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.

Example - ① Ordinary Job

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

scope.launch {
    try {
        throw RuntimeException()
    } catch (ex: Exception) {
        Print("Exception: $ex")
    }
}
```

O/P: Exception: RuntimeException ✓

Example - ②

```
runBlocking {

    try {
        launch {
            throw RuntimeException()
        }
    } catch (ex: Exception) {
        Print("Exception: $ex")
    }
}
```

O/P: Crash: Due to RuntimeException ✗

But what happen if CoroutineExceptionHandler installed in child coroutine which is not recomended:

Example (3)

```
val exceptionHandler = CoroutineExceptionHandler { _, _ }

val scope = CoroutineScope(Job())

scope.launch {
    launch(exceptionHandler) {
        throw RE();
    }
}
```

X

O/P: Crash

The cancellation exception which are the result when Cancelling coroutine, are not get handled by coroutine exception handler.

③ Try-Catch VS CoroutineExceptionHandler

Example (1)

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

scope.launch {
    launch {
        try {
            P.f("Job-1 Started")
            throw RuntimeException();
        } catch (ex: Exception) {
            P.f(ex)
        }
    }
```

Example (2)

```
    launch {
        P.f("Job-2 started")
        delay(200)
        P.f("Job-2 completed")
    }
}
```

O/P: Job-1 Started
RuntimeException
Job-2 started
Job-2 completed

```
val exceptionHandler = CoroutineExceptionHandler { _, throwable →
    P.f("exception: $throwable")
}

val scope = CoroutineScope(Job())

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")
    }
}
```

O/P: Job-1 Started
Job-2 Started
Exception: RE

In Example-①, Coroutine-1 thrown exception which is catched. But that did not impacted on Coroutine-2 and it completed.

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

④ Launch { } Coroutine builder
Vs
async { } — " —

Example ①

```
val exceptionHandler = CoroutineExceptionHandler { }
val scope : CoroutineScope( Job() )

scope.launch {
    throw RIException()
}

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

Example ②

```
val exceptionHandler = CoroutineExceptionHandler {
    _, throwable →
    P.f("Exception: $throwable")
}
```

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

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

O/P: { Nothing Printed } → Exception also not printed in CoroutineExceptionHandler

Example-④

```
val exceptionHandler = CoroutineExceptionHandler {
    _, throwable →
    P.f("Exception: $throwable")
}

val scope = CoroutineScope( Job() + exceptionHandler)

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

O/P: Exception: RuntimeException

Example-⑤

```
val exceptionHandler = CoroutineExceptionHandler { _, throwable →
    P.f("Exception: $throwable")
}
```

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

scope.launch {
    async {
        throw RuntimeException()
    }
}
o/p: Exception: RuntimeException

scope.async {
    async {
        throw RuntimeException()
    }
}
o/p: { Nothing printed even exception handler }
```

Uncaught exceptions in both launch and async coroutines are immediately propagated up the job hierarchy.

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

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

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

⑤ CoroutineScope

```
try {
    coroutineScope {
        launch {
            throw RuntimeException()
        }
    }
} catch (ex: Exception) {
    p.f(ex)
}
```

o/p:

RuntimeException

The scoping function is called in scope rethrows exceptions of its failed child coroutines instead of propagating them up the job hierarchy which allows us to handle exceptions of failed coroutine with an outer try/catch.

⑥ SupervisorScope

Example ①

```
runBlocking {
    try {
        supervisorScope {
            launch {
                throw RuntimeException()
            }
        }
    } catch (ex: Exception) {
        p.f(ex)
    }
```

Top level coroutine → ✗

The Supervisor scope create 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.

And because of these reasons we have to handle the exception by ourself.

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 deal with all exception ourselves.

Run intended, If an exception is not thrown in a coroutine 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 try catch.

Example ②

```
runBlocking {

    try {
        supervisorScope {
            throw RuntimeException()
        }
    } catch (ex: Exception) {
        print("exception: $ex")
    }

}
```

o/p: exception: RuntimeException

Imp

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

Imp

The supervisor scope create new independent scope here with supervisor scope scoping function coroutine that we launch directly are now also top level coroutine.

Example ③

```
try {
    supervisorScope {
        async {
            throw RuntimeException()
        }
    }
} catch (ex: Exception) {
    print("exception: $ex")
}
```

o/p:

{ Nothing printed }

Example ④

```
val exceptionHandler = CoroutineExceptionHandler {
    _, throwable ->
    print("exception: $throwable")
}
val scope = CoroutineScope(Job() +
```

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

Top Level Coroutine →

o/p:
Exception: RuntimeException

Example ⑤

```
val exceptionHanding = CoroutineExceptionHandler {
    _, throwable →
    print("Exception: $throwable");
}

val scope = CoroutineScope(Job() + exceptionHandler)

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

top level Coroutine →

o/p: Exception: RuntimeException

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

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

So as per SupervisorScope document
" The provided scope is inherits ~~from~~ its coroutine context from the outer scope, but overrides context's Job with supervisorJob.

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

NP (verified) ①

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

op: {Nothing} (Not crashed) ✓

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

②

```
val deferred = async {
    launch {
        throw RE()
    }
}
deffered.await()
```

o/p: Crash: RuntimeException ✗

PTO

```
try {
    launch {
        throw RE()
    }
} catch (e) {
}
```
[X]

```
try {
    async {
        throw RE()
    }
} catch (e) {
}
```
[✓]

```
try {
    CoroutineScope {
        launch {
            throw RE()
        }
    }
} catch (e) {
}
```
[✓]

```
try {
    CoroutineScope {
        async {
            throw RE()
        }
    }
} catch (e) {
}
```
[✓]

top level coroutine

```
try {
    SupervisorScope {
        launch {
            throw RE()
        }
    }
} catch (e) {
}
```
[X]

In this case all coroutines started in that scope will be cancled

```
try {
    SupervisoryScope {
        async {
            throw RE()
        }
    }
} catch (e) {
}
```
[✓]

Sequential or Series Excuation

```
suspend fun networkCall1(): Any {
    ==
    return data
}

suspend fun networkCall2(): Any {
    ==
    return data
}

suspend fun networkCall3(): Any {
    ==
    return data
}

launch {
    val data1 = networkCall1()
    val data2 = networkCall2()
    val data3 = networkCall3()
}
```

} Sequential OR Series

Concurrent or Parallel Execution

① With launch

```
launch {
    launch {
        networkCall1()
    }
    launch {
        networkCall2()
    }
}

launch {
    val job1 : Job = launch {
        delay(5000)
    }
    val job2 : Job = launch {
        delay(5000)
    }
    job1.join()
    job2.join()
}
```

Concurrent

Total time taken 5000ms

② with async

```
launch {
    val deferred1 = async {
        delay(5000)
        "data-1"
    }
    val deferred2 = async {
        delay(5000)
        "data-2"
    }
    deferred1.await()   // data-1
    deferred2.await()   // data-2
}
```

Parallel

Total time taken 5000ms

ok

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

Timeout

① withTimeout (delay : Int)

```
launch {

    try {
        val data = withTimeout ( 1000) {
                delay ( 1500)
                return "data"
            }
    } catch ( ex : TimeoutCancellationException) {
        P.f( ex)                    // Timeout
    } catch ( ex: Exception) {
        P.f( ex)
    }
}
```

② withTimeoutOrNull (delay : Int)

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

    P.f( data)      // null
}
```

Higher Order function

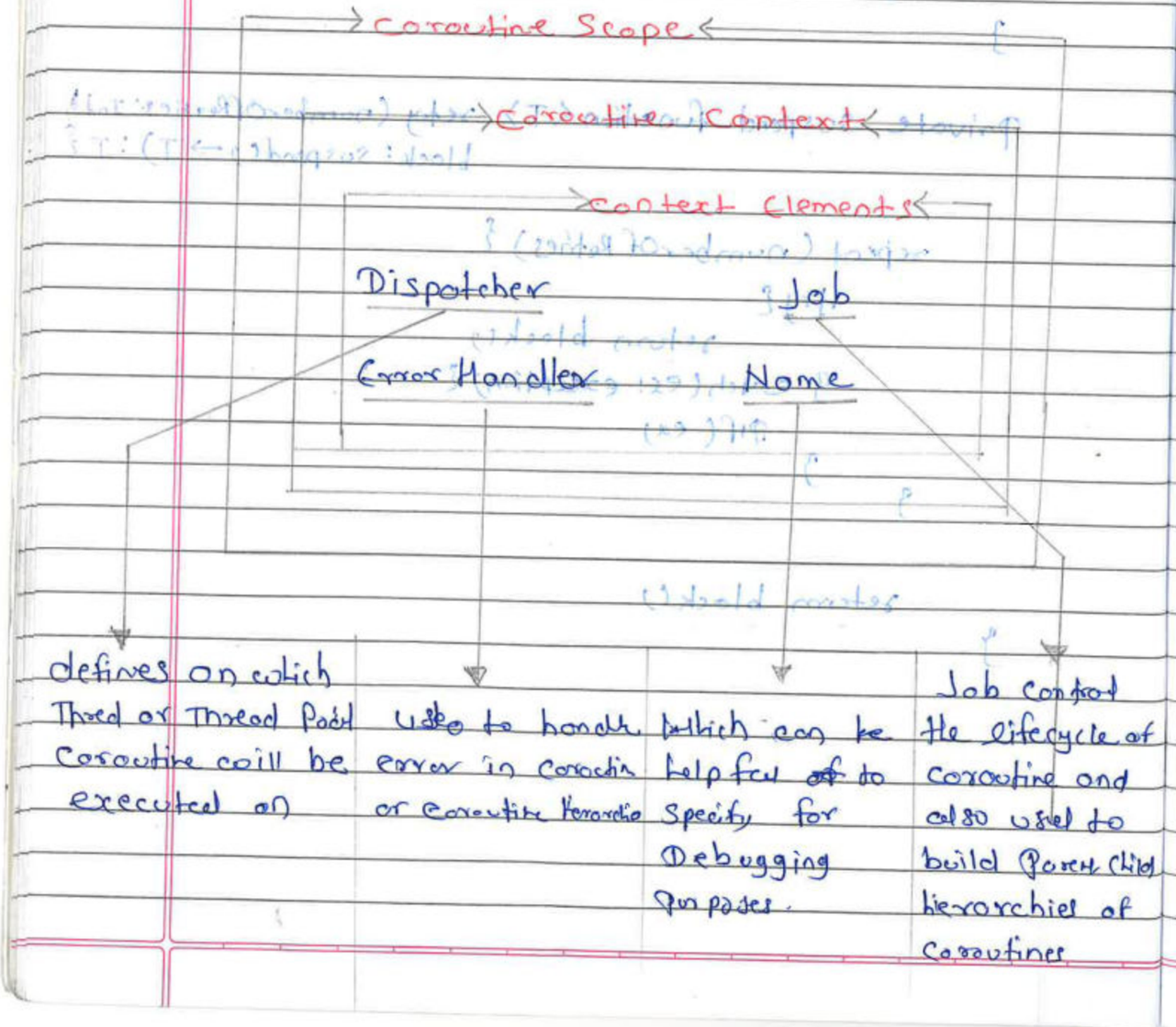
```
launch {

    val numberOfRetries = 2
    try {
        retry ( numberOfRetries) {
            networkCall()
        }
    } catch ( ex : Exception) {
        P.f( ex)
    }
}

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

    repeat ( numberOfRetries) {
        try {
            return block()
        } catch ( ex: exception) {
            P.f( ex)
        }
    }

    return block()
}
```

Coroutine Context

Coroutine Context is at the core of every coroutine.

Every coroutine starts is tied to a specific coroutine context.

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



ViewModel Scope

- Context Elements
  1. Dispatcher — Main
  2. Job — SupervisorJob
  3. Error Handler — Null
  4. None - Default

Dispatchers

① **Dispatchers.Main**

- Will launch coroutine on Android Main thread (UI thread)

② **Dispatchers.IO**

- Perform IO-related operations (N/W, file handle)
- Uses shared thread pool internally,
- Limited to 64 threads (But can configure for more)

③ **Dispatchers.Default**

- Default dispatcher (if no other is define)
- Optimised for CPU-intensive work (like, Sorting list, heavy calculations, Parse big JSON file)
- Uses shared thread pool internally, the max number of threads is equal to number of CPU cores (at least two) (2, 4, or 8 threads)

Page No.
Date

(4) Dispatchers. Unconfined

- Not confined to any thread
- Initial running on thread the coroutine was started on.
- Might switches thread on context switches with suspend functions
- 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 Coroutine Dispatcher

```
Coroutine public fun CoroutineScope.launch(
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.Default,
    block: suspend CoroutineScope.() → Unit): Job {
}
```

All our Coroutines started till now operated on an empty coroutine context EmptyCoroutineContext?
NO

Page No.
Date

When a new Coroutine starts with launch then the context from coroutine scope on which launch called is inherited for the new coroutine.

If a specify certain context elements in the context parameter of launch then these context elements then override the context elements of the scope.

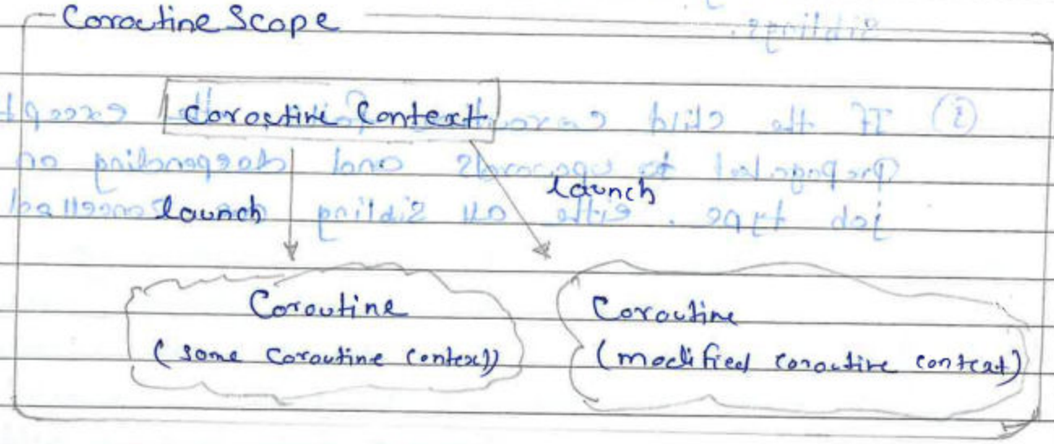
Suspend function →

withContext(Dispatcher. { Main / Default / IO }) {

}

for switching the context.

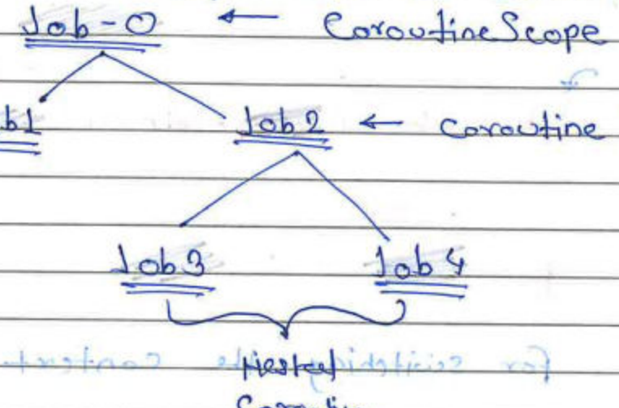
withContext is most specifical used for switching the Dispatcher. But can also switch to another ErrorHandler, Job and Name but not a good practice



Structured Concurrency

① Every Coroutine needs to be started in a logic scope with a limited life-time.

② Coroutines started in the same scope form a hierarchy.



③ Parent Job won't complete until all of its children have completed.

④ Cancelling a Parent will cancelling 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 the job type, either all Sibling are cancelled or not

```
val scope = CoroutineScope(Dispatcher.Default)
scope.launch {
}
val job = scope.launch {
}
```

scope.cancel()

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 the job of its scope as its parent job.

```
val scopeJob = Job()
val scope = CoroutineScope(Dispatcher.Default
                           + scopeJob)
val coroutineJob = scope.launch {
}
scopeJob.children.contains(coroutineJob) // true
```

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

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

Here a new job hierarchy created which would have to manage by ourselves.

According to Android team" We don't recomended passing Jobs in the context parameter to Coroutine builders in modern code, through"

↓

```
val scopeJob = Job()
val scope = CoroutineScope(Dispatchers.Default +
                           scopeJob)
val passedJob = Job()
val coroutineJob = scope.launch(passedJob) {
}
passedJob == coroutineJob // false
scopedJob.children.contains(coroutineJob) // false
```

Job & SupervisorJob

① Job (Ordinary Job)

example :

```
val exceptionHandler = CoroutineExceptionHandler {
        coroutineContext, throwable →
        P.f("Exception: $throwable")
}
val scope = CoroutineScope(Job() + exceptionHandler)

scope.launch {
    P.f("Coroutine-1 starts")
    delay(50)
    P.f("Coroutine-1 fails")
    throw IRE()
}
```

→ In Same Scope and are siblings

```
scope.launch {
    P.f("Coroutine-2 starts")
    delay(500)
    P.f("Coroutine-2 got finish")
}.invokeOnCompletion { throwable →
    if(throwable is CancellationException) {
        P.f("Coroutine-2 got cancelled")
    }
}

P.f("Is Scope Active: ${scope.isActive}")
```

o/p:

Coroutine - 1 starts
— 11 — -2 —11—
— 11 — — 1 fails
— 11 — -2 got cancelled!
Exception: java.long.RuntimeException
IsScope is Active: false

"In ordinary Job if any child Coroutine fails then all its siblings and its scope (Parent Job) get cancelled."

(2) Supervisor Job

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

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

then the op as below

o/p: Coroutine - 1 Starts
— 11 — -2 —11—
— 11 — 1 fails
Exception: java.long.RuntimeException
Coroutine - 2 Complete
Is Scope Active: true

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

"ViewModelScope and LifeCycleScope has SupervisoryJob"

Structured Concurrency

"Coroutines needs to run in a certain scope with limited lifetime and these tasks then form a hierarchy by automatically. The hierarchy put some very useful automatic cancellation and exception handling mechanisms in place."

- ViewModelScope
- LifeCycleScope

Unstructured Concurrency

Unstructured Concurrency can achieved by GlobalScope.

Global Scope

A Global Scope or Global Coroutine Scope is not bounded to any Job.

Global scope is used to launch top-level coroutines which are operated in whole application lifetime and are not cancelled permanently.

Global Scope operators running in Dispatcher.Unconfined which does not have any job assisted attached to it.

GlobalScope.coroutineContext[Job] // null

The GlobalScope does not have any

So no job object associated with the Global Scope. This mean that no hierarchy of job objects will be formed when you launched a new coroutine, in the global scope.

So when a new Coroutines will be formed in global Scope started wouldn't have any connection to the scope. It has initial behind the

The only way to form relationship bet^n coroutin and scope is could be via their job objects.

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

Dispatcher Main vs Main.Immediate

Ex-①: launch(Dispatcher.Main){
p.f("first Statement In Coroutine")
delay(10)
} Pf("Second Statement in Coroutine")
p.f("first statement after launch coroutine")

o/p: First statement after launch Coroutine
first statement in coroutine
Second Statement in coroutine

Ex-②: launch(Dispatcher.Main.immediate){
p.f("first statement in coroutine")
delay(10)
} Pf("Second Statement in Coroutine")
p.f("first statement after launch coroutine")

o/p: first statement in coroutine
first statement after launch coroutine
Second statement in coroutine

ViewModelScope has Dispatcher.Main.immediate.
LifecycleScope has Dispatcher.Main.immediate.

Life Cycle Scope

for Activity & fragment

① lifecycleScope.launch { }
This coroutine scope is tied to LifecycleOwner's of Lifecycle. The scope will be cancled when the lifecycle is destroyed. Also the running coroutine also cancelled when device is rotated because a configuration change also destroys and recreates the activity.

② lifecycleScope.launchWhenCreated { }

③ lifeCrckScope.launchWhenResumed { }

④ lifecycleScope.launchWhenStarted { }

Ex-①

```
val scope = CoroutineScope(Job())
scope.launch {
    launch { }
    launch { }
}
```

Parallel Execution

Recommended

```
scope.launch {
    coroutineScope {
        launch { }
        launch { }
    }
    launch { }
}
```

Parallel

Serial

(A)

```
scope.launch {
    val job1 = launch { }
    val job2 = launch { }
    job1.join()
    job2.join()
    launch { }
}
```

Parallel

wait until Coroutine ① & ② Complete

Serial

(B)

```
scope.launch {
    launch {
        launch { }
        launch { }
    }.join()
    launch { }
}
```

Parallel

Serial

(C)

```
scope.launch {
    doSomeTask()
    launch { }
}
```

Parallel

like
It is Coroutine builder
~~some like~~ CoroutineScope.launch()

```
fun CoroutineScope.doSomeTask() {
    launch { }
    launch { }
}
```

Parallel

Page No.
Date

(5)

```
scope.launch {
    doSomeTask()
    launch { }
}
```

Serial

it is coroutine scope

```
suspend fun doSomeTask() = coroutineScope {
    launch { }
    launch { }
}
```

Parallel

fun CoroutineScope.launch
fun CoroutineScope.myCoroutine { }

It is a coroutine builder or launcher behaves like CoroutineScope.launch{}

```
suspend fun myCoroutineScope = coroutineScope { }
```

It is a coroutine scope

Cancelling Coroutines

```
val job = launch {
    repeat(10) {
        P.f("$it")
    }
    delay(100)
}
delay(250)
P.f("Cancelling Coroutine")
job.cancel()        // cancel() is normal function not a suspend function
```

o/p : 0
1
2
Cancelling 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) {
        P.f(it)
        delay(100)
    }
}
delay(250)
P.f("Cancelling Coroutine")
job.cancel()
```

o/p: 0
1
2
Cancelling Coroutine
CancellationException thrown

```
val job = launch {
    repeat(10) {
        P.f(it)
        try { delay(100) }
        catch (e: Exception) {
            P.f("CancellationException thrown")
            throw CancellationException()
        }
    }
}
delay(250)
P.f("Cancelling Coroutine")
job.cancel()
```

Rethrown for cancelling coroutine otherwise coroutine did not cancel and loop execute continously.

Every suspend function of the Kotlin Co-routine library checks weather the coroutine in which it called is still active otherwise throws Cancellation exception.

This behaviour of our 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 therefor are cooperative regarding cancellation

```
val job = launch {
        repeat(5) {
            p.f(it)
            delay(100)
        }
}
delay(250)
P.f("Cancelling Coroutine")
job.cancel()
```

O/P: 0
1
2
Cancelling Coroutine
3
4

Above example the Coroutine does not stop even the job is called cancel on its job. That mean we have to make our own code in coroutines or own suspend functions cooperative regarding cancellation

(1) EnsureActive() & Yield()

```
val job = launch {
        repeat(10) {
            ensureActive()   // yield()
            P.f(it)
            Thread.sleep(100)
        }
}
delay(250)
P.f("Cancelling Coroutine")
job.cancel()
```

O/P: 0
1
2
Cancelling Coroutine

(2) isActive

```
val job = launch {
        repeat(10) {
            if(isActive) {
                p.f(it)
                Thread.sleep(100)
            } else {
                p.f("Clean up code")
                throw CancellationException() // return@launch
            }
        }
}
delay(250)
P.f("Cancelling Coroutine")
job.cancel()
```

O/P: 0
1
2
Cancelling Coroutine
Clean up code

Non Cancellable Code

```
val job = launch {
        repeat(10) {
            if(isActive) {
                P.f(i++)
                Thread.sleep(100)
            } else {
                delay(100)
Not reachable {  P.f("cleaning code")
  code        }  throw CancellationException()
            }
        }
delay(250)
job.cancel()  P.f("cancelling coroutine")
```

O/P: 0
1
2
cancelling coroutine

In above example print statements below the delay(100) in else block never will execute because delay has already thrown a cancellation Exception. So so cleaning code will never execute.

So to make else block executable need to write code in

```
withContext(NonCancellable) {
        delay()
}       // This code will not be cancelled
```

A Non Cancellable job is always Active. It 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(i++)
                Thread.sleep(100)
            } else {
                withContext(NonCancellable) {
                    delay(100)
                    P.f("clean up code")
                    throw CancellationException()
                }
            }
        }
delay(250)
P.f("cancelling coroutine")
job.cancel()
```

O/P: 0
1
2
cancelling coroutine
clean up code

Flow

Flow Builders

```
launch {
    collectNumbers(10).collect { it ->
        P.f($it)
    }
}

fun collectNumbers(n: Int): Flow<Int> = flow {
    for (i in 1..n) {
        delay(10)
        emit(i)
    }
}.flowOn(Dispatcher.Default)

flowOf<Int>(1).collect { it -> P.f(it) }

flowOf<Int>(1, 2, 3, 4, 5).collect{it -> P.f(it)}

listOf(1, 2, 3).asFlow().collect { it -> P.f(it) }

flow {
    delay(100)
    emit("Hi")
    emitAll(anotherFlow)
}.collect { it ->
    P.f(it)
}
```

Collect is a suspend function

Terminal Operators

① collect { it → }

② flow.first()
flow.firstOrNull()
flow.first { it > 2 }

③ flow.last()

④ flow.single() // It will crash if flow emits more than one.

⑤ toList()
toSet()

⑥ fold()
Initial value ↓ (initially accumulator is equal to initial value)
val item = flow.fold(10) { accumulator, emittedItem →
accumulator + emittedItem
}

⑦ reduce()

⑧ launchIn
val scope = CoroutineScope(EmptyCoroutineContext)
flow.onEach { it → }
.launchIn(scope)

⑨ ~~zip~~ ~~map~~

launchIn vs Collect

```
val flow = flow {
    delay(100)
    p.f("Emitting first value")
    emit(11)

    delay(100)
    p.f("Emitting second value")
    emit(22)
}

val scope = CoroutineScope(EmptyCoroutineContext)

flow.onEach { p.f("Received launchIn-1 : $it") }
    .launchIn(scope)

flow.onEach { p.f("Received launchIn-2 : $it") }
    .launchIn(scope)
```

Concurrently

O/P:

```
Emitting first value
Received launchIn-1 : 11
Received launchIn-2 : 11
Emitting second value
Received launchIn-1 : 22
Received launchIn-2 : 22
```

```
scope.launch {
    flow.collect {
        P.f("Received collect-1 : $it")
    }
    flow.collect {
        P.f("Received collect-2 : $it")
    }
}
```

Sequential

O/P:

```
Emitting first value
Received collect-1 : 11
Emitting second value
Received collect-2 : 22
Emitting first value
Received collect-1 : 11
Emitting second value
Received collect-2 : 22
```

Lifecycle Operator

① onStart()

② onCompletion()

↳ onCompletion { throwable : Throwable? → }

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

asLiveData()

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

Can pass Coroutine Context elements

LiveData operator waits for this specified timeout after the live data observer gets inactive and only then cancels the flow — 5_000 ms

When observer is in the inactive state only for a very short time (default timeOutInMs) we don't need to cancel the flow and restart it again.

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

Intermediate Operator

① map and mapNotNull

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

true → emit

false → emit

filterNotNull()

filterIsInstance<T>() ← Type to filter

③ Take (count = 3) → first 3

takeWhile { it > 3 } → take

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

O/P : (Nothing)

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

O/P : 1, 2, 3

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

```
flowOn(1,2,3,4,5)
    .map { it →
        it * 10
    }
    .collect { value →
        P.f(value)
    }

// 10, 20, 30, 40, 50
```

Map can emit only a single value ability

```
flowOn(1,2,3,4,5).
    .transform { this: FlowCollector<Int> }
        emit(it)
        emit(it*10)
    }.
    .collect { value →
        P.f(value)
    }

// 1, 10, 2, 20, 3, 30, 4, 40, 5, 50
```

Transform can emit mutiple values emits ability

⑥ withIndex()

```
flowOn(1, 2, 3).withIndex()
    .collect {
        P.f(it)
    }
```

IndexedValue(index=0, value=1)
(-"- 1, -"- 2)
(-"- 2, -"- 3)

⑦ distinctUntilChanged()

I/P: 1, 1, 1, 2, 2, 3 → O/P → 1, 2, 3

I/P: 1, 2, 1, 3, 1 → O/P → 1, 2, 1, 3, 1

Flow Exception Handling

```
val flow = flow {
        emit("Apple")
        emit("Google")
        throw Exception("Flow Exception")
    }
Launch {
    flow.map {
        throw Exception("Flow Pipline Exception")
    try {
        flow.onCompletion { cause: Throwable? ->
            if(cause == null) {
                P.f("Flow completed successfully"
            } else {
                P.f("Exception: $cause")
            }
        • collect {
            Pof("$stock")
        }
    } catch(e: Exception) {
        P.f("Exception Handled: $e")
    }
```

Coroutine Exception Handler

① ② ③ ④ ⑤

④ → catch { throwable → }

```
flow.onCompletion { cause → P.f($cause) }
    • catch { throwable → P.f($throwable) }
    • collect {
        P.f(it)
    }
```

Catch can emit value the collect

```
catch { throwable → P.f(throwable)
        emit("value") }
```

```
flow.onCompletion { }
    • onEach {
        throw Exception("Exception")
        P.f("Data: $it")
      }
    • catch { throw → }
    • launchIn(Dispatcher.Default)
}
```

Exception Transparency

Violation of Exception Transparency Rule

```
flow {
    try {
        emit(1)
    } catch(e: Exception) {
        P.f("Exception: $e")}
    }
}, collect { value →
    throw Exception("Throw Exception in Collect")
}
```

O/P:

Exception: Throw Exception in collect

Code sucessfully ran without any crash

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

emit()

↓

collect()

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

Exception Transparency Aspects

① A downstream exception must always be propagated to the collector

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

So now above example is violation of Exception Transparency Rule

fix:
```
flow {
    emit(1)
}.catch { throwable →
    Pf("Exception:" $throwable");
}.
    catch{}
.collect {
    throw exception("Throw R(")
}
```

Opp: Exception: Throws Rf
Crash the code,
Use no more violation of Exception Transparency Rule)

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

Do not violate Exception Transparency Rule

Retry() & RetryWhen

retry(

At consumer 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(retries: Long = Long.MAX_VALUE,
        predicate: suspend (cause: Throwable)
        → Boolean = {true}): Flow<T>
```

```
retryWhen { cause, attempt → Long
    Pf("No of Attempts: $attempt")
    cause is Network
}
```

Life Cycle Aware Collecting flow

① By repeatOnLifecycle

ViewModel: 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)
    }
}
```

Cold flow

```
var job = launch {
fun coldflow() = flow {
    repeat(5) { pf("emitting: $it")
        emit(it)
        delay(1000)
    }
}

fun main = runBlocking {
fun 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.

(710)

```
    delay(250)
}
```

Page No.

Page No.:

" When collecting coroutine cancled then the flow builder stopes being executing. So flow of collector on @Completed is executed and so next values are not emitted."

){

){

" Flows are cold "
" Shared flows are Hot "

Page No.
Date

Cold Flow

(Normal) flows are cold

Cold flows

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 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 bet all collectors

Producer: val sharedflow = MutableSharedflow<Int>()
launch {
repeat(5) {
P.f(" Sharedflow Emit $it ")
sharedflow.emit(it)
delay(250)
}
}

① Without Consumer:

o/p:

Without Collector — Hot flows emit value

SharedFlow emits 0
— " — 1
— " — 2
— " — 3
— " — 4

② With Consumer:

```
Thread.sleep(550)

launch {
    sharedflow.collect {
        pf("collected: $it")
    }
}
```

o/p

SharedFlow emits 0
— " — 1
— " — 2
— " — 3
} lost emission

collected 3
SharedFlow emits 4
collected 4

③

```
launch {
    SharedFlow.collect { }
        pf("SharedFlow-1 collected: $it")
    }
}
```

④

```
launch {
    sharedFlow.collect {
        pf("SharedFlow-2 collected: $it")
    }
}
```

o/p:

SharedFlow emits 0
— " — 1
SharedFlow-1 collected 1
— " — 2
SharedFlow emits 2
SharedFlow-1 collected 2
— " — 2
— " — 2
SharedFlow emits 3
SharedFlow-1 collected 3
— " — 2
— " — 3
SharedFlow emits 4
SharedFlow-1 collected 4
— " — 2
— " — 4
SharedFlow emits 5
SharedFlow-1 collected 5
— " — 2
— " — 5

Convert flow / cold flow into SharedFlow

```
val flow: Flow<Int> = flow {
    repeat(10) {
        emit(it)
        delay(100)
    }
}.shareIn(
    scope = viewModelScope,
    started = SharingStarted.WhileSubscribed(), replay = 1
)
```

shareIn

Page No.
Date

(A) Started :-

(1) SharingStarted.Eagerly

- The collection of upstream flow will happen immediately, even if nobody currently collects from our shared flow.
- And so we could lose 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 flow even when all collectors stop collecting.

(2) SharingStarted.Lazily

- This one work similar to Eagerly option with the only difference that the collection and sharing of emmision start lazily when first collector starts collecting from the shared flow.
- This guaranteed that the first collector gets all of the emitted values.
- The upstream flow continues to be collected even when all collectors stop collecting.

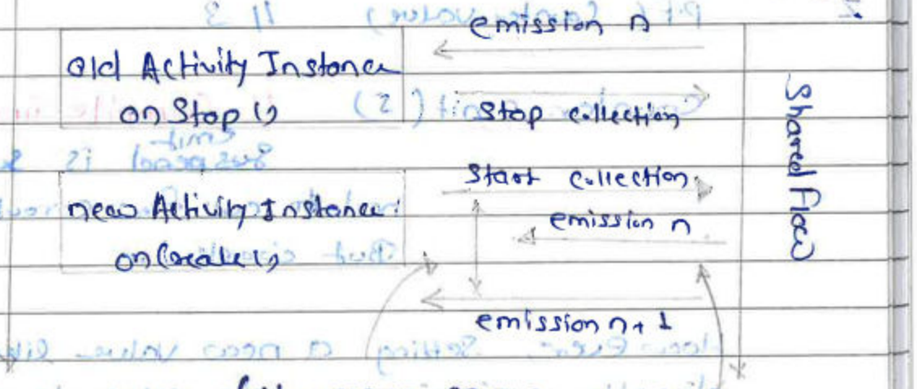
(3) SharingStarted.WhileSubscribed()

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

- * fun WhileSubscribed(stopTimeoutMillis: Long=0, replayExpirationMillis: Long=Long.Max.value) : SharingStarted

- StopTimeoutMilles parameter is specifies if collecter is inactive for very short time (i-e StopTimeoutMilles = 5000) then we don't need to cancel the shared flow and restart it.

(B) Replay :-



(No response span if replay (reply) is equal to zero or no value set explicitly.)

(But if reply >1 then will get more one (1,2,3..n) recent emission)

So reply = n then when starts the re observe flow / re collect flow then after configuration changes then that numbers most recent emissions we get.

State flow is a 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.

State flow always has an initial value to new subscribers.

No need of Coroutine for collection

```
val counter = MutableStateFlow(0)
p.f( counter.value)          // 0
counter.value = 3
p.f( counter.value)          // 3

counter.emit(5)              // Compile Time Error
```

emit is suspend function need to call from coroutine or other suspend fn. But overkill

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)          // 0
repeat
CoroutineScope {
    repeat (10,000) {
        launch {
            counter.value = counter.value + 1
        }
    }
}
```

```
p.f( counter.value)          // 7812
                             (expected 10000)
```

so counter.value is not thread safe.

Now,

```
val counter = MutableStateFlow(0)
p.f( counter.value)          // 0
CoroutineScope {
    repeat (10,000) {
        launch {
            counter.update { value ->
                value + 1
            }
        }
    }
}
p.f( counter.value)          // 10000
```

so update {} function on state flow made code thread safe.

Convert (flow/cold flow into State flow

```
val flow: Flow<Int> = flow {
    repeat (10) {
        emit(it)
        delay (100)
    }
}.stateIn(
    scope = viewmodelScope,
    initialValue = -1
    started = SharingStarted.WhileSubscribed(5000)
)
```

Difference Between Shared flow vs State flow

	Shared flow	State flow
Initial value	No	Yes
Replay cache	customizable	fixed size to 1
Emission of subsequent equal values	Yes	No

Distint until Change Behaviour

Third difference is the behaviour when two equal values are subsequently emitted to flow.

Shared flow : (No distint until change behaviour)

```
val SharedFlow = flow {
        emit (1)
        delay (10)
        emit (1)
        delay (10)
        emit (2)
}. SharedIn(
    Scope = viewmodelScope
    Started = SharingStarted.
    replay = 1
)

launch {
    SharedFlow. collect {
        P.P( it)
    }
}
```

O/P: 1
1
2

State flow (Distint until change behaviour)

```
val StateFlow = flow {
        emit (1)
        delay (10)
        emit (1)
        delay (10)
        emit (2)
}. StateIn(
    Scope = viewmodelScope
    initialValue = 1
)

launch {
    StateFlow . collect { P.P( it) }
}
```

O/P: 1
2

Here, initially value (1) is emitted three times (inital value on time 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 has Distint until change behaviour.

Thumb Rule for
Rule of Thumb of usage: StateFlow & SharedFlow

① Whenever you to want to use a hot flow, use a ~~sta~~ state flow by default.

② State flows are most efficient coken used for state.

③ State flow provides convenient option to read and write its value in a non-suspending fashion by synchronosly ~~acces~~ accessing the .value property.

④ ~~If~~ only if you have special requirements switch to State flow.
- Not have inital value
- Required ~~flow~~ customized Replay ~~beh~~ cache.
- Required emission of subsequent equal values

Note: In state flow if collector is slow then Potentially lose emissions.

Concurrent flows

Buffer

```
val flow = flow {
    repeat(5) {
        p.f("Emitter: Start Cooking Cake $it")
        delay(100)
        p.f("Emitter: Cake $it ready")
        emit(it)
    }
}
flow.collect {
    p.f("Collector: Start eating Cake $it")
    delay(300)
    p.f("Collector: Finished eating Cake $it")
}
```

In single coroutine

Direct function call

O/P:

```
Emitter: Start Cooking Cake 0
—11— Cake 0 ready
Collector: Start eating Cake 0
—11— Finished eating Cake 0
Emitter: Start Cooking Cake 1
—11— Cake 1 ready
Collector: Start eating Cake 1
—11— finished eating Cake 1
:
Emitter: Start Cooking Cake 4
Emitter: Cake 4 ready
Collector: Start eating Cake 4
Collector: finished eating Cake 4
```

Execution of flow is completely sequential

```
val flow = flow {
    repeat(5) {
        println("Emitter: Start cooking cake $it")
        delay(100)
        println("Emitter cake $it ready")
        emit(it)
    }
}.buffer()

flow.collector {
    println("Collector: Start eating cake $it")
    delay(300)
    println("Collector: finished eating cake $it")
}
```

Coroutine-1 → repeat(5) { ... }

channel

Coroutine-2 → collector { ... }

Now when whole flow is running in a single Coroutine, as call to emit is basically just a function call to the collect block with the emission as the input argument.

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

Buffer operator used channels under the hood so that coroutine can communicate with each other.

o/p

⇒

```
Emitter: Start cooking cake 0
 —11—  : Cake 0 ready
Collector: Start eating cake 0
Emitter: Start cooking cake 1
 —11—  : Cake 1 ready
 —11—  : Start cooking cake 2
 —11—  : Cake 2 ready
 —11—  : Start cooking Cake 3
Collector: finished eating cake 0
 —11—  : Start eating cake 1
Emitter : Cake 3 ready
 —11—  : Start cooking Cake 4
 —11—  : Cake 4 ready
Collector: finished eating cake 1
 —11—  : Start eating cake 2
 —11—  : finished eating cake 2
 —11—  : Start eating cake 3
 —11—  : finished eating cake 3
 —11—  : Start eating cake 4
 —11—  : finished eating cake 4
 —11—  : Start eating cake 5
 —11—  : finished eating cake 5
```

buffer

64

```
fun <T> Flow<T>.buffer(capacity: Int = BUFFERED,
    onBufferOverflow: BufferOverflow =
        BufferOverflow.SUSPEND):
    Flow<T>
```

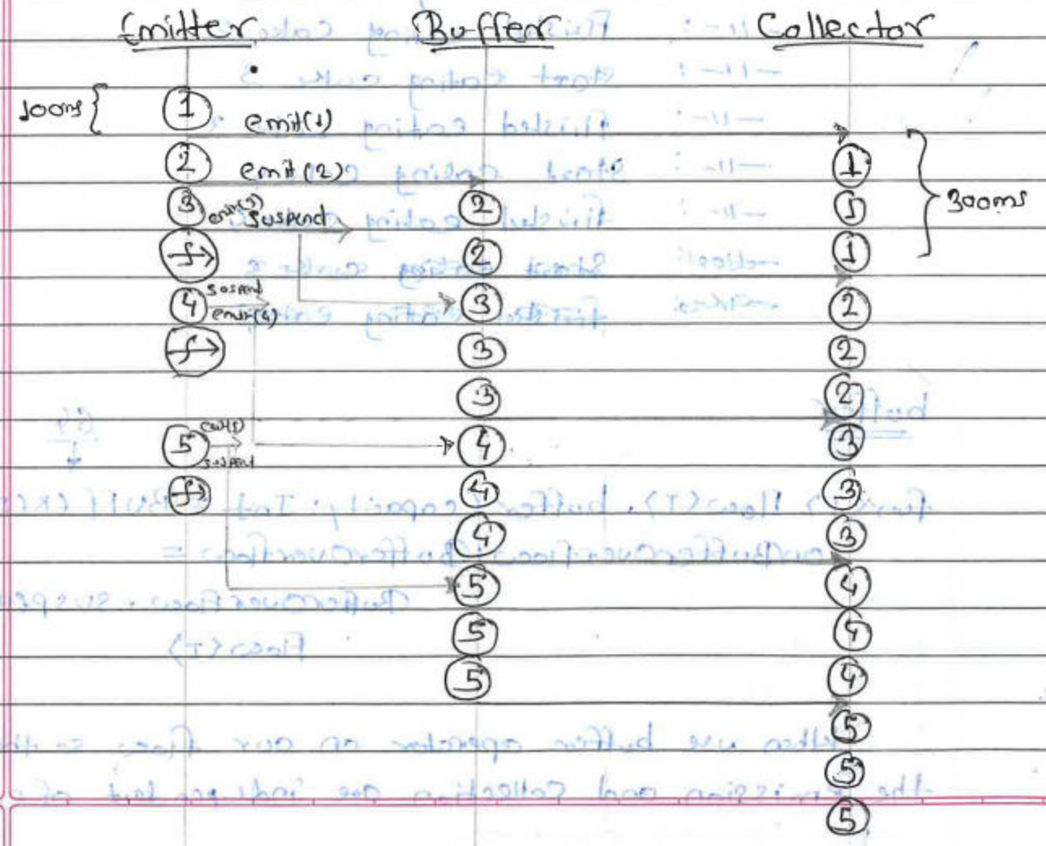
When use buffer operator on our flow so that the emission and collection are independent of eachother.

So this 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 Back pressure. However in kotlin this is called as Buffer overflow.

(1) onBufferOverflow = BufferOverflow.SUSPEND



If the buffer is full then emitter is not stop emiting instead of that emitter will emit but once it tries to emit it, they call to emit will suspend. Since the buffer is full.

```
Op: Start cooking cake 1

val flow = flow {
    repeat(5) {
        val cakeIndex = it + 1
        p.f("Emitter: Start cooking cake $
            cakeIndex");
        delay(100)
        p.f("Emitter: cake $ cakeIndex ready")
        emit(cakeIndex)
    }
}.buffer(capacity = 1, onBufferOverflow =
    BufferOverflow.SUSPEND)
```

The collector on other hand continues to process.

```
flow.collect {
    p.f("Collector: Start eating cake $it")
    delay(300)
    p.f("Collector: Finished eating cake $it")
}
```

O/P: Emitter: Start cooking cake 1
—11— : Cake 1 ready
Collector: Start eating cake 1
Emitter: Start cooking cake 2
—11— : Cake 2 ready
—11— : Start cooking cake 3
—11— : Cake 3 ready
Collector: Finished eating cake 1
—11— : Start eating cake 2
Emitter: Start cooking cake 4
—11— : Cake 4 ready
Collector: Finished eating cake 2
Emitter Start eating cake 5
—11— : Cake 5 ready

Collector: finished eating cake 3
— II — : Start eating cake 4
— II — : finished eating cake 4
— II — : Start eating cake 5
— II — : finished eating cake 5

So if the Buffer is full then emitter is no. Stop emiting 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.

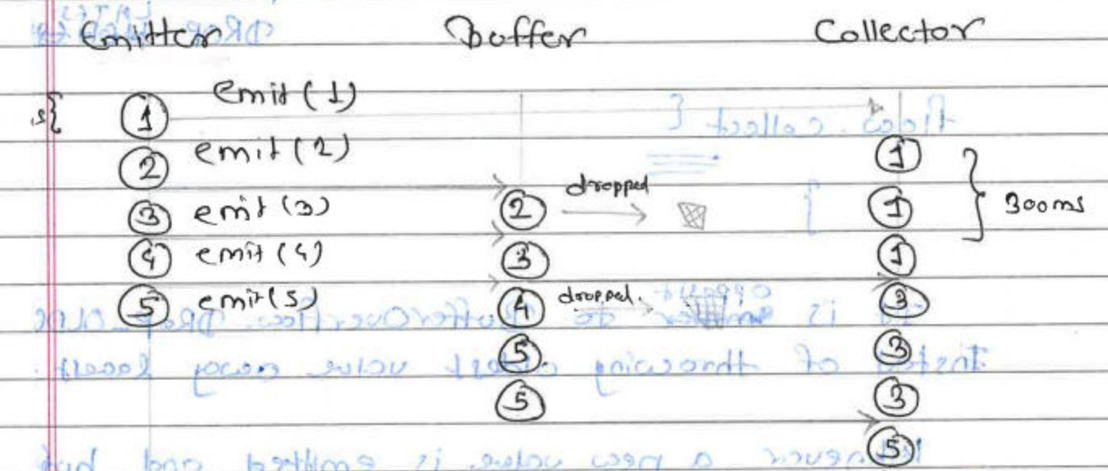
(2) onBufferOverflow = BufferOverflow.DROP_OLDEST

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) {
            val cake = it+1
            println("Emitter: Start cooking cake $it")
            delay(100)
            println("Emitter: cake $it ready")
            emit(cake)
        }
    }.buffer(capacity = 1, onBufferOverflow
        = BufferOverflow.
        DROP_OLDEST)
```

```
flow.collect {
    println("Collector: Start eating cake $it")
    delay(300)
    println("Collector: finished eating cake $it")
}
```



③ onBufferOverflow = BufferOverflow.DROP_~~OLDEST~~ LATEST

```
val flow = flow {
    ==========
}.buffer( capacity = 1,
          onBufferOverflow = BufferOverflow.
                             DROP_~~OLDEST~~ LATEST)

flow.collect {
    ==========
}
```

It is opposit ~~Emitter~~ to BufferOverflow.DROP_OLDEST. Instead of throwing oldest value away latest value

Whenever a new value is emitted and buffer is full, new value is thrown away which will not collect by collector.

Emitter		Buffer		Collector
Boom {	① emit(1)			① } Boom
	② emit(2)			②
	③ emit(3)	dropped	②	①
	④ emit(4)		③	④
	⑤ emit(5)	dropped	④	④
				④
				④

① Collect Latest

~~val flow = flow {~~
~~repeat (5) {~~
~~val cakeIndex = it + 1~~
~~println("Emitter: Start cooking cake $it")~~

```
val flow = flow {
    repeat(5) {
        val cakeIndex = it + 1
        println("Emitter: Start cooking cake $cak")
        delay(100)
        println("Emitter: cake $ cake ready")
        emit(cake)
    }
}

flow.collectLatest {
    println("Collector: Start eating cake $it")
    delay(300)
    println("Collector: finished eating cake $it")
}
```

Op: Emitter: Start cooking cake 1
—||— : Cake 1 ready
Collector: Start eating cake 1
~~cake~~ : ~~finished eating cake 1~~
Emitter: Start cooking cake 2
—||— : cake 2 ready
Collector: Start eating cake 2
Emitter: Start cooking cake 3
—||— : cake 3 ready
Collector: Start eating cake 3
Emitter: Start cooking cake 4
—||— cake 4 ready

Collector: Start eating cake 4
Emitter: Start cooking cake 5
—11— : cake 5 ready
Collector: Start eating cake 5
—11— : finished eating cake 5

Collector Latest :

Every time upstream emits a new item, the collectLatest block is immediately cancelled and restarted with the new item.

So collectLatest 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.

collectLatest runs concurrently instead of sequentially.

```
Public suspend fun <T> flow<T>.collectLatest(action:
                    suspend (value: T) → Unit) {
    mapLatest(action).buffer(capacity = 0).collect()
}
```

Buffer of capacity zero is inserted in flow. And that explains why our flow here runs concurrently.

② mapLatest :

```
val flow = flow {
    repeat(5) {
        val cake = it + 1
        println("Emitter: Start cooking cake $cake")
        delay(100)
        println("Emitter: Cake $cake ready")
        emit(cake)
    }
}.mapLatest {
    println("Adding icing on $it")
    delay(200)
    "it"
}

flow.collect {
    println("Collector: Start eating cake $it")
    delay(300)
    println("Collector: Finished eating cake $it")
}
```

O/P : Emitter: Start cooking cake 1
Emitter: Cake 1 ready
Adding icing on 1
Emitter: Start cooking cake 2
—11— : Cake 2 ready
Adding icing on 2
Emitter: Start cooking cake 3
—11— : Cake 3 ready
Adding icing on 3
Emitter: Start cooking cake 4
—11— : cake 4 ready)
Adding icing on 4

Collector: Start eating cake 4
Emitter: Start cooking cake 5
—||— : cake 5 ready
Collector: Start eating cake 5
finished eating cake 5

Collector Latest :

Every time upstream emits a new item, the collectLatest block is immediately cancelled and restarted with the new item.

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

collectLatest runs concurrently instead of sequentially.

Public suspend fun <T> Flow<T>.collectLatest(action: suspend (value: T) -> Unit) {

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

Buffer of capacity zero is inserted in flow. And that explains why our flow here runs concurrently.

② mapLatest : transformLatest

```
val flow = flow {
    repeat(5) {
        val cake = it + 1
        println("Emitter: Start cooking cake $cake")
        delay(100)
        println("Emitter: Cake $cake ready")
        emit(cake)
    }
}.mapLatest {
    println("Adding icing on $it")
    delay(200)
    "$it"
}

flow.collect {
    println("Collector: Start eating cake $it")
    delay(300)
    println("Collector: Finished eating cake $it")
}
```

O/P : Emitter: Start cooking cake 1
Emitter: Cake 1 ready
Adding icing on 1
Emitter: Start cooking cake 2
—||— : Cake 2 ready
Adding icing on 2
Emitter: Start cooking cake 3
—||— : Cake 3 ready
Adding icing on 3
Emitter: Start cooking cake 4
—||— : cake 4 ready
Adding icing on 4

Emitter: Start cooking cake 5
— " — : Cake 5 ready
Adding icing on 5
Collector: Start eating cake 5
— " — finished eating cake 5

MapLatest & TransformLatest

Now mapLatest and transformLatest is useful in situations where we have a slow operation in its block and we don't care about outdated emissions.

In mapLatest and transformLatest every time there's a new emission in the upstream, its flow is cancelled and restarted with the new value.

③ Conflate operator

```
val flow = flow {
    ...
}.conflate()

flow.collect {
    ...
}
```

Conflate operator is shortcut of buffer operator with capacity of Channel.CONFLATED and onBufferOverflow = BufferOverflow.DROP_OLDEST

1 (value)

Buffer in shared flow

```
val flow = MutableSharedFlow<Int>()

// Collector 1
launch {
    flow.collect {
        println("Collector 1 Processes $it")
    }
}

// Collector 2
launch {
    flow.collect {
        println("Collector 2 Processes $it")
        delay(100)
    }
}

// Emitter
launch {
    val timeToEmit = measureTimeMillis {
        repeat(5) {
            flow.emit(5)
            delay(10)
        }
    }
    println("Time to emit all values: $timeToEmit ms")
}
```

Collector 1	Processes	0
"	2	" 0
"	1	" 1
"	2	" 1

Collector 1 Processes 2
-11- 2 -11- 2
-11- 1 -11- 3
-11- 2 -11- 3
-11- 1 -11- 4
-11- 2 -11- 4

Time to emit all values: 432ms

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

If the Shared flow has two different collectors and out of one is slower than 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.

MutableSharedFlow

```
fun <T> MutableStateFlow(
        replay: Int = 0,
        extraBufferCapacity: Int : 0
        onBufferOverflow: BufferOverflow.=
                BufferOverflow.SUSPEND
    ): MutableSharedFlow<T>
```

Buffer in Shared flow

Replay + Extra = Total
Cache Buffer Buffer.

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

so the o/p of previous code by adding extraBufferCapacity = 10 i.e.

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

O/P. Collector 2 Processes 1
-11- 1 -11- 1
-11- 2 -11- 2
-11- 3 -11- 3
-11- 4 -11- 4

Time to emit all values: 61ms
Collector 2 Processes 1
-11- 2 -11- 2
-11- 2 -11- 3
-11- 2 -11- 4

Buffer in StateFlow

```
val flow = MutableStateFlow(0)

// Collector 1
launch {
    flow.collect {
        println("Collector 1 Process $it")
    }
}

// Collector 2
launch {
    flow.collect {
        println("Collector 2: Process $it")
        delay(100)
    }
}

// Emitter
launch {
    val timeToEmit = measureTimeMillis {
        repeat(5) {
            flow.emit(it)
            delay(10
        }
    }
}
```

O/P : Collector 1 Process 0
2
1 1
1 2
1 3
4

Time to emit all values: 64ms
Collector 2 process 4

In StateFlow, the emitter is never suspended. our fast collector processes all items however slow collector collect to only process most recent items.

Mutable StateFlow

real :

// MutableStateFlow (initial value) is a shared flow with the following

```
val shared = MutableSharedFlow(
    replay = 1
    onBufferOverflow = BufferOverflow.DROP_OLDEST)
shared.tryEmit(initialValue) // emit initial value
val state = shared.distinctUntilChanged()
// get StateFlow like behavior
```

So In StateFlow if Collector is slower than emitter then potentially loses the emissions.

Flow Cancellation

```
val scope = CoroutineScope(EmptyCoroutineContext)

scope.launch {
    intFlow().
        onCompletion { throwable -> Throwable? ->
            if(throwable is CancellationException){
                P.f("Flow got cancelled")
            }
        }.collect {
            P.f("$it")
            if(it == 2) {
                cancel()          // scope.cancel()
            }
        }
}

Private fun intFlow() = flow {
                        emit(1)
                        emit(2)
                        emit(3)
                        emit(4)
                    }
```

O/P: 1/2
Flow got cancelled.

Above code is co-oprative regrading cancellation.

The flow builder is internally performing additional ensure active checks for cancellation on each emitted value.

```
val scope = CoroutineScope(EmptyCoroutineContext)
scope.launch {
    intFlow().
        onCompletion { throwable ->
            if(throwable is CancellationException){
                P.f("Flow got cancelled")
            }
        }.
        collect {
            P.f(it)
            if(it==2) {
                cancel()
            }
        }
}

Private fun intFlow() = flow {
                        emit(1)
                        emit(2)
                        currentCoroutineContext().ensureActive()
                        P.f("start calculation")
                        calculateFactorialOf(1_000)
                        P.f("calculation finish")
                        emit(3)
                        emit(4)
                    }

In above code.
Private fun calculateFactorialOf(n: Int): BigInteger{
    var factorial = BigInteger.ONE
    for(i in 1..n) {
        factorial = factorial.multiply(BigInteger.valueOf(i.toLong()))
    }
    return factorial
}
```

In above example if we comment the line currentCoroutineContext.ensureActive() then the code is not in cooperative regarding cancellation and after the cancel() call we get following o/p.

o/p: 1
2
Start calculation
Flow got cancelled.

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

Now if we remove the comment of currentCoroutine-Context().ensureActive() then the code is cooperative regarding cancellation and after cancel() call we get following o/p

o/p: 1
2
Flow got cancelled.

Another approach to make code cooperative regarding cancellation is check ensureActive in every before every iteration of for loop.

```
private fun inFlow() = flow{
                        emit(1)
                        emit(2)
                        println("Start calculation")
                        calculateFactorialOf(1-00)
                        println("calculation finish")
                        emit(3)
}
```

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

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

o/p: 1
2
Start calculation
Flow got cancelled.

```
val scope = CoroutineScope(EmptyCoroutineContext)

scope.launch {
    flowOf(1,2,3,4)
        .onCompletion { throwable ->
            if(throwable is CancellationException){
                println("Flow got cancelled")
            }
        }.onEach { collect {
            println(it)
            if(it==2){
                cancel()
            }
        }
}
```

O/P: 1
2
3
4

Now here an interesting thing occurs happens that flow is never cancelled and all of the values collected.

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

So we have to manually / explicitly checks by ourself.

```
val scope = CoroutineScope(EmptyCoroutineContext)

scope.launch {
    flowOf(1,2,3,4)
        .onCompletion { throwable ->
            if(throwable is CancellationException){
                P.f("flow got cancelled")
            }
        }.onEach {
            P.f("Receive $it in onEach")
            if(!currentCoroutineContext().job.isActive){
                throw CancellationException()
            }
        }.collect {
            P.f(" collect $it")
            if(it==2)
                cancel()
        }
}
```


O/P: 1/2 Receive 1 in onEach
1
Receive 2 in onEach
2
Receive 3 in onEach
flow got cancelled.

Also .cancellable() can be use as follows.

```
val scope = CoroutineScope(EmptyCoroutineContext)

scope.launch{
    flowOf(1,2,3,4).
        onCompletion { throwable ->
            if(throwable is CancellationException){
                P.f("flow got cancelled")
            }
        }
        .cancellable()
        .collect {
            P.f(" it)
            if(it==2){
                cancel()
            }
        }
}
```

O/P: 1/2
flow got cancelled.