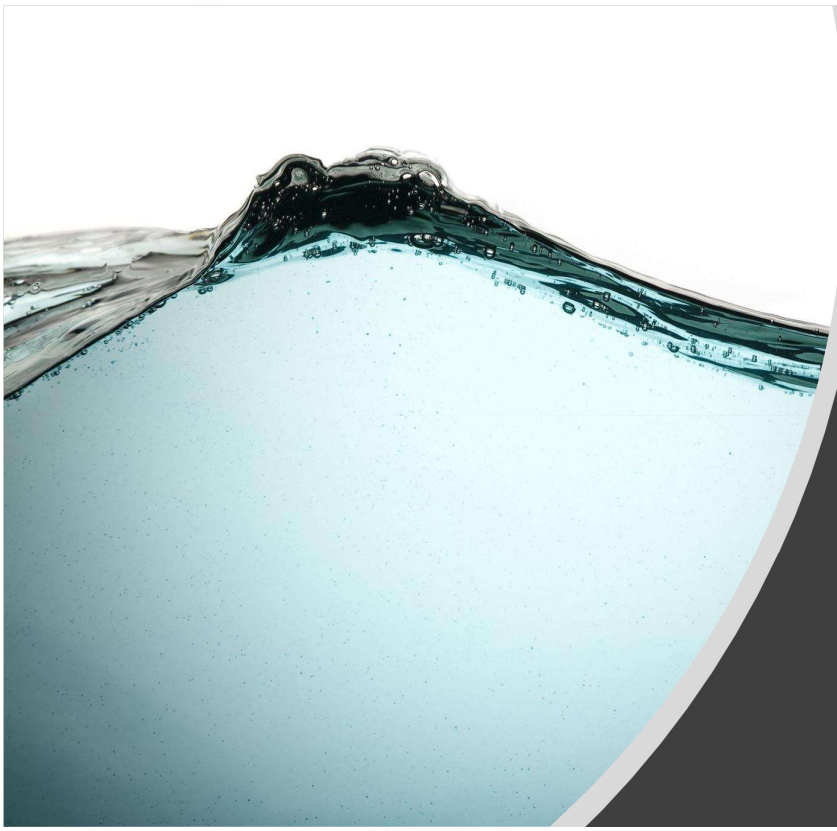


Flows

Reactive Streams

Agenda

- Coroutines Review
- Channels
- Kotlin Flows and Reactive Streams
- Flows Unit Tests
- Simplifying Callback-based APIs with Coroutines and callbackFlow
- Cold Flows and Hot Flows
- Flows in Android

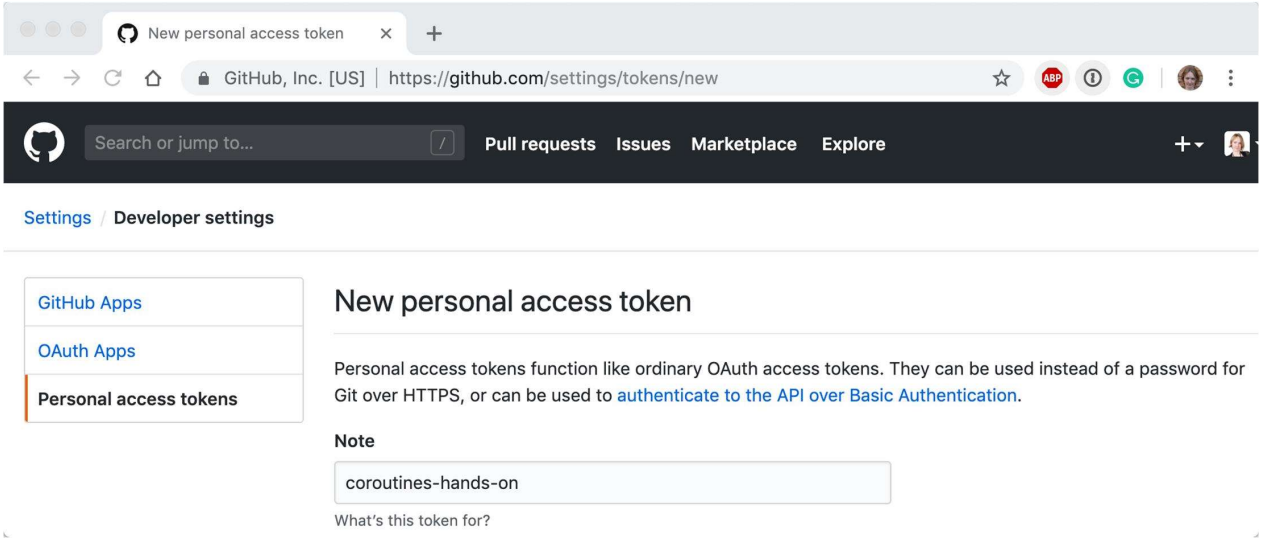
Warming Up Labs for Coroutines Review

We'll learn:

1. Why and how to use suspend functions to perform network requests.
2. How to send requests concurrently using coroutines.
3. How to share information between different coroutines using channels (and flows).

Generating GitHub developer token

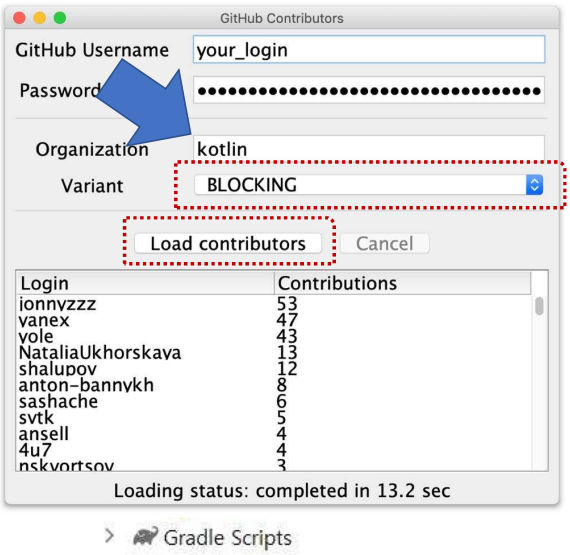
- We'll be using GitHub API.
- You need to specify your Github account name and a token.
  - If you have two-factor authentication enabled on GitHub, then only a token will work.
- You can generate a new GitHub token to use the GitHub API from your account here: https://github.com/settings/tokens/new.
- Specify the name of your token, for example, coroutines-hands-on.



- There is no need to select any scopes, click on "Generated token" at the bottom of the screen.

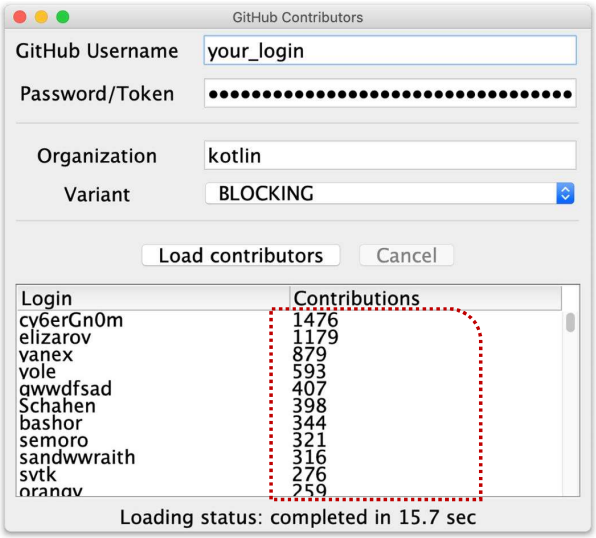
Running the code

- Open the **src/contributors/main.kt** file in **GitHub** module and run the main function.
- Make sure that in the variant dropdown menu the 'BLOCKING' option is chosen, then click on "Load contributors".
- Our program loads the contributors for all the repositories under the given organization. By default, the organization is "**kotlin**" but it could be any other one. Later we'll add logic to sort the users by the number of their contributions.



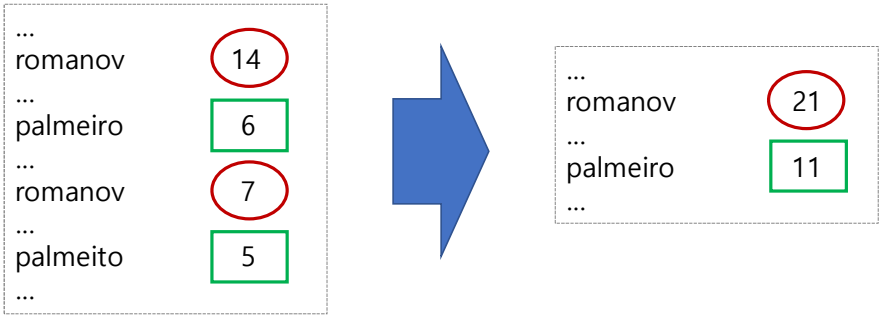
Warming Up Exercise:

- Display users sorted by the total number of their contributions.
- Open `src/tasks/Aggregation.kt` and implement `List<User>.aggregate()` function.
- The corresponding test file `test/tasks/AggregationKtTest.kt` shows an example of the expected result.



Tips

- Use either `groupBy()` or `groupingBy()`.
- Use `sortedByDescending()`.



groupBy

- Extension functions for grouping collection elements.
- `groupBy()` takes a λ function (*keySelector*) and returns a `Map`.

```
val numbers = listOf("one", "two", "three", "four", "five")

numbers.groupBy { it.first() }                  // {o=[one], t=[two, three], f=[four, five]}
numbers.groupBy { it.first().uppercase() }      // {O=[one], T=[two, three], F=[four, five]}
numbers.groupBy(                                // {o=[ONE], t=[TWO, THREE], f=[FOUR, FIVE]}
    keySelector = { it.first() },
    valueTransform = { it.uppercase() })
)
```


groupingBy

- Group elements and then apply an operation to all groups at one time.
  - `eachCount()`, `fold()`, `reduce()`, `aggregate()` etc.

```
val nums = listOf("one", "two", "three", "four", "five")
nums.groupingBy { it.first() }.eachCount()) // {o=1, t=2, f=2}
nums.groupingBy { it.first() }.reduce { key, acc, elem -> acc + elem })
// {o=one, t=twothree, f=fourfive}
nums.groupingBy { it.first() }.fold(emptyList<String>()) { acc, elem -> acc + elem })
// {o=[one], t=[two, three], f=[four, five]}
nums.groupingBy { it.first() }
    .aggregate { key, acc: StringBuilder?, elem, first ->
        if (first) StringBuilder().append(elem.uppercase()) else acc?.append(elem)
    }
// {o=ONE, t=TWOthree, f=FOURfive}
```

Step 1: Blocking Request

```
interface GitHubService {
    @GET("…")
    fun getOrgReposCall(
        @Path("org") org: String
    ): Call<List<Repo>>


  @GET("…")
    fun getRepoContributorsCall(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Call<List<User>>
}
```

```
fun loadContributorsBlocking(
    service: GitHubService, req: RequestData
): List<User> {
    val repos = service
        .getOrgReposCall(req.org)
        .execute() // Executes request and blocks the current thread
        .also { logRepos(req, it) }
        .body() ?: listOf()

    return repos.flatMap { repo →
        service
            .getRepoContributorsCall(req.org, repo.name)
            .execute() // Executes request and blocks the current thread
            .also { logUsers(repo, it) }
            .bodyList()
    }//.aggregate()
}
```

```
fun <T> Response<List<T>>.bodyList(): List<T> {
    return body() ?: listOf()
}
```


loadContributors() in UI

- This solution works, but blocks the thread and therefore freezes the UI.

```
fun loadContributors() {
    ...
    when (getSelectedVariant()) {
        BLOCKING -> { // Blocking UI thread
            val users = loadContributorsBlocking(service, req)
            updateResults(users, startTime)
        }
        ...
}
```

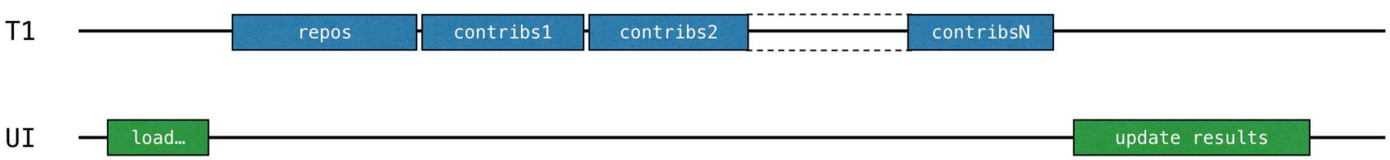
Step 2: Using Callbacks

- To make the UI responsive, we can either
  1) Move the whole computation to a *separate thread* or
  2) Switch to *async Retrofit API* and start using *callbacks* instead of blocking calls.

Calling loadContributors in the background thread

```
fun loadContributors() {
  ...
  when (getSelectedVariant()) {
    BACKGROUND -> { // Blocking a background thread
      loadContributorsBackground(service, req) { users ->
        SwingUtilities.invokeLater {
          updateResults(users, startTime)
        }
      }
    }
```

```
fun loadContributorsBackground(
  service: GitHubService, req: RequestData,
  updateResults: (List<User>) -> Unit)
{
    thread {
        val users = loadContributorsBlocking(service, req)
    }
}
```



Task to Do

- Fix the `loadContributorsBackground()` in `src/tasks/Request2Background.kt` so that the resulting list was shown in the UI.

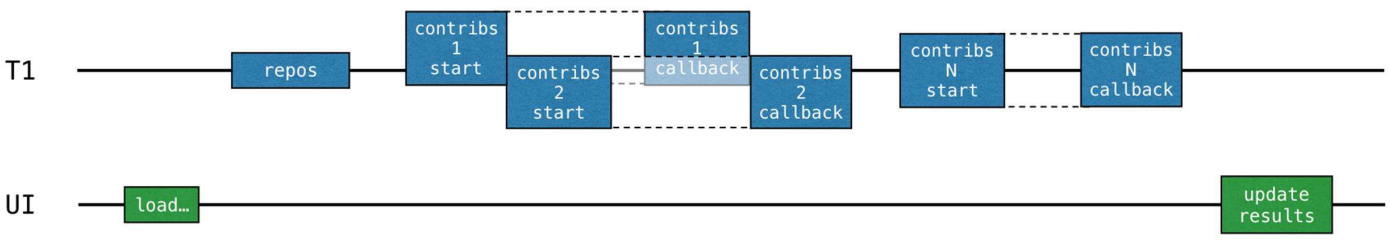
```
fun loadContributors() {
  ...
  when (getSelectedVariant()) {
    BACKGROUND -> { // Blocking a background thread
      loadContributorsBackground(service, req) { users ->
        SwingUtilities.invokeLater {
          updateResults(users, startTime)
        }
      }
    }
```

```
fun loadContributorsBackground(
  service: GitHubService, req: RequestData,
  updateResults: (List<User>) -> Unit)
{
    thread {
        val users = loadContributorsBlocking(service, req)
    }
}
```


Using Retrofit callback API

```
fun loadContributorsBlocking(
    service: GitHubService, req: RequestData
): List<User> {
    val repos = service.getOrgReposCall(req.org).execute().bodyList()
    return repos.flatMap { repo ->
        service
            .getRepoContributorsCall(req.org, repo.name)
            .execute() // Executes request and blocks the current thread
            .bodyList()
    }.aggregate()
}
```

Make this concurrent



onResponse() extension function

```
interface GitHubService {
    @GET("…") fun getOrgReposCall(@Path("org") org: String): Call<List<Repo>>


    @GET("…") fun getRepoContributorsCall(
        @Path("owner") owner: String, @Path("repo") repo: String): Call<List<User>>
}
```

```
inline fun <T> Call<T>.onResponse(crossinline callback: (Response<T>) -> Unit) {
    enqueue(object : Callback<T> {
        override fun onResponse(call: Call<T>, response: Response<T>) {
            callback(response)
        }
        override fun onFailure(call: Call<T>, t: Throwable) {
            log.error("Call failed", t)
        }
    })
}
```


Maybe Solution …

```
fun loadContributorsCallbacks(service: GitHubService, req: RequestData,
    updateResults: (List<User>) -> Unit) {
    service.getOrgReposCall(req.org).onResponse { responseRepos ->
        val repos = responseRepos.bodyList()

        val allUsers = mutableListOf<User>()
        for (repo in repos) {
            service.getRepoContributorsCall(req.org, repo.name).onResponse { responseUsers ->
                val users = responseUsers.bodyList()
                allUsers += users
            }
        }
        updateResults(allUsers.aggregate())
    }
}
```

Task to Do

- However, the provided solution doesn't work. If we run the program and load contributors choosing CALLBACKS option, we can see that nothing is shown.
- The tests that immediately return the result, however, pass. Why?

```
...
val allUsers = mutableListOf<User>()
for (repo in repos) {
    service.getRepoContributorsCall(req.org, repo.name).onResponse { responseUsers ->
        val users = responseUsers.bodyList()
        allUsers += users
    }
}
// TODO: Why this code doesn't work? How to fix that?
updateResults(allUsers.aggregate())
```


Task to Do (Cont'd)

- We're starting many requests concurrently which lets us decrease the total loading time.

  However, we don't wait for the loaded result.

  We call the `updateResults` callback right after we started all the loading requests, currently, the `allUsers` list is not yet filled with the data.

- Rewrite the code so that the loaded list of contributors was shown.

Solution (first attempt)

```
fun loadContributorsCallbacks(service: GitHubService, req: RequestData,
    updateResults: (List<User>) -> Unit) {
    service.getOrgReposCall(req.org).onResponse { responseRepos ->
        ...
        val allUsers = mutableListOf<User>()
        for ( (index, repo) in repos.withIndex() ) {  // #1
            service.getRepoContributorsCall(req.org, repo.name).onResponse { responseUsers ->
                logUsers(repo, responseUsers)
                val users = responseUsers.bodyList()
                allUsers += users
                if (index == repos.lastIndex) {       // #2
                    updateResults(allUsers.aggregate())
                }
            }
        }
    }
}
```

- In line #1 we iterate over the list of repos with an index. Then from each callback, we check whether we're on the last iteration (#2). And if that's the case, we update the result.
- However, this code is also incorrect. Why? What's the source of the problem? Spend some time trying to find an answer to this question.

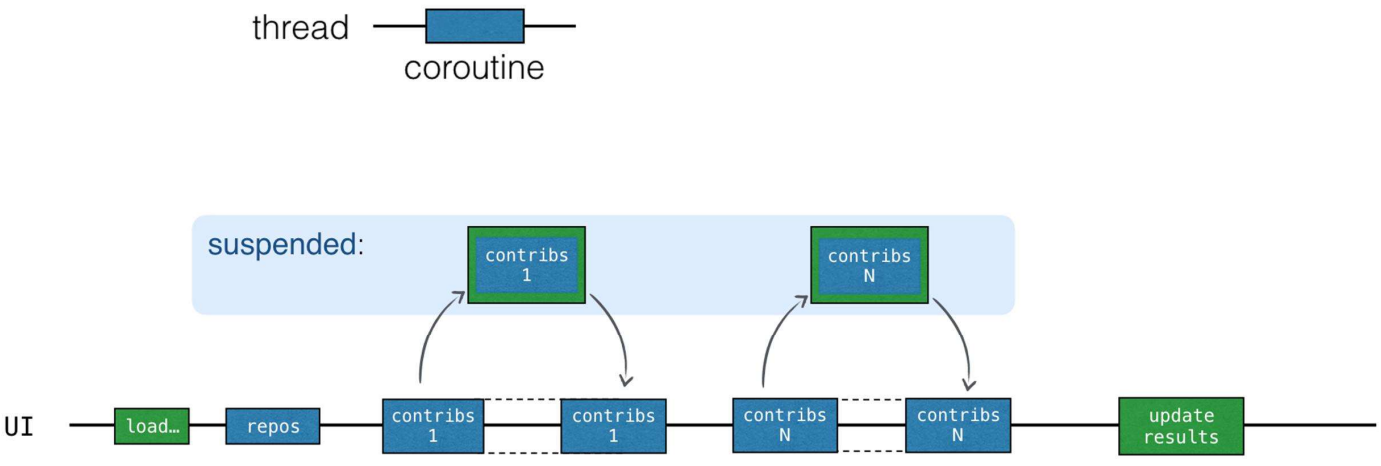
Tips

1. Use `Collections.synchronizedList(…)`.
2. Use either `AtomicInteger` or `CountDownLatch`.

Step 3: Using suspend functions

```
interface GitHubService {
    @GET("orgs/{org}/…")
    fun getOrgReposCall(
        @Path("org") org: String
    ): Call<List<Repo>>

    @GET("repos/{owner}/{repo}/…")
    fun getRepoContributorsCall(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Call<List<User>>
}
```

```
interface GitHubService {
    @GET("orgs/{org}/…")
    suspend fun getOrgRepos(
        @Path("org") org: String
    ): Response<List<Repo>>

    @GET("repos/{owner}/{repo}/…")
    suspend fun getRepoContributors(
        @Path("owner") owner: String,
        @Path("repo") repo: String
    ): Response<List<User>>
}
```

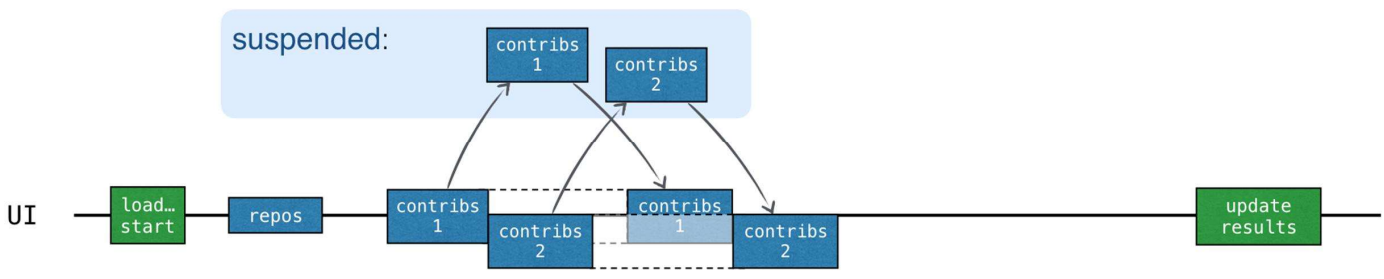



Task to Do

- Copy the implementation of `loadContributorsBlocking` (defined in `src/tasks/Request1Blocking.kt`) into `loadContributorsSuspend` (defined in `src/tasks/Request4Suspend.kt`).
- Then modify it in a way so that the new suspend functions are used instead of ones returning `Call`s. (Ignore the concurrent repositories processing for now.)
- Run the program choosing the `SUSPEND` option and make sure that the UI is still responsive while the GitHub requests are performed.
- The log can also show you what coroutine the corresponding code runs on. To enable it,
  - add the `-Dkotlinx.coroutines.debug` VM option.

Step 5: Concurrency

- Kotlin coroutines are extremely inexpensive in comparison to threads.
- Each time when we want to start a new computation asynchronously, we can create a new coroutine.



Coroutine Builders

- To start a new coroutine, use "coroutine builders":
  - `launch`: fire-and-forget
  - `async`: get async result
  - `runBlocking`*: a bridge between blocking and non-blocking worlds

```
fun main() = runBlocking {
    val deferred: Deferred<Int> = async {
        loadData()
    }
    println("waiting...")
    println(deferred.await())
}
```

```
fun main() = runBlocking {
    val deferreds: List<Deferred<Int>> = (1..3).map {
        async {
            delay(1000L * it)
            println("Loading $it")
            it
        }
    }
    val sum = deferreds.awaitAll().sum()
    println("$sum")
}
```

*It works as an adaptor for starting the top-level main coroutine and is intended primarily to be used in main functions and in tests.

Task to Do

- Implement a `loadContributorsConcurrent` function in the `Request5Concurrent.kt` file.
- Use the previous `loadContributorsSuspend` function.

Tips

- We can only start a new coroutine inside a coroutine scope. So, copy the content from `loadContributorsSuspend` to the `coroutineScope` call, so that we can call `async` functions there:

```
suspend fun loadContributorsConcurrent(
    service: GitHubService,
    req: RequestData): List<User> = coroutineScope {
    // ...
}
```

- Base the solution on the following scheme:

```
val deferreds = repos.map { repo ->
    async {
        // load contributors for each repo
    }
}
deferreds.awaitAll() // List<List<User>>
```


Step 6: Structured Concurrency

- Coroutine scope is responsible for parent-child structural relationships between different coroutines.
- Coroutine context stores additional technical information used to run a given coroutine (*name*, *job*, *dispatcher*, *exception handler*).
- **Benefits structured concurrency** has over global scopes:
  - The scope is generally responsible for child coroutines, and their lifetime is attached to the lifetime of the scope.
  - The scope can automatically cancel child coroutines.
  - The scope automatically waits for completion of all the child coroutines.
    - Therefore, if the scope corresponds to a coroutine, then the parent coroutine does not complete until all the coroutines launched in its scope are complete.

Task to Do

- Let's compare two versions of the `loadContributorsConcurrent` function: one using `coroutineScope` to start all the child coroutines and the other using `GlobalScope`.
- Add a 3-second delay to all the coroutines sending requests, so that we have enough time to cancel the loading before the requests are sent:

```
suspend fun loadContributorsConcurrent(service: GitHubService, req: RequestData):
List<User> = coroutineScope {
    // ...
    async {
        log("starting loading for ${repo.name}")
        delay(3000)
        // load repo contributors
    }
    // ...
    result
}
```

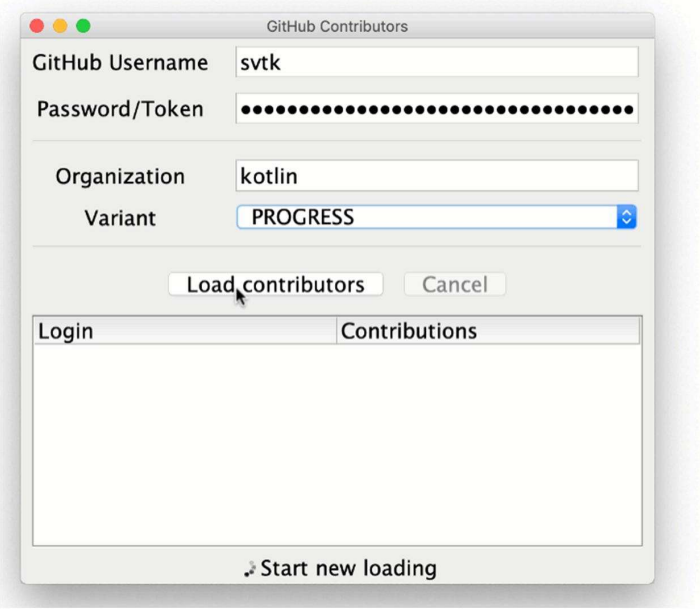

Task to Do (Cont'd)

- Copy the implementation of `loadContributorsConcurrent` to `loadContributorsNotCancellable` (in `Request5NotCancellable.kt`) and remove the creation of a new `coroutineScope`. We need to use `GlobalScope.async`:

```
suspend fun loadContributorsNotCancellable(service: GitHubService, req: RequestData):
List<User> {
    // ...
    GlobalScope.async {
        log("starting loading for ${repo.name}")
        delay(3000)
        // load repo contributors
    }
    // ...
    return result
}
```

- Run the program first with CONCURRENT and then with NOT_CANCELLABLE option.

Step 7: Showing Progress



- The user only sees the final resulting list once all the data is loaded.
- We could show the intermediate results earlier and display all the contributors after loading the data for each of the repositories.

Tips

- To implement this functionality, we'll need to pass logic updating the UI as a callback, so that it is called on each intermediate state:

```
suspend fun loadContributorsProgress(
    service: GitHubService,
    req: RequestData,
    updateResults: suspend (List<User>, completed: Boolean) -> Unit
) {
    // loading the data
    // calling `updateResults` on intermediate states
}
```

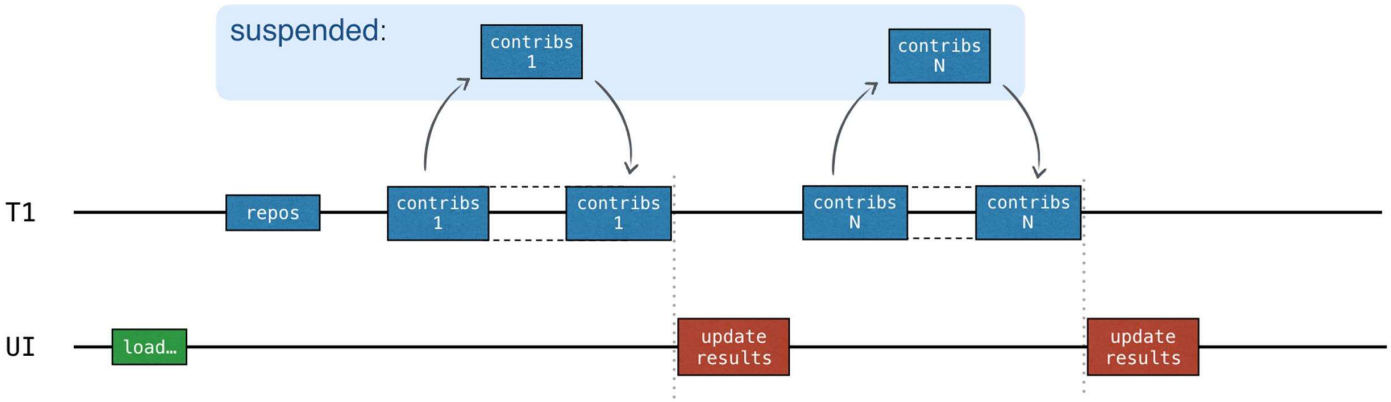
- On the call site, we pass the callback updating the results from the Main thread:

```
launch {
    loadContributorsProgress(service, req) { users, completed →
        withContext(Dispatchers.Main) {
            updateResults(users, startTime, completed)
        }
    }
}.setUpCancellation()
```

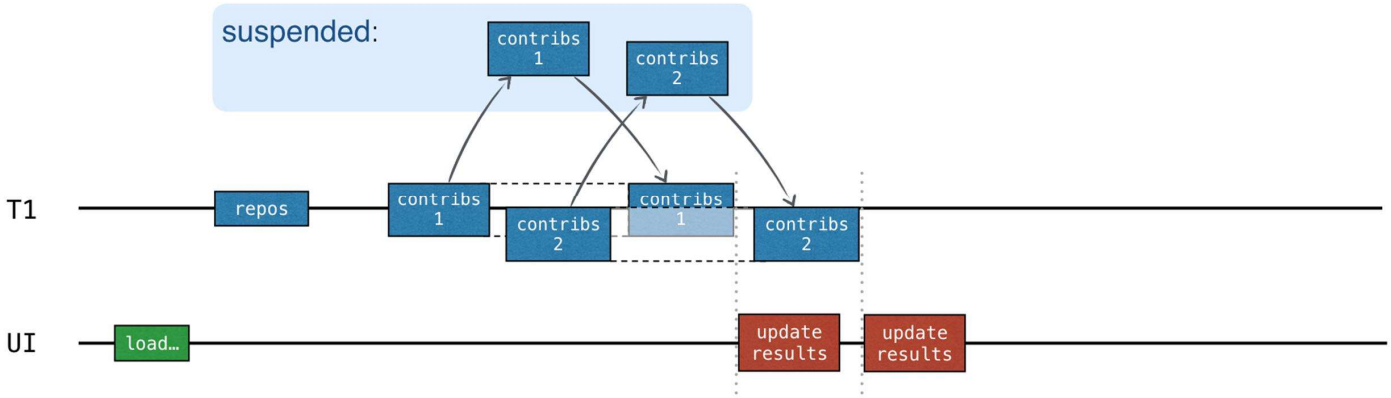
Task to Do

- Implement the function `loadContributorsProgress` that shows the intermediate progress (in the `Request6Progress.kt` file). Base it on the `loadContributorsSuspend` function (from `Request4Suspend.kt`).
- We'll use a simple version without concurrency for now.
- Note that the intermediate list of contributors should be shown in an "aggregated" state, not just the list of users loaded for each repository.
- The total numbers of contributions for each user should be increased when the data for each new repository is loaded.

An `updateResults` callback is called after each request is completed.
Since this code is sequential, we don't need synchronization, yet.



Step 8: How to add concurrency?



Task to Do for Channels

- Implement the function `loadContributorsChannels` that requests all the GitHub contributors concurrently, but shows intermediate progress at the same time.
- Use these two previous functions: `loadContributorsConcurrent` from `Request5Concurrent.kt` and `loadContributorsProgress` from `Request6Progress.kt`.

Tips

- Different coroutines that concurrently receive contributor lists for different repositories can send all the received results to the same channel:

```
val channel = Channel<List<User>>()
for (repo in repos) {
    launch {
        val users = ...
        // ...
        channel.send(users)
    }
}
```

- Then the elements from this channel can be received one by one and processed:

```
repeat(repos.size) {
    val users = channel.receive()
    ...
}
```

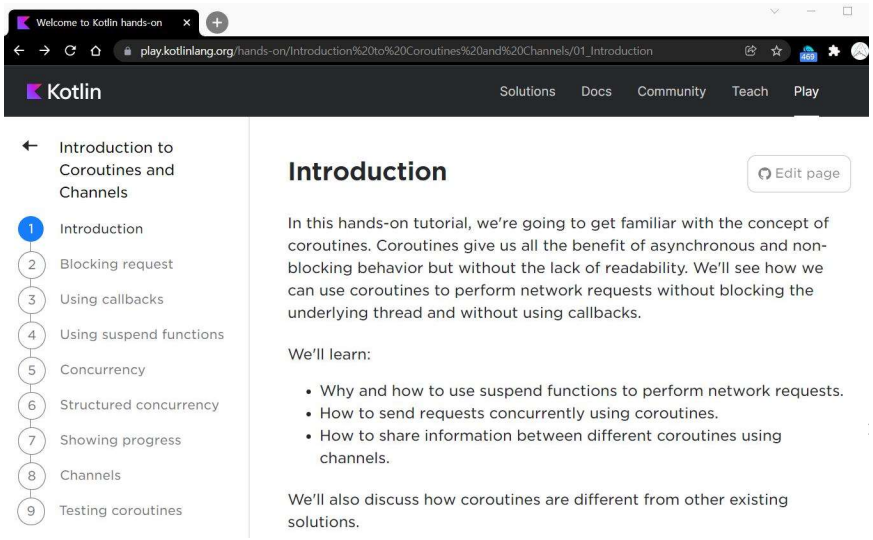
- Since we call the receive calls sequentially, no additional synchronization is needed.

Step 9: How to add concurrency? - Flows

- Do the same thing as in Step 8, except using appropriate flows instead of channels.

Original Source URL

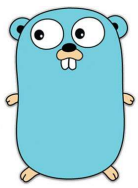
https://play.kotlinlang.org/hands-on/Introduction to Coroutines and Channels



Shared Mutable State

"A widely accepted method of communication is by inspection and updating of a common store … However, this can create severe problems in the construction of correct programs and it may lead to expense and … unreliability (e.g., glitches) …"

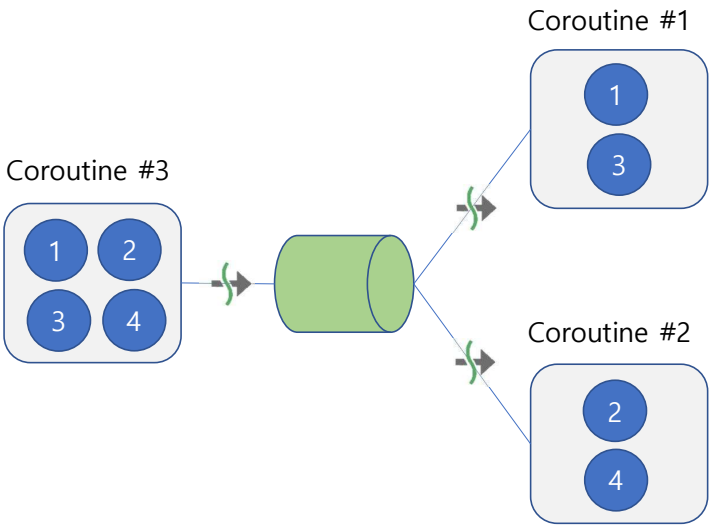
– Tony Hoare, CSP 1978



"Don't communicate by sharing memory;
instead,
share memory by communicating"

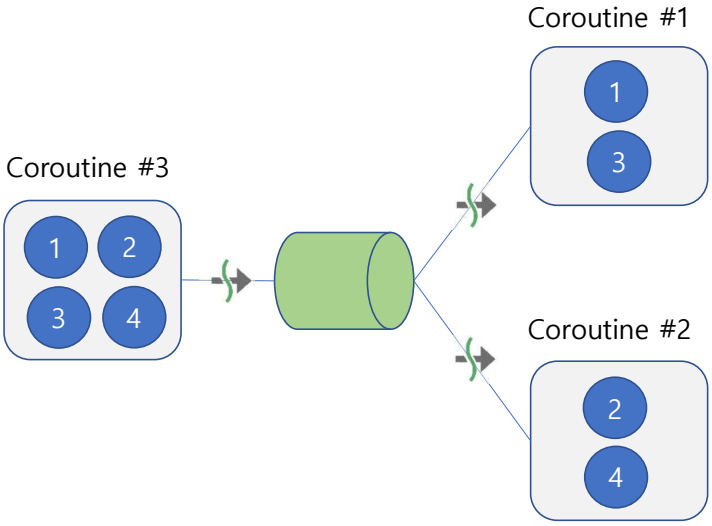
Simple Channel Demo

```
val channel = Channel<Int>()

launch { // Coroutine #1
    for (n in channel) { TODO() }
}

launch { // Coroutine #2
    for (n in channel) { TODO() }
}

launch { // Coroutine #3
    for (i in 1..4) {
        channel.send(i)
    }
    channel.close()
}
```



Simple Channel Demo

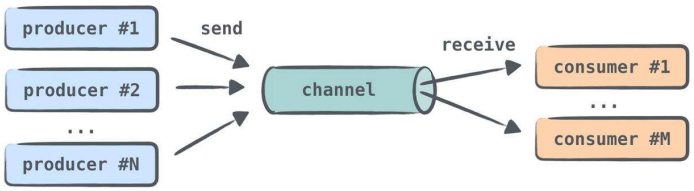
```
var channel: ReceiveChannel<Int>

launch { // Coroutine #1
    for (n in channel) { TODO() }
}

launch { // Coroutine #2
    for (n in channel) { TODO() }
}

channel = produce { // Coroutine #3
    for (i in 1..4) {
        channel.send(i)
    }
    // channel automatically closed
}
```



Channels

- Coroutines can communicate with each other via *channels*.

- Multiple producers and/or consumers can be associated with a channel.
  - Each element is handled (i.e., removed from the channel) only once by one of the consumers.



Channels

```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E> { … }
```

- Channels provide a pipel

```
public fun <E> Channel(
    capacity: Int = RENDEZVOUS,
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND,
    …
): Channel<E>
```

- A channel can be created

```
val rendezvousChannel = Channel<String>()
val conflatedChannel  = Channel<String>(CONFLATED)
val bufferedChannel   = Channel<String>(10)
val unlimitedChannel  = Channel<String>(UNLIMITED)
```

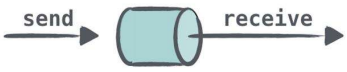
- The (optional) buffer size parameter determines the number of items that can be sent before the channel suspends (*default = 0*).

```
Channel.RENDEZVOUS = 0              // RendezvousChannel
Channel.CONFLATED = -1              // ConflatedChannel
Channel.BUFFERED = -2 (or explicit) // ArrayChannel
Channel.UNLIMITED = Int.MAX_VALUE   // LinkedListChannel
```

Channel Types

```
Channel.RENDEZVOUS = 0        // RendezvousChannel
```

No buffer and transfer occurs only when both receiver and sender met.




```
Channel.CONFLATED = -1        // ConflatedChannel
```

Always buffers the most recent item.
The receiver always get the most recently sent item.


```
Channel.BUFFERED = -2 (or explicit)  // ArrayChannel
```


```
Channel.UNLIMITED = Int.MAX_VALUE    // LinkedListChannel
```

Operations

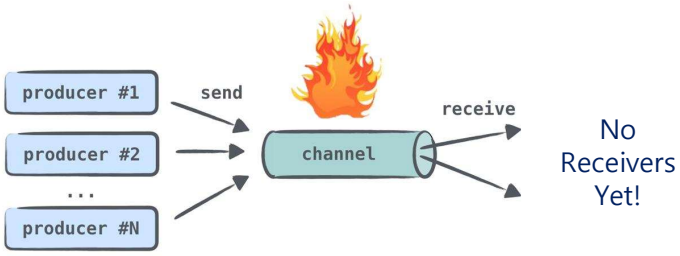
```
interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

```
interface SendChannel<in E> {
    suspend fun send(element: E)
    fun trySend(element: E): ChannelResult<Unit>
    fun close(cause: Throwable? = null): Boolean
}

interface ReceiveChannel<out E> {
    suspend fun receive(): E
    fun tryReceive(): ChannelResult<E>
    fun cancel(cause: CancellationException? = null)
}
```

- By default, the `send` suspends if there is no receiver. The `receive` also suspends if no item has been sent through the `Channel`.
- Calling `close` closes the channel and terminates the stream.
  - All items sent to the channel before calling `close` are guaranteed to be sent to the receiver.

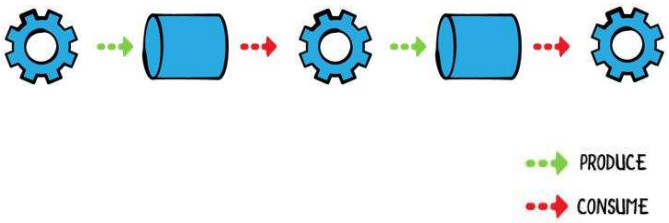
Channel is Hot

- A channel represents a *hot stream* of data that emits items without the presence or subscription from a receiver.
- Once an item is consumed from a channel, the same item cannot be re-consumed by the same or another receiver.

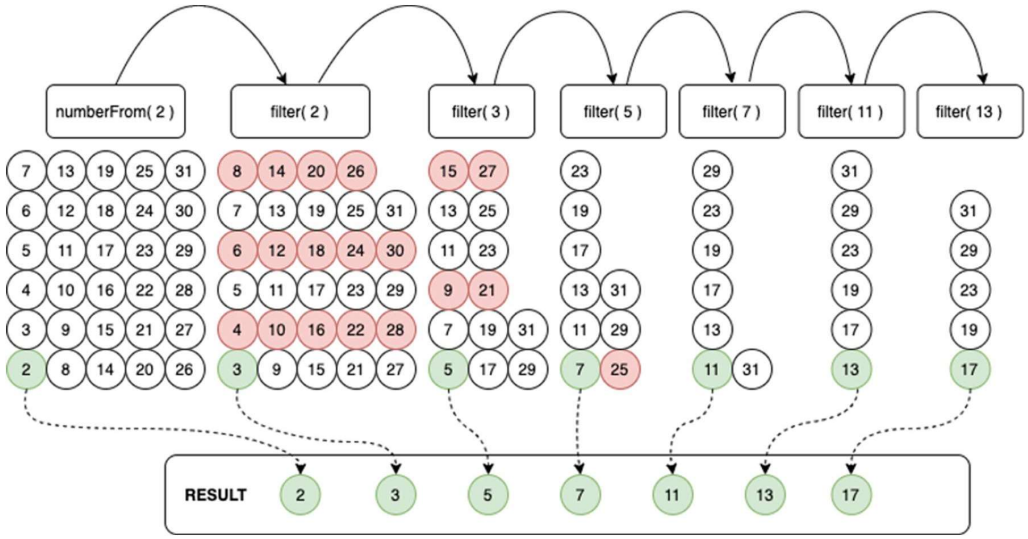


Channels Pipeline

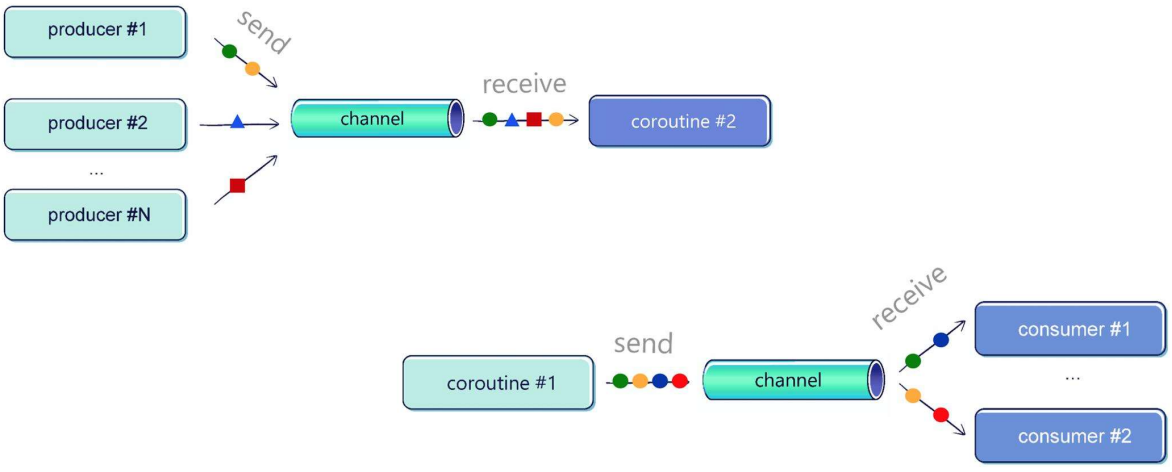
```
val channel1 = Channel<Int>()
val channel2 = Channel<Double>()

suspend fun produceNumbers() {
    var x = 1
    while (true) channel1.send(x++)
}

suspend fun square() {
    for (x in channel1) channel2.send((x * x).toDouble())
}

launch { produceNumbers() }
launch { square() }

repeat(5) {
    log(channel2.receive())
}
```

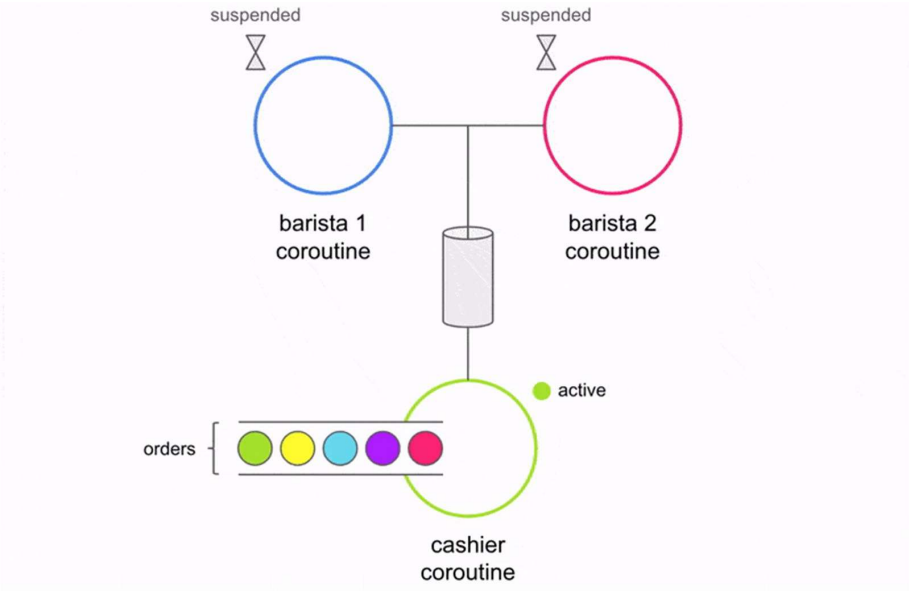


Sieve of Eratosthenes



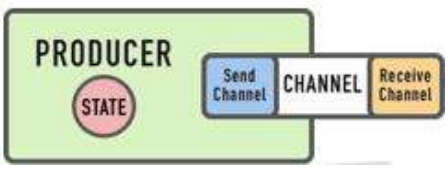
Channel Fan-In/Fan-Out





produce *vs.* actor

- A **producer** coroutine has a dedicated send channel.
- Only `ReceiveChannel` is exposed.



- An **actor** coroutine has a dedicated receive channel.
- Only `SendChannel` is exposed.

BroadcastChannel/ConflatedBroadcastChannel

- Unlike an ordinary channel, the `send` operation on a `BroadcastChannel` does not suspend (i.e., non-blocking) if there are no receivers.

```
val channel = BroadcastChannel<Int>(1)

repeat(10) {   // Runs and ends successfully without suspending
    channel.send(it)
}
```

- Consumers need to subscribe to channel to receive data.
  - Anything sent before obtaining the subscription is not received by the subscribed consumers.
- The `send` on `ConflatedBroadcastChannel` never suspends at all.
- `BroadcastChannel` → `SharedFlowConflated`
- `BroadcastChannel` → `StateFlow`

BroadcastChannel (Cont'd)

- Whereas `Channel` implements `SendChannel` and `ReceiveChannel`, `BroadcastChannel` only implements `SendChannel`.
- Multiple receivers can subscribe for the elements using `openSubscription()` function and unsubscribe using `ReceiveChannel.cancel()` function.

TL;DR

- Conceptually, channels behave like a Blocking Queue.
- Channels are synchronization primitives.

Github Issues #254

- Channels are hot!
  - They consume resources before any consumer receives the first value from the channel: somewhat inefficient, not easy to use

```
produce<Int> {
    while (true) {
        val x = computeNextValue()
        send(x)
    }
}
```

Need an abstraction for cold streams that is lazy and computes data in a "push" mode.

And finally the Flows

Kotlin Flow

https://www.youtube.com/watch?v=tYcqn48SMT8&t=73s

- Kotlin Flow is a *declarative* mechanism for working with *sequential asynchronous data streams*.
- It builds on top of Kotlin *coroutines* and *structured concurrency*.

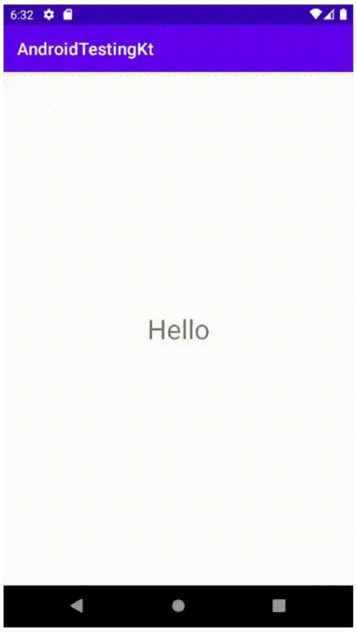
Types of Async Requests

- **One-shot request** returning a *single* value
  - Suspending functions

```
suspend fun loadData(): Data
```

```
uiScope.launch {
    val data = loadData()
    updateUI(data)
}
```


Types of Async Requests



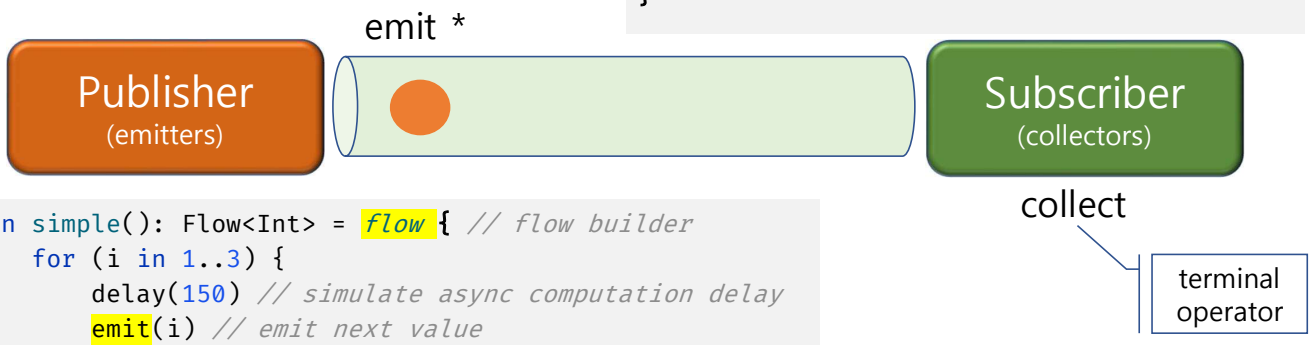
- **Stream request** returning an *asynchronous stream* of values.
  - Each value is computed *asynchronously* and delivered *in sequence*.
  - This is where Kotlin **Flow**s come in.

```
fun dataStream(): Flow<String>
```

```
uiScope.launch {
    dataStream().collect {
        updateUI (it)
    }
}
```

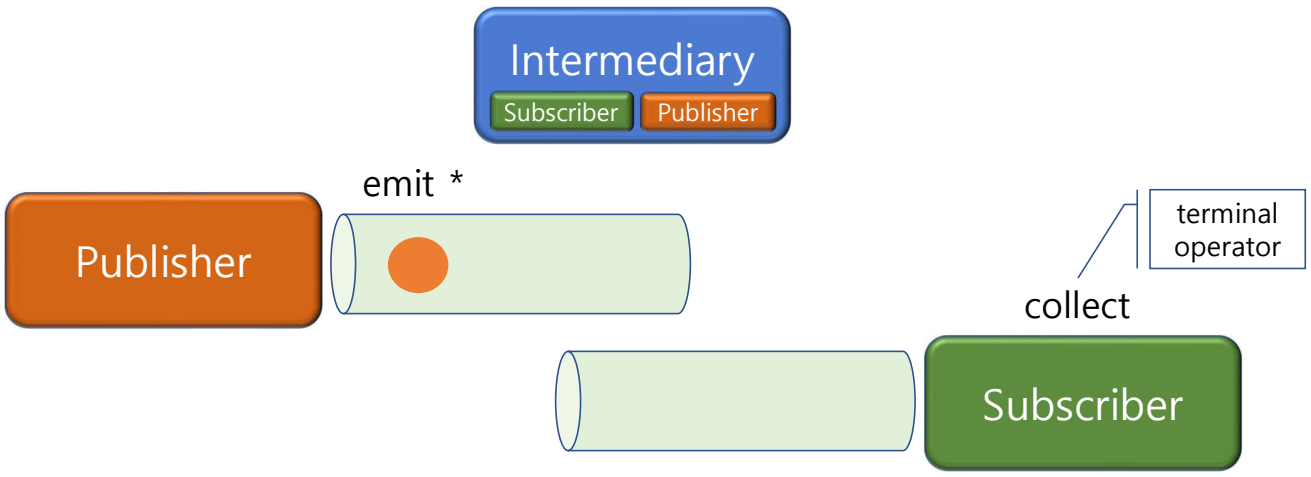
Components of Flows

```
fun main() = runBlocking {
    simple().collect { // collect the flow
        value -> println(value)
    }
}
```

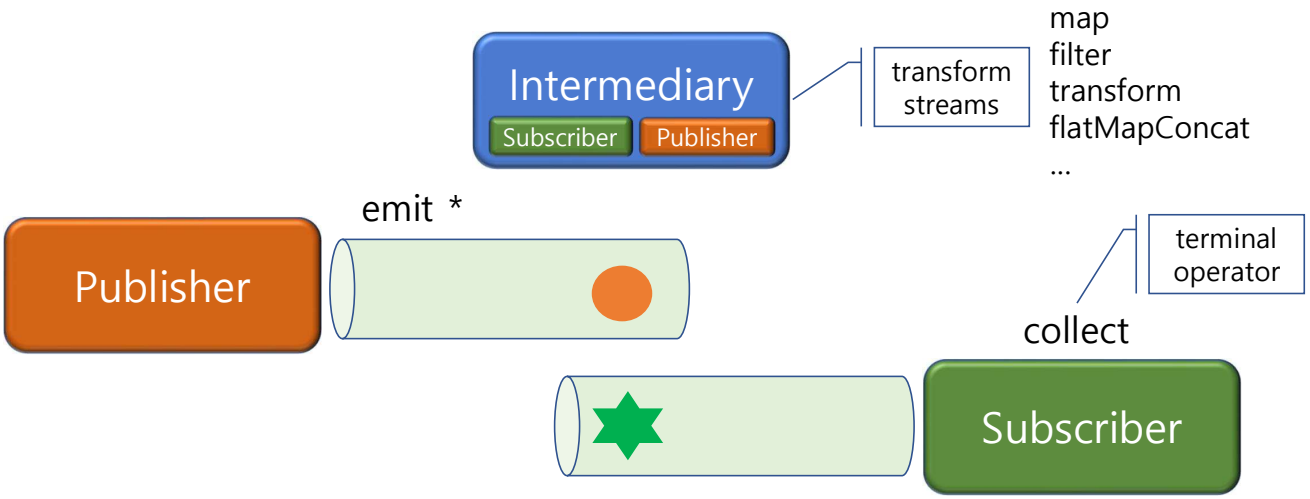



```
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(150) // simulate async computation delay
        emit(i) // emit next value
    }
}
```


Components of Flows



Components of Flows



Flows

`Flow<T>` represents the stream of asynchronously computed values.

- Values are *emitted* from the flow using `emit` function.

suspending function

```
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(150) // simulate async computation delay
        emit(i) // emit next value
    }
}
```

- Values are *collected* from the flow using `collect` function

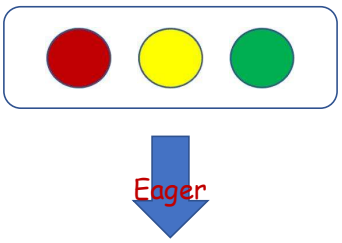
suspending function

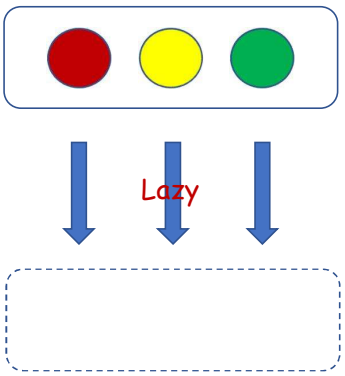
```
fun main() = runBlocking {
    // Collect the flow
    simple().collect { value -> println(value) }
}
```

suspending lambda

Returning multiple values as a whole *vs.* as a stream

- As a whole (i.e. entire collection)
  - List, Set, Map, …
- As a Stream (i.e., one by one)
  - Sequence, Flow





Collection Builders as of Kotlin 1.3

- buildList
- buildSet
- buildMap
- buildString

With builder, a whole list can be constructed in *non-blocking* way, if needed.

But, **lists** are still *eager*, while **sequences** are *lazy**.

*The next element is always calculated on-demand, when it is needed.

```
fun foo(): List<Result<String>> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

Build in blocking way

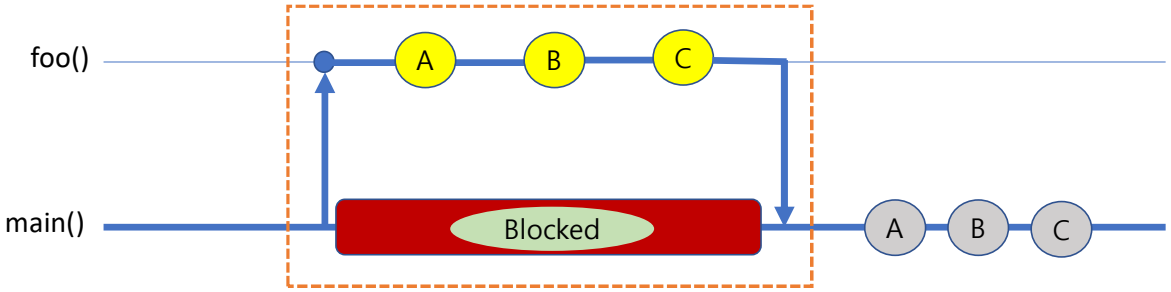
```
suspend fun foo(): List<Result<String>> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

Build in non-blocking way

Lists

```
fun foo(): List<Result<String>> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

```
fun compute(i: String): Result<String> {
    sleep(100)
    return Result.success(i)
}
```




```
fun main() {
    val list = foo()
    list.forEach { println(it) }
}
```

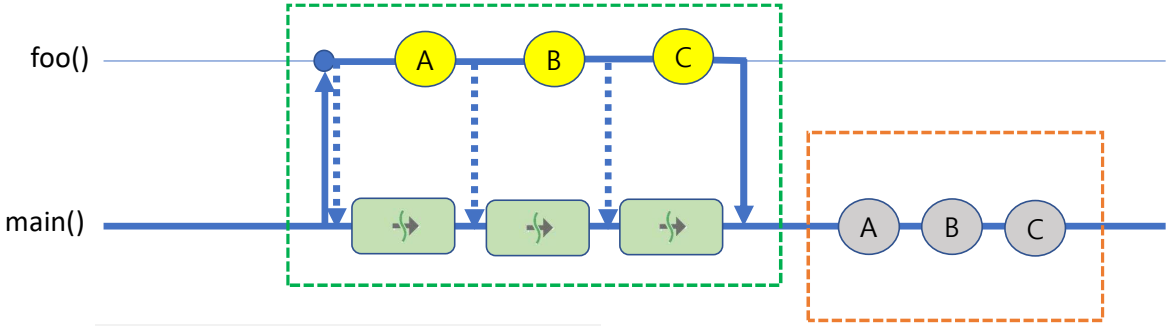
List construction is sync and blocking

The whole list must be ready before its use

Lists

```
suspend fun foo(): List<Result<String>> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```

```
suspend fun compute(i: String): Result<String> {
    delay(100)
    return Result.success(i)
}
```




```
fun main() = runBlocking{
    val list = foo()
    list.forEach { println(it) }
}
```

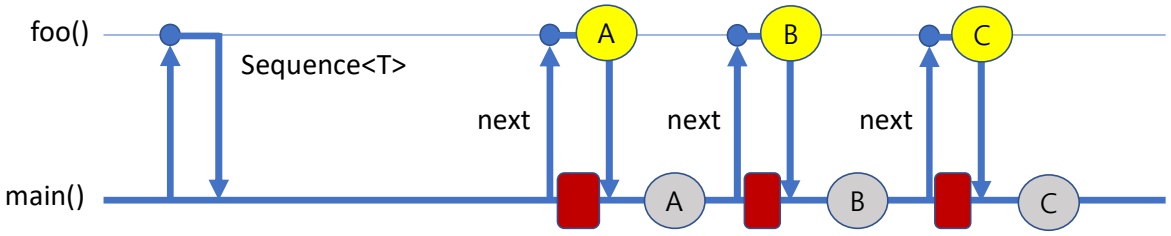
List construction is sync, but non-blocking

The whole list *still* must be ready before its use because list is eager in nature

Sequences

```
fun foo(): Sequence<Result<String>> = sequence {
    yield(compute("A"))
    yield(compute("B"))
    yield(compute("C"))
}
```

```
fun compute(i: String): Result<String> {
    Thread.sleep(100)
    return Result.success(i)
}
```




```
fun main() = runBlocking{
  val seq = foo()
  seq.forEach { println(it) }
}
```

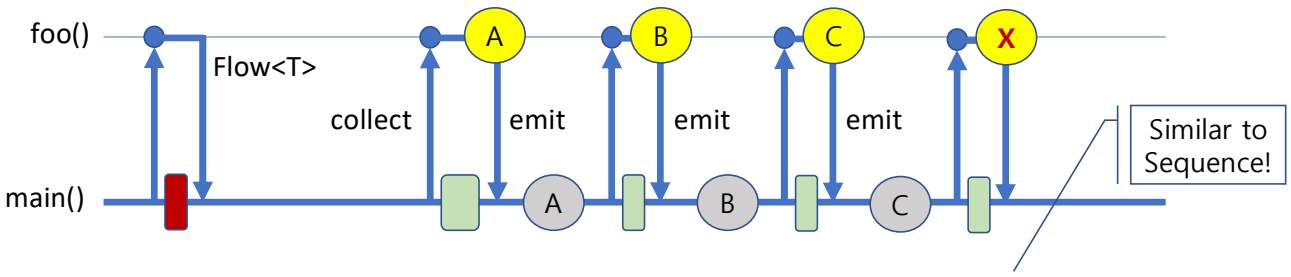
Sequence is async, but element generation is sync and blocking

Do not need to wait until the whole sequence is ready because sequence is lazy in nature

Flow

```
fun foo(): Flow<Result<String>> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```

```
suspend fun compute(i: String): Result<String> {
    delay(100)
    return Result.success(i)
}
```




```
fun main() = runBlocking{
    val flow = foo()
    flow.collect { println(it) }
}
```

Flow is async and element generation is push-based async and non-blocking

Do not need to wait until the whole flow is ready because flow is lazy in nature like sequences

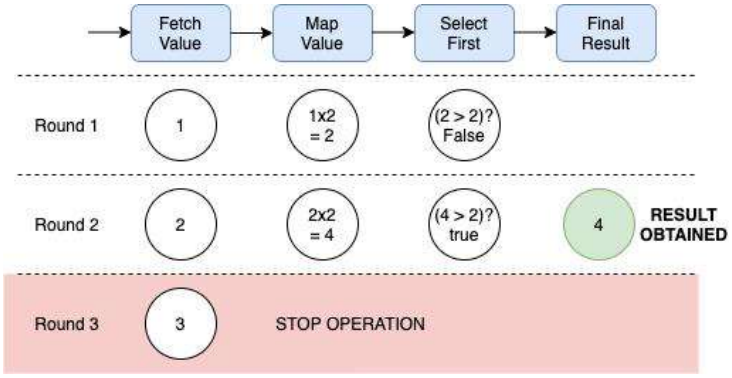
Both Sequences and Flows are sequential, but lazy

```
runBlocking {
    (1..3).asSequence()
        .map { println("sequence mapping $it"); it * 2 }
        .first { it > 2 }
        .let { println("sequence $it") }

    (1..3).asFlow()
        .map { println("flow mapping $it"); it * 2 }
        .first { it > 2 }
        .let { println("flow result $it") }
}
```

```
sequence mapping 1
sequence mapping 2
sequence 4
flow mapping 1
flow mapping 2
flow result 4
```

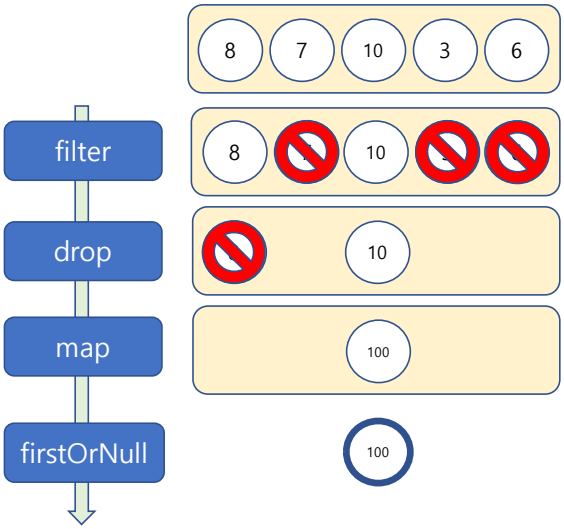

Both Sequences and Flows are sequential, but lazy

- While **List** is *eager*, both **Sequences** and **Flows** are *lazy*.
  - Sequences and Flows process each item and terminate the process as soon as it get the result.
- **Flow** can be considered as *Sequence in steroids*.



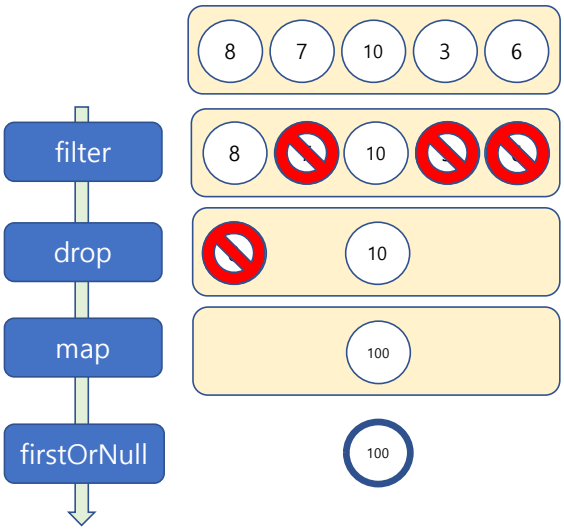
Find the square of the second even number which is greater than 7.

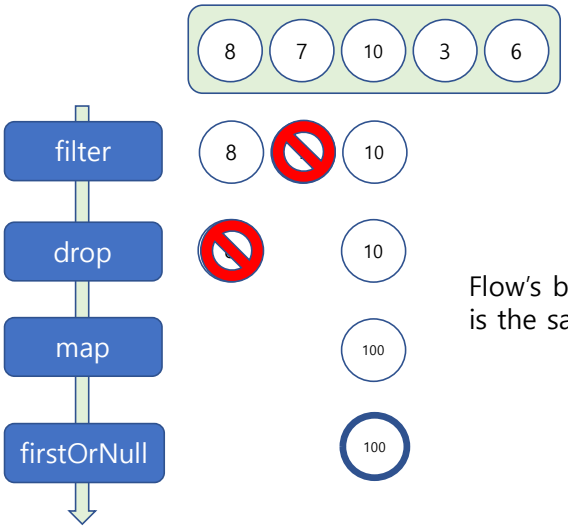
```
listOf(8, 7, 10, 3, 6)
    .filter { it > 7 && it % 2 == 0 }
    .drop(1)
    .map { it * 2 }
    .firstOrNull()
```



Find the square of the second even number which is greater than 7.

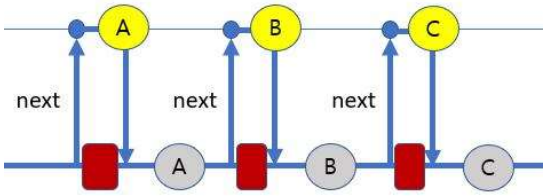
```
sequenceOf(8, 7, 10, 3, 6)
    .filter { it > 7 && it % 2 == 0 }
    .drop(1)
    .map { it * 2 }
    .firstOrNull()
```





Flow's behavior is the same

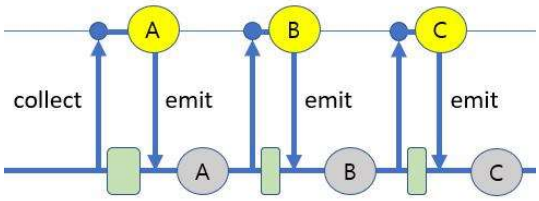
Sequence is blocking
(while each element is generated)




```
@Test
fun `sequence is blocking`() = runBlocking {
    fun simple() = sequence {
        (1..3).forEach { Thread.sleep(100); yield(it) }
    }

    val job = launch {
        for (k in 1..3) {
            println("\t\tFrom launch $k")
            delay(100)
        }
    }

    simple().forEach { value -> println("From sequence $value") }
    job.join()
}
```

```
From sequence 1
From sequence 2
From sequence 3
                    From launch 1
                    From launch 2
                    From launch 3
```


Flow is non-blocking
(while each element is generated)



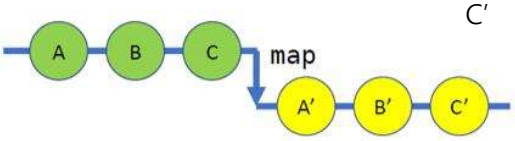

```
@Test
fun `flow is non-blocking`() = runBlocking {
    fun simple() = flow {
        (1..3).forEach { delay(100); emit(it) }
    }

    val job = launch {
        for (k in 1..3) {
            println("\t\tFrom launch $k")
            delay(100)
        }
    }

    simple().collect { value -> println("From flow $value") }
    job.join()
}
```

```
                    From launch 1
From flow 1
                    From launch 2
From flow 2
                    From launch 3
From flow 3
```

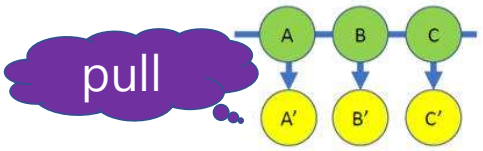
List *vs.* Sequence *vs.* Flow

```
listOf("A", "B", "C").map {
    println("$it")
    compute(it)
}.forEach { println(it) }
```

A
B
C
A′
B′
C′

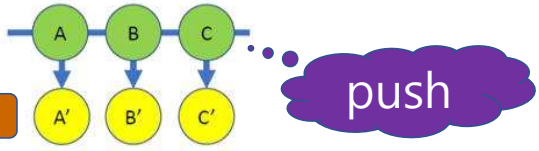



```
sequenceOf("A", "B", "C").map {
    println("$it")
    compute(it)
}.forEach { println(it) }
```



A
A′
B
B′
C
C′

```
flowOf("A", "B", "C").map {
    println("$it")
    compute(it)
}.collect { println(it) }
```



Reactive on emitted values

Sequences *vs.* Flows

- Kotlin Flow a much better version of Sequence?

https://medium.com/mobile-app-development-publication/kotlin-flow-a-much-better-version-of-sequence-d2555ba9eb94

- Use Sequence instead of Kotlin Flow when…

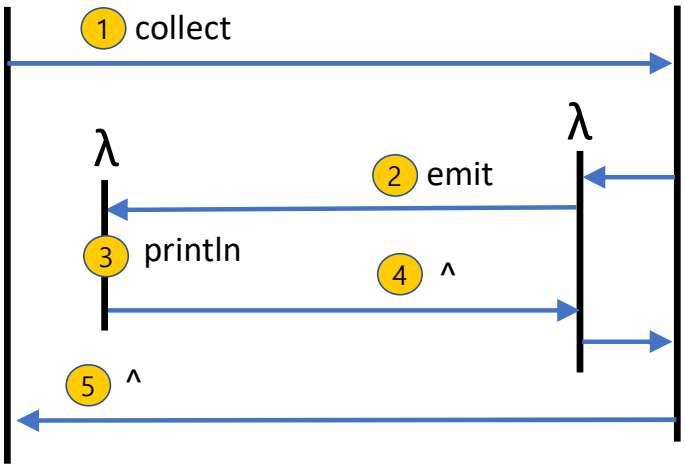
https://medium.com/mobile-app-development-publication/use-sequence-instead-of-kotlin-flow-when-ad577316ce51

Flow

- Flows are similar to **Sequence**s, except that each step of a flow can be *asynchronous*. It is also easy to integrate flows in structured concurrency, to avoid leaking resources.
- In **Flow**, you can suspend anywhere:
  - in the builder function or
  - in any of the operators provided by the Flow API.
- *Suspension* in **Flow** behaves like *backpressure control.*
  - but you don't have to do anything – the compiler will do all the work.

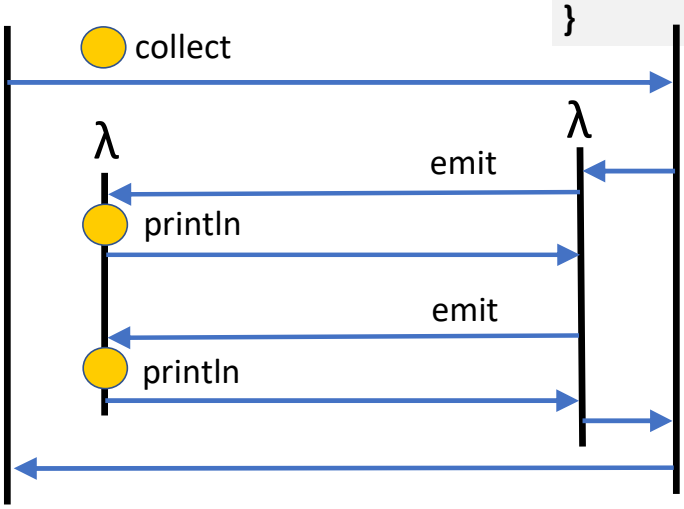

```
flow.collect { value ->
    println(value)
}
```

```
val flow = flow {
    emit("A")
}
```



```
flow.collect { value ->
    println(value)
}
```

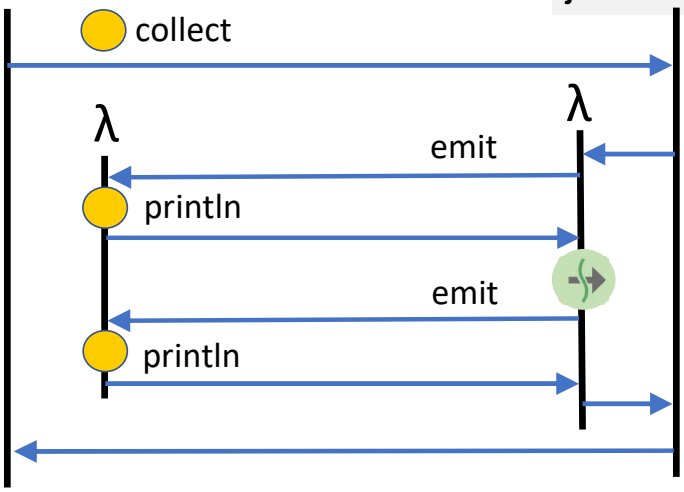
```
val flow = flow {
    emit("A")
    emit("B")
}
```



Asynchronous emitter

```
flow.collect { value ->
    println(value)
}
```

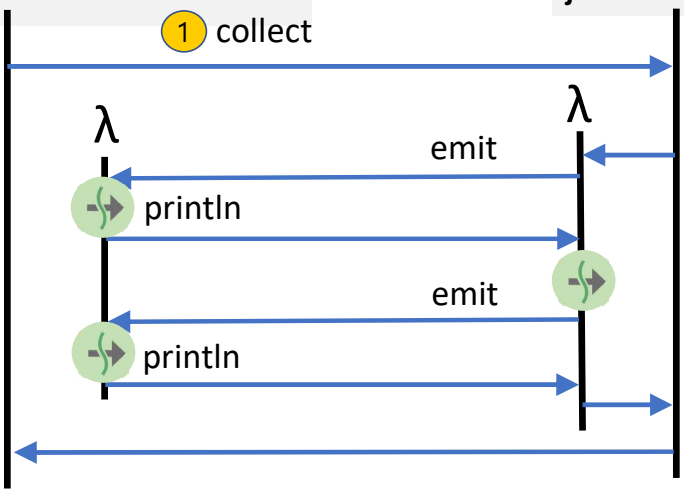
```
val flow = flow {
    emit("A")
    delay(100)
    emit("B")
}
```



Backpressure

```
flow.collect { value ->
    delay(200)
    println(value)
}
```

```
val flow = flow {
    emit("A")
    delay(100)
    emit("B")
}
```



Flow is Cold

```
fun foo(): Flow<String> = flow {
    emit("A")
    emit("B")
    emit("C")
}
```

```
fun main() = runBlocking {
    val flow = foo()
    // flow.collect { x -> println(x) }
}
```

Created on-demand and emit data when they're being observed (i.e., terminal operator is called)

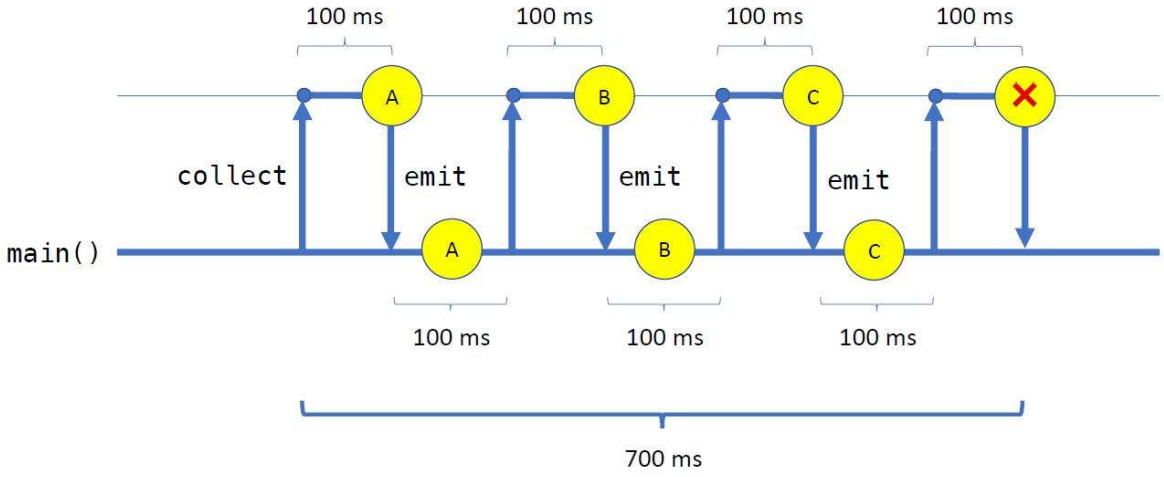
```
fun main() = runBlocking {
    val flow = foo()
    println("First collector")
    flow.collect { println(it) }

    println("Second collector")
    flow.collect { println(it) }
}
```

```
First collector
A
B
C
Second collector
A
B
C
```

Triggers the same code **every time** it is collected.

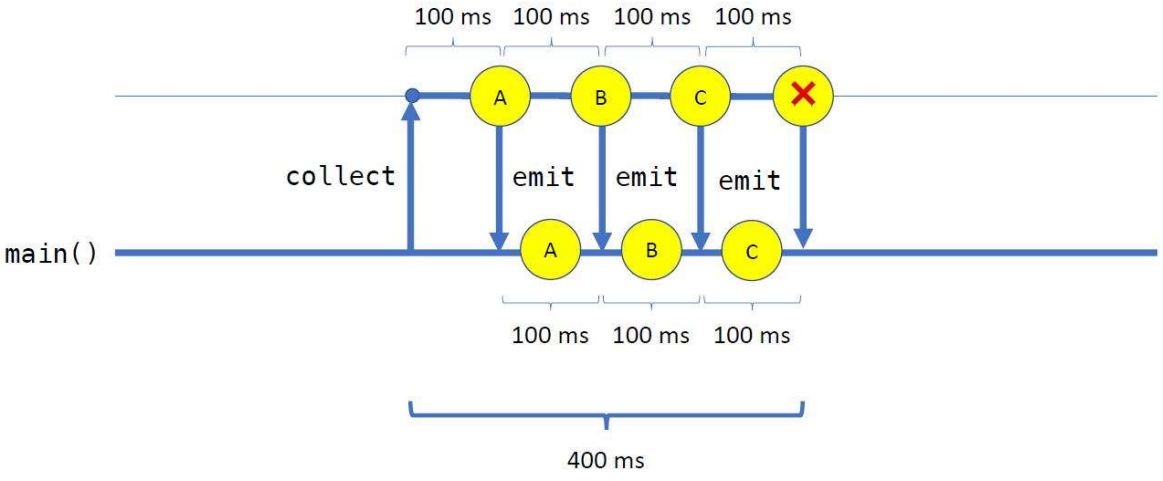
Flow is asynchronous, yet sequential

Single coroutine



Going Concurrent with Flow

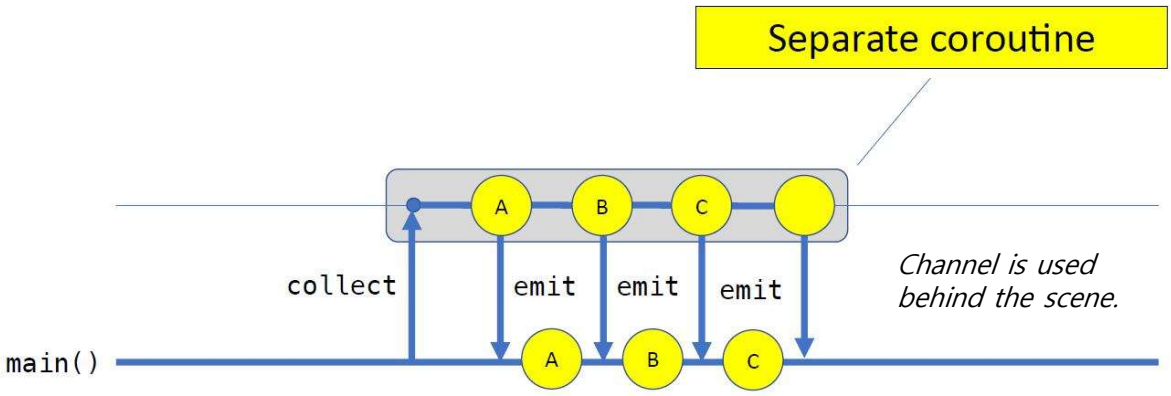
```
flow.buffer().collect { … }
```



Going Concurrent with Flow

```
flow.buffer().collect { … }
```

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



Flow is declarative

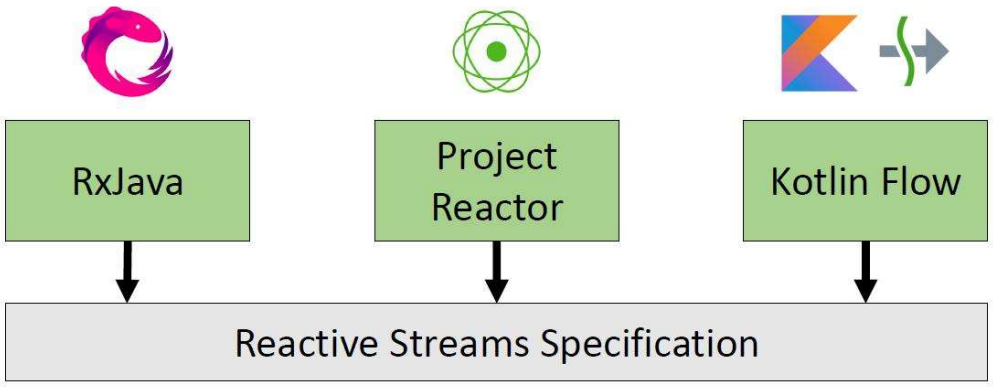
```
val userId: Flow<String> = flow {
    emit("A001")
}

fun userDetails(id: String): Flow<UserDetails> = flow {
    …; emit(UserDetails(id, "Alice", 33, …)))
}

val phones: Flow<Phone> =
  userId.flatMapLatest { id →
    userDetails(id).filter { details →
      details.gender == Gender.FEMALE && details.age > 18
    }.map { details →
      details.phone
    }
  }

phones.collect { makeCall(it) }
phones.toList()
```

Operators

Defined - Declarative

Runs the flow

Flow is Reactive




```
fun <T : Any> Publisher<T>.asFlow(): Flow<T>
fun <T : Any> Flow<T>.asPublisher(): Publisher<T>
```


What's so special about Kotlin Flow

- Null safety in streams
- Interoperability between other reactive streams and coroutines
- Supports Kotlin multiplatform
- No special handling for back pressure [thanks to coroutines]
- Fewer and simple operators [because single operator can handle synchronous and asynchronous logic]
- Perks of structured concurrency

Basic ways to create a flow

- `flowOf(...)` functions to create a flow from a fixed set of values.

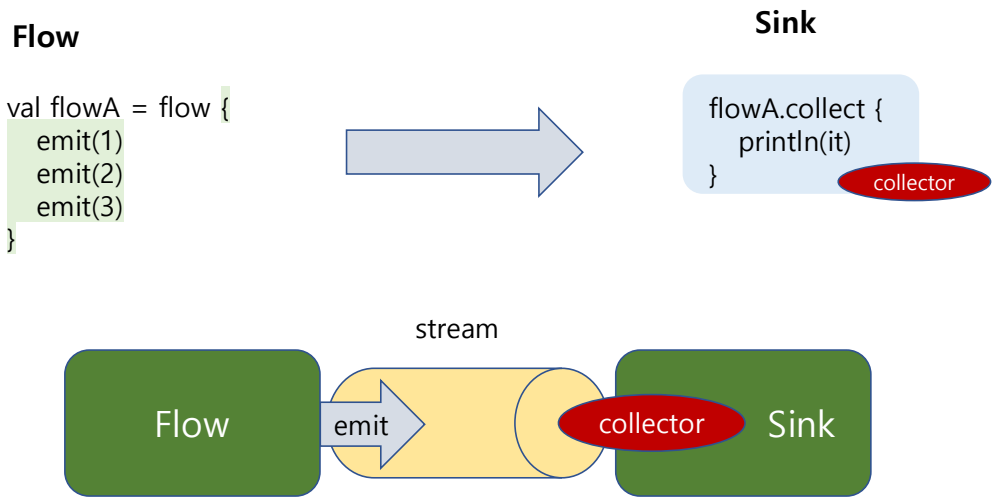
```
flowOf(1, 3, 5, 7, 9)
```

- `asFlow()` extension functions on various types to convert them into flows.

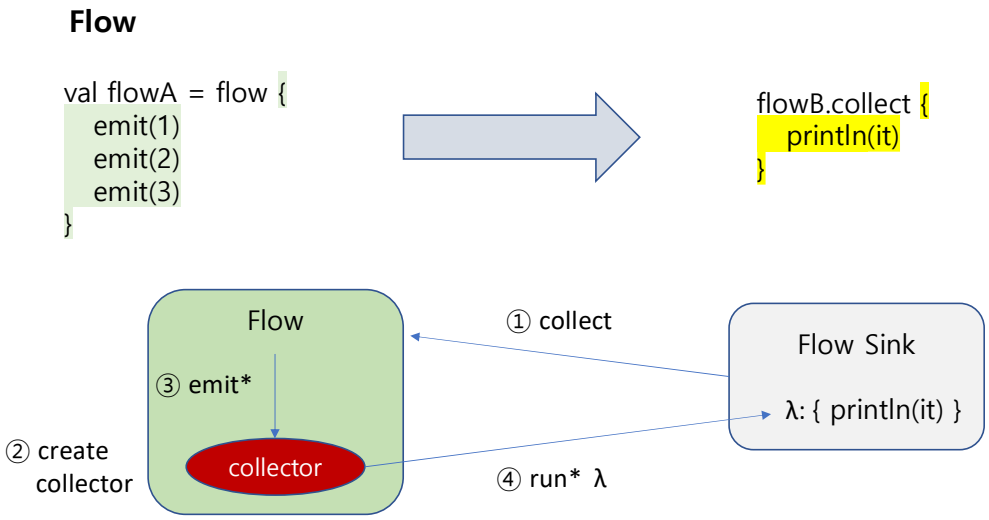
```
(1..3).asFlow()
```

- `flow { ... }` builder function to construct arbitrary flows from sequential calls to emit function.
- `channelFlow { ... }` builder function to construct arbitrary flows from potentially concurrent calls to the send function.
- `MutableStateFlow` and `MutableSharedFlow` define the corresponding constructor functions to create a ***hot flow*** that can be directly updated.

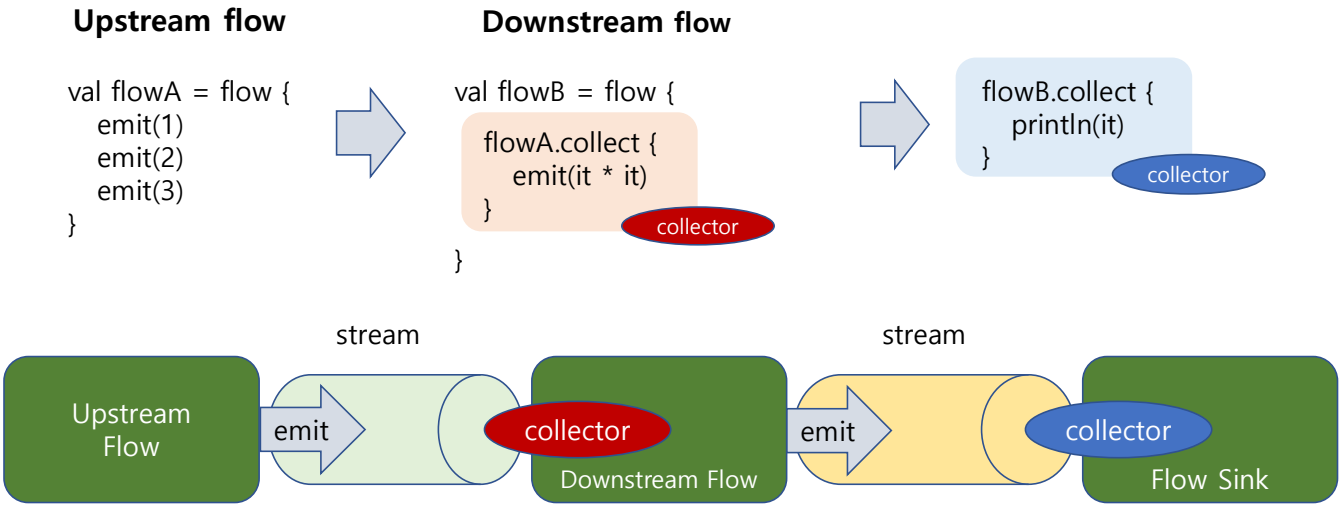
Hyporthetical Flows



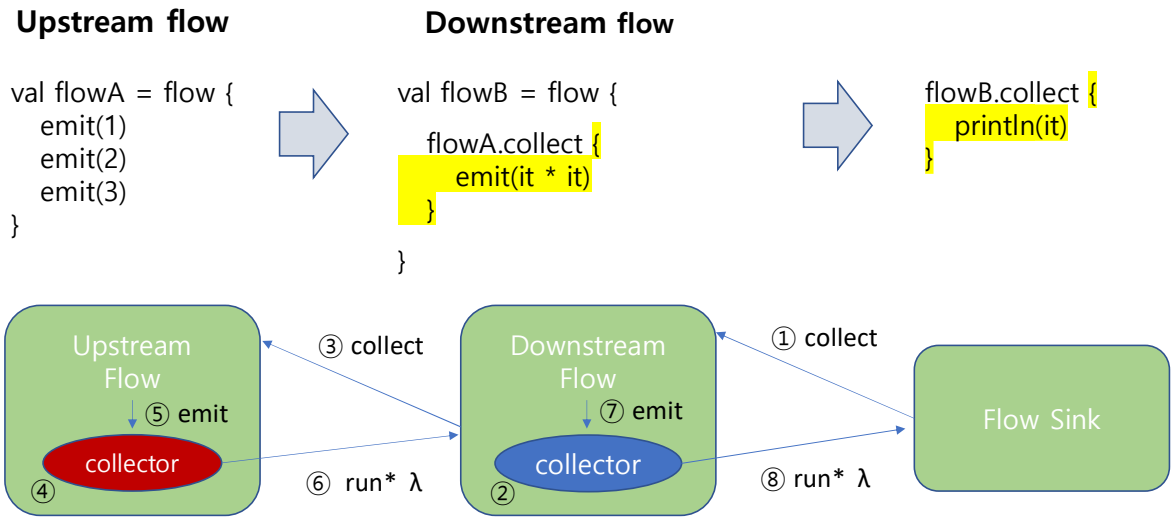
Real Flows



Hypothetical Flows



Real Flows



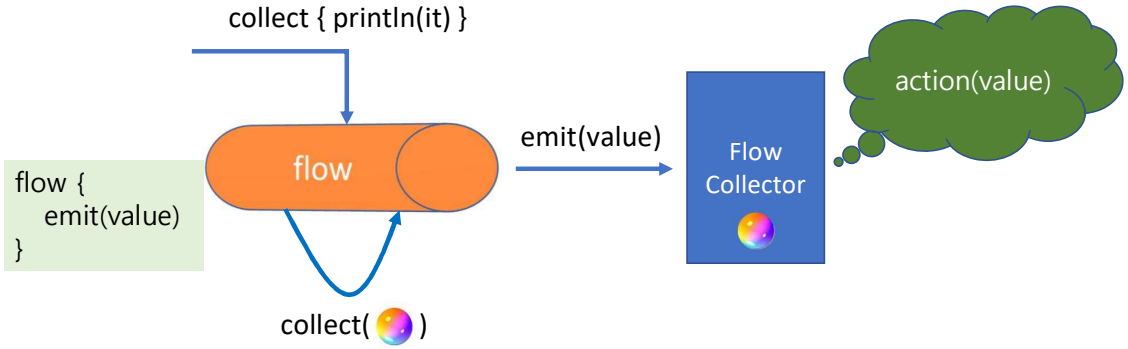
Flow interfaces (before Kotlin 1.6.0)

```
interface Flow<out T> {
    /**
     * Accepts the given collector and emits values into it.
     * This method should never be implemented or used directly.
     */
    public suspend fun collect(collector: FlowCollector<T>)
}
```

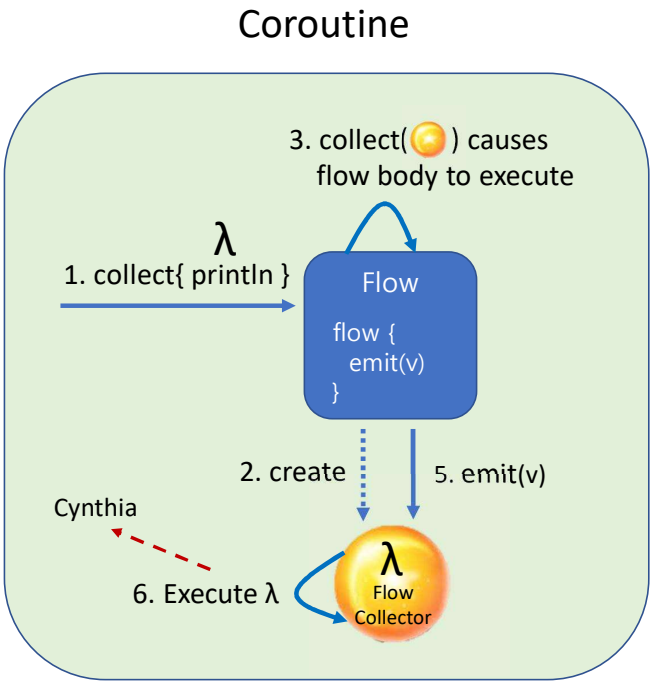
```
/**
 * Used as an intermediate or a terminal collector of the flow and
 * represents an entity that accepts values emitted by the Flow.
 */
interface FlowCollector<in T> {
    /**
     * Collects the value emitted by the upstream.
     */
    public suspend fun emit(value: T)
}
```

Subscriber (before Kotlin 1.6.0) - *deprecated*

```
suspend fun <T> Flow<T>.collect(action: suspend (value: T) -> Unit): Unit =
    collect(object : FlowCollector<T> {
        override suspend fun emit(value: T) = action(value)
    })
```



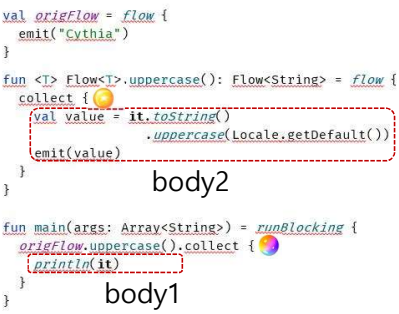

```
val origFlow = flow {
    emit("Cynthia")
}

fun main() = runBlocking {
    origFlow.collect {
        println(it)
    }
}
```

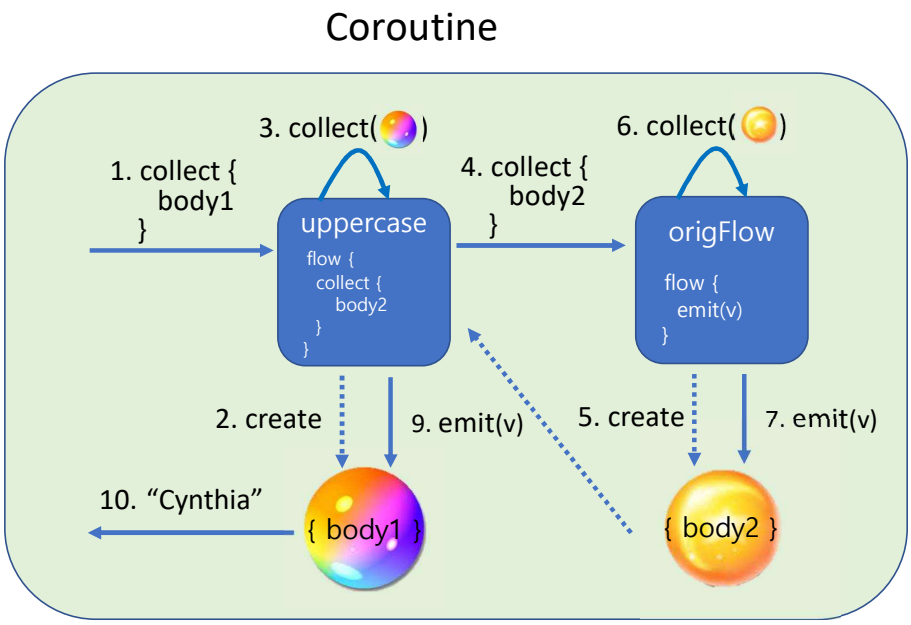


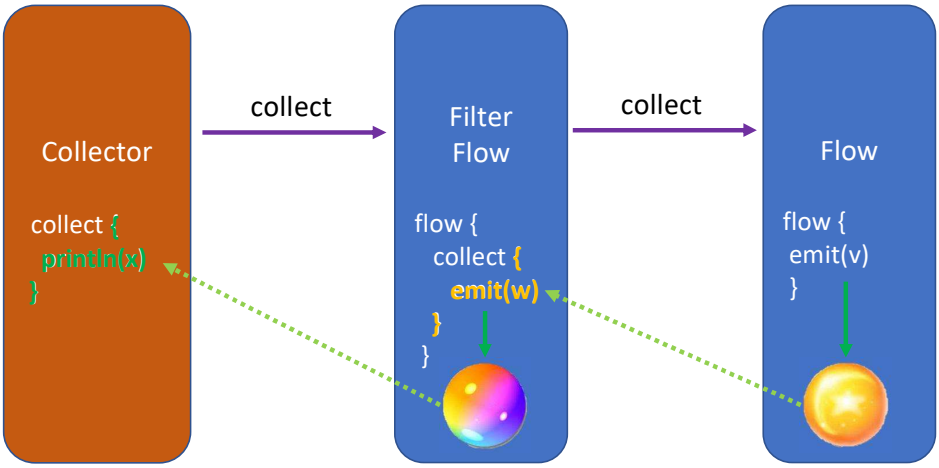
```
val origFlow = flow {
  emit("Cythia")
}

fun <T> Flow<T>.uppercase(): Flow<String> = flow {
  collect {
    val value = it.toString()
                  .uppercase(Locale.getDefault())
    emit(value)
  }
}

fun main(args: Array<String>) = runBlocking {
  origFlow.uppercase().collect {
    println(it)
  }
}
```

body2

body1

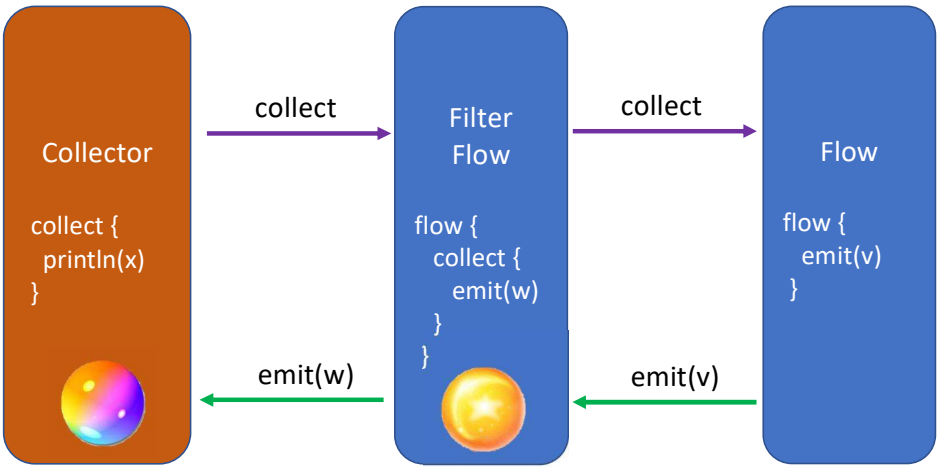






Terminal collector

Intermediate collector



Flow interface (as of Kotlin 1.6.0)

```
interface Flow<out T> {
    /**
     * Accepts the given collector and emits values into it.
     * This method should never be implemented or used directly.
     */
    public suspend fun collect(collector: FlowCollector<T>)
}
```

```
/**
 * Used as an intermediate or a terminal collector of the flow and
 * represents an entity that accepts values emitted by the [Flow].
 */
fun interface FlowCollector<in T> { // SAM
    /**
     * Collects the value emitted by the upstream.
     */
    public suspend fun emit(value: T)
}
```


Subscriber (before Kotlin 1.6.0) - *deprecated*

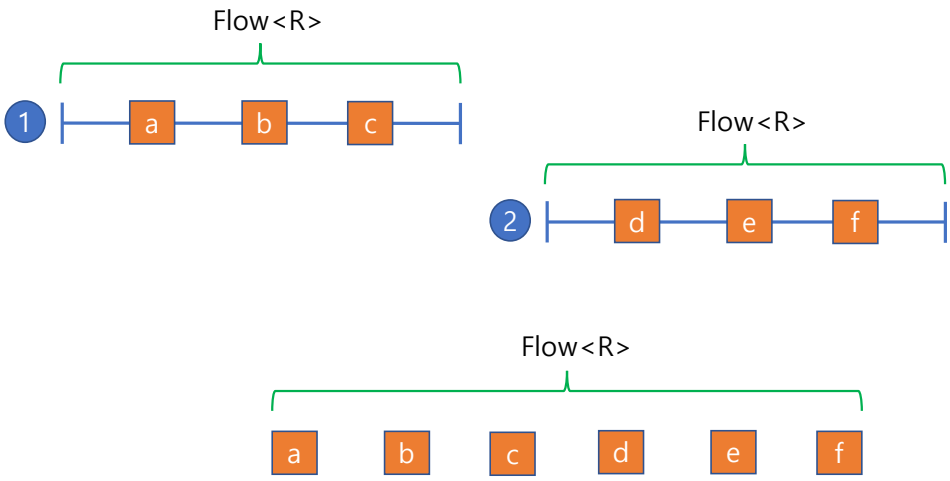
```
suspend fun <T> Flow<T>.collect(action: suspend (value: T) -> Unit): Unit =
    collect(object : FlowCollector<T> {
        override suspend fun emit(value: T) = action(value)
    })
```

Subscriber (as of Kotlin 1.6.0) – *SAM conversion*

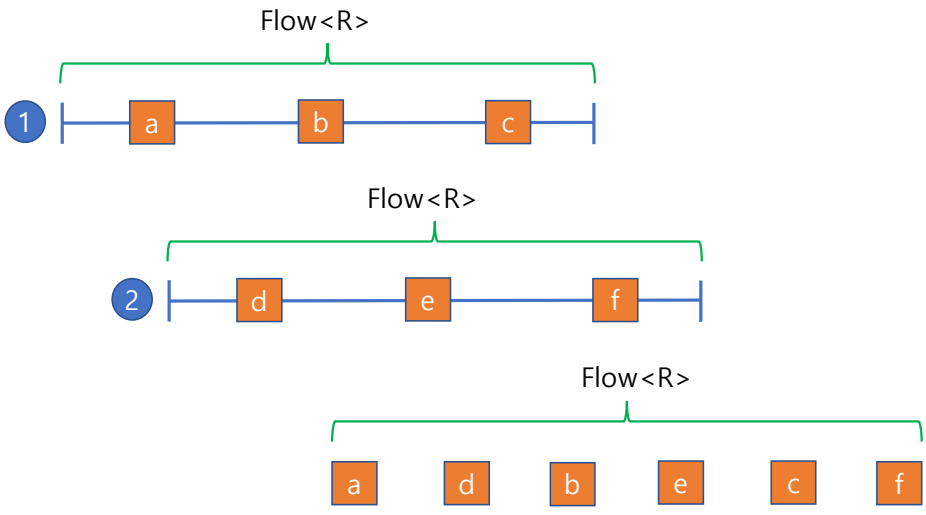
```
flow.collect(object : FlowCollector<Int> {
    override suspend fun emit(value: Int) {
        println("Received $value")
    }
})
```

```
flow.collect { value →
    println("Received $value")
}
```

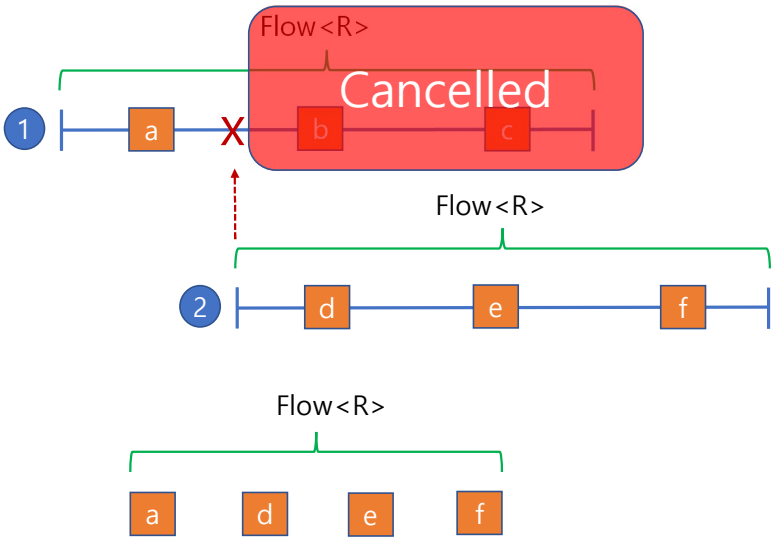
Flow<T>.flatMapConcat(f: (T) -> Flow<R>): Flow<R>



Flow<T>.flatMapMerge(f: (T) -> Flow<R>): Flow<R>



Flow<T>.flatMapLatest(f: (T) -> Flow<R>): Flow<R>



Flow Constraints

All implementations of the Flow interface must adhere to

- **Context preservation**
- **Exception transparency**

to ensure

- **Modularization**: separation of upstream emitters from downstream collectors
- **Local reasoning** about the code with flows

A user of a flow does **not** need to be aware of *implementation details of the upstream flows* it uses.

What would happen if this is allowed?

```
launch(Dispatchers.Main) { // launch in the main thread
    initDisplay() // prepare ui
    dataFlow().collect {  // block of the collector begins
        updateDisplay(it) // update ui
    }
}
```

```
fun dataFlow(): Flow<Int> = flow { // create emitter
    withContext(Dispatchers.Default) {
        while (isActive) {
            emit(someDataComputation())
        }
    }
}
```

- Then `updateDisplay` in the collector's code would try to update UI from the wrong thread.

Romanov's Arguments

https://elizarov.medium.com/execution-context-of-kotlin-flows-b8c151c9309b

- Force every flow collector to write *some boiler-plate code* to ensure that execution of its block happens in the right context or to establish some *project-wide conventions* on the context in which elements of the flow are allowed to be emitted.

```
launch(Dispatchers.Main) { // launch in the main thread
    initDisplay() // prepare ui
    dataFlow().collect { // block of the collector begins
        withContext(Dispatchers.Main) {
            updateDisplay(it) // update ui
        }
    }
}
```

- These conventions are hard to maintain as project becomes larger and 3rd party libraries and operators are taken into account — some of them might fail to document their emission context at all, but even when they do, it places *too much cognitive load* on developers who have to always consult with documentation and should not forget to explicitly specify the context they need.
- The worst part of this story is that if you forget about the context then you might end up with a code that passes all the tests but sometimes produces *subtle and hard to reproduce errors at runtime*.

Context Preservation

- Every flow implementation has to *preserve the collector's context*.
- Collectors can always be sure that their execution context is preserved.

Flow Execution Context

```
fun foo(): Flow<Result<String> =
 flowOf("A", "B", "C").map { name ->
   compute(name)
 }
```

By default, the producer of a flow builder executes **in the** `CoroutineContext` **of the coroutine that collects** from it.

Where does it execute?

```
fun main() = runBlocking {
  val flow = foo()
  flow.collect { x -> println(x) }
}
```

The producer **cannot** `emit` values from a different `CoroutineContext`.

Context Preservation

- The flow encapsulates its own execution context and never propagates or leaks it downstream.
- *Flows should only emit from the same coroutine as the collector.*
  - should not start any coroutines (except scoped primitives like `coroutineScope`)

```
val myFlow = flow {
    // GlobalScope.launch { // is prohibited
    // launch(Dispatchers.IO) { // is prohibited
    // withContext(CoroutineName("myFlow")) // is prohibited
    emit(1) // OK
    coroutineScope {
        emit(2) // OK -- still the same coroutine
    }
}
```

- Use `channelFlow` if the collection and emission of a flow are to be separated into multiple coroutines. (kotlinx-coroutines-core/kotlinx.coroutines.flow/channelFlow)

Correct Emitter Implementations

1. Remove `withContext` around `emit`.

```
fun dataFlow(): Flow<Int> = flow {
    while (currentCoroutineContext().isActive) {
        emit(someDataComputation())
    }
}
```

2. Encapsulate the appropriate context in `someDataComputation` itself:

```
private fun someDataComputation(): Data =
    withContext(Dispatchers.Default) {
        // implementation here
    }
```

3. Use `flowOn()` to change upstream's context:

```
fun dataFlow(): Flow<Int> = flow {
    while (currentCoroutineContext().isActive) {
        emit(someDataComputation())
    }
}.flowOn(Dispatchers.Default) // ^ works on the flow before it
```

Flow Execution Context

```
fun foo(): Flow<Result<String> =
 flowOf("A", "B", "C").map { name ->
  compute(name)
 }
 .flowOn(Dispatchers.Default)
```

flowOn is the only way to change the upstream context ("everything above the flowOn operator").

Executes in Background

```
fun main() = runBlocking {
  val flow = foo()
  flow.collect { x -> println(x) }
}
```

Executes in Collector's Context

Executing in a different CoroutineContext

- `flowOn` changes the `CoroutineContext` of the *upstream flow*, meaning the producer and any intermediate operators applied *before* (or *above*) `flowOn`.
- The *downstream* flow (the intermediate operators after `flowOn` along with the consumer) is not affected and executes on the `CoroutineContext` used to `collect` from the flow.
- If there are multiple `flowOn` operators, each one changes the upstream from its current location.

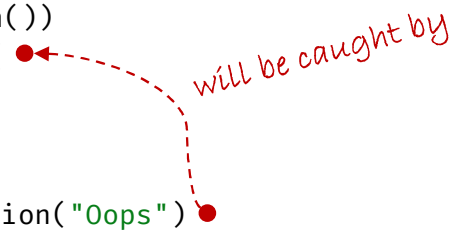
```
withContext(Dispatchers.Main) {
    val singleValue = intFlow // will be executed on IO
        .map { ... }          // Will be executed in IO
        .flowOn(Dispatchers.IO)
        .filter { ... }       // Will be executed in Default
        .flowOn(Dispatchers.Default)
        .single()             // Will be executed in Main
}
```

How do we want to handle exceptions?

```
try {
    flow {
        …    // exceptions may be thrown here
    }.collect {
        …    // or here
    }
} catch (e: Throwable) {
    // handle any uncaught exceptions
    showErrorMessage(e)
}
```


What if using try/catch inside flow {…}?

```
try {
    flow {
        try {
            emit(someComputation())
        } catch (e: Throwable) {
            showErrorMessage(e)
        }
    }.collect {
        throw IllegalStateException("Oops")
    }
} catch (e: Throwable) {
    // useless
}
```



Exception Transparency

> *Terminal operators like `collect` should throw any unhandled exceptions that occur in their code or in upstream flows.*
>
> *Flow implementations never catch or handle exceptions that occur in downstream flows.*

How to ensure Exception Transparency?

- **Never** emit values in the `flow { ... }` builder from inside of a `**try/catch**` block.
- Use `catch` operator that catches exceptions coming from upstream flows while passing all downstream exceptions.

If not followed, an exception in `collect` could be "caught" by an upstream flow!!

catch() operator

- The `catch` can analyze an exception and react to it in different ways:
  - Exceptions can be rethrown.
  - Exceptions can be turned into emission of values using `emit`.
  - Exceptions can be ignored, logged, or processed by some other code.

```
someFlow()
    .catch { exception → emit("Caught ... $exception") }
    .collect { value → log(value) }
```

- The `catch` catches only upstream exceptions (that is an exception from all the operators above `catch`, but not below it).
- If the block in `collect {…}` (placed below `catch`) throws an exception then it escapes.

Declarative Style of Flow Processing

```
scope.launch {
  flow
  .onCompletion { cause -> updateUi(if (cause == null) "Done" else "Failed") }
  .catch { cause -> LOG.error("Exception: $cause") }
  .collect { value -> updateUi(value) }
}
```

```
scope.launch {
  flow
  .onEach { value -> updateUi(value) }
  .onCompletion { cause -> updateUi(if (cause == null) "Done" else "Failed") }
  .catch { cause -> LOG.error("Exception: $cause") }
  .collect()
}
```

launchIn() operator

```
fun <T> Flow<T>.launchIn(scope: CoroutineScope): Job = scope.launch {
    collect()
}
```

- `launchIn()` is a terminal operator that launches flow collection in the scope.
- Usually used with `onEach`, `onCompletion` and `catch` operators, for example:

```
flow
.onEach { value → updateUi(value) }
.onCompletion { cause → updateUi(if (cause == null) "Done" else "Failed") }
.catch { cause → LOG.error("Exception: $cause") }
.launchIn(scope)
```


Flow Testing using Turbine

Testing Kotlin flows

Check a **finite number of items** coming from the flow.

```
@Test fun myRepositoryTest(): Unit = runBlocking {
    counter().first()          // Take the first item
    counter().drop(1).first()  // Take the second item
    counter().take(2).toList() // Take the first 2 items
    // Take the first 2 items matching a predicate
    counter().takeWhile { it < 5 }.take(2).toList()

    // Finite data streams
    counter().toList() // Take all items
    counter().count()  // emits exactly N elements
    counter().count { it % 2 != 0 })

    // Takes the first item verifying that the flow is closed after that
    counter().single()
    // repository.counter().drop(9).single()
}
```

```
fun counter() = flow {
  repeat(10) {
    emit(it)
  }
}
```

```
0
1
[0, 1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1]
10
5
java.lang.IllegalArgumentException: Flow has more than one element
...
```

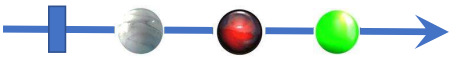

Turbine

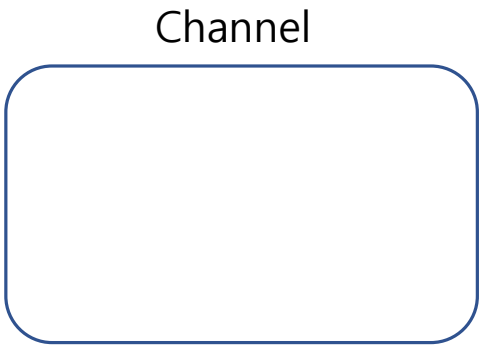
https://github.com/cashapp/turbine

- Small testing library for kotlinx.coroutines Flow.
- The library provides a test extension that internally launches a coroutine and collects from the flow.

```
testImplementation 'app.cash.turbine:turbine:$version'
```

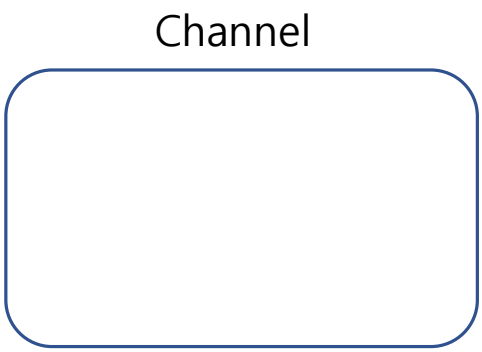
How Turbine Works



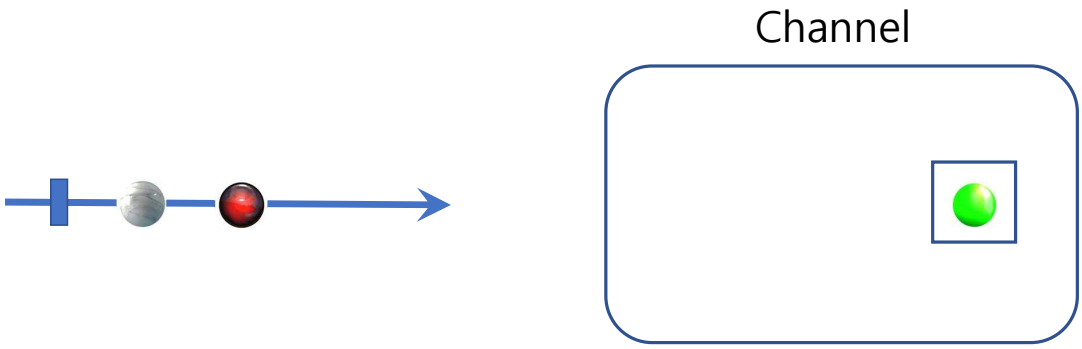


How Turbine Works

```
sealed class Event<out T> {
  object Complete
  data class Error(...)
  data class Item(val value: T)
}
```

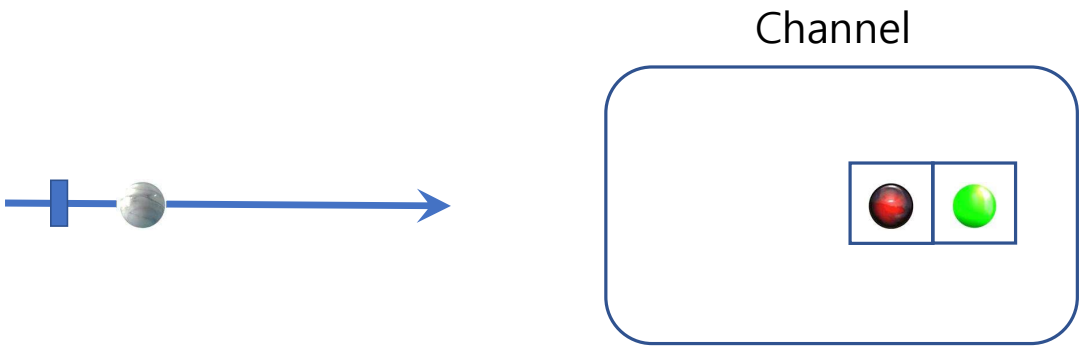


How Turbine Works



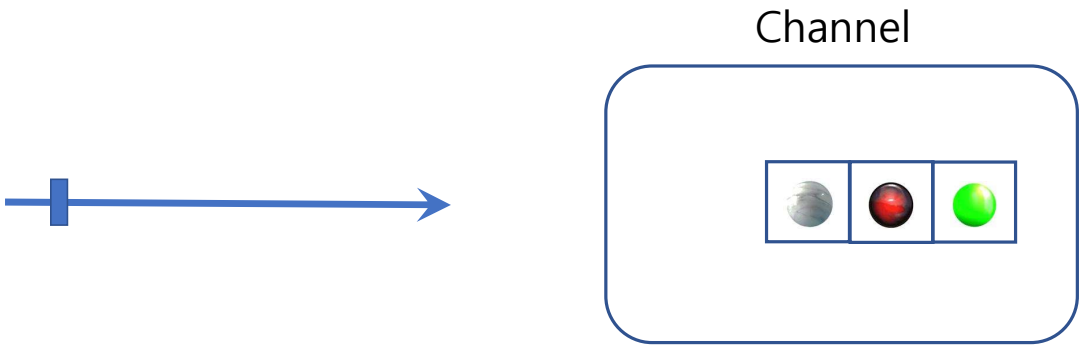

```
data class Item(val value: T): Event<T>
```


How Turbine Works



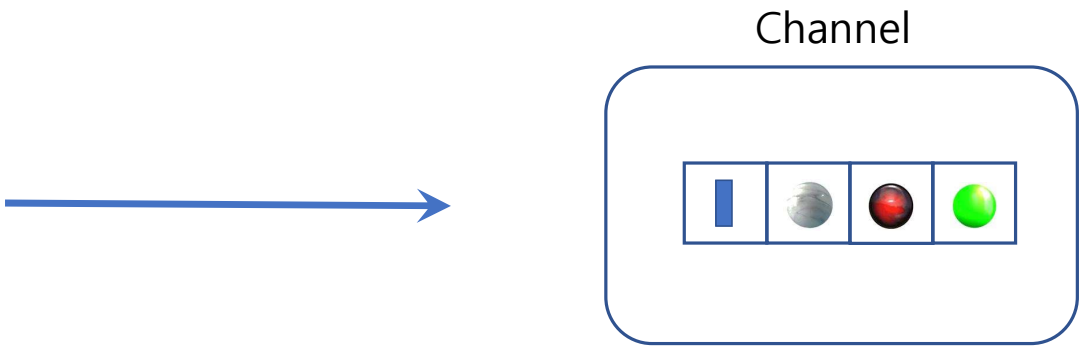

```
data class Item(val value: T): Event<T>
```

How Turbine Works



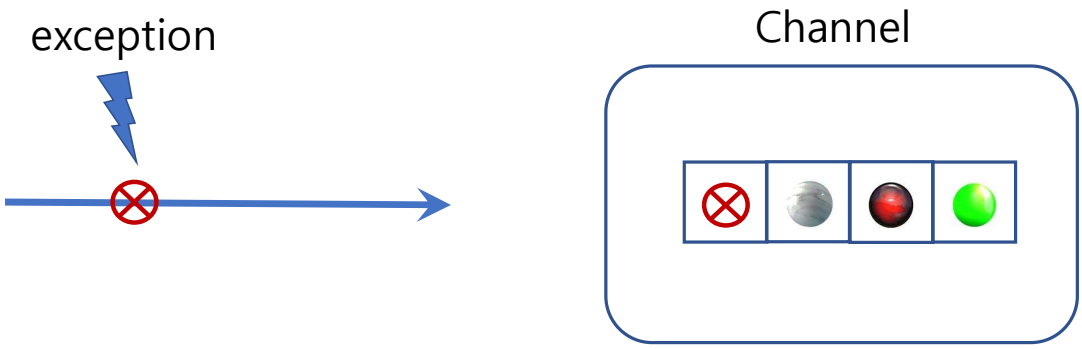

```
data class Item(val value: T): Event<T>()
```


How Turbine Works




```
object Complete: Event<Nothing>()
```

How Turbine Works




```
data class Error(val throwbale: Throwbale): Event<Nothing>()
```


Turbine Testing API

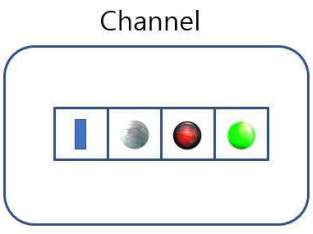
```
public suspend fun <T> Flow<T>.test(
  timeout: Duration = Duration.seconds(1),
  validate: suspend FlowTurbine<T>.() -> Unit
) {
    ...
}
```

FlowTurbine: API to query channel

```
public interface FlowTurbine<T> {
  suspend fun awaitItem(): T
  suspend fun awaitComplete()
  suspend fun awaitError(): Throwable
  suspend fun awaitEvent(): Event<T>
  fun expectMostRecentItem(): T
  fun cancelAndIgnoreRemainingEvents(): Nothing
  fun cancelAndConsumeRemainingEvents(): List<Event<T>>
  fun expectNoEvents()
  ...
}
```

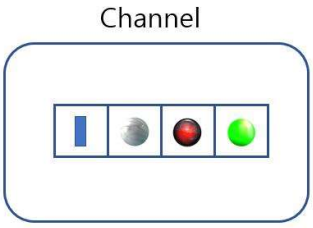

Test with Turbine

```
@Test
fun `should get tokens`() = runTest {
  tokensFlow.test {
    assertThat(awaitItem()).isEqualTo(token1)
    assertThat(awaitItem()).isEqualTo(token2)
    assertThat(awaitItem()).isEqualTo(token3)
    awaitComplete()
  }
}
```



Test with Turbine

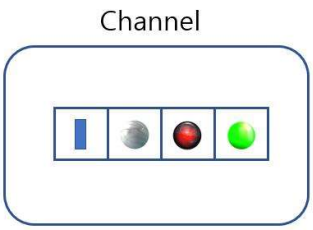
```
@Test
fun `should get tokens`() = runTest {
  tokensFlow.test {
    assertThat(awaitItem()).isEqualTo(token1)
    assertThat(awaitItem()).isEqualTo(token2)
    // token3 verification missing!
    expectComplete()
  }
}
```



app.cash.turbine.AssertionError: Expected complete but found Item(Token(...))

Test with Turbine

```
@Test
fun `should get tokens`() = runTest {
  tokensFlow.test {
    assertThat(awaitItem()).isEqualTo(token1)
    assertThat(awaitItem()).isEqualTo(token2)
    assertThat(awaitItem()).isEqualTo(token3)
    // completion verification missing!
  }
}
```



app.cash.turbine.AssertionError: Unconsumed events found:
- Complete

Migration: Callback-based APIs

Callbacks are a very common solution for asynchronous communication. However, they come with some drawbacks:

- Incomprehensible code with nested callbacks
- Complicated error handling
  – as there isn't an easy way to propagate them

Migration: Callback-based APIs (Cont'd)

For **one-shot** async calls,

- Use the `suspendCoroutine/suspendCancellableCoroutine` API.

For **streaming** data,

- Use the `callbackFlow` API.

```
interface Operation<T> {
    fun performAsync(callback: (T?, Throwable?) -> Unit)
    fun cancel() // cancels ongoing operation
}
```

Single-shot callback

```
suspend fun <T> Operation<T>.perform(): T =
    suspendCancellableCoroutine { continuation →
        performAsync { value, exception →
            when {
                exception ≠ null → // operation had failed
                    continuation.resumeWithException(exception)
                else → // succeeded, there is a value
                    continuation.resume(value as T) {}
            }
        }
        continuation.invokeOnCancellation { cancel() }
    }
```


Multi-shot callback

```
fun <T : Any> Operation<T>.perform(): Flow<T> = callbackFlow {
    performAsync { value, exception ->
        when {
            exception != null -> // operation had failed
                close(exception)
            value == null -> // operation had succeeded
                close()
            else -> // there is a value
                trySend(value as T)
        }
    }
    awaitClose { cancel() }
}
```

Cold flows

- Created on-demand and emit data when they're being observed.
- A flow that triggers the same code **every time** it is collected.
- A new stream for each collector is created and started every time it is collected.

Hot flows

- Always active and can emit data regardless of whether or not they're being observed.
- A stream whose active instance exists **independently** of the presence of collectors.
- Collectors share the same hot stream unless you return new streams (which is discouraged).

Shared & State Flows

- **Shared** and **State Flows** are *hot streams* that can propagate items to *multiple consumers*.
- **Shared Flows** allow you to *replay and buffer* emissions.
- **State Flows** have features such as *sharing strategies* and *conflation*.
- Complete replacement of `BroadcastChannel` and `ConflatedBroadcastChannel` with the `SharedFlow` and `StateFlow`, respectively.

Shared Flow

```
interface SharedFlow<out T> : Flow<T> {
    public val replayCache: List<T>
}
```

- A *hot flow* that shares emitted values among all its collectors in a broadcast fashion, so that all collectors get all emitted values.
- Shared flow *never completes*.
  - Most terminal operators like `toList` would also not complete.
  - But, flow-truncating operators like `take` and `takeWhile` can be called.
- A *subscriber* of a shared flow *can be cancelled*.
- **SharedFlow** is useful for broadcasting events to subscribers that can come and go.

SharedFlow & MutableSharedFlow

```
public interface SharedFlow<out T> : Flow<T> {
    public val replayCache: List<T>
}
```

```
public interface MutableSharedFlow<T> : SharedFlow<T>, FlowCollector<T> {

    // Emits a value to this shared flow
    override suspend fun emit(value: T)

    // Tries to emit a value to this shared flow without suspending
    public fun tryEmit(value: T): Boolean

    // The number of subscribers (active collectors) to this shared flow
    public val subscriptionCount: StateFlow<Int>

    // Resets the replayCache of this shared flow to an empty state
    public fun resetReplayCache()
}
```

MutableSharedFlow

```
public fun <T> MutableSharedFlow(
    replay: Int = 0,
    extraBufferCapacity: Int = 0,
    onBufferOverflow: BufferOverflow = BufferOverflow.SUSPEND
): MutableSharedFlow<T> { … }
```

- *replay* - the number of values replayed to new subscribers
- *extraBufferCapacity* - the number of values buffered in addition to replay
- *onBufferOverflow* - configures an emit action on buffer overflow
  - BufferOverflow.**SUSPEND** (default)
  - BufferOverflow.**DROP_OLDEST**
  - BufferOverflow.**DROP_LATEST**

Replay Count

A replay count specifies how many previous emissions to replay to any subscribers.

```
@Test
fun `replay test`() = runBlocking {
  val sharedFlow =
      MutableSharedFlow<Int>(replay = 0)

  sharedFlow.emit(1)

  withTimeout(1000) {
    val value = sharedFlow.first()
    assertThat(value).isEqualTo(1)
  }
}
```

Timed out waiting for 1000 ms
kotlinx.coroutines.TimeoutCancellationException: ...

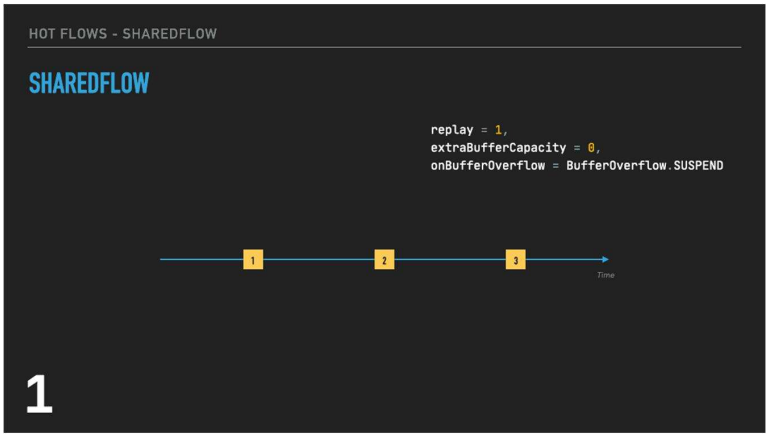
```
@Test
fun `replay test`() = runBlocking {
  val sharedFlow =
      MutableSharedFlow<Int>(replay = 1)

  sharedFlow.emit(1)

  withTimeout(1000) {
    val value = sharedFlow.first()
    assertThat(value).isEqualTo(1)
  }
}
```

passed

Replay = 0

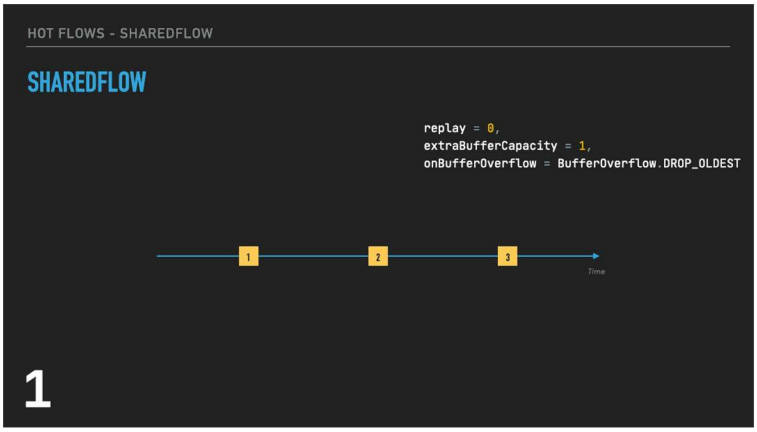
1. This shared flow has three events and two subscribers. The first event is emitted when there are no subscribers yet, so it gets lost forever.
2. By the time the shared flow emits the second event, it already has one subscriber, which gets said event.
3. Before reaching the third event, another subscriber appears, but the first one gets suspended and remains like that until reaching the event. This means emit() won't be able to deliver the third event to that subscriber. When this happens, the shared flow has two options: It either buffers the event and emits it to the suspended subscriber when it resumes or it reaches buffer overflow if there's not enough buffer left for the event.
4. In this case, there's a total buffer of zero — replay + extraBufferCapacity. In other words, buffer overflow. Because onBufferOverflow is set with BufferOverflow.SUSPEND, the flow will suspend until it can deliver the event to all subscribers.
5. When the subscriber resumes, so does the stream, delivering the event to all subscribers and carrying on its work.

Replay = 1



1. When the shared flow reaches the first event without any active subscribers, it doesn't suspend anymore. With replay = 1, there's now a total buffer size of one. As such, the flow buffers the first event and keeps going.
2. When it reaches the second event, there's no more room in the buffer, so it suspends.
3. The flow remains suspended until the subscriber resumes. As soon as it does, it gets the buffered first event, along with the latest second event. The shared flow resumes, and the first event disappears forever because the second one now takes its place in the replay cache.
4. Before reaching the third event, a new subscriber appears. Due to replay, it also gets a copy of the latest event.
5. When the flow finally reaches the third event, both subscribers get a copy of it.
6. The shared flow buffers this third event while discarding the previous one. Later, when a third subscriber shows up, it also gets a copy of the third event.

replay = 0, extraBufferCapacity = 1



1. The behavior is the same at first: With a suspended subscriber and a total buffer size of one, the shared flow buffers the first event.
2. The different behavior starts on the second event emission. With onBufferOverflow = BufferOverflow.DROP_OLDEST, the shared flow drops the first event, buffers the second one and carries on. Also, notice how the second subscriber does not get a copy of the buffered event: Remember, this shared flow has extraBufferCapacity = 1, but replay = 0.
3. The flow eventually reaches the third event, which the active subscriber receives. The flow then buffers this event, dropping the previous one.
4. Shortly after, the suspended subscriber resumes, triggering the shared flow to emit the buffered event to it and cleaning up the buffer.

Convert Flow to SharedFlow

```
public fun <T> Flow<T>.shareIn(
    scope: CoroutineScope,
    started: SharingStarted,
    replay: Int = 0
): SharedFlow<T> {
```

- *scope* - the coroutine scope in which sharing is started
- *started* - the **sharing policy** that controls when sharing is started and stopped
- *replay* - the number of values replayed to new subscribers

started - Sharing Policy

- *Eagerly* — the upstream flow is started even *before* the first subscriber appears.
- *Lazily* — starts the upstream flow *after* the first subscriber appears.
  - The first subscriber gets all the emitted values.
  - Subsequent subscribers are only guaranteed to get the most recent replay values.
- *WhileSubscribed()* — starts the upstream flow when the first subscriber appears, immediately stops when the last subscriber disappears, keeping the replay cache forever.

For **one-shot** operations you can use `Eagerly` or `Lazily`.

For **stream operations**, you should use `WhileSubscribed` to do small but important optimizations.

The WhileSubscribed strategy

- `WhileSubscribed` cancels the upstream flow when there are no collectors.
- When your app goes to the background, you should stop these coroutines.

```
public fun WhileSubscribed(
    stopTimeoutMillis: Long = 0,
    replayExpirationMillis: Long = Long.MAX_VALUE
): SharingStarted = { … }
```

stopTimeoutMillis configures a delay between the disappearance of the last subscriber and the stopping of the upstream flow. It defaults to zero (stop immediately).

replayExpirationMillis configures a delay between the disappeareance of the last subscriber and the clearing of the replay cache. Use zero to expire the cache immediately.

Tip for Android apps!

- You can use `WhileSubscribed(5000)` most of the time to keep the upstream flow active for 5 seconds more after the disappearance of the last collector.
- That avoids restarting the upstream flow in certain situations such as configuration changes.
- This tip is especially helpful when upstream flows are expensive to create and when these operators (`sharedIn` & `stateIn`) are used in `ViewModel`s.

StateFlow & MutableStateFlow

```
interface StateFlow<out T> : SharedFlow<T> {
    val value: T
}

interface MutableStateFlow<T> : StateFlow<T>, MutableSharedFlow<T> {
    override var value: T
    fun compareAndSet(expect: T, update: T): Boolean
}
```

`StateFlow` is a special kind of `SharedFlow`, closest to `LiveData`:

- It always has *only one value*, independently of the number of active observers
- It supports multiple observers (so the flow is shared).
- Any update to the value will be reflected in all flow collectors by emitting a value with state updates.

When exposing UI state to a view, use **StateFlow**. It's a safe and efficient observer designed to hold UI state.

StateFlow is close to LiveData

```
class DownloadingModel {

    private val _state = MutableStateFlow<DownloadStatus>(DownloadStatus.NOT_REQUESTED)
    val state: StateFlow<DownloadStatus> = _state  // or use asStateFlow()

    suspend fun download() {
        _state.value = DownloadStatus.INITIALIZED
        initializeConnection()
        processAvailableContent {
            partialData: ByteArray, downloadedBytes: Long, totalBytes: Long ->
                storePartialData(partialData)
                _state.value = DownloadStatus.IN_PROGRESS
        }
        _state.value = DownloadStatus.SUCCESS
    }
}
```


StateFlow is a subtype of SharedFlow optimized for sharing state

In `StateFlow`, the last emitted item is replayed to new collectors, and items are *conflated* using `Any.equals`.

Use `SharedFlow` when you need a `StateFlow` with tweaks in its behavior such as *extra buffering*, *replaying* more values, or *omitting the initial value*.

```
// MutableStateFlow(initialValue) is a shared flow with the following parameters:
val shared = MutableSharedFlow(
    replay = 1,
    onBufferOverflow = BufferOverflow.DROP_OLDEST
)
shared.tryEmit(initialValue)             // emit the initial value
val state = shared.distinctUntilChanged() // get StateFlow-like behavior
```

StateFlow never completes

- A call to `Flow.collect` on a state flow never completes normally, and neither does a coroutine started by the `Flow.launchIn` function.

```
val stateFlow = MutableStateFlow<UIState>(UIState.Success)
```

```
fun main() = runBlocking {
  stateFlow
    .onCompletion {
       println("ON COMPLETE")
    }.collect {
       println(it)
    }
}
```

Output

UIState.Success
(hang forever)

```
@Test
fun `emit default`() = runBlockingTest {
  stateFlow
    .onCompletion {
       println("ON COMPLETE")
    }.test {
      assertThat(awaitItem())
        .isEqualTo(UIState.Success)
    }
}
```

Output

ON COMPLETE

Convert Flow to StateFlow (StateIn)

```
public fun <T> Flow<T>.stateIn(
    scope: CoroutineScope,   // scope to run this flow
    started: SharingStarted, // sharing policy
    initialValue: T
): StateFlow<T> { … }
```

- *initialValue*
  - Initial value of the state flow; Also used when the state flow is reset using the `WhileSubscribed` with the `replayExpirationMillis` parameter.

```
val result: StateFlow<Resource<Recipe>> =
    someFlow.stateIn(
        scope = viewModelScope,
        started = WhileSubscribed(5000),
        initialValue = Resource.Loading)
```

Suspending Version of StateIn

```
@Test
fun `suspending version of stateIn`() = runBlocking {
    val stateFlow = flow {
        delay(1000)
        emit(1)
        delay(1000)
        emit(2)
    }.stateIn(this) // suspends until the first value is emitted

    val collector = launch {
        stateFlow.collect {
            println(it)
        }
    }
    // …
}
```


Things to know about **shareIn** and **stateIn**

- `StateFlow` allows you to access the last emitted value synchronously by reading its value property, while not the case with `SharedFlow`.
- The `shareIn` and `stateIn` operators convert cold flows into hot flows.
- Can *multicast* the information that comes from a cold upstream flow to multiple collectors.
- Often used to
  - *to improve performance* by sharing the same instance of the flow to be observed by all collectors,
  - *to add a buffer* when collectors are not present, or
  - to be used *as a caching mechanism* for the last emitted item.

WATCH OUT!

Do not create new instances on each function call

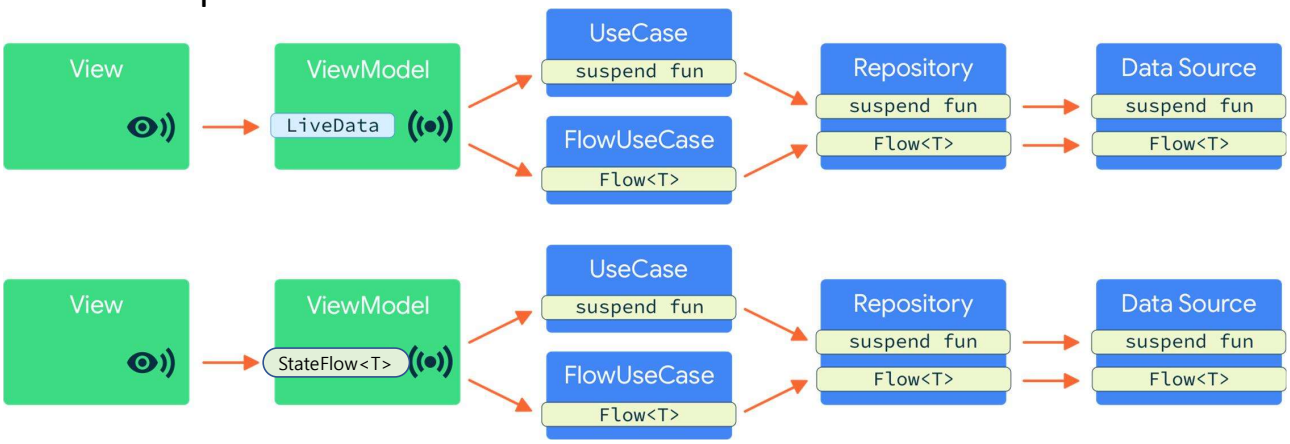
- NEVER use `shareIn` or `stateIn` to create a new flow that's returned when calling a function.

```
class UserRepository(
    private val userLocalDataSource: UserLocalDataSource,
    private val externalScope: CoroutineScope
) {
    // DO NOT USE shareIn or stateIn in a function like this.
    // It creates a new SharedFlow/StateFlow per invocation which is not reused!
    fun getUser(): Flow<User> =
        userLocalDataSource.getUser()
            .shareIn(externalScope, WhileSubscribed())

    // DO USE shareIn or stateIn in a property
    val user: Flow<User> =
        userLocalDataSource.getUser().shareIn(externalScope, WhileSubscribed())
}
```


Flow in Android

Recommended Architecture

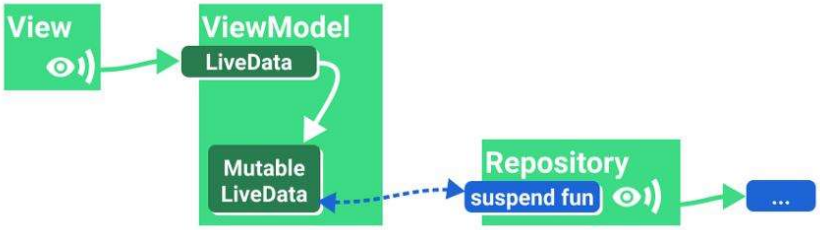
- One-shot operations
- Stream operations



LiveData

#1: Expose the result of a one-shot operation with a Mutable data holder

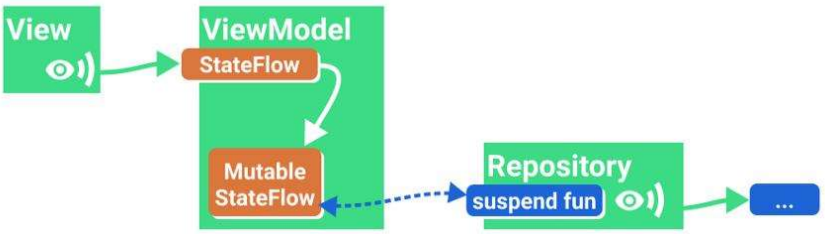
```
class ViewModelLive(val query: String, val repository…) : ViewModel() {
    private val _recipes = MutableLiveData<Resource<List<Recipe>>>(Resource.Loading)
    val recipes: LiveData<Resource<List<Recipe>>> = _recipes

    // Load data from a suspend fun and mutate state
    init {
        viewModelScope.launch {
            _recipes.value = repository.getRecipes(query)
        }
    }
}
```



StateFlow

#1: Expose the result of a one-shot operation with a Mutable data holder

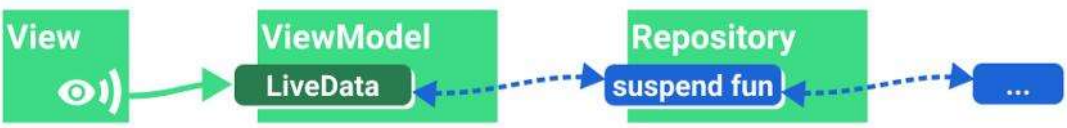
```
class ViewModelFlow(val query: String, val repository…) : ViewModel() {
    private val _recipes = MutableStateFlow<Resource<List<Recipe>>>(Resource.Loading)
    val recipes: StateFlow<Resource<List<Recipe>>> = _recipes

    // Load data from a suspend fun and mutate state
    init {
        viewModelScope.launch {
            _recipes.value = repository.getRecipes(query)
        }
    }
}
```



LiveData

#2: Expose the result of a one-shot operation without a mutable backing property

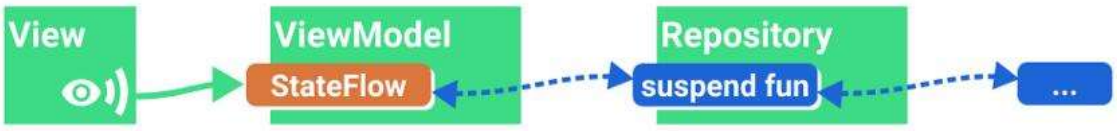
```
class ViewModelLive(
  private val query: String,
  private val repository: Repository
) : ViewModel() {
    val recipes: LiveData<Resource<List<Recipe>>> = liveData {
        emit(Resource.Loading)
        emit(repository.getRecipes(query))
    }
}
```



StateFlow

#2: Expose the result of a one-shot operation without a mutable backing property

```
val recipes: StateFlow<Resource<List<Recipe>>> =
    MutableStateFlow<Resource<List<Recipe>>>(Resource.Loading).apply {
    viewModelScope.launch {
        value = repository.getRecipes(query)
    }
}
```

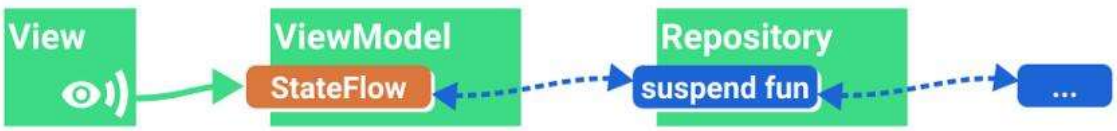


StateFlow: Alternative

#2: Expose the result of a one-shot operation without a mutable backing property

```
class ViewModelFlow(val query: String, val repository: Repository) : ViewModel() {
    val recipes: StateFlow<Resource<List<Recipe>>> = flow {
        emit(repository.getRecipes(query))
    }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.Lazily, // Eagerly or Lazily because it's a one-shot
        initialValue = Resource.Loading
    )
}
```

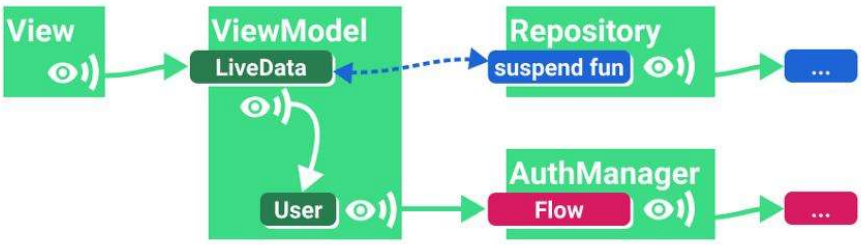
stateIn is a Flow operator that converts a **Flow** to **StateFlow**.



LiveData

#3: One-shot data load with parameters

```
class ViewModelLive(val repository:…, val authManager: …) : ViewModel() {

    private val user: LiveData<User> = authManager.observeUser().asLiveData()

    val favorites: LiveData<Resource<List<Recipe>>> =
        user.switchMap { user ->
            liveData {
                emit(Resource.Loading)
                emit(repository.getFavoriteRecipes(user.id))
            }
        }
```

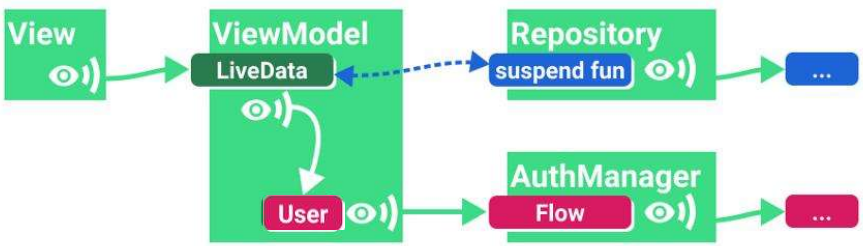


LiveData: Alternative

#3: One-shot data load with parameters

Note that if you need more flexibility you can also use transformLatest and emit items explicitly.

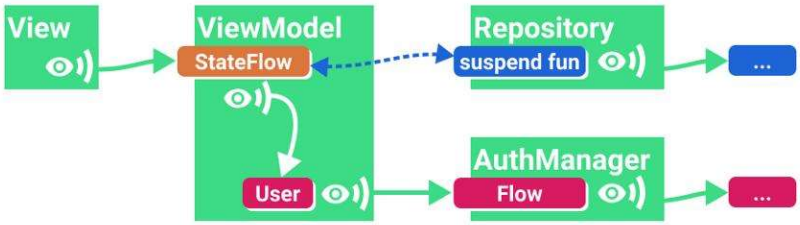
```
class ViewModelLive(val repository…, val authManager…) : ViewModel() {

    private val user: Flow<User> = authManager.observeUser()

    val favorites: LiveData<Resource<List<Recipe>>> = user.mapLatest { user →
        repository.getFavoriteRecipes(user.id)
    }.asLiveData()

}
```



StateFlow

#3: One-shot data load with parameters

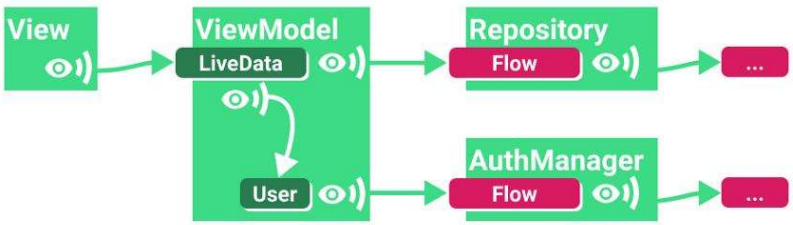
```
class ViewModelFlow(val repository…, val authManager…) : ViewModel() {
    private val user: Flow<User> = authManager.observeUser()

    val favorites: StateFlow<Resource<List<Recipe>>> = user.mapLatest {
        user -> repository.getFavoriteRecipes(user.id)
    }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.Lazily,
        initialValue = Resource.Loading)
}
```



LiveData

#4: Observing a stream of data with parameters

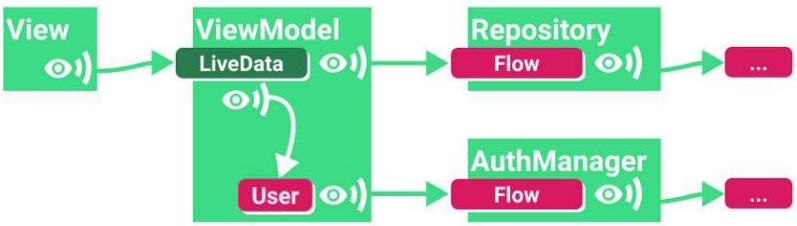
```
class ViewModelFlow(val repository…, val authManager…) : ViewModel() {
    private val user: LiveData<User> =
        authManager.observeUser().asLiveData()

    val favorites: LiveData<Resource<List<Recipe>>> = user.switchMap { user ->
         repository.getFavoriteRecipesFlow(user.id).asLiveData()
    }
}
```



LiveData: Alternative

#4: Observing a stream of data with parameters

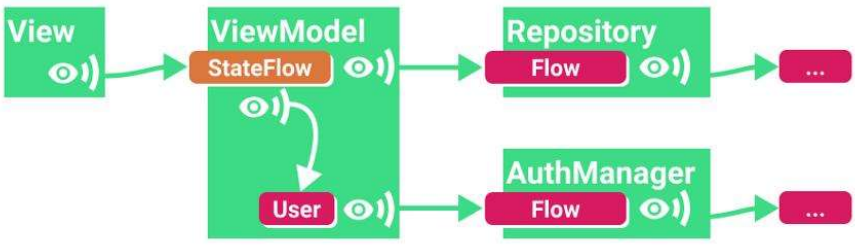
```
class ViewModelFlow(val repository…, val authManager…) : ViewModel() {
    /* Or, preferably, combine both flows using flatMapLatest and convert
     * only the output to LiveData: */
    private val user: Flow<User> = authManager.observeUser()

    val favorites: LiveData<Resource<List<Recipe>>> = user.flatMapLatest {
        user -> repository.getFavoriteRecipesFlow(user.id)
    }.asLiveData()
}
```



StateFlow

#4: Observing a stream of data with parameters

```
class ViewModelFlow(val repository…, val authManager…) : ViewModel() {
    private val userId: Flow<User> = authManager.observeUser()

    val favorites: StateFlow<Resource<List<Recipe>>> = user.flatMapLatest {
        repository.getFavoriteRecipes(it.id)
    }.stateIn(
        scope = viewModelScope,
        started = WhileSubscribed(5000),
        initialValue = Resource.Loading // LoadingUser
    )
}
```



The exposed StateFlow will receive updates whenever the user changes or the user's data in the repository is changed.

MediatorLiveData -> Flow.combine

#5: Combining multiple sources

```
val liveData1: LiveData<Int> = ...
val liveData2: LiveData<Int> = ...

val result = MediatorLiveData<Int>()

result.addSource(liveData1) { value ->
    result.setValue(liveData1.value ?: 0 + (liveData2.value ?: 0))
}
result.addSource(liveData2) { value ->
    result.setValue(liveData1.value ?: 0 + (liveData2.value ?: 0))
}
```

```
val flow1: Flow<Int> = ...
val flow2: Flow<Int> = ...

val result = combine(flow1, flow2) { a, b -> a + b }
```

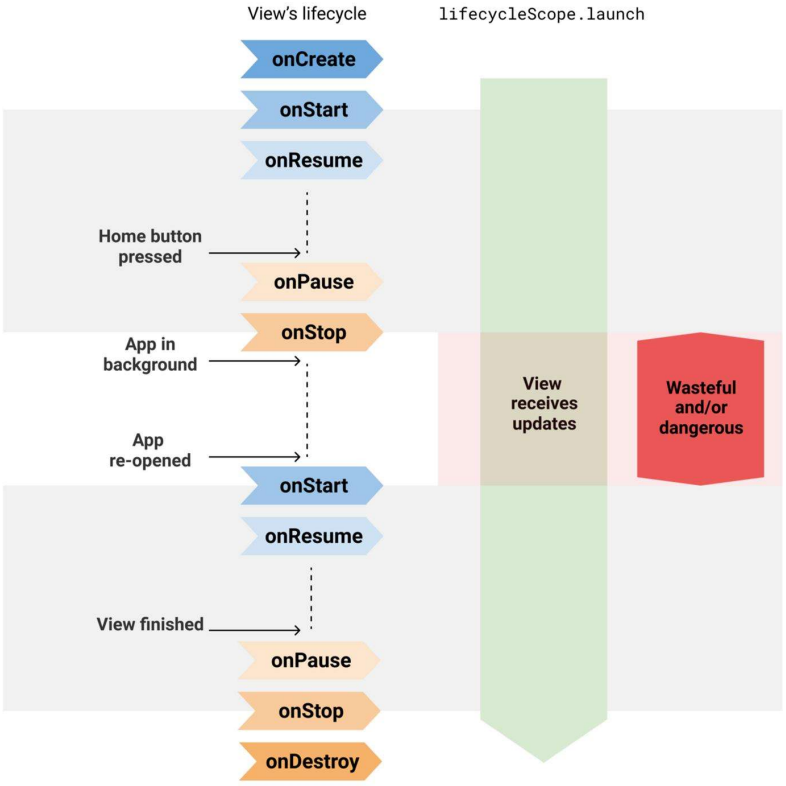

Wasting Resources

- A cold flow backed by a **channel** or using operators with buffers such as `buffer`, `conflate`, `flowOn`, or `shareIn` is not safe to collect with existing APIs such as
  - `CoroutineScope.launch`,
  - `Flow<T>.launchIn`,
  - `LifecycleCoroutineScope.launchWhenX`,

  unless you manually cancel the `Job` that started the coroutine when the activity goes to the background.
- These APIs will *keep the underlying flow producer active* while emitting items into the buffer in the background, and thus wasting resources.

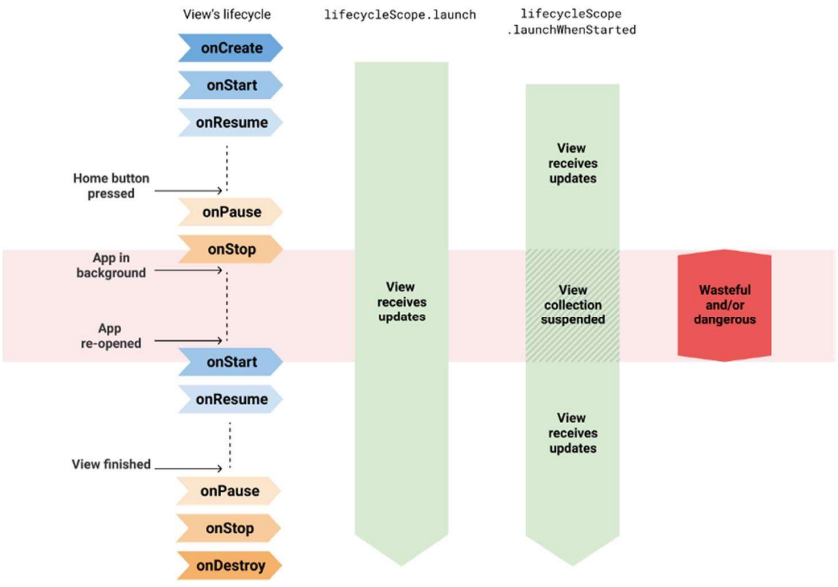
launch

- It's important to note that *it doesn't cancel the coroutine until its lifecycle owner is destroyed*.
- Using the `lifecycleScope.launch` or `launchIn` APIs are dangerous as the view keeps consuming locations even if it's in the background!



LaunchWhenStarted, launchWhenResumed…

- It's important to note that *they also don't cancel the coroutine until their lifecycle owner is destroyed*.



callbackFlow Example

- `callbackFlow` uses a channel, which is conceptually very similar to a blocking queue, and has a default capacity of 64 elements.

```
// Implementation of a cold flow backed by a Channel that sends Location updates
fun FusedLocationProviderClient.locationFlow() = callbackFlow<Location> {
    val callback = object : LocationCallback() {
        override fun onLocationResult(result: LocationResult?) {
            result ?: return
            try { offer(result.lastLocation) } catch(e: Exception) {}
        }
    }
    requestLocationUpdates(createLocationRequest(), callback, Looper.getMainLooper())
        .addOnFailureListener { e ->
            close(e) // in case of exception, close the Flow
        }
    // clean up when Flow collection ends
    awaitClose {
        removeLocationUpdates(callback)
    }
}
```

Collecting this flow from the UI layer using any of the aforementioned APIs keeps the flow emitting locations even if the view is not displaying them in the UI!

UnSafe Collection

```
class LocationActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    // Collects from the flow when the View is at least STARTED and
    // SUSPENDS the collection when the lifecycle is STOPPED.
    // Collecting the flow cancels when the View is DESTROYED.
    lifecycleScope.launchWhenStarted {
        locationProvider.locationFlow().collect {
          // New location! Update the map
        }
    }
    // Same issue with:
    // - lifecycleScope.launch { /* Collect from locationFlow() here */ }
    // - locationProvider.locationFlow().onEach { /* ... */ }.launchIn(lifecycleScope)
  }
}
```

Safe Flow Collection Manually

```
class LocationActivity : AppCompatActivity() {

    // Coroutine listening for Locations
    private var locationUpdatesJob: Job? = null

    override fun onStart() {
        super.onStart()
        locationUpdatesJob = lifecycleScope.launch {
            locationProvider.locationFlow().collect {
                // New location! Update the map
            }
        }
    }

    override fun onStop() {
        // Stop collecting when the View goes to the background
        locationUpdatesJob?.cancel()
        super.onStop()
    }
}
```


A safer way to collect flows from Android UIs

- Not doing more work than necessary
- Wasting resources (both CPU and memory)
- Leaking data when the view goes to the background

- Best practices:
  - `Lifecycle.repeatOnLifecycle`, and `Flow.flowWithLifecycle`

Lifecycle.repeatOnLifecycle

available in the lifecycle-runtime-ktx library:2.4.0-alpha01 or later

The solution needs to be

1. simple,
2. friendly or easy to remember/understand, and more importantly
3. safe!

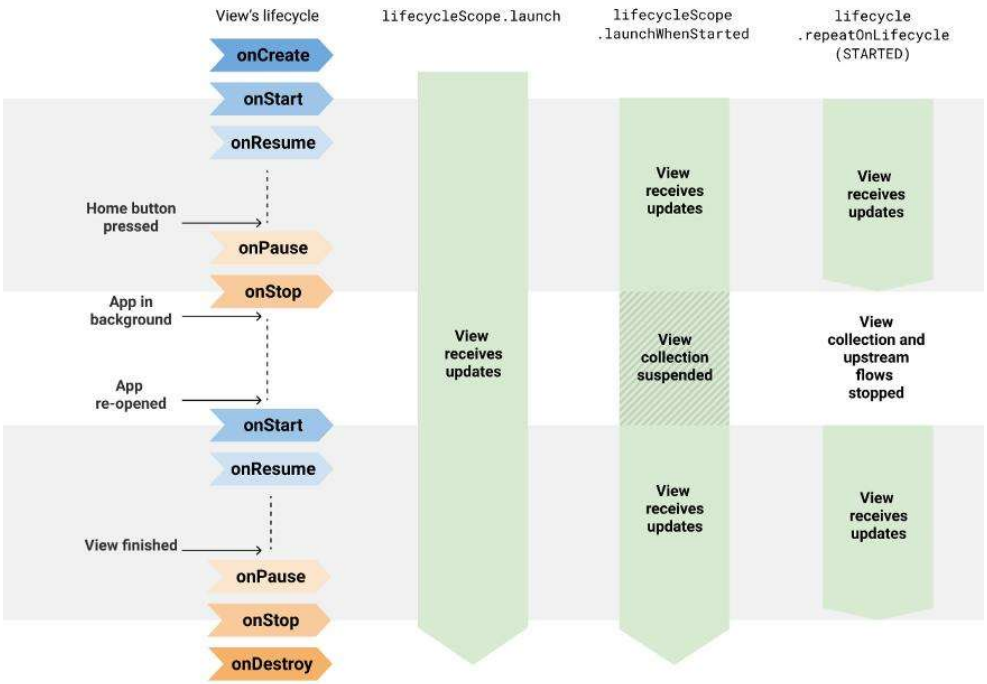
It should work for all use cases regardless of the flow implementation details.


```
class LocationActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Create a coroutine since `repeatOnLifecycle` is a suspend function
        lifecycleScope.launch {
            // The block passed to `repeatOnLifecycle` is executed when the lifecycle
            // is at least STARTED and is cancelled when the lifecycle is STOPPED.
            // It automatically restarts the block when the lifecycle is STARTED again.
            lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
                // Safely collect from locationFlow when the `lifecycle` is STARTED
                // and stops collection when the lifecycle is STOPPED
                locationProvider.locationFlow().collect {
                    // New location! Update the map
                }
            }
            // Resumes the calling coroutine when the `Lifecycle` is destroyed.
        }
    }
}
```

it's recommended to call this API in the activity's **onCreate** or fragment's **onViewCreated** methods to avoid unexpected behaviors.

```
class LocationFragment: Fragment() {
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        // ...
        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                locationProvider.locationFlow().collect {
                    // New location! Update the map
                }
            }
        }
    }
}
```

Important: Fragments should *always* use the `viewLifecycleOwner` to trigger UI updates. However, that's not the case for `DialogFragment`s which might not have a `View` sometimes. For `DialogFragment`s, you can use the `lifecycleOwner`.



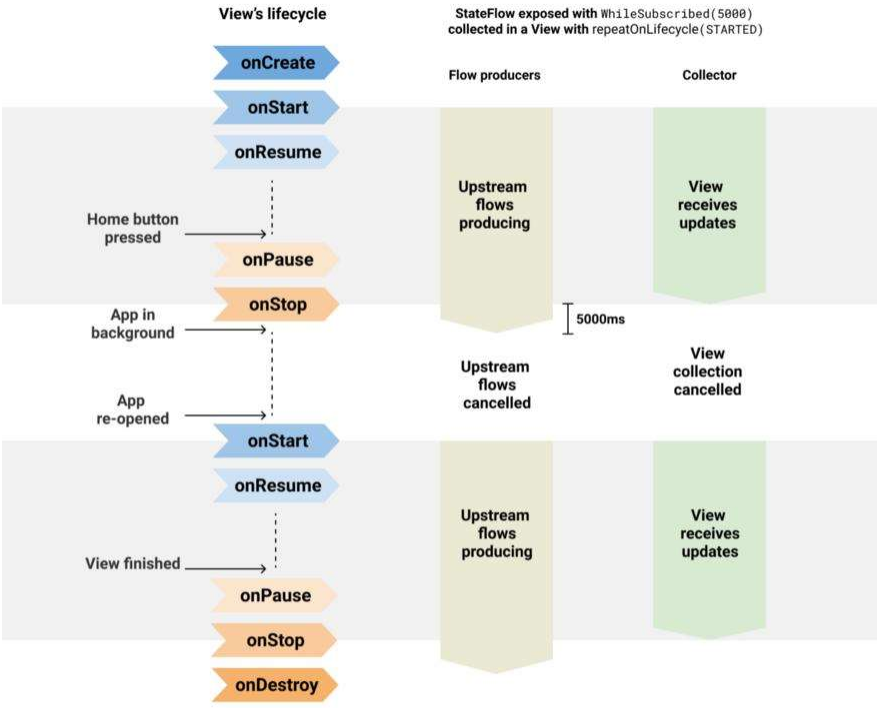
Flow.flowWithLifecycle

- You can also use the `Flow.flowWithLifecycle` operator when you have only one flow to collect.
- This API uses the `repeatOnLifecycle` API under the hood, and emits items and cancels the underlying producer when the Lifecycle moves in and out of the target state.


```
class LocationActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Listen to one flow in a lifecycle-aware manner using flowWithLifecycle
        lifecycleScope.launch {
            locationProvider.locationFlow()
                .flowWithLifecycle(this, Lifecycle.State.STARTED)
                .collect {
                    // New location! Update the map
                }
        }

// continue on next slide
```

```kotlin
// Listen to multiple flows
        lifecycleScope.launch {
            lifecycle.repeatOnLifecycle(Lifecycle.State.STARTED) {
                // As collect is a suspend function, if you want to collect
                // multiple flows in parallel, you need to do so in
                // different coroutines
                launch {
                    flow1.collect { /* Do something */ }
                }

                launch {
                    flow2.collect { /* Do something */ }
                }
            }
        }
    }
}
```


- Mixing the **repeatOnLifecycle** with the **StateFlow** will get you the best performance while making a good use of the device's resources.



Configuring underlying producers

- Even if you use these APIs, watch out for *hot* flows that could waste resources even if they aren't collected by anyone!
- **Depending on the use case, decide whether the producer needs to be always active or not**.
- The `MutableStateFlow` and `MutableSharedFlow` APIs expose a `subscriptionCount` field that you can use to stop the underlying producer when `subscriptionCount` is zero.
- Similarly, the `Flow.stateIn` and `Flow.shareIn` operators can be configured with the sharing started policy for this. `WhileSubscribed()` will stop the underlying producer when there are no active observers!

  On the contrary, `Eagerly` or `Lazily` will keep the underlying producer active as long as the `CoroutineScope` they use is active.

Summary

The best way to expose data from a `ViewModel` and collect it from a view is:

- ✔ Expose a `StateFlow`, using the `WhileSubscribed` strategy, with a timeout.
- ✔ Collect with `repeatOnLifecycle`

Any other combination will keep the upstream Flows active, wasting resources:

- ✘ Expose using `WhileSubscribed` and `collect` inside `lifecycleScope.launch/launchWhenX`
- ✘ Expose using `Lazily`/`Eagerly` and collect with `repeatOnLifecycle`

Of course, if you don't need the full power of `Flow`… just use `LiveData`. :)

Flow over LiveData

1. **More powerful transformations.** Transform, filter and do anything with the data.
2. **Can be combined.** Combine as much flows as you need.
3. **Collectable from any dispatcher.** No more need to set `Dispatchers.Main` after fetching data on asynchronously on `Dispatchers.IO`.
4. **Adjustable (replays, buffers, etc.).** More flexibility on how to handle data and errors.
5. **Convertible from cold to hot (`stateIn`).** We can easily convert a cold flow to `StateFlow` using `stateIn()`.
6. **SingleLiveEvent made easy.** (because we don't want our error dialogs pop-up on each rotation) A `Channel` is a great solution that is convertible to a `Flow` with a simple extension function.
7. **Easier to test.** No more `getOrAwaitValue` or `observeForTesting`.
8. **Better for multiplatform projects.** No need to re-write code since they're not tied to specific platform, unlike `LiveData`.

Conclusion for Android

Use the `Lifecycle.repeatOnLifecycle` or `Flow.flowWithLifecycle` APIs to safely collect flows from the UI layer in Android.

- Collecting state flows using these APIs is a natural replacement for `LiveData` in Kotlin-only apps.

Coroutines best practices

1. Inject Dispatchers into classes
Don't hardcode them when creating new coroutines or calling `withContext`.
✅ *Benefits*: ease of testing as you can easily replace them for both unit and instrumentation tests.

2. The ViewModel/Presenter layer should create coroutines
If it's a UI-only operation, then the UI layer can do it. If you think this is not possible in your project, it's likely you're not following best practice #1 (i.e. it's more difficult to test VMs that don't inject `Dispatchers`; in that case exposing suspend functions makes it doable).
✅ *Benefits*: The UI layer should be dumb and not directly trigger any business logic. Instead, defer that responsibility to the ViewModel/Presenter layer. Testing the UI layer requires instrumentation tests in Android which need an emulator to run.

3. The layers below the ViewModel/Presenter layer should expose suspend functions and Flows
If you need to create coroutines, use `coroutineScope` or `supervisorScope`.
✅ *Benefits*: The caller (generally the ViewModel layer) can control the execution and lifecycle of the work happening in those layers, being able to cancel when needed.