map .

with flat map we can have our first flux and then we can retrieve next flux ant it will flaten all the data into one stream of data.

Note that in flatmap the items will be emitted whenever they are ready and the order will not remain the same.

* concatMap(): the only diff with flatMap() is that it will keep the order, it will tell reactor to move on to the next item only if the previous done it’s job.

**Publishers :**

* there are 2 kind of publisher : Hot and Cold
* as we mentioned before nothing happends untill the subscriber subscribe to publisher

this is the concept of Cold publisher , they start emitting data just when some subscriber start to listen to it. Think about Netflix.

* Hot publisher does’nt work that way, Hot publishers don’t need a subscribe to start emitting items, think about tv, they emitt items with no need of a subscriber so when i start to watch tv anyone around the world would see exact items not from the begining.

Operators :

* Share() : one of the waysto change a cold publisher to hot

Share is internally equal to publish().refCount(1)

* Publish(): Prepare a ConnectableFlux which shares this Flux sequence and dispatches values to subscribers in a backpressure-aware manner.

refCount() : minimum subscribers to start emitting items

* Resubscribe : assume the publisher finish emitting items then another subscriber joins , what will happen is that the new subscriber begin to have the items from 0 !

It is what refCount does, if new subscriber joins during emitting items , it will not receive the items from the begging but if the data finished emitting then a new subscriber joins, it will have the data from begining.

* autoConnect() : instead of refCount we can use autoConnect so we not resubscribe anymore and it will make it Real HOT HOT

with this operator, the new subscriber can not get the data if it joins after emitting all the items.

Using this operator we don’t even need the subscribers to start emitting items !!

* Cache : using cache it will stores the data that published already so for second subscribers they don’t need to get the data again from publisher , it is already there !

Cache also takes maximum items to cache as an argument.

Why we need hot publishers ?

Asume we have a stock price or a news feed and we want to push information periodically and we don’t need to listen for a request by the users , we want the frontend to have the most new data just now without any listeners or readers . we can use hot publisher in this sitautions.

**Threading & Schedulers :**

**subscribeOn :**

* Reactor is a library for async-event based programming
* It uses java tools to handle conurency but thiis tools is very hard to implement by ourselves
* Reactor does this job for developers.
* By default every task excutes in one thread ( L01)
* For executing task asynchronously and not blocking the main thread we have to create the pipeline as a Runnable and get it excuted in a different thread.
* For this job reactor created different polls and optimized for us.
* boundedElastic : optimized for Network/time-consuming calls
* parallel : best for CPU intensive tasks
* single : a single dedicated thread for one-off tasks
* immediate : current thread

Note that elastic is deprecated and we should use boundedElastic

How to use ?

* we have two operators :
  + subscribeOn -> for upstream
  + publishOn -> for downstream

Schedulers thread

how subscribeOn works ?

Main thread

Publisher

Publisher

subscriber

subscriber

When the pipline created and the subscriber begins to subscribe to the pipline, it will face a sub-on operator (remember the flow ? it begins from bottom to top ) it will take all of the pipeline to another Thread.

* boundedElastic : usually 10 times more threads than parallel ( if we have a 4 core cpu we have 40 threads ) – that why we use boundedElastic for time consuming tasks.

If we not sure we can use boundedElastic.

* What happens if we have multiple subscribeOn ?

Everything switches to the subscribeOn that closest to the publisher ( upper in the pipeline)

so if we want to force the stream to be on the bondedElastic or whatever schedulers that we want, we can add it to the publisher pipeline not the subscriber .

* When we use schedulers like boundedElastic with subscribeOn operator , we still running all of the pipeline for all items in just one thread, we moved to boundedElastic but we still having one of them . and all of them ran in boundedElastic-1

Why ?

* All the opertions are always executed in sequential
* Data is processed one by one via 1 thread in the threadpool for the subscriber
* Schedulers.parallel() is a thread pool for cpu-tasks. Does not mean parallel execution ( schedulers.parallel != parallel execution )
* The assumption is to have multiple jobs to be done in multiple threads for one publisher is WRONG !
* The correct assumption is that we have 3 subscriber that getting data from one or more publisher – in this sitatuion we goona have the thread pool and the tasks will be done in different threads .
* Parallel : usually the number of threads is very less ( if we have a 4 core cpu we have 4 thread )

**publishOn:**

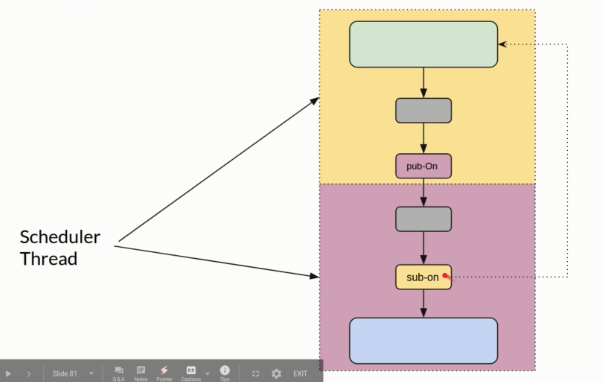
as a developer of a publisher he has some information that he knows what is going to happen so he can select the proper schedulers using publishOn operator. To publish data correctly

but the developer of subscriber may want to have some time consuming task to do and he may choose another scheduler.

* subscribeOn for upstream
* publishOn for downstream
* keep the publishOn before the specific operator ( for example use boundedElastic brefore the time consuming operator)

then again use publishOn parallel before the cpu intensive task.

How to use publishOn and subscribeOn together ?



Assume we have this pipeline, the subscriber starts to subscriber , as soon as we face the subOn , the pipeline goes to the speciefied schedulers to reach the publisher, the publisher starts emitting data in that schedulers, then when we face the pub-on , the thread will switch to that schedulers.

In summarize : when we have sub-on it will affect the source and when we have publish-on everything below that will happen on that scheduler.

Look at L05. ( remember the pipeline begins from bottom) the create will be in boundedElastic because this is what subscriber wants, then all the operators before the publishOn will do their job in boundedElastic , then when they face publishOn , everything event the subscriber will go inside parallel.

Parralel-Execution :

* by default the things is done sequantially .
* but if we want the parallel processing (parallel execution) we can use two operators of parrallel or runOn.

In L06 we see that we only have one publisher and one subscriber but still the things will happens in different threads. But we loose the sequential and the order.

* And if we want to go back to sequential after some point we can use sequential() operator.

**Backpressure & Overflow :**

Assume that the subscriber requests for all the data and publisher starts emitting data in a high rate and the pipeline doing some time consuming or heavy processes.

What will happen now ?

The publisher emitted 10 items in parallel thread , and below that the boundedElastic thread processed and emitted only 2 items . now where are the rest of items ?

Reactor provides strategies for that.

Expain : look at demo, before publishOn operator everything was very fast , all the 500 items was emitted , but after publishOn we sleep thread for items so it will be slower, how is this working ? reactor will keep all the data emitted in the memory before the subscriber can use them , and this is not good.

* Buffer startegy – default - keep in memory
* Drop – once the queue is full, new items will be dropped
* Latest – once the queue is full, keep 1 latest item as and when it arrives, drp old
* Error – throw error to downstream

Operators :

* onBackPressureDrop() : we can use onBackPressureDrop() to switch to drop strategy

L02 – why it get only 256 items ?

Check the Queues class from project reactor package

We have a small buffer size property which set to 256, we can change it in our code.

In the demo , we change it to 16, and also we add a sleep to publisher, what happens is that when we publish the items, ( remember the 75% ? ) it gets the 75 percent of the data which is 12 items then it will drain and then the next item after 12 will go the buffer .

Now for saving the dropped values we can capture them inside the operator , this operator accepts a consumer of a sink. ( in this example we store them in a list )

* onBackPressureLatest() : using this operator we are getting the latest data just before the 75% of data that want to be drained.
* onBackPressureBuffer() (with size): we can pass a value to this operator to change the size of in memory items to keep

tip : we can pass the Strategy as a second parameter to flux.create().

**Batching :**

Is a teqnique to group similar tasks .

* Buffer

Consider a flux that publish items at a fast rate, now assume we need to instert them into a database, instead of insterting them one-by-one which is very time consuming, we can collect items for like every 5 seconds and then store them.

After adding buffer(int) to the pipeline of a string publisher , the downstream data type will change from for example String to List<String>

Tip : when we use buffer(5) it waits for 5th item to array ( it’s not blocking )

This is not good, for example in a high pressure to our system the items can be emitted very fast and vise-versa .

What we can do instead of waiting for the 5th item we can use duration instead of items count.

But in this case, assume the emitting is very very fast !

We can use buffetTimeOut instead. Using this we can pass items cound and also a duration . and whatever comes first, it will use it.

Now assume we are batching items using duration or items count and we want to find a pattern inside them. In this situation for example we want to have a pattern of simillar items, but one of them can be batched in another list .

We can use 2nd parameter of buffer() operator which is skip

Usings this parameter we can skip the specified number in next batch .

For example batch(3,1) will give us 3 items in a batch with only one new item.

* Window : like buffer but instead of giving a list, this will give a flux – another different is in buffer it has to wait for items to construct it list object ( because list is a data structure) but flux is not a data structure so it does not should wait for items so subscriber can proccess it immediatly.

Using this method, we will have a flux in downstream and someone should subscribe to it , we can create a processor for it

Then() : using this we can return a Mono<Void> as a signal

* Group : assume we have some Colorized Balls and you want to group them by they colors . this is what GroupBy do.

**Repeat and Retry :**

Two ways of closing a flux is to have either a complete signal or error signal

If we want to reconnect or resubscribe to a publisher we can take advantage of this concept.

* Repeat : resubscribe after complete signal
* Retry: resubscribe after error sign

**Context :**

We can create a context that keeps an information that in the pipeline can be available.

Context internally returns a mono and it is a map ( key value pair )

**Combining Publishers :**

In real life we could have multiple publisher and some time we might want to combine them.

Options :

* startWith

assume we have 2 publishers, A (assume as a non-primary publisher) and B(primary publisher) . this will be exposed as one single publisher to our subscriber . when subscriber subscribes to this flux of publishers it will always drains the items from publisher A and once it fully drained then it will go to publisher B to get items.

Subscriber can cancel subscriber after getting all data from publisher A of satisfied somehow.

* Concat

The opposite of startWith – it will start from the Main ( primary publisher ) and then it will drains item from next publisher. This process is easy , it means unles main publisher not fully drained, we do’nt use next publisher.

* Merge

Used to merge all data from different publishers so we do not need to subscribe to all the publisher .

* Zip

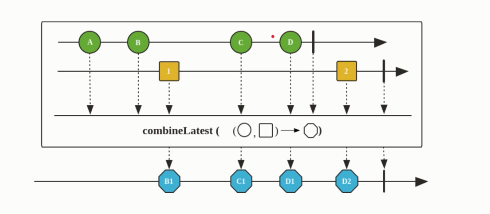
Assume the pipeline that builds a car , and the subscriber wants a car . now we have a publisher that produce car body parts, one responsible for engine and one for producing tires , we need these 3 publishers to create a car . now to build a car we need a car body, an engine and a tire , what zip does is to take one item from each publisher and zip it.

Tip – assume one publisher only has one item, now we only can create one car .

It means to keep emitting items all the publisher needs to emitt items.

* combineLatest

combine latest emitted items from all publishers .



**Sinks :**

Emitting and sending signals manually can be achieved with sinks.

* Sinks.One() : used for Mono Streams. 1 publisher – n subscriber
* Sinks.many() – unicast : for flux Streams - 1 pub 1 sub
* Sinks.many() – multicast : for flux Streams - 1 pub n sub
* Sinks.many() – replay : for flux Streams – 1 pub n sub ( replay all values for late subscribers )

**Testing :**

Step Verifier : using step verifier we can test different behaviours of our pipeline.

* expectNext -> to predicate next emitting value
* verifyComplete -> will the pipeline emmits complete signal
* verifyError -> will pipeline finish with error
* expectNextCount -> how much items will emitts
* thenConsumeWhile -> predicate the quality of items emitted ( I.E: items less than 100 )
* verify() -> using this method we can test time consuming methods
* virtualTimeTest() -> using this operator instead of waiting for test, we simulate that the time has been passed .
* thenAwait() -> used for time simulating
* expectSubscription() -> used when we want to test subscription event
* expectNoEvent() -> used when we want to test that there is no event happening.
* scnearoName() -> used to add a name to the scenario we are testing.
* as() -> operator to add name to the pipeline