

## ① Try - catch

If the child coroutine fails, the exception is propagated upwards and depending on the parent's job type, either all siblings are cancelled or not.

*(JobScope2 dot 17.9)*

Example - ① ~~incorrect without word~~ Ordinary Job

```
val scope = CoroutineScope(Job(Y)) {
    scope.launch {
        try {
            println("Job start")
            throw Runtimeexception()
        } catch (ex: Exception) {
            print("exception: $ex")
        }
    }
}
```

*(JobScope2 dot 17.9)*

O/P: Job start  
Exception: Runtimeexception ✓  
(JobScope2 dot 17.9)

Example - ②

```
runBlocking {
    try {
        launch {
            throw Runtimeexception()
        }
    } catch (ex: Exception) {
        print("exception: $ex")
    }
}
```

*(JobScope2 dot 17.9)*

O/P: Crash : Due to Runtime exception ✗

But what happens if CoroutineExceptionHandler installed in child coroutine which is not recommended?  
 It will not propagate exception to parent.  
 val exceptionHandler = CoroutineExceptionHandler { e ->

Example ③ val scope = CoroutineScope(Job(1))

```
    launch {
        delay(100)
        throw RuntimeException("Job-1")
    }
}
```

X claiming out

O/P: crash

old world, exception not handled?

old world Job-1 delay

The cancellation exception which are the result of cancelling coroutine, are not get handled by coroutine exception handler.

Try-Catch VS CoroutineExceptionHandler

Example ①

val scope = CoroutineScope(Job(1))

scope.launch {

launch {
 delay(100)
 throw RuntimeException("Job-1")
 }
}

P.F("Job-1 Started")

delay(100)

throw RuntimeException()

catch (exception: Exception) {

P.F("exception: \$exception")

}

}

}

launch {

delay(100)
 P.F("Job-2 Started")

delay(200)
 println("Job-2")

P.F("Job-2 completed")

}

catch (exception: Exception) {

println("exception caught in child Job-1")

O/P: Job-1 started

Runtime exception

Job-2 started

Job-2 completed

val exceptionHandler = CoroutineExceptionHandler { e ->

e.printStackTrace()
 P.F("exception: \$e")

}

val scope = CoroutineScope(Job(1))

scope.launch(exceptionHandler) {

launch {

P.F("Job-1 Started")

delay(100)

throw RuntimeException()

}

launch {

P.F("Job-2 Started")

delay(200)

P.F("Job-2 completed")

(Job-2 completed by "Job-1")

O/P: Job-1 Started {  
Job-2 Started }  
Exception: RuntimeException  
Job-1 completed {  
Job-2 completed }

In Example-①, Job-1 thrown exception which is caught. But that did not impact on Job-2 and it completed. Job-2 is not affected by Job-1's exception.

But in Example-②, Job-1 thrown exception which is again caught by coroutine exception handler - but that impact on Job-2 and Coroutine-1 and 2 get canceled.

#### ④ Launch { } Coroutine builder

VS Launch { } Coroutine builder

async { } ———

Example ①

val exceptionHandler = CoroutineExceptionHandler {

val scope: CoroutineScope = Job()

{ (withCancellation) launch { print }

scope.launch {

throw RuntimeException() } }

{ (withCancellation) launch { print }

scope.async { }

throw RuntimeException() }

{ (withCancellation) launch { print }

Example ②

val exceptionHandler = CoroutineExceptionHandler {

-> throwable -> do { }

P.F("exception: \$throwable")

(Individually) val scope = CoroutineScope(Job() + exceptionHandler)

```
scope.async {
    throw RuntimeException()
}
```

O/P: { Nothing printed } → exception also not printed  
caught without in CoroutineExceptionHandler

#### Example-④

3.2020 39.22  
3.2020

val exceptionHandler = CoroutineExceptionHandler {

-> throwable ->

P.F("exception: \$throwable")

{ nothing printed and handled prints }

No scope = CoroutineScope (Job() + exceptionHandler)

but still not in monitored stage

val deferred = scope.async { do { print }

when (canceling) last job all throws RuntimeException()

if both are not canceled at }. launch this later

at later no cancel or cancel after launching in

scope.launch { no ignorance about all

and deferred, await() back nothing no

total, job is not ignore all + done this before

and awaiting has with cancel handled

O/P: Exception: RuntimeException in launch

with nothing this is not here in scope initial

do { Example-⑤ (using) println seen nothing

and all nothing no println has printed

val exceptionHandler = CoroutineExceptionHandler {

-> throwable -> do { }

P.F("exception: \$throwable") }

`val Scope = CoroutineScope(Job() + exceptionHandler)`

```
scope.lanch {  
    async void with (int a) {  
        throw RuntimeException();  
    }  
}
```

```
OP: Exception: RuntimeException  
Scope. sync {  
    async {  
        throw RuntimeException()  
    }  
}  
OP: { Nothing printed even exception handler }
```

uncaught exceptions in both launch and  
async coroutines are immediately propagated  
up the job hierarchy.  $\Rightarrow$  long task low

However, if the top level coroutine was started with `launch`, the exception is handled by a coroutine exception handler, or passed to the threads' uncaught exception handler.

On other hand, if top-level coroutine was started with `async`, the exception is encapsulated in deferrable deferred return type, and rethrown when `await` is called on it.

When `async` is used for a child coroutine, the exception also immediately propagates up the job hierarchy and depending on whether the top level coroutine was started with `launch` or `async`, the exception is handled differently.

⑤ Coroutine Scope (Scope) 2022-02-02 2022-02-02 2022-02-02

The scoping function is called in scope without exceptions of its failed child coroutines instead of propagating them up the job hierarchy which allows us to handle exceptions of failed coroutines with an outer try catch section after it is rebound.

## ④ Supervisor Scope

Esempio

Example ① runBlocking {

800-892-2892 | [www.brownlow.com](http://www.brownlow.com)

```
try {  
    supervisorScope {
```

Top level —  
Corporate

3 catch (exception) {  
    P.f(ex)

The supervisor scope creates a new independent sub scope in job hierarchy. Because of this independence, no one else will handle exceptions in coroutines that are started in our new supervisor scope and therefore the scope doesn't rethrow the exception. So supervisor scope uses a supervisor job in its context, and exceptions also won't be propagated above supervisor jobs.

So we can either have to use try catch in our coroutine or install a coroutine exception handler.

So In Supervisor Scope, we have to handle all exception ourselves. I writing

If an exception is not thrown in a ~~catch~~ block of a supervisor scope, but in supervisor scope itself. The exception will be thrown by supervisor scope and therefore we can handle it with the outer ~~catch~~ block.

### Example

runBlocking {

```
try {  
    SupervisorScope {  
        throw Runtimeexception?  
    }  
}  
} catch (ex: Exception) {  
    only print ("exception: " + ex)  
}
```

## O/p: exception: Runtime Exception

三

When supervisor scope fails like this all of the started coroutines will be canceled. That is different to failure of child coroutine in supervisor scope which won't affect its others.

T

The Supervisor Scope Create new, independent Scope here with Supervisor scope Scoping function coroutine that we launch directly are now all top level coroutines `mbnaf init(93%)`

Exon

```
③ try {  
    } catch (SupervisionScopeException e)  
    {  
        → async { ← oldsworth, -  
        level (oldsworth) throws Runtimeexception()  
        casein  
    }  
}
```

`catch (Exception ex) {` Unit test = 9/22 ✓  
 `P.F(ex: Exception; ex")`

이

{ Noting printed } } don't  
{ writing } without writing

Econ  
④

```
val exceptionHandler = CoroutineExceptionHandler {  
    - throwable →
```

```
    print("Exception: $throwable")  
    throw new RuntimeException("Uncaught exception in  
    job scope. Job got an unhandled exception. Details:  
    $throwable")  
}
```

```

    scope.launch {
        supervisorScope {
            launch {
                launch(exceptionHandler) {
                    coroutineScope {
                        throw RuntimeException()
                    }
                }
            }
        }
    }
  
```

Top Level → launch(exceptionHandler) →  
 CoroutineScope → throw RuntimeException()

O/P: exception handler is both situations  
 Exception: RuntimeException just got

Example

⑤

```

val exceptionHandling = CoroutineExceptionHandler {
    throwable → {
        println("exceptionHandling")
    }
}
  
```

```

val scope = CoroutineScope(Job() + exceptionHandler)
  
```

top level →

Coroutine

```

scope.launch {
    supervisorScope {
        launch {
            launch {
                throw RuntimeException()
            }
        }
    }
}
  
```

O/P: exception: RuntimeException was low.

("oldHandler: nothing") was:

Why above example not crash even we not  
 install exception handler in top level launch or  
 in coroutines of supervisor scope because when

Coroutine context is inherited when launching  
 new coroutines and the exception handler is also  
 a context element.

Also per SupervisorScope document  
 "The provided scope is inherits from its  
 CoroutineContext from the outer Scope, but  
 overrides context's Job with supervisor job."

try {

```

    launch {
        throw RuntimeException()
    }
}
  
```

NP (verified)

①

```

async {
    launch {
        throw RuntimeException()
    }
}
  
```

O/P: {Nothing} (Not crashed)

②

```

val deferred = async {
    launch {
        throw RuntimeException()
    }
}
  
```

deferred.await() ~~EX~~  
 O/P: crash: RuntimeException

try {

```

    async {
        throw RuntimeException()
    }
}
  
```

try {

```

    launch {
        throw RuntimeException()
    }
}
  
```

PTO

Not allowed to throw exception from inside a try block if you want to return value

```

try {
    launch {
        throw RE()
    }
} catch(e) {
}

```

[X]

```

try {
    launch {
        async {
            throw RE()
        }
    }
} catch(e) {
}

```

[✓]

top level coroutine

```

try {
    CoroutineScope {
        launch {
            throw RE()
        }
    }
} catch(e) {
}

```

[✓]

[X]

In this case  
all coroutines started  
in that scope will

```

try {
    CoroutineScope {
        try {
            throw RE()
        }
    }
} catch(e) {
}

```

[✓]

[✓]

## Sequential or Series Execution

Suspend fun networkCall1(): Any {

```

    = return data
}

```

Suspend fun networkCall2(): Any {

```

    = return data
}

```

Suspend fun networkCall3(): Any {

```

    = return data
}

```

(002) point 1  
val data1 =

networkCall1()  
val data2 =

networkCall2()  
val data3 =

networkCall3()

}

Sequential

OR

Series

(003) point 1  
data1, data2

## Concurrent or Parallel Execution

### ① With launch

```
launch {
```

(1) duration out hangs

```
    launch {
```

data printer

```
        networkCall 1()
```

```
    } (2) duration out hangs
```

```
    launch {
```

networkCall 2() dataPrinter

```
        } (3) duration out hangs
```

```
    } (4) duration out hangs
```

```
    launch {
```

data printer

```
val job1 : Job = launch {
```

delay (5000)

```
    networkCall 1()
```

```
    } (1) duration = data 1
```

```
    delay (5000)
```

```
val job2 : Job = launch {
```

delay (5000)

Concurrent

job1.join()  
job2.join()

Total time taken  
5000ms

### ② with async

```
launch {
```

3 ms

```
let deferred1 = async {
```

```
    (0001) two emit data = data 1 delay (5000)
```

```
    (002L) prints "data-1"
```

"data" number

```
let deferred2 = async {
```

```
    (0019) delay (5000)
```

```
    (0020X) data 2 delay (5000)
```

(0019) prints "data-2"

Total time

```
deferred1.await()
```

// data-1 taken 3000ms

```
deferred2.await()
```

// data-2

ok

else

```
val data1: awaitAll(deferred1, deferred2)
```

```
{ val data1: List<T> = awaitAll(deferred1, deferred2)}
```

```
(002L) prints
```

"data"

Total time (data 17.9)

## Timeout

① `withTimeout(delay: Int)`

`launch {`

```
'dry' { launch = launchJobs(1)
  (suspend) val data = withTimeout(1000) {
    "I waited"
    delay(1500)
    return "data"
  }
}
```

`} catch (ex: TimeoutCancellationException) {`

`p.f(ex) // Timeout`

`} catch (ex: Exception) {
 p.f(ex)
}`

`} (whenIdleJobs)
 (whenAllJobs)`

② `withTimeoutOrNull(delay: Int)`

`launch {`

```
(suspend) val data = withTimeoutOrNull(1000) {
  delay(1500)
  "data"
}
```

`p.f(data) // null`

`}`

## Higher Order function

`fun launch(block: suspend () -> Unit)`

`val numberRetries = 2`

`try {
 block()
}`

`retry(numberOfRetries) {`

`block()
}`

`catch (ex: Exception) {
 p.f(ex)
}`

`}`

`private suspend function<T> retry(numberOfRetries: Int,
 block: suspend() -> T): T {`

`repeat(numberOfRetries) {`

`try {`

`block()`

`} catch (ex: Exception) {
 p.f(ex)
 }
}`

`}`

`return block()`

`hosting do`

`to always off set no direct access of other host or host to host`

`no access of one host to another or vice versa`

`or less often not using shared memory to access hosts`

`with own block triggered`

`in mid process`

`multiple`

## Coroutine Context

Page No.	
Date	

Coroutine Context is at the core of every Coroutine.

Every Coroutine starts is tied to a specific Coroutine Context.

↳ (initially dormant) prior

Coroutine has to be started in a certain Coroutine scope, which has a Coroutine context as its sole property.

↳ (starts)

→ Coroutine Scope

↳ (initially dormant) prior → Coroutine Context starting  
↳ (T →, happens: void)

↳ Context Elements  
↳ (initially dormant) prior  
Dispatcher → Job  
↳ (initially dormant)  
Error Handler → Name  
↳ (initially dormant)

↳ (initially dormant)

defines on which

Thread or Thread Pool used to handle, which can be the lifecycle of Coroutine will be error in Coroutine help few off to Coroutine and also used to execute on or Coroutine Hierarchy Specify for

Debugging  
Run Pages

Job Context

build Parent Child  
hierarchies of  
Coroutines

Page No.	
Date	

## ViewModel Scope

↳ (initially dormant) prior  
↳ (initially dormant) Context Elements have nothing

1. Dispatcher → Main

↳ (initially dormant) 2. Job → Dispatcher's own Job

3. Error Handler → Null

4. None - Default

↳ (initially dormant) 5. Job → Job's own Job  
↳ (initially dormant) 6. Job's own Job

→ Dispatcher → Job's own Job

### ① Dispatcher. Main

→ Will launch Coroutine on Android Main thread  
(UI thread) ↳ (initially dormant)

②

Dispatcher. IO → (initially dormant)  
↳ (initially dormant)

→ Perform IO-related operations (Network, file handle)  
→ Uses shared thread pool internally,  
↳ (initially dormant) limited to 64 threads (but can config  
for more) ↳ (initially dormant) not scaling

③ Dispatcher. Default → (initially dormant)

↳ (initially dormant) ↳ (initially dormant) ↳ (initially dormant)

→ Default dispatcher (if no other is defined)  
→ Optimised for CPU-intensive work  
(like, Sorting list, heavy calculations,  
Parse big JSON file)

→ (initially dormant) uses shared thread pool internally,  
↳ (initially dormant) if the number of threads is equal

to number of CPU cores (at least two)  
(2, 4, or 8 threads)

## ④ Dispatchers. Unconfined

- Not confined to any thread
- Initial running on thread the coroutine was started from
- ↳ Might be switches thread on context switcher with Suspend function
- As per official Android document "The unconfined dispatcher should not normally be used in code".

## ⑤ Custom Dispatcher

- newSingleThreadContext()
- newFixedPoolContext
- java.util.concurrent.Executor → as CoroutineDispatcher

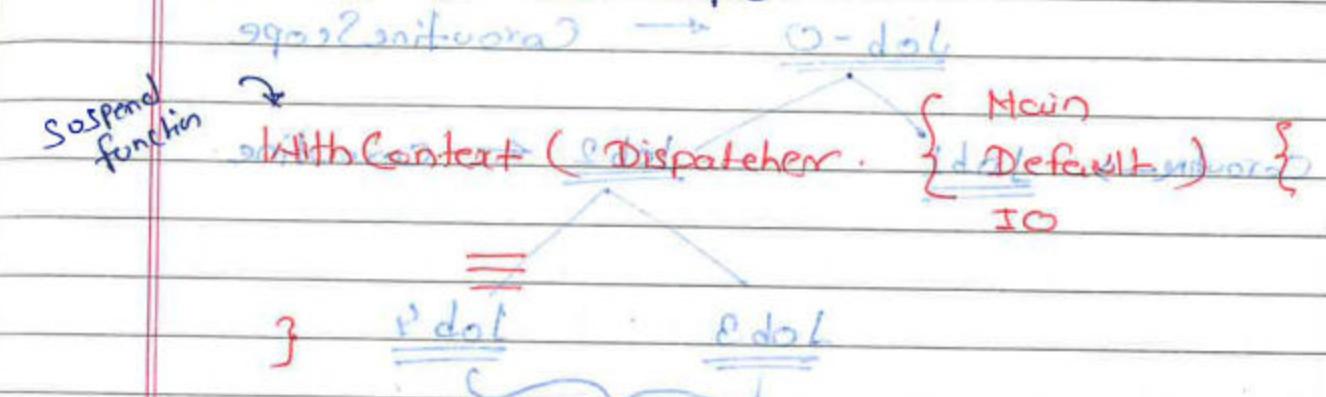
```
Coroutines
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.Default,
    block: suspend CoroutineScope.() -> Unit): Job {
    // ...
}
```

↳ All coroutines started from now operate on an empty Coroutine context. Empty Coroutine Context?

No. To answer this question we need to understand what happens when we launch a coroutine from an empty Coroutine context.

When a new Coroutine starts with launch then the context from Coroutine scope on which launch is called is inherited by the new Coroutine. Initially both initial + block scope

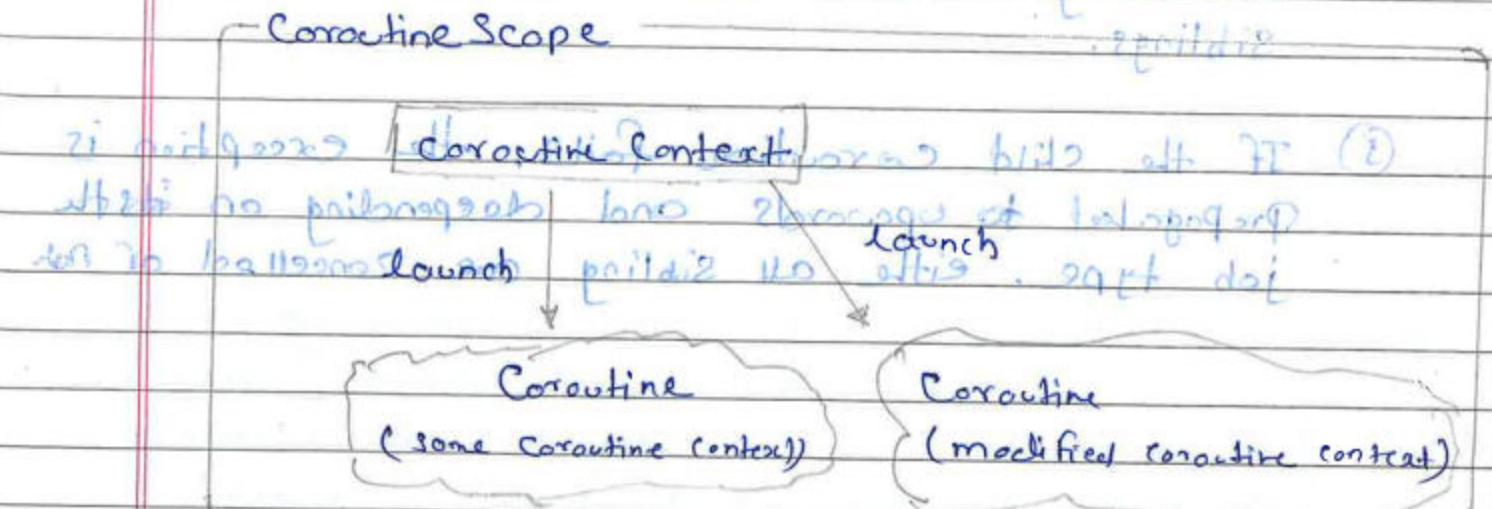
If we specify certain context elements in the context parameter of launch then those context elements then override the context elements of the scope.



for switching context.

With Context is most specifically used for switching context. But also can be used with another Emulator, so it's not a good practice.

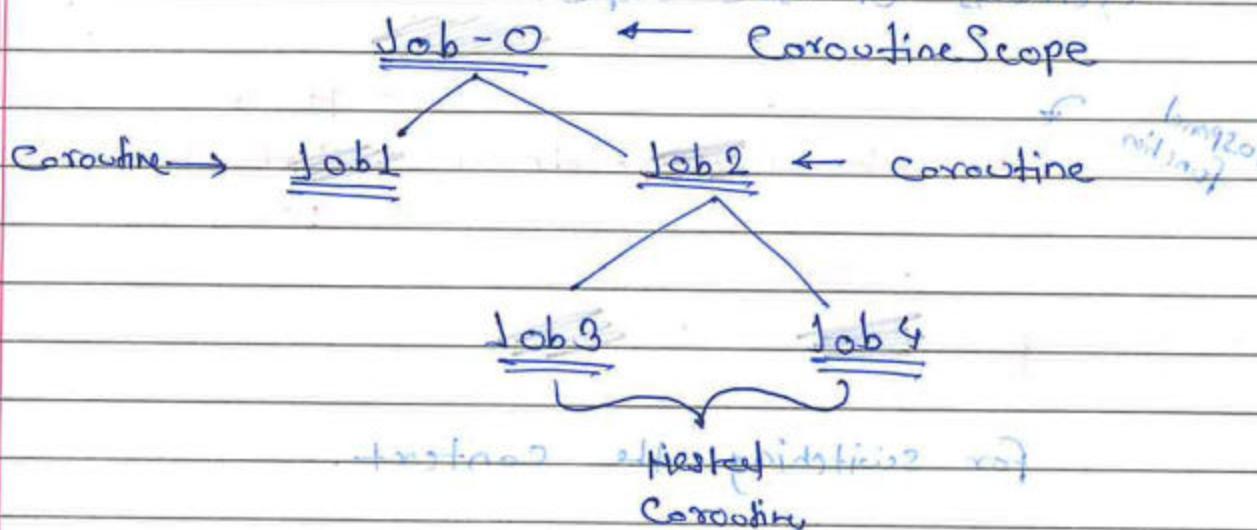
It's not good to mix them in parallel. It's not good to mix them in parallel.



## Structured Concurrency

- ~~structured~~ ~~concurrency~~ can be used  
when we want all coroutines to work at once
- ① Every Coroutine needs to be started in a logic scope with limited life-time.

- ② Coroutines started in the same scope form a hierarchy.



- ③ When launching from a function itself, ParentJob won't complete until all of its children have completed.

- ④ Cancelling a parent will cancel all children. Cancelling a child won't cancel the parent or siblings.

- ⑤ If the child coroutine fails, the exception is propagated upwards and depending on its type, either all sibling are cancelled or not.

~~(writing without parent)~~ ~~(writing without parent)~~

but as `val scope = CoroutineScope(Dispatcher.Default)`  
~~scope.cancel()~~ ~~cancel~~ ~~and ignore it~~

↳ ~~cancel~~ ~~cancel~~ ~~most important of all~~  
~~when we start a job of scope launch & do not cancel~~  
~~it will show outcome of execution~~

3

`Scope.cancel()` ~~if dot = do scope.launch~~  
~~that's what scope.launch() does. so it = do scope.launch +~~

~~(dot) whenever a new coroutine launch then~~  
~~the coroutine inherits the context elements from~~  
~~its parent, but except for the job. For Job~~  
~~coroutine creates its own new job object and~~  
~~defines its own job = scope.launch its parent job.~~

↳ `val scopeJob = Job()`  
`val scope = CoroutineScope(Dispatcher.Default)`  
~~if dot dot dot == dot launching +~~  
~~scopeJob~~

~~if dot dot dot == scope.launch { }~~  
3

`scopeJob.children.contains(coroutineJob) // true`

By installing our own job for the parent job when calling launch, we could override the mechanism of structured concurrency. And because of that if canceled the scope for instance then our started coroutine wouldn't also be canceled anymore.

val scope = CoroutineScope(Dispatcher.Main)  
scope.coroutineContext[Job] !!.join()

Page No.	
Date	

Page No.	
Date	

(This) Here a new job hierarchy is created which would have to manage by ourselves. 29/02

According to Android team "We don't recommend passing jobs in the context parameter to coroutine builders in modern code, though"



val scopeJob = Job() (11/02 29/02)  
val scope = CoroutineScope(Dispatcher.Default)

+

and don't mix them in a nonScopeJob)

with standard syntax like starting with the dot  
dot val passedJob = Job() and then the dot

dot with the dot now can't make difference

dot then val CoroutineJob = scope.launch(passedJob){

} (dot = } dot 29/02 now)

↳ (dot 29/02) ==> return = 29/02 now

+ passedJob ==> coroutineJob // false

(dot 29/02)  
· scopeJob.children.contains(coroutineJob) // false

↳ (dot 29/02) returns nullity of dot 29/02

↳ (dot 29/02) dot now has children of  
the children block - 29/02. Don't point out  
both - because it's hierarchy for coroutines  
and not 29/02 all 29/02 is just a named  
variable and after 29/02 it's not mixed  
variable and after 29/02 it's not mixed

## Job & SupervisorJob

①

### Job (Ordinary Job) 1 - ordinary

-> S - -> ->

29/02 L - ->

↳ (dot 29/02) S - ->

val exceptionHandler: CoroutineExceptionHandler?

coroutineContext.throwable? →

p.f("exception: \$throwable")

"With exception block it is prohibited to  
use val scope = CoroutineScope(Job(exceptionHandler))"

### scope.launch {

p.f("Coroutine - 1 starts")

delay (5s)

→ if p.f("in Coroutine - 1 fails")

caused throwException(true) or dot 29/02

}

→ In Same Scope and are

scope.launch{ 29/02 return } = siblings

p.f("Coroutine - 2 starts")

(method delay (5s))

p.f("Coroutine - 2 got finish")

} . invokeOnCompletion { throwable →

if( throwable is cancellationException ) {

p.f("Coroutine - 2 got cancellation")

}

↳ 1 - ordinary 29/02

-> S - ->

p.f("isScopeActive: \${Scope.isActive} ") -

whenScopeActive.endPoint : int 29/02

↳ 2 - ordinary

point : with scope 29/02

o/p:

Coroutine - 1 starts

- 1 - 2 - 1

- 1 - 1 fails

- 1 - 2 got cancelled!

Exception: java.lang.RuntimeException

IScope is Active: false

(old with cancellation) 11.0

"In ordinary job if <sup>any</sup> child Coroutine fails  
then all its siblings and its <sup>(parent job)</sup> scope get cancelled."

②

## Supervisor Job

In above example (previous example), if we replace Job by SupervisorJob like below:

val scope = CoroutineScope(SupervisorJob(0.0))  
(11.0) plus  
(exceptionHandler)

(child job 3 - without cancellation)  
+ old with cancellation 2 & down.  
Finally no cancellation in old with 11.0  
then the option below 11.0

o/p: Coroutine - 1 starts

- 1 - 2 - 1

- (Coroutine 1 fails: without cancellation 2) 11.0

Exception: java.lang.RuntimeException

Coroutine - 2 complete

IScope Active: true

In SupervisorJob if any child Coroutine fails it won't affect its siblings and parent job and they won't get cancelled.

11.0 (11.0) without cancellation 11.0

ViewModelScope and LifecycleScope has SupervisoryJob like 11.0

old with cancellation 11.0  
to cancel or task or parent 11.0. 11.0 local 11.0

internal scope under branch of 11.0 11.0 local 11.0  
StructuredConcurrency + withdraw soon to cancellation and NewConcurrency soon after 11.0

"Coroutine needs to run inside a certain scope with limited lifetime and these tasks then form a hierarchy by automatically. The hierarchy put some useful automatic cancellation and exception handling mechanisms in place."  
11.0 plus and for 11.0 scope local self 11.0  
due to NewModel (scope in loc or in right)  
- Lifecycle Scope

## Unstructured Concurrency

Unstructured Concurrency is activated by GlobalScope.launch(11.0) 11.0 plus

## Global Scope

11.0 launch without cancellation 11.0

A Global Scope or Global Coroutine Scope is not bounded by boundaries of Job.launch(11.0)

Global Scope is used to launch top-level Coroutines which are operated throughout application lifetime and are not cancelled permanently.

Global Scope operators running in `IDispatcher`.  
which does not have any job <sup>assisted</sup> object attached to it.

`GlobalScope.coroutineContext[Job] == null`

The GlobalScope object has no Job

So no Job object associated with the GlobalScope. This means that no hierarchy of Job objects will be formed when you launch a new coroutine in the global scope.

So when a new coroutine is launched in globalScope it will not have any connection to other scope. It has similar behavior to `Dispatchers.global`.

This is because there is no information of parent job object. The only way to form a relation between two scopes is via a third job object.

As the Global Scope does not have any Job object. There is no way of building some kind of parent child relation.

### Dispatcher Main vs Main.Immediate

Ex-①: `launch(Dispatcher.Main) {`

`Pf("first statement in Coroutine")`

`delay(10)`

`Pf("Second statement in Coroutine")`

`Pf("first statement after launch coroutine")`

Op: First statement after launch Coroutine  
↳ first statement in coroutine is delayed  
↳ first statement in coroutine is delayed

Ex-②: `launch(Dispatcher.Main.immediate) {`

`Pf("first statement in Coroutine")`

`delay(10) == 0ms`

`Pf("Second statement in Coroutine")`

`Pf("first statement after launch Coroutine")`

Op: first statement in Coroutine  
first statement after launch Coroutine  
↳ second statement in Coroutine. ①  
↳ second statement in Coroutine. ②

ViewModelScope has `Dispatcher.Main.immediate`.  
LifecycleScope ||

### Lifecycle Scope

`[ ] disposal = 10ms`

form Activity & fragment

`[ ] disposal = 10ms`

① `lifecycleScope.launch { }`

`signals life cycle`

`main.10ms`

② This coroutine scope is tied to LifeCycleOwner's of LifeCycle. The scope will be canceled when the lifecycle is destroyed. Also the running coroutine also canceled when device is rotated because a configuration change also destroys and recreates the activity.

`10ms.10ms`

③ `lifecycleScope.launchWhenCreated { }`

`[ ] disposal`

④ `lifecycleScope.launchWhenResumed { }`

`main.10ms`

⑤ `lifecycleScope.launchWhenStarted { }`

`[ ] disposal`



## Cancelling Coroutines

```
    while(1) {
        if (Serial.available() > 0) {
            delay(100);
            Serial.write('A');
        }
    }
}
```

without blocking main thread so it works  
without delay (250) until both return at end so  
with P. f("Concelling Coroutine") p-2 in normally has  
job concely // cancel() is normal function not  
a suspend function [ so  
OIP: 

0	{ (2) integer
1	
2	(if np

  
concelling (coroutine)

The suspending function is cancellable. If the job of current coroutine is cancelled or completed while this suspending function is waiting, this function immediately resumes with CancellationException.

val job = launch {  
 repeat(10) {  
 println("Hello")  
 }  
}

• cancel (cancel) continues delay (100) delay { delay (100) }  
room Host .doi hi on cancel will catch (e: exception) {  
vo initiator ni cancel now we want if "Cancellation exception"  
"will" cancel after cancel without throws CancellationException  
} { } ↑

```
delay(250)  
printf("Creating coroutine")  
Job.cancel();
```

$\Theta(P)$

Canceling Coroutine

Rehearsals for cancelling  
coronavirus

~~forests~~ did not cancel  
and loop execute  
continually.

Every suspend function of the Kotlin Coroutines library checks whether the coroutine it's being called from is still active. Otherwise it throws cancellation exception.

This behaviour of an implementation is called to be cooperative regarding cancellation.

However if we create our own suspend functions, we have to make sure that they support cancellation and therefore are cooperative regarding cancellation.

val job = launch {

```
repeat(5) {
```

    P.F(it)

    delay(100) polymorphic

}

    delay(250) is without polymorphic return  
    P.F("canceling Coroutine") returns Unit

    when(job.cancel()) {  
        is without polymorphic return

O/P: 0 : without polymorphic return

1

canceling Coroutine if not = do!

2 : canceling

3 : canceling

Above: canceling Coroutine does not stop  
the job when called cancel on its job. That means  
we have to make our own code in coroutines or  
our suspend functions cooperative regarding cancellation

## ① ensureActive() & yield()

val job = launch {

```
repeat(10) {
```

ensureActive()

    // yield

P.F(it), when cancel

(0.0) polo

1 : cancel

2 : cancel

3 : cancel

4 : cancel

5 : cancel

6 : cancel

7 : cancel

8 : cancel

9 : cancel

10 : cancel

Thread.sleep(100) and then = do!  
    ? around = do!  
    ? Job Hoop  
    delay(250) (return Unit)  
    P.F("canceling Coroutine")  
    job.cancel()  
    O/P: 0 : cancel  
        1 : cancel  
        2 : cancel  
        3 : cancel  
        4 : cancel  
        5 : cancel  
        6 : cancel  
        7 : cancel  
        8 : cancel  
        9 : cancel  
        10 : cancel  
    midpoint canceling Coroutine

## ② isActive

val job = launch {

```
repeat(10) {
```

    if(isActive){  
        P.F("canceling Coroutine")

    }

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

    delay(250) is without polymorphic return

    P.F("canceling Coroutine") is without polymorphic return

</

### Non cancellable code

```

val job = launch {
    repeat(10) {
        if(isActive) {
            Thread.sleep(100)
        } else {
            delay(100)
        }
    }
    Not reachable
    P.F("Cleaning code")
    Code
    throw CancellationException()
}
delay(250)
job.cancel(P.F("canceling coroutine"))

```

OIP:

1 (2) : 0092  
2 Cancelling Coroutine

(HP7.5)

In above example, point statements below the `delay(100)` in else block never will execute because `delay` has already thrown a cancellation exception. So `Not reachable` & `Cleaning code` will never execute.

So to make else block executable, need to wrap code in

```

withContext(NonCancellable) {
    delay(250)
    // This code won't be cancelled
}

```

A Non Cancellable job is always Active. If is designed for withContext function to prevent cancellation of code block that needs to be executed without cancellation.

val job = launch {

```

repeat(10) {
    if(isActive) {
        P.F()
        Thread.sleep(100)
    } else {
        withContext(NonCancellable) {
            delay(100)
            P.F("Cancellation")
            throw CancellationException()
        }
    }
}

```

delay(250)

P.F("canceling coroutine")  
job.cancel()

OIP:

1 Cancelling coroutine  
2 Clean up code

## Flow

### Flow Builders

flow main { }

{ <-- flow builder ②

launch { }

() build ( ) ③

collectNumbers (10). collect { it → emit  
P.F(it) }

{ <-- it → emit ④

}

( ) build ( ) ⑤

}

main fun flowCollectNumbers (n: Int) : flow<Int> = flow { }

for (i in 1..n) { }

delay (10)

emit (i)

}

( ) build ( ) ⑥

( ) build ( ) ⑦

}. flowOn (Dispatcher.Default)

( ) build ( ) ⑧

flowOf (Int) (0). delay (10) = multiFlow

multiFlow (Int) (1). collect { it → P.F(it) } flowOn,

flowOf (Int) (1, 2, 3, 4, 5). collect { it → P.F(it) } flowOn,

( ) build ( ) ⑨

listOf (1, 2, 3). asFlow(). collect { it → P.F(it) }

flow { }

= delay (100)

( ) build ( ) ⑩

( ) build ( ) ⑪

emitAll (anotherFlow) ( ) build ( ) ⑫

}. collect { it →

( ) build ( ) ⑬

P.F(it) }

( ) build ( ) ⑭

( ) build ( ) ⑮

Collect is a suspend function

## Terminal Operators

① collect { it → }

{ emit item }

② flow.first()

{ await }

flow.firstOrNone().or( )

flow.first { it > 2 }

(it) { p }

③ flow.last()

{ }

④ flow.single() (if it collides with flow, emits more)

than one.)

(it) { p }

⑤ toList()

(1) { p }

toSet()

(1) { p } initial value

⑥ fold()

initial value

↓ is equal to initial value

val item = flow.fold ( 10 ) { accumulator, emittedItem → }

emittedItem + 1 } + accumulator + emittedItem }

{ }

resulting after (10, 1, 2, 3, 4, 5) (start > 10) { }

⑦ reduce()

(initial value) { it + 1 } (10, 1, 2, 3, 4, 5) { start > 10 } { }

{ }

⑧ launchIn

(root) { p }

val scope = CoroutineScope(EmptyCoroutineContext)

flow.onEach { it → p } // it is

• launchIn(scope) { it } { p }

(it) { p }

⑨ repeat & mapAsFlow

{ }

## launchIn vs Collect

val flow = flow { }

(it) { delay(400) { p } (it) { p }

Q.F("Emitting first value")  
emit(1)

{ }

(it) { delay(500) { p } (it) { p }

Q.F("Emitting Second value")  
emit(2)

{ }

Focus order: 1 (Received 1st p)

val scope = CoroutineScope(EmptyCoroutineContext)

1 (Received 2nd p)

flow.onEach { Q.F("InLaunchIn-1: \$it") }

• launchIn(scope) { p }

1 (Received 1st p)

flow.onEach { Q.F("InLaunchIn-2: \$it") }

• launchIn(scope) { p }

O/P:

Emitting first value

Received LaunchIn-1: 1

Emitting first value

Received LaunchIn-2: 1

Emitting Second value

Received LaunchIn-1: 2

↑ (Received LaunchIn-2: 2)

↑ Received LaunchIn-2: 2

## Scope: launch {

```
flow.collect {
    P.F("Received collect-1:$it")
    flow.emit(1)
}
flow.collect {
    P.F("Received collect-2:$it")
    flow.emit(2)
}
```

Sequential

O/P:

```
Emitting first value
Received collect-1:1
Emitting second value
Received collect-2:2
Emitting first value
Received collect-1:1
Emitting second value
Received collect-2:2
```

## Lifecycle Operator ~~also~~ wait finishing

① onStart()

other wait finishing

② onCompletion(): \$E -> Unit having

other base finishing

onCompletion{ Throwable? } →

\$E -> Unit having

If null then flow completed without exception  
else it holds exception.

## as LiveData ()

```
fun < T : Any? > Flow< T > asLiveData( @param ①
    context : CoroutineContext = EmptyCoroutineContext,
    timeoutInMs : Long = DEFAULT_TIMEOUT ②
) : LiveData< T >
```

Can pass Coroutine Live Data Operator waits  
Context elements for this. Specified timeout  
after the live data observer  
gets inactive and only then  
cancels the flow

③ start ← (e = time) start

When observer is in the inactive  
state only for a very short time (default)

we don't need to cancel the flow

Execution restart if again. ↓

(onStart) : 9/0

So if the live data observer is inactive more  
than timeoutInMs then only flow will cancel.  
and onCompletion block will execute. ↓

(e = time) start

2.11 (e) starts .(2,3,4,5) onstart

{S>H} with works

2.22 (e) starts .{2,3,4,5} onstart

## Intermediate Operator

① map along & mapNotNull

```
map: (T -> S)
mapNotNull: (T -> S)
    if (value == null) return null
    else return value
```

② filter { it > 3 } & filterNot { it > 3 }

```
filter: (T -> Boolean)
filterNot: (T -> Boolean)

if true emit
if false emit
```

### filterNotNull()

↳ If input contains null values, it will ignore nulls  
 Example: `filterNotNull<T>(): Flowable<T>`  
 • works with nullable values  
 • nothing has nullable Type to filter

③ Take (count = 3) → first 3

↳ If input contains null values, it will ignore nulls  
`takeWhile { it > 3 } → take`  
 until it finds a value less than 3

flowOf(1, 2, 3, 4, 5). takeWhile { it > 3 }

OIP: (Nothing)

↳ If input contains null values, it will ignore nulls

flowOf(1, 2, 3, 4, 5). takeWhile { it < 3 } → null

OIP: 1, 2, 3 → null should emit because it has been

④ drop (count = 3)

flowOn { 1, 2, 3, 4, 5 }. drop(3) // 4, 5

dropWhile { it < 2 }

flowOn { 1, 2, 3, 4, 5 }. dropWhile { it < 2 } // 3, 4, 5

⑤

## Transform

### Map VS Transform

↳ count = count (no change)

flowOn(1, 2, 3, 4, 5) → flowOn(1, 2, 3, 4, 5).

(with map { it -> 1 }) configuration. transform(this: FlowCollector<Int>)

it \* 10

}

• collect { value ->

(map { it -> P.F(it) }) configuration. collect { value ->

}

P.F(value)

}

→ [10, 20, 30, 40, 50] (it = 1, 2, 20, 3, 30)

3 (it = 3, 4, 40, 5, 50)

"Map has emit only" ↳ Transform has emit multiple

• Single value ability ↳ Map has emit ability

⑥

### WithIndex("value")

flowOn(1, 2, 3). withIndex(index = 0, value = 1)

• withIndex(index = 3 → 4, 2 → 1, 1 → 2)

• collect {

P.F(it)

}

⑦ { distinctUntilChanged(3) no algorithm can do }

{ addCount(3) → oldCount + 3 values }

OIP: 1, 1, 1, 2, 2, 3 → 1, 2, 3

OIP: 1, 2, 1, 3, 1 → 1, 2, 1, 3, 1

{ (oldCount + 1) - addCount(3) (3) }

→ 1, 2, 1, 3, 1

## Flow Exception Handling

```
val flow = flow {
    emit("Apple")
    emit("Google")
    throws exception("Flow Exception")
}
```

Launch {

flow.map {

```
    emit() throws exception("Flow Pipeline Exception")
```

try {

```
    flow.onCompletion { cause: Throwable? ->
        if (cause == null) {
```

P.F("Flow completed successfully")

```
    } else { P.F("exception: $cause") }
    catch (e) {
```

collect {

P.F("stock")

```
    } catch (e: Exception) { P.F("Exception Handled: $e") }
    } }
```

③ → catch { throwable → }

```
flow.onCompletion { cause -> P.F("$cause") }
```

• catch { throwable → P.F("\$throwable") }

• collect { id -> P.F(id) }

Catch value  
emit the concerned

```
catch { throwable -> P.F(throwable)
       emit("value") }
```

onSuccess { }

before onCompletion { crashing in onComplete }

- onEach { throwing bad "cause" of throw exception("Exception") }
- P.F("Data: \$1F")

- catch { throw → }

at launchIn ( dispatcher.default ) to make A

assertive code ambiguous behavior on run

## Exception Transparency

inner functionality

flow {

try {

emit(j)

catch (e: Exception) {

P.F("Exception: \$e") }

}

}, collect { value -> }

throw exception("Throw exception in collect")

O/P:

Exception: Throw exception in collect

Code successfully ran without any crash.

emit() is just a function that internally calling collect() like callback or calls

emit() functioned as return did in  
with user defined function in many languages

collect()

Violation of  
Exception OR  
Role Transparency

"Exceptions in downstream are handled somewhere in upstream" but explicitly, means:

(caught & Ignored) or

(not caught)

### Exception Transparency Aspects

① A downstream exception must always be propagated to the collector

② Once an uncought exception was thrown downstream, the flow isn't allowed to emit additional values.

Break

So now above example is violation of Exception Transparency Rule 1

Fix: ~~if~~ ~~try~~ ~~catch~~ ~~else~~ ~~return~~

```
fix: flow {
    emit(1)
}. catch { throwable ->
    if (throwable instanceof NetworkException) {
        throw new RuntimeException("Throw RF")
    }
}
```

call: ~~on~~ ~~emit~~ ~~throw~~ ~~exception~~ ("Throw RF")

exp: ~~exception~~ ~~throws~~ ~~RF~~

~~OPR~~: Crash the code, but it is fine

→ condition (so no more violation of Exception Transparency Rule)

!! Catch operator can catches only upstream exceptions and passes all downstream exceptions"

### retry() & retryWhen()

retry()

At consumer side end:

```
val flow = stockflow(). retry { throwable ->
    throwable is NetworkException
}. catch { throwable ->
    collect { }
```

At Producer end:

flow {

```
. retry { throwable ->
    throwable is NetworkException
}
```

```
fun<T> Flow<T>.retry (attempts: Long = Long.MAX_VALUE,
    predicate: Suspend<cause: Throwable> -> Boolean = {true}): Flow<T>
```

if ~~attempts~~ if ~~attempts~~ is

Long

retryWhen { cause, attempt ->

```
if ("No of Attempts: ${attempt}") {
    cause is Network
```

## Life Cycle Aware Collecting flow

Page No.	
Date	

### ① By repeatOnLifecycle

ViewMode: val flow: Flow<UIState>

Activity:

```
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.flow.collect { uiState →
            render(uiState)
        }
    }
}
```

Fragment:

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.repeatOnLifecycle(
        Lifecycle.State.STARTED) {
        viewModel.flow.collect { uiState →
            render(uiState)
        }
    }
}
```

### ② By flowWithLifecycle

```
lifecycleScope.launch {
    viewModel.flow.flowWithLifecycle(
        lifecycle, Lifecycle.State.STARTED)
        .collect { uiState →
            render(uiState)
        }
}
```

val job = launch {
 fun coldFlow() = flow {
 repeat(5) { pf("emitting:\$it")
 premit(it)
 delay(1000)
 }
 }
}

fun main = runBlocking {
 suspend fun main(): Unit = coroutineScope {
 val job = launch {
 coldFlow()
 onCompletion {
 pf("flow of collector complete")
 }
 collect {
 pf("collected:\$it")
 }
 delay(1500)
 job.cancelAndJoin()
 }
 }
}

O/P: emitting 1  
collected 1  
emitting 2  
collected 2  
flow of collector complete.

delay (250)

"Flows are cold"  
"Shared Flows are hot"

"When collecting coroutine canceled then the flow <sup>builder</sup> code stops being executing. So flow of collector on @Completed is executed and so next values are not emitted."

### Cold Flow

(Normal) flows are cold if they are not shared

#### Cold Flows characteristics

- ① Become active on collection
- ② Become inactive on cancellation of the collecting coroutine.
- ③ emit individual emissions to every collector.

### Hot Flow

Shared flows are not hot

#### Hot flow

- ① Are active regardless of whether there are collectors
- ② stay active even when there is no more collector
- ③ emission are shared between all collectors

Producer: val sharedFlow = MutableSharedFlow<Int>()

launch {

(it: Int) -> repeat(5) { }

P.F("SharedFlow emit \$it")

sharedFlow.emit(it)

delay(250)

### Consumer ① Without consumer:

OIP:

sharedflow emits 0

- " - 1

Without

Collector

Hot flows emit value

- " - 2

initials no emission

- " - 3

- " - 4

- " - 5

### ② With Consumer:

producer (process of initializing sharedflow) ③

④ Thread-Sleep (550)

Launch {

sharedflow.collect {

PF("collected:\$it") ⑤

}

OIP

initial Sharedflow emits 0 { when no } ①

- " - 1 initial 2 lost emission

- " - 2

initials 2 emit 3 after action ⑥

collected 3

Sharedflow emits 4

initials " collected 4 because no emission ⑦

⑧

⑨ launch { 2 sharedflow = new sharedflow }

sharedflow.collect { 3 done }

PF("sharedflow-1.collected:\$it")

("Hi" 6) waitThread 7 ⑩

{ 8 ("Hi") timer.waitThread

(0.2 1) 10 ⑪

launch {

sharedflow.collect {

PF("sharedflow-2.collected:\$it") ⑫

}

initials 2 emit 3 after action ⑬

initials 2 emit 3 after action ⑭

OIP:

initial Sharedflow emits 0 { half -

break 2 emit 2 " to wait using wait

sharedflow-1.collected 1 of period

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 2 ⑮

- " - 2 - 1 - 2

sharedflow emits ⑯ half - half period ⑰

sharedflow-1.collected 3

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 4 ⑱

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 5 ⑲

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 6 ⑳

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 7 ㉑

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 8 ㉒

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 9 ㉓

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 10 ㉔

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 11 ㉕

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 12 ㉖

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 13 ㉗

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 14 ㉘

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 15 ㉙

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 16 ㉚

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 17 ㉛

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 18 ㉜

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 19 ㉝

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 20 ㉞

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 21 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 22 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 23 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 24 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 25 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 26 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 27 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 28 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 29 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 30 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 31 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 32 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 33 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 34 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 35 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 36 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 37 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 38 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 39 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 40 ㉟

initials 2 emit 2 " to wait 2 half -

initials 2 emit 2 " to wait 2 half -

sharedflow-1.collected 41 ㉟

initial

(A) Started :-

① Sharing Started: Eagerly and "174

- The collection of upstream flow will happen immediately, even if nobody currently collects from our shared flow.
    - And so we could slice emissions when Shared flow emits flow values before we start collecting in the `onCreate` method.
    - With that option Shared flow will continue to collect from upstream flows even when all collectors stop collecting.

② Sharing started. Lazily ~~limit - with wonder~~

$$\Sigma \text{ (bottom)} = a_{\text{min}}^2 \text{ (area)}$$

- This one works ~~as~~<sup>similar</sup> to Eagerly option with the only difference that the collection and sharing of emission / start lazily when first collector starts collecting from the shared flow.
    - This guarantees that the first collector gets all of the emitted values.
    - The upstream flow continues to be collected even when all collectors stop collecting.

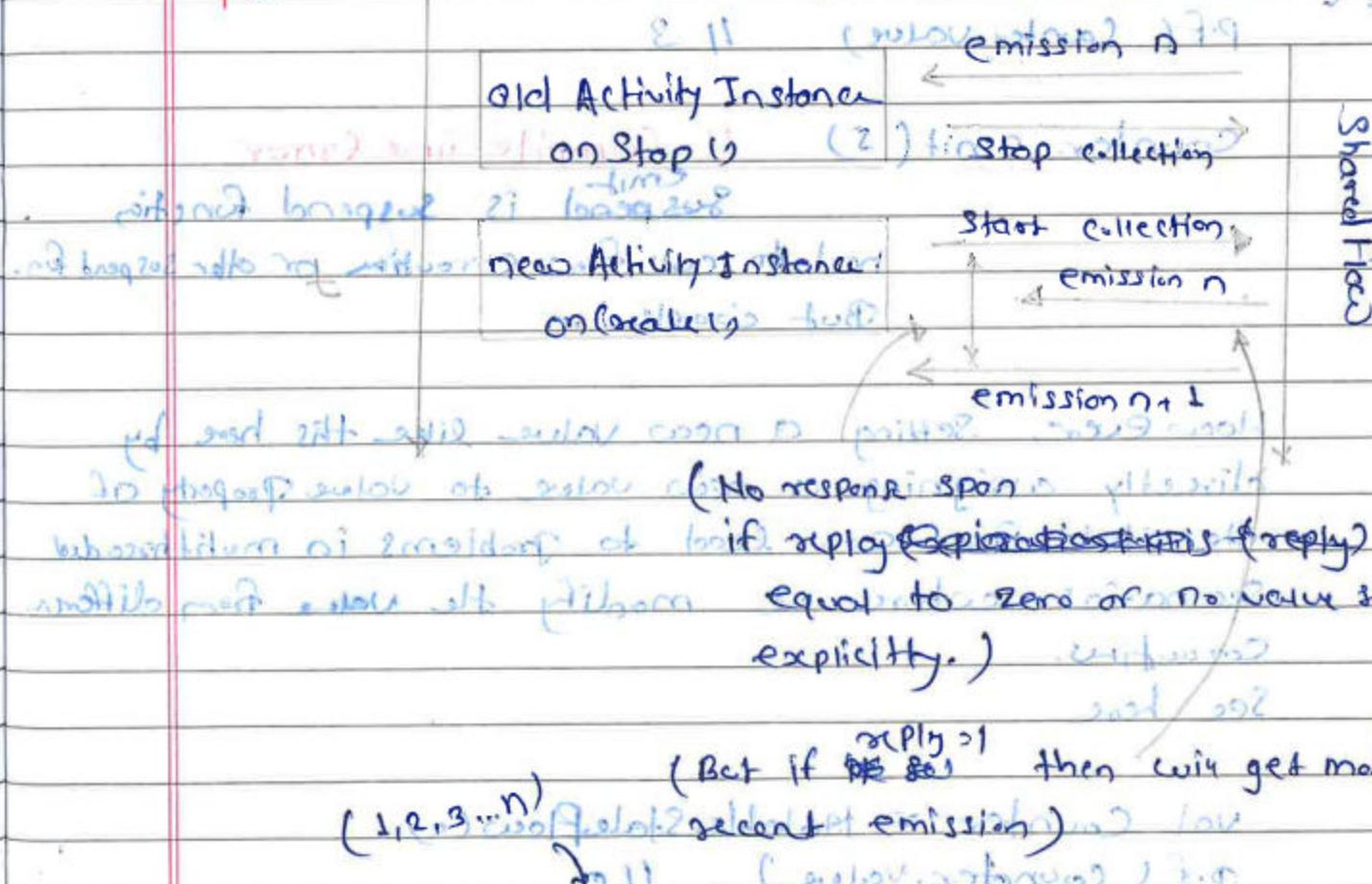
③ Sharing Started: While Subscribing

- With this option, shared flow starts the collection and re-emissions once the first collector starts collecting and immediately stops when the last collector stops collecting.

- \* fun`onWhileSubscribed` (stopTimeOutMillis: Long = 0, `replayExpirationMillis`: Long = Long.Max.value)
    - ↳ `log.info("Sharing Started")`
  - toCollectorFromMapper - `Mapper` → `long` → `Long`
  - toStopTimeOutMillis parameter `0` specifies if collector is inactive for very short time (i.e. `stopTimeOutMillis(5000)`) then we don't need to cancel it for short flashes and small values

(B) Replay!

PAGE



So reply = In (replay expression is missing) then when starts the reobserve flow / recollect flow traces after configuration changes (other) that numbers most recent emissions we get it down.

State flow is a shared flow

(source: https://pub.dev/packages/shared\_flow)

State flow is a special purpose, high-performance and efficient implementation of SharedFlow for narrow but widely used case of sharing a state across multiple subscribers.

→ though StateFlow always has an initial value to return subscriber's form of best links

↳ Value Counter = MutableStateFlow(0)

p.f(counter.value) → 11 (0) → 11 (3)

Counter.value = 3

p.f(counter.value) → 11 (3)

↳ initial value → 3

counter.emit(5) → 11 (Compile Time Error)

emit is suspend function

need to call from coroutine or other suspend fun.

But works as

However, setting a new value like this here by directly assigning a new value to value property of the state flow can lead to problems in multithreaded

scenarios where we modify the value from different coroutines.

See here

val Counter = MutableStateFlow(0)

p.f(counter.value) → 11 (0)

and repeat → 11 (1000) → 11 (1000)

↳ coroutineScope{ conf suggests to 11 (1000)

more information repeat(1000) leads to infinite loop

↳ launch { + 10 sec wait time } loop

↳ counter.value = counter.value + 1

p.f(counter.value)

11 7812

(expected 10000)

↳ so it's not thread safe

Now,

Let's fix this addressing

↳ Value Counter = MutableStateFlow(0)

p.f(counter.value) → 11 (0) → 11 (3)

CoroutineScope {

repeat (10\_000) {

launch {

counter.update { value → }

value + 1 → 11 (10000)

→ error of breaking p.f(p.f(counter.value)) → 11 (10000)

}

(unintended infinite loop)

p.f(counter.value) → 11 (10000)

↳ conf = coroutineScope {

↳ so update { } function on state flow mode code

↳ threadSafeMode

(1) time

(2) value

Convert (float) / (double) into State Flow

(1) time → 11 (10000)

↳ ValueFlow: (float | double) = flow {

↳ Introduce repeat(10) {

↳ Introduce emit(it)

↳ delay(100)

}

↳ stateIn

↳ initialValue = -1

↳ started = SharedStartedWhileSubscribed(5000)

## Difference Between Shared Flow vs State Flow

(cont'd. page 2)

```
emit (1)    2  
delay (10)  hand  
emit (1)  
delay (10)  
emit (2)  
}. SharedIn(1)
```

Init =  $\{ \text{Scope} \in \text{VicinoModelScope} \}$

3 Started = Sharing Started.

(ii) + while statements

Call reply 2/1

Launch }

## Latitude - L

Steady flow. Collected {  
} (contd) Had 33.4 g. of P.P.C. (H) + 10 g. of sand + water.

O/p: 1

1

2

4

State flow (distinct until change of behaviour)

horizontal stateflow flow from event emit  
emit(1) . state2 emit  
delay(10)  
at position -fraction 0.2 emit(1) p. -201 state2  
1920-000 > ai value delay(10).1 from base  
is positioned 2920 H200 emit(2) pd mindest  
g. stateIn(EmptyP. 20100 \*  
Scope = Viewmodel Scope

2-branch loop lineage initial values =  $\{0, 0\}$  (1)  
 launch {  
 } const char\* str\_dblout  
 stockflow.collect { P.FC(it) }  
 3. initial work bolt =  
QIP: add folg 2 → this work = 3 next lineage  
 loops  $\frac{2}{1}$  supposed to raise lineage number  
 problem

on time

Initially, value (1) is emitted three times (initial value and two times emit(1)). but value 1 is collected only once at initially. When value 2 emitted then after value 1 is emitted.

So state flow <sup>has</sup> is distinct until change behaviour.



val flow = flow {

repeat(5) { - emit 1st

? (2) if off("Emitter: Start cooking cake \$it")

Coroutine-1 → delay(100)

channel<sup>1</sup>(1) pif(pif("Emitter: cake \$it ready"))

emit(\$it)

1. buffer()

flow. collector {

Coroutine-2 printf("Collector: Start eating cake \$it")

delay(300)

pif("Collector: finished eating cake \$it")

}

("no other prints before this")? .p

Now when whole flow is running in a single  
flow coroutine, as call to emit is basically just  
a function call to the collect block with the emission  
as the input argument. (no other prints before this)

But when flow builder running in different coroutines  
than collector, the communication via function  
call doesn't work anymore and therefore a channel  
is used internally so that these two coroutines can  
communicate with each other.

Buffer operator uses channels under the hood  
so that coroutines can communicate with each other

(no other prints before this)

O/p: three notifications with zoomed width - 2

Emmiting by Emitter: Start cooking cake \$1 (no zoom)

-!!- : cake 0 ready

→ Other Collector: Start eating cake with hot

water for Emitter and Start cooking cake \$1, rotating

water until cake is steady and notification ni

-!!- : start cooking cake 2

notification node: -!!- : cake 2 ready and cook

notification node: -!!- : start cooking cake 3 (no zoom)

notification node: collector: finished eating cake \$1 (no zoom)

notification node: -!!- : start eating cake \$1 (no zoom)

Emitter: cake 3 ready

-!!- : start cooking cake 4

-!!- : cake 4 ready

Collector: finished eating cake \$1

-!!- : start eating cake 2

-!!- : finished eating cake 2

-!!- : start eating cake 3

-!!- : finished eating cake 3

-!!- : start eating cake 4

-!!- : finished eating cake 4

Collector: start eating cake \$3

Collector: finished eating cake \$3

## buffer

fun<T> flow<T>.buffer(capacity: Int = BUFF\_SIZE,  
onBufferOverflow: BufferOverflow =

BufferOverflow.SUSPEND):

Flow<T>

When we use buffer operator on our flow so that  
the emission and collection are independent of each other.

So that means the emitter doesn't emit anymore until the collector has processed previously emitted item.

And this work because by using the buffer operator, the code in flow builder and the code in collector runs in two separate coroutines.

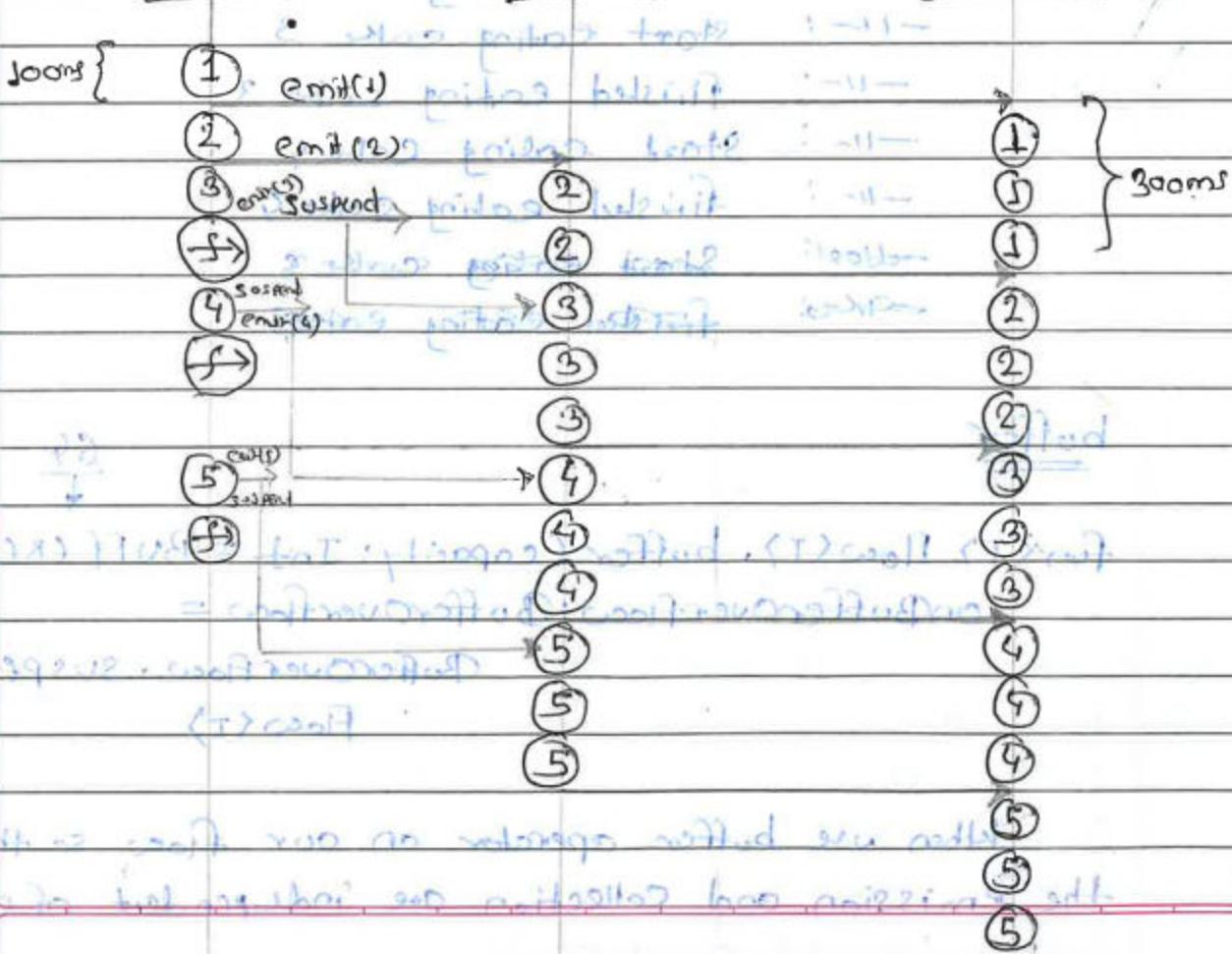
Now here emitter is more faster than collector. In other words Producer is more faster than consumer and probably that situation is known as Backpressure. However in kotlin this is called as BufferOverflow.

### ① onBufferOverflow = BufferOverflow.SUSPEND

↓ when padding built-in

↓ when padding built-in

Emitter,onBufferOverflow = Collector



If the buffer is full then emitter is not emitting instead of that emitter will emit but once it tries to emit

onBufferOverflow: StartCookingCake it. They call to emit coil  
↓ when padding built-in  
Suspend. Since the buffer is full.  
val flow = flowBuilder.libIndex : -11-  
2. when repeat(5) { it + 2 } : -11-  
2. when padIndex = it + 1

The Collector on other hand continues to process.

when nothing with it if it delay(100) fi or  
it will be a nothing p.f("Emitter: p cake is cakeIndex ready")  
while loop true emit(cakeIndex) fi when  
it nothing break do .117 fi while still  
}. buffer(capacity = 1, onBufferOverflow =  
(BufferOverflow, SUSPEND))

flow.collect {

p.f("Collector: Start eating cake \$it")

when repeat delay(300) of 1000000 fi it  
nothing p.f("Collector: finished eating cake \$it")  
}

has value true nothing and no suspension of  
to OIP: Emitter: StartCookingCake 1 i will off  
waited after 2.11- when cake is ready can resume without  
Collector: Start eating cake 1

Emitter: Start cooking cake 2 until inv  
-11- : cake 2 ready  
-11- = startCooking cake 3

Emitter: Cake 3 ready

Collector: Finished eating cake 1

Emitter: 2.11- isStart eating cake 2

Emitter: 3.11- StartCooking cake 4

-11- : cake 4 ready

when repeat(10) collector: finished eating cake 2

when repeat(10) Emitter: Start eating cake 5

Emitter: 4.11- : cake 5 ready

Collector: finished eating cake 13:12 : 9:13

-11- : Start eating cake 4

-11- : finished eating cake 4 wait 10s

-11- : Start eating cake 5

-11- : finished eating cake 5

"11:12:13:14:15:16:17:18"

what was

So if the Buffer is full then emitter is no.

Stop emitting whereas emitter will tries to

emits it, they call to emit will suspend since

the buffer is full. On other hand collector is

continues to process that.

"11:12:13:14:15:16:17:18"

② onBufferOverflow = BufferOverflowStrategy.DROP\_OLDEST

"11:12:13:14:15:16:17:18"

It is contrast to Suspend strategy and the  
coroutine of emitter will never be suspended.

So whenever we have emitter emit values and  
the buffer is full then simply oldest value out of  
buffer throws away and puts new value into buffer,  
which my collect by collector.

val flow = flow {

repeat(5) {

for (i in 1..5)

if (i == 1) p("Emmiter: Start cooking cake i")

1 emit private val cake = i + 1

if (i == 5) p("Emmiter: cake i is ready")

1 emit private emit(cake) = i + 1

1 emit private val i = i + 1

1 emit private val flow = flow {

1 emit private val flow = flow {

1 emit private val flow = flow {

flow.collect {

p("Collector: Start eating cake i")

delay(300) } wait = wait . 10s

p("Collector: finished eating cake i")

i = (integer) wait .

private val flow = flow {

emit(i)

emit(2)

emit(3)

emit(4)

emit(5)

emit(6)

emit(7)

emit(8)

emit(9)

emit(10)

emit(11)

emit(12)

emit(13)

emit(14)

emit(15)

emit(16)

emit(17)

emit(18)

emit(19)

emit(20)

emit(21)

emit(22)

emit(23)

emit(24)

Buffer

Collector

full . 10s

③  $\text{onBufferOverflow} = \text{BufferOverflow} \cdot \text{DROP}_{\text{OLDEST}}$

$\text{val\_flow} = \text{flow} \{$

$\} \cdot \text{buffer}(\text{capacity} = 1,$

$\text{onBufferOverflow} = \text{BufferOverflow} \cdot$

$\text{DROP}_{\text{LATEST}}$

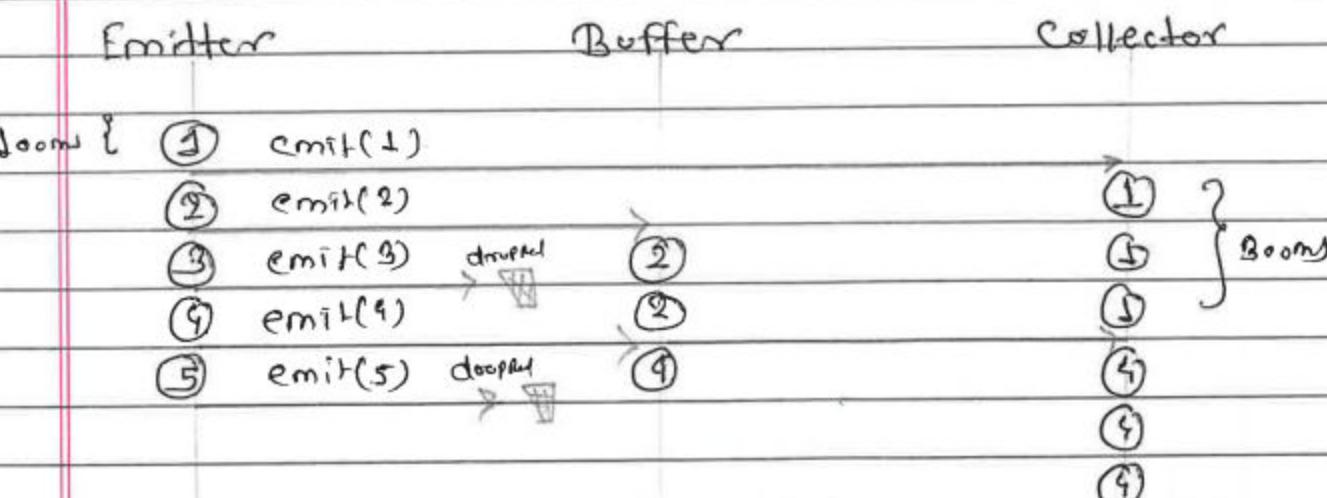
$\text{flow.collect} \{$

$\} \equiv$

$\} \}$

It is <sup>opposite</sup> to BufferOverflow.  $\text{DROP OLDEST}$ .  
Instead of throwing oldest value away newest value

Whenever a new value is emitted and buffer is full, new value is thrown away which <sup>will</sup> be collected by collector.



① Collect Latest

$\text{tot flow} = \text{flow} \{$

$\} \cdot \text{repeat}(5) \{$

$\text{val\_flow} = \text{flow} \{$

$\} \cdot \text{repeat}(5) \{$

$\text{emit("emitter": start cooking cake")}$

$\text{emit("emitter": coke & cake ready")}$

$\text{delay(300)}$

$\text{emit("emitter": coke & cake ready")}$

$\text{emit("emitter": finished eating cake")}$

$\text{delay(300)}$

$\text{emit("emitter": finished eating cake")}$

flow.collectLatest {

To box or pbar:  $\text{p.f("collector": Start eating cake $it")}$

$\text{delay(300)}$

$\text{p.f("collector": finished eating cake $it")}$

3 (final) OP: - Emitter: Start cooking cake 1  
- : Coke 1 ready  
Collector: Start eating cake 1 ignored  
- : Finished eating cake 1  
Emitter: Start cooking cake 2  
- : Coke 2 ready  
Collector: Start eating cake 2  
Emitter: Start cooking cake 3  
- : Coke 3 ready  
Collector: Start eating cake 3  
Emitter: Start cooking cake 4  
- : Coke 4 ready

Collector: start eating cake 4

Emitter: Start cooking cake 5

-||- : cake 5 ready -> wolf eat

Collector: start eating cake 5

-||- : finished eating cake 5

(phase 1 emitting nothing) T.P.

Collector Latest:

{ wolf = wolf + 10v

{ (2) sugar

Every time collector upstream emits a new

item then collector latest block is immediately

concluded and restarted with the new item.

(phase 2 starts emitting) T.P.

So collector latest is useful when you have a slow collector and only care about the most recent emission and therefore you don't want to process outdated emissions.

{ 1. wolf += .wolf

(if collector latest runs concurrently instead of

sequentially. (002) p10n

(it emits nothing until now) T.P.

public suspend fun <T> flow<T>. collectLatest(action:

+ along pipeline toSuspendIf(value: T) → Unit {  
phase 1 emit : -||-

mapLatest(action). buffer(capacity=0). collect()

+ along pipeline toSuspendIf(value: T) → Unit {  
phase 2 emit : -||-

Buffer of capacity zero is inserted

firstly inflow. And that explains why our

flow-hands runs concurrently.

phase 2 emit : -||-

+ along pipeline toSuspendIf(value: T) → Unit {  
phase 3 emit : -||-

+ along pipeline toSuspendIf(value: T) → Unit {  
phase 4 emit : -||-

② mapLatest & 4+docs for Latest, n/a (continues)

(phase 2 emit : -||-

val flow = flow { 2 as pizza val. wolf

{ repeat(5) { emit("rotating") }

{ emit val.cake = it + 1 -||-

q.f("Emitter: Start cooking cake & cake")

delay(100)

q.f("Emitter: Cake 1 & cake ready")

emit(cake)

(phase 2 emit : -||-

(if no collisions with last value in

buffer -> runs mapLatest("Adding icing on 1") id

delay(200)

(if no collisions with last value in

buffer -> runs mapLatest("Adding icing on 2") id

(if no collisions with last value in

buffer -> runs mapLatest("Adding icing on 3") id

flow.collect {

q.f("Collector: Start eating cake 1")

delay(300)

q.f("Collector: Finished eating cake 1")

{ emit : -||-

O/P: Emitter: Start cooking cake 1.

Emitter: cake 1 ready

Adding icing on 1 { emit : -||-

Emitter: Start cooking cake 2.

-||- : cake 2 ready

Adding icing on 2

along with Emitter to start cooking cake 3 { emit : -||-

along with Emitter to start cooking cake 3 { emit : -||-

Emitter: Start cooking cake 4

-||- : cake 4 ready

Adding icing on 4



Emitter: Start cooking cake 5  
- n - : cake 5 ready

Adding top icing on 5 [count = count + 1]

Collector: start heating cake 5

- n - finished heating cake 5

(002) plug 1 rot2 (rot1) "17.p"

(002) plug 1

### MapLatest & TransformLatest

(order 2 times)

Now mapLatest and transformLatest is useful in situations where there's a slow operation in the blocking code (again) don't care about outdated emissions.

(002) plug 1

In mapLatest and transformLatest every time

there's a new emission it's upstream - its plug is

cancelled and restarted with the new value.

3 blocks total

("this value going back to block 1") 7.q

### ③ Conflate operator

(008) plug 1

("this value going back to block 1") 7.q

val flow = flow {

    } =

    } . conflate (maxTime: 900)

        where t who's within

        flow.collect { who's plug 1 within }

            start (plug 1) before maxTime

            } where t who's : -1 -

            . 0 no [prior] (nilData)

conflate operator is a short cutting of buffer operator with capacity of channel: CONFLATED and onBufferOverflow

= BufferOverflow: PROP\_OVERFLOW, plug 1

    1 0 to 3 plug 1 from 2 (within?)

        (boot p 0 to 3 : -1 - 1

            (value)

    0 no [prior] (nilData)

### Buffer in shared flow

val flow = MutableSharedFlow<Int>()

// Collector 1

launch {

    flow.collect { println("1: \$it") }

    p.f1("Collector 1 processes \$it")

}

    } // initializing all 3 collector stations at

    // message with values will return only one to

    // the collector that triggered extraction

    // launch 3 from 3 in waiting state goes to a

    flow.collect {

        println("2: \${p.f1("Collector 2 processes \$it")}")

        // settings delay(100) so no active han-

        // dle type setting message when loop 2 has nothing

        // will go to log calm long if both han - same hi

        // will error out since handle to han - remains but

        // using 11 (emitter). what goes around comes around

        // now launch 3 start at 1st han after other didn't

        val timeToFirst = measureTimeMillis {

            repeat(5) {

                flow.emit(5)

                delay(10) until flow.isEmitted(5) and

                } } = first : (01920)

        } } // for i in range(0, 1000000)

        = com.p.f1("Time to emit all values: \${timeToFirst ms}")

        } // conflate operator

        } // conflated default :

        Collector 1 processes 0

        -1n 2 -1n 0

        -1n 1 -1n 1

        -1n 2 -1n 1

Collector 1 processes 2

-1n 2 -1n 2

-1n (1 + 1) mod 3 and 2. idletime = 100 ms low

-1n 2 -1n 3

-1n 1 -1n 4

L rot 110 11

-1n 2 -1n 4

I down

Time to emit all values: 4.32 ms

("110 298202000 2 rot 110 11" p)

In next slot Shared flow if the collector is slower than emitter then emitter will suspend till collector process not complete and once collector block emit complete emitter will emit another value.

] rot 110 11

If (the shared flow has two different collectors and one of them is slower) then emitter still emitter will suspend until slower collector not process it value and that impact also get on another fast consumer and it should wait for new value. Once slower consumer complete then only emitter emit new value and get to both consumer and so on.

Waiting time = time of emit low

MutableSharedFlow (2) longer

(2) long wait

fun <T> MutableStateFlow<L>() {

replay: Int = 0,

extraBufferCapacity: Int = 0

(on sendPoint: T) on BufferOverflow: BufferOverflow =

bufferOverflow, SUSPEND

): MutableSharedFlow<T>

0 110 298202000 1 rot 110 11

0 -1n 2 -1n 2

L -1n 2 L -1n 2

L -1n 2 L -1n 2

Buffer 10 Shared flow

Replay + (0 Extra 7 11 12 Total 14 = 298202000 low

Cache Buffer Buffer

L rot 110 11

Increase the buffer capacity either increases by Replay each or extra buffer both by increase both.

(48 ms wait L rot 110 11 "11" p)

so the OIP of previous code by adding extraBufferCapacity = 10 i.e.

val flow = MutableSharedFlow<Int>(extraBufferCapacity: 10)

? down

OIP: Collector 2 processes 2

0 110 298202000 1 rot 110 11

-1n 2 -1n 2 (only 2 produced)

-1n 2 -1n 3

-1n 2 -1n 4

Time to emit all values: 61 ms

Collector 2 processes 1

-1n 2 -1n 2 (all 2 produced)

-1n 2 -1n 3

-1n 2 -1n 4

Waiting time = time of emit low

-1n 2 -1n 4

(11) 110 ms wait

0 110 298202000 1 rot 110 11

L -1n 2 L -1n 2

L -1n 2 L -1n 2

L -1n 2 L -1n 2

0 110 298202000 1 rot 110 11

L -1n 2 L -1n 2

L -1n 2 L -1n 2

L -1n 2 L -1n 2

Buffer in State flow with bound of 27ms

```
val flow = MutableStateFlow(0) +  
    ~After 27ms 27ms
```

// Collector 1

if launch { emits values with current  
and not flow.collect{ emit values to flow}

```
    pf("Collector 1 Process $it")
```

```
} possible for this waiting to go with or  
    .onEach =
```

```
+1 ms //Collector(2) conf 1ms old 1ms = 2ms late
```

launch {

```
flow.collect{ is now 1ms 1ms}
```

```
pf("Collector 2 process $it")
```

```
delay(100) -> 1 ->  
    & -> 1 ->  
    } 1 -> 1 ->
```

2nd : now 1ms time of 3ms

// Emitter 1 ms 1ms 1ms

launch {

val timeToEmit = measureTimeMillis {

repeat(3) {

```
    flow.emit(it)
```

```
    delay(10)
```

```
}
```

```
}
```

O/P : Collector 1 Process 0

```
-1 2 -1 0  
-1 1 -1 1  
-1 1 -1 2  
-1 1 -1 3  
-1 1 -1 4
```

Time to emit all values: 64ms

(from) collector (2 processes) 64ms = 27ms inv

3 don't 27ms

In Stateflow, the emitter is never suspended.

→ Stateflow collector always processes all items however

? (Custom-Block-Collector) collects only processes most

recent items until it is

Mutable State flow 3.7.11-2

(H) 7ms

real \$ 3 (s = fi 2)i

(initial) Mutable State flow (initial value) is a Shared flow  
with the following

val shared = MutableSharedFlow( )

replay = 1

On BufferOverflow (BufferOverflow.DROP  
OLDEST)

shared.emit(initialValue) // emit initial value

val state = shared.distinctUntilChanged

// get LStateflow-like behavior

So In Stateflow if collector is slower than  
emitter then potentially loses the emissions.

now 1ms 1ms 1ms 1ms 2 1 2 3 4

emit later problem if you emit 2 1 before 1st  
bottom line no notifications sent outside within 20ms

.1ms



(In above example if we comment the line `currentCoroutineContext().ensureActive()` then the code is not in cooperative reggrading cancellation and after the cancel() call we get following OLP.

```
OLP: 1. oldmethod() {
    2. if(gasx) unitlloop() {
        3. if(cancelled()) {
            "hasStartCalculation"
        }
        calculationFinish {
            flow got cancelled.
        }
    }
}
```

Here after cancel() call still calculateFactorial function is executed.

Now if we remove the comment of `currentCoroutineContext().ensureActive()` then the code is cooperative regarding cancellation and after cancel() call we get following OLP.

```
OLP: 1. flow = ()>Unit! not start()
2. (1) times
Flow got cancelled.
```

(\* iteration after cancel() call no more

(\*) Another approach to make code cooperative by using cancellation first check `ensureActive` in every before every iteration of `inflow` loop.

```
private fun inflow() = flow{
```

```
emit(1)
```

```
    ... emit(2) ...
```

```
    repeat(10): (it: Int) {
        if("Start calculation")
            sum += it
        calculateFactorial(1..it)
    } (n... if("calculation finish")
```

```
    n+1
    emit(3)
}
```

```
private suspend fun calculateFactorial(number: Int) {
    val BigInteger = coroutineScope {
```

```
        val factorial = BigInteger.ONE
```

```
        for(i in 1..number) {
            factorial = factorial.multiply(BigInteger(
                valueOf(i.toLong())))
        }
    }
}
```

```
    if(!canceling || !ensureActive()) {
        flow = it
    }
    factorial
}
```

```
    if(!canceling || !ensureActive()) {
        flow = it
    }
    factorial
}
```

```
OLP: 1
"flow" = ()>Unit! not start()
Start calculation
```

```
Flow got cancelled. 3 times
(2, 3, 4) times
← oldmethod() notInScope
```

```
3 (emptyUnitLoop() zi oldmethod())
val scope = CoroutineScope(emptyCoroutineContext)
```

```
scope.launch {
```

```
"flowOf(1,2,3,4)" IT
```

```
val doIt() {
    flowOf(1,2,3,4).onCompletion { throwable -->
        if(throwable is CancellationException) {
            Unit
        } else {
            if("flow got cancelled") {
                ...
            }
        }
}
```

```
.onEach { collect {
```

```
(it: Int) &gt; if(it == 1) {
```

```
(it == 2) &gt; if(it == 2) {
```

```
(it == 3) &gt; conceal()
```

(1, 2, 3, 4)  $\rightarrow$  initial state & no flow being shown  
 $\{ \text{scope} \text{.init}(\text{it}(1, 2, 3, 4)) \} = \text{repeat}(\text{it}) :$   
 1  
 2  
 3  
 4

$\text{it}(1, 2, 3, 4) = \text{initial-flow}$

Now here an interesting thing no flow happens that flow  
 $\text{it}(1, 2, 3, 4)$  is never cancelled and all of the values collected.

$(\text{repeat}.i)$  reason

The flow of () builder does not internally check  
 whether the coroutine is still active or not.

initial

So we have to manually / explicitly checks by  
 our self.

$\pm : 912$

```
val scope = CoroutineScope(EmptyCoroutineContext)
    .init(it(1, 2, 3, 4))
scope.launch {
    flowOf(1, 2, 3, 4)
        .onCompletion { throwable ->
            if (throwable is CancellationException) {
                (it as Job).cancel("if flow got cancelled")
            }
        }
    }.onEach {
        it.collect { value ->
            if (value == 2) throw CancellationException("throw cancellation exception")
        }
    }.ensureActive()
    (it as Job).cancel("cancel")
}
```

• collect { it ->

(+ i) & if (it == 2)

{ cancel() }

{ cancel() }

{ cancel() }

O/P:  $\frac{1}{2}$  Receive 1 in onEach

1  
 Receive 2 in onEach

2  
 Receive 3 in onEach  
 flow got cancelled.

Also .cancelable() can be use as follows.

val scope = CoroutineScope(EmptyCoroutineContext)

scope.launch {

flowOf(1, 2, 3, 4).

.onCompletion { throwable ->
 if (throwable is CancellationException) {
 if ("flow got cancelled")
 }
}

• cancellable()

• collect {

if (it == 2) {

cancel()

}

}

O/P:  $\frac{1}{2}$

flow got cancelled.