Kotlin Flows and Channels for Android

Ryan Pierce

Mobile App Developer

Android Developer @ Capital One

Aerospace Engineer

Ryan Pierce

Challenges

Challenges of Asynchrony

- Race conditions
- Back pressure
- Leaked resources
- Threading
  - Expensive
  - Starvation
  - Deadlocks
- and more...

Shared Mutable State

```
A widely adopted 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

Do not communicate by sharing memory;

instead,

share memory by communicating.

Coroutines

A prettier callback?

```
fun getValueAsync(onCompletion:(R)->Unit) {
   val result: R = getValue()
   onCompletion(result)
}
```

```
getValueAsync() { result ->
     print(result)
}
```

A prettier callback?

```
suspend fun getValue(): R {
    return getValueAsync()
}
```

```
launch {
    val result1 = getValue()

    print(result1)
}
```

A prettier callback?

```
suspend fun getValue(): R {
    return getValueAsync()
}
```

```
launch {
    val result1 = getValue()
    val result2 = nextValue(result1)
    print(result2)
}
```

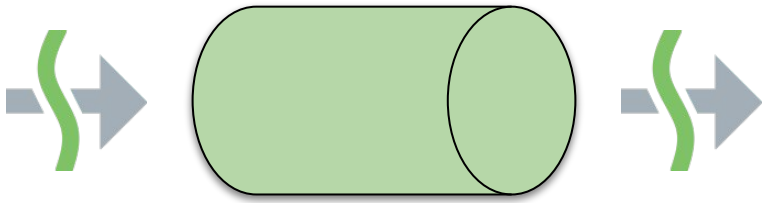
A prettier callback ... **that suspends**

A prettier callback … **that suspends**

A prettier callback ... **that suspends** 

```
launch {
    val result1 = getValue()
    val result2 = //...
    print(result2)
}
```

```
launch {
    val result1 = getValue()
    val result2 = //...
    print(result2)
}
```

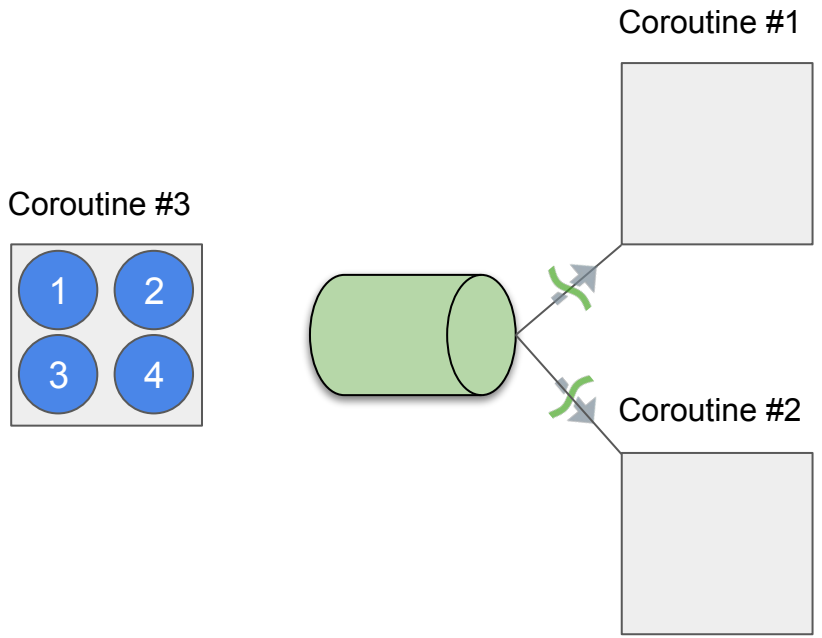


Channels

Simple Channel Demo

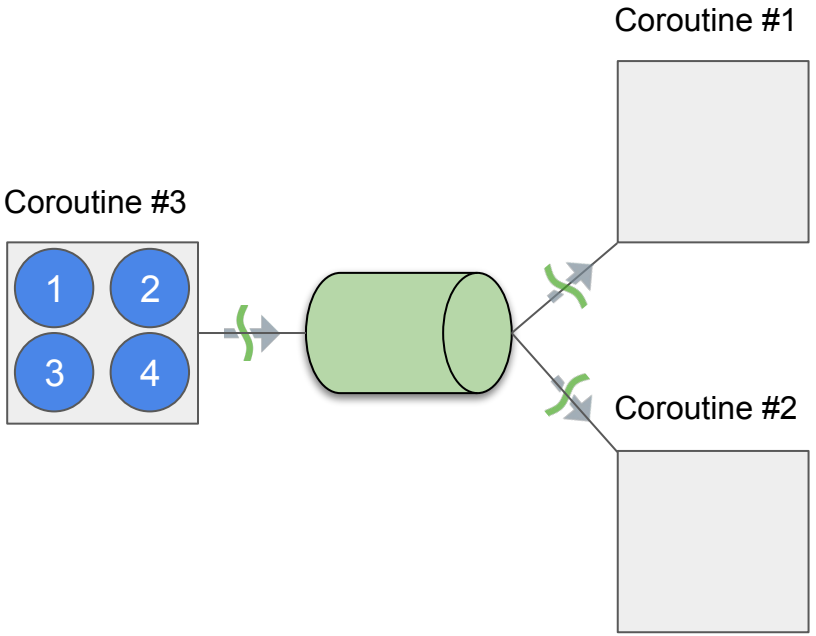
```
val channel = Channel<Int>()

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4)


}
```

Note: Using `for (it in channel)` instead of `channel.consumeEach` will lead to safer and more consistent behavior in fan-out channels.



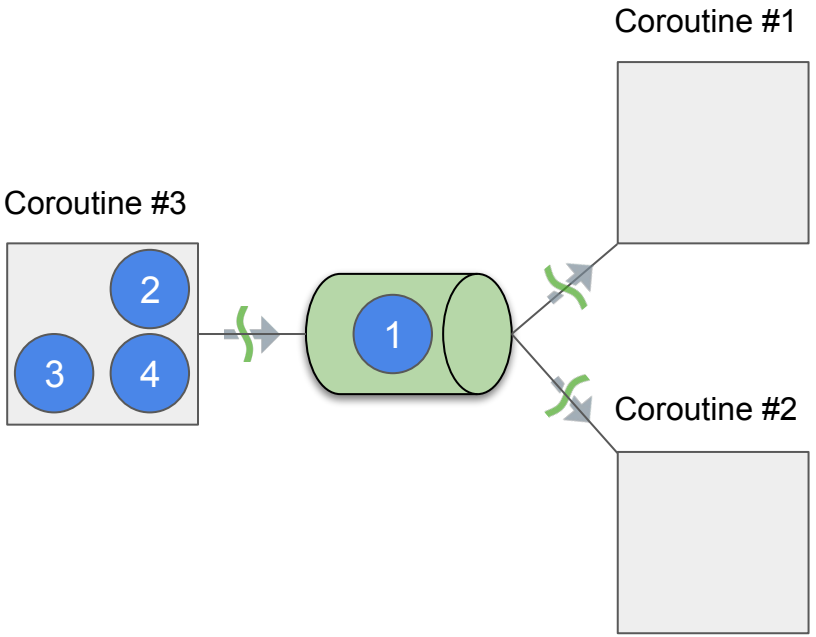
Simple Channel Demo

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

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4).forEach {
        channel.send(it)
    }
}
```



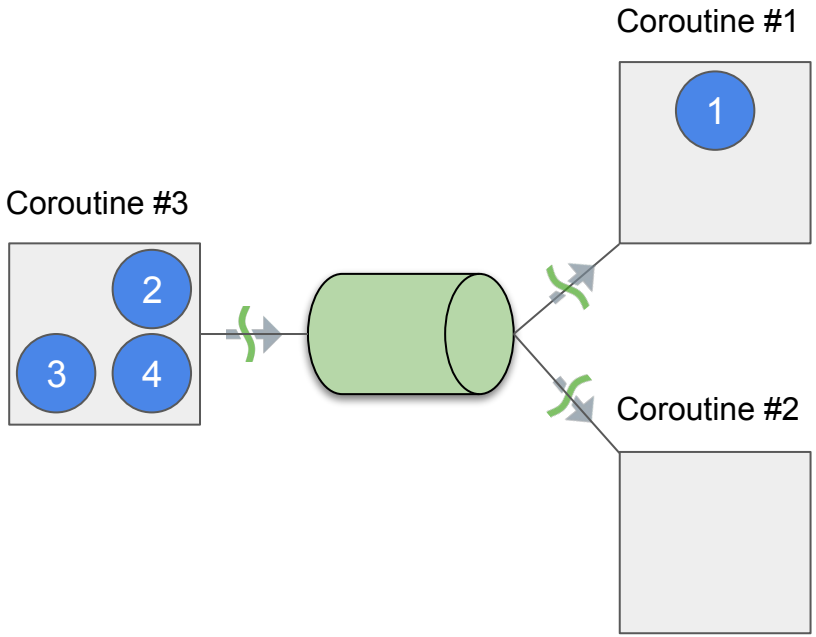
Simple Channel Demo

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

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4).forEach {
        channel.send(it)
    }
}
```



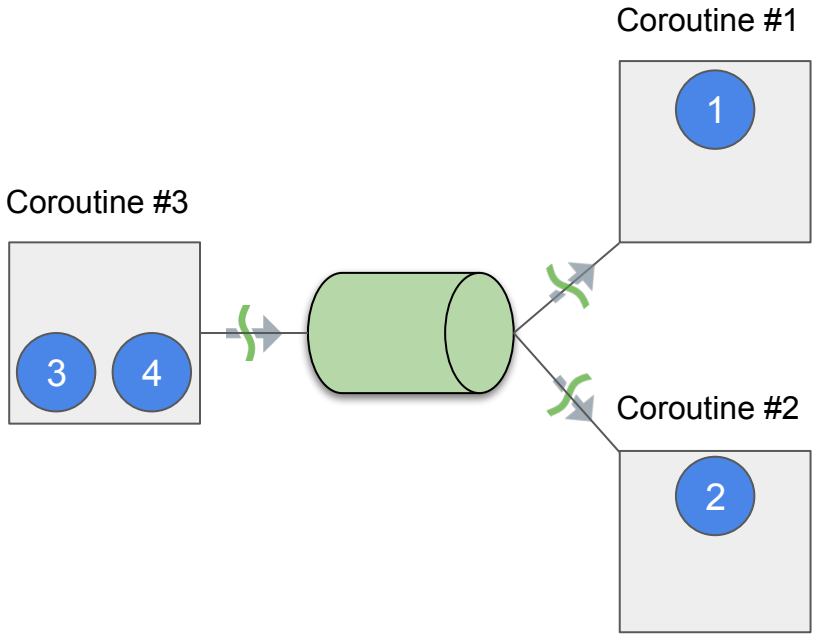
Simple Channel Demo

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

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4).forEach {
        channel.send(it)
    }
}
```



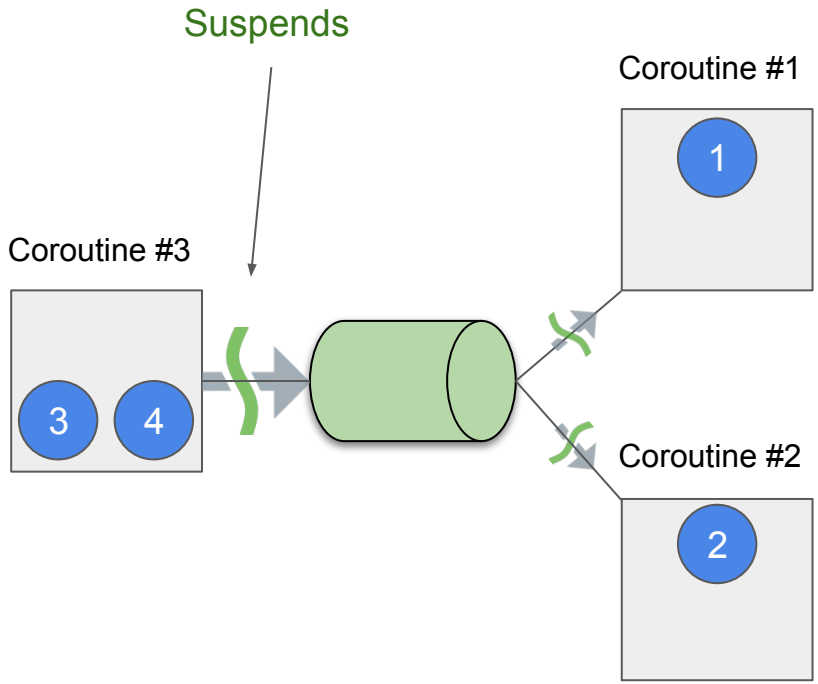
Simple Channel Demo

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

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4).forEach {
        channel.send(it)
    }
}
```



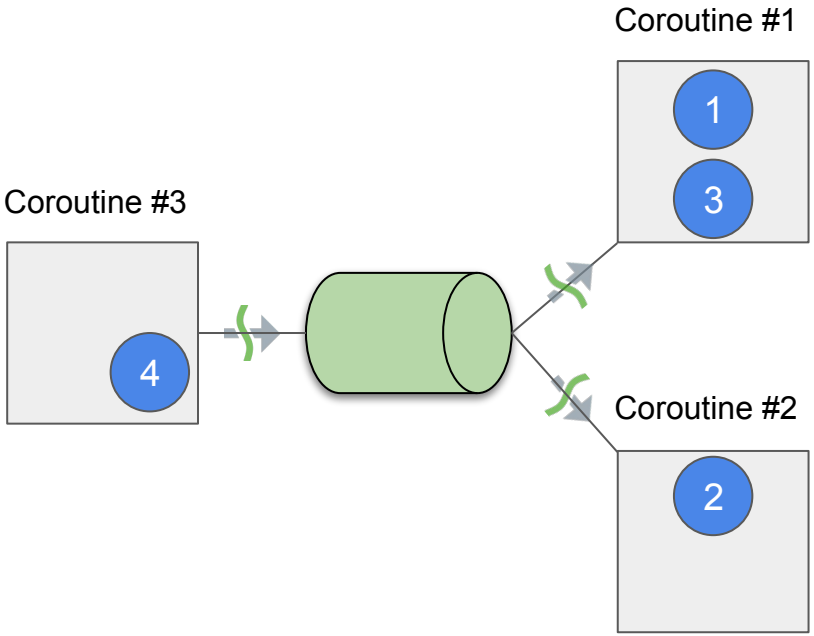
Simple Channel Demo

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

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4).forEach {
        channel.send(it)
    }
}
```



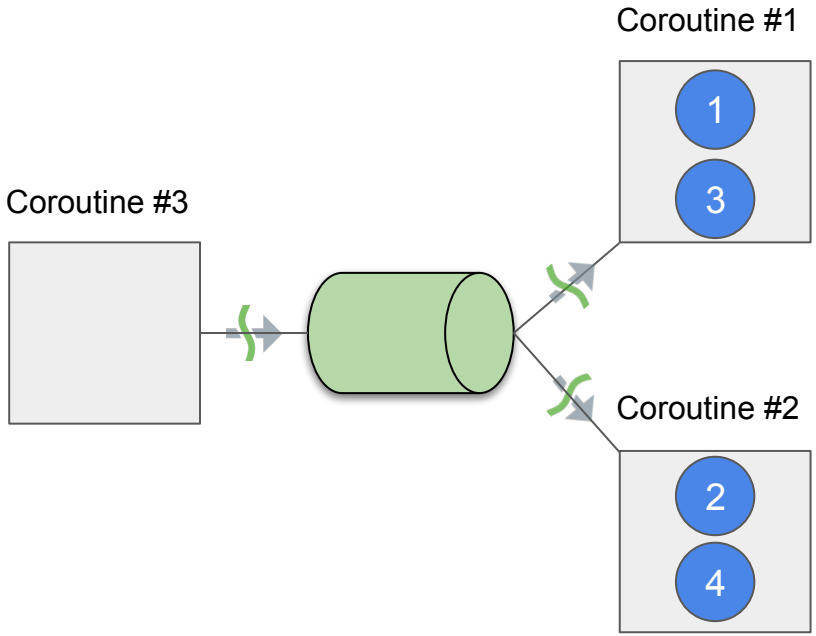
Simple Channel Demo

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

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4).forEach {
        channel.send(it)
    }
}
```



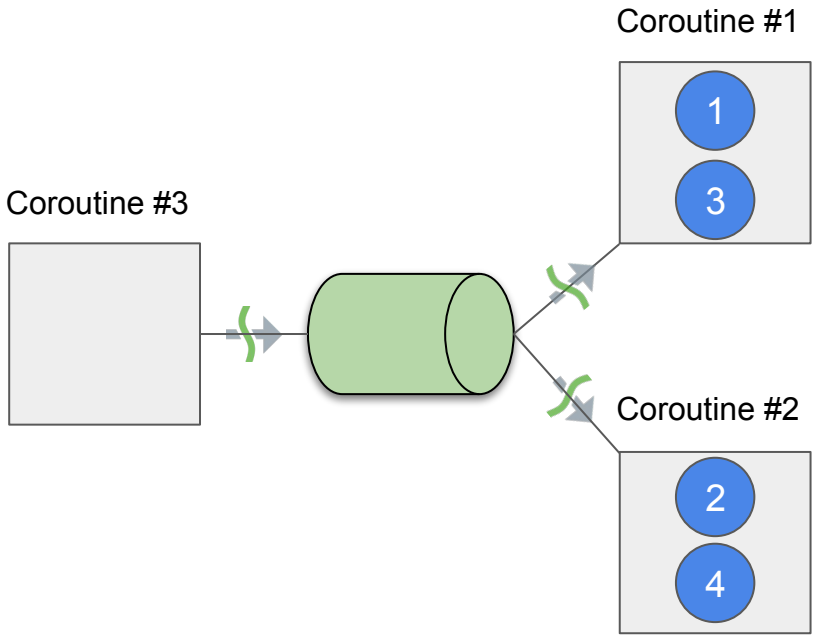
Simple Channel Demo

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

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4).forEach {
        channel.send(it)
    }
}
```



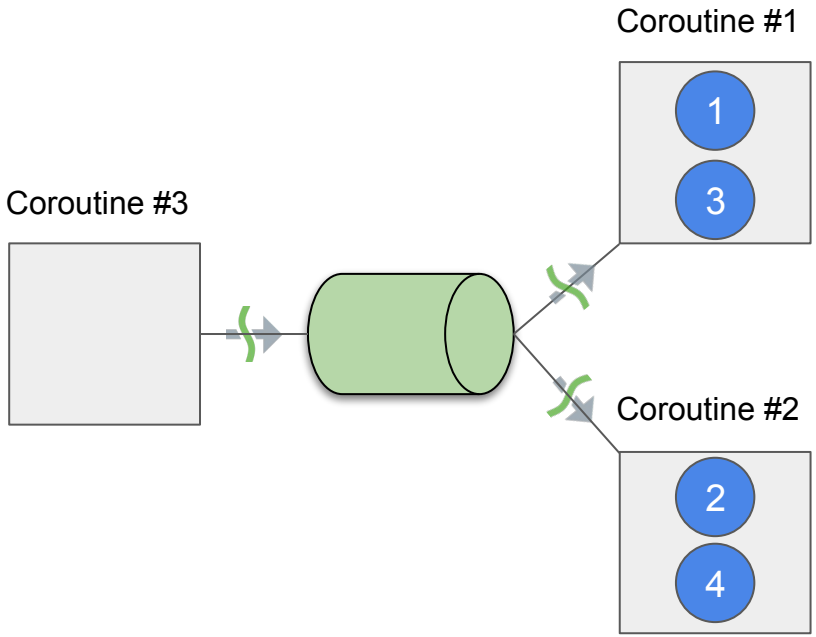
Simple Channel Demo

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

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

launch { // Coroutine 3
    listOf(1,2,3,4).forEach {
        channel.send(it)
    }
    channel.close()
}
```



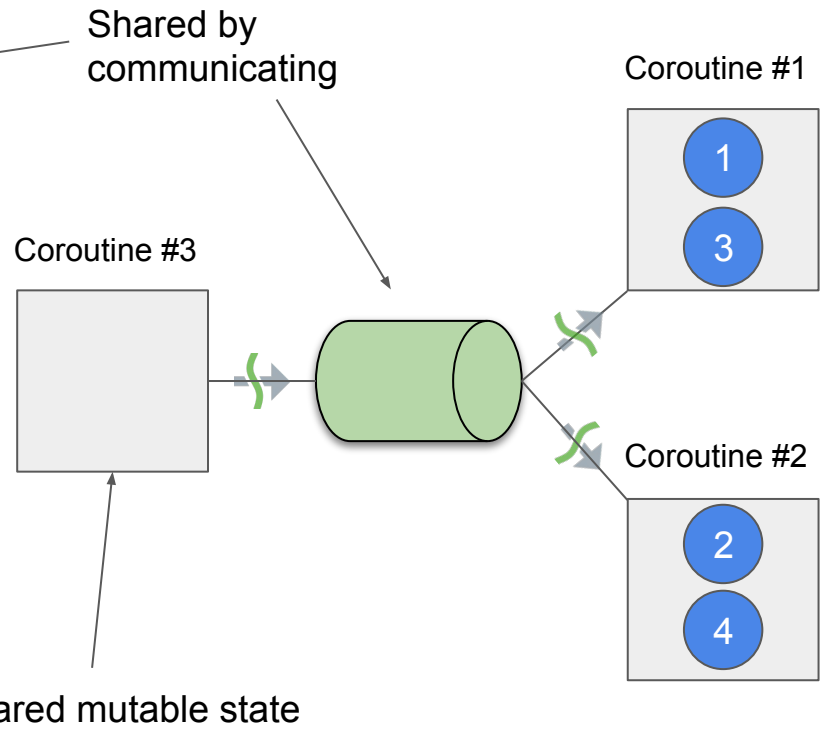
Simple Channel Demo

```
var channel: ReceiveChannel<Int>

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

channel = produce<Int> { // Coroutine 3
    listOf(1,2,3,4).forEach {
        send(it)
    }
}
```



Take away

```
var channel: ReceiveChannel<Int>

launch { // Coroutine 1
    for (it in channel) { //... }
}

launch { // Coroutine 2
    for (it in channel) { //... }
}

channel = produce<Int> { // Coroutine 3
    listOf(1,2,3,4).forEach {
        send(it)
    }
}
```



Take away

Channels behave like a non-blocking **Queue**

Channels are a **Synchronization Primitives**

Github Issue #254

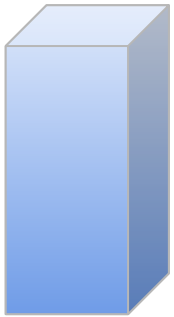
Channels are hot!

Needed an abstraction for cold streams that was lazy and computed data in a "push" mode.

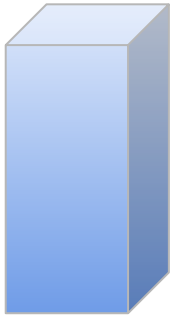
Flow

Simplest Flow Demo

```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}
```

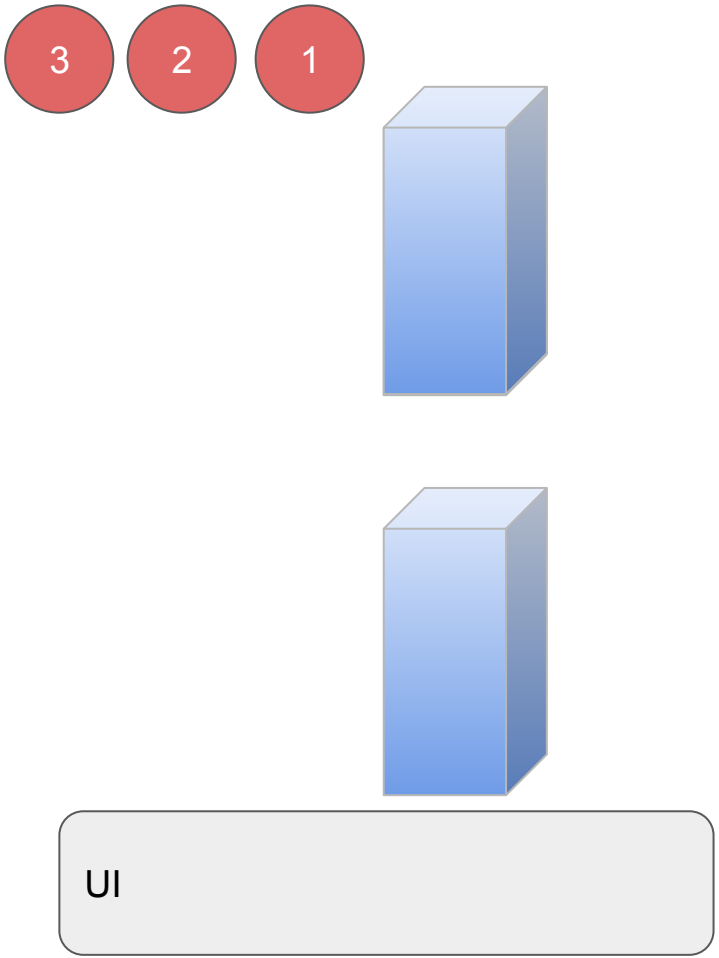


```
launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```

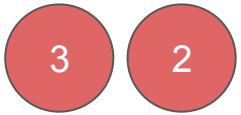


Simplest Flow Demo

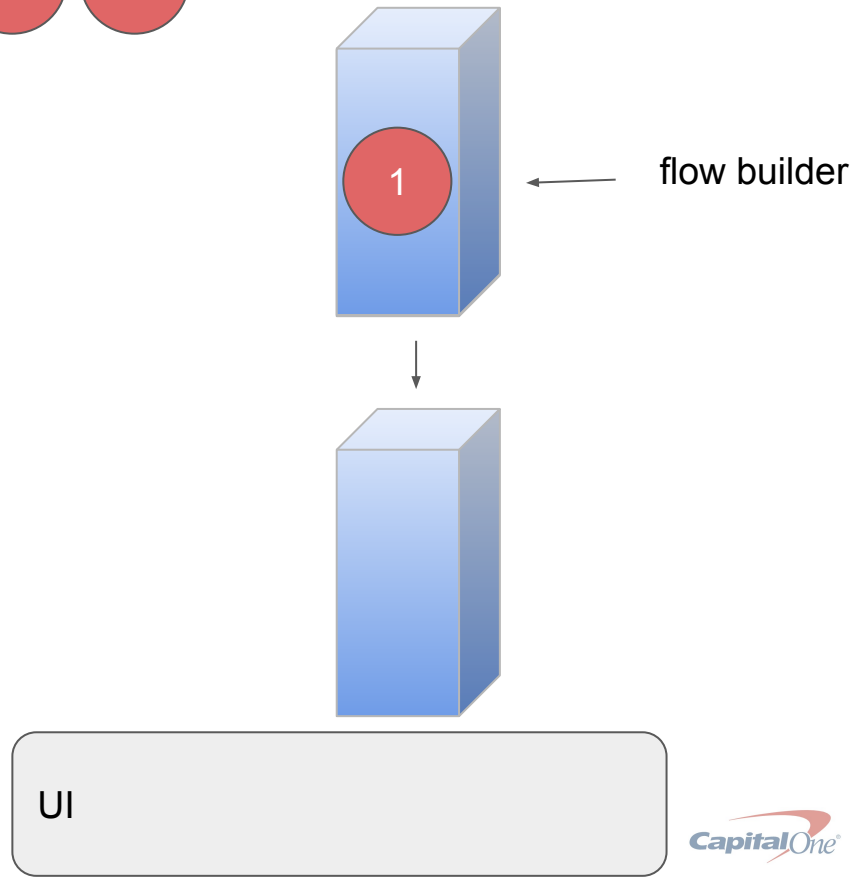
```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}

launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```



Simplest Flow Demo




```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}


launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```

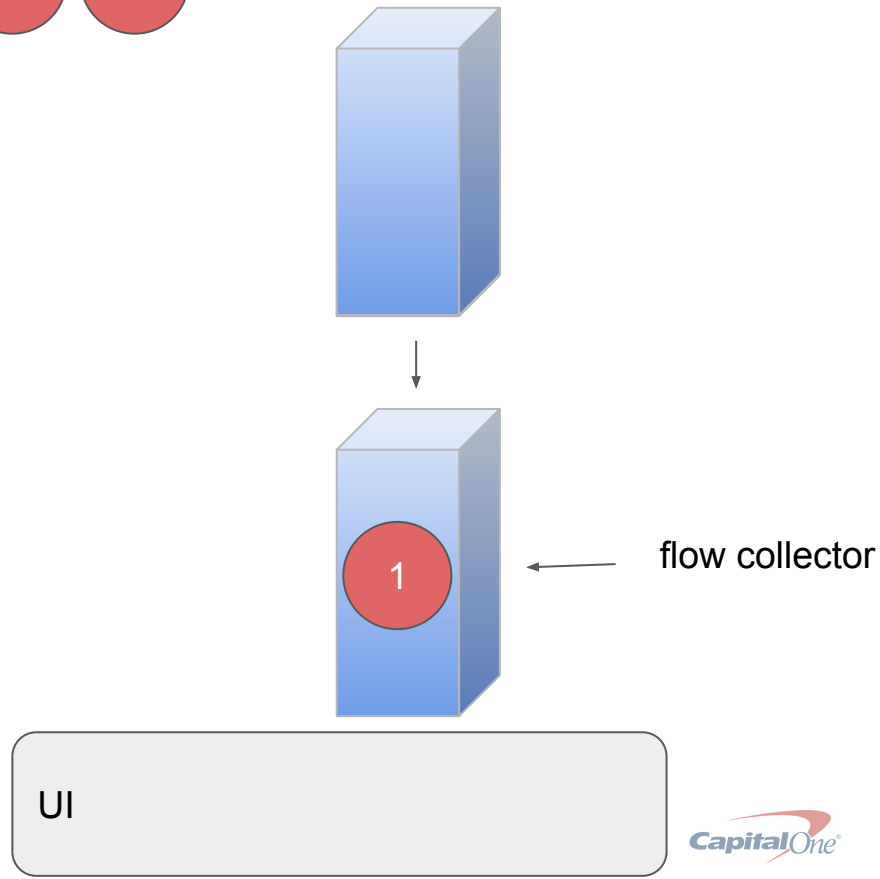


Simplest Flow Demo

3 2

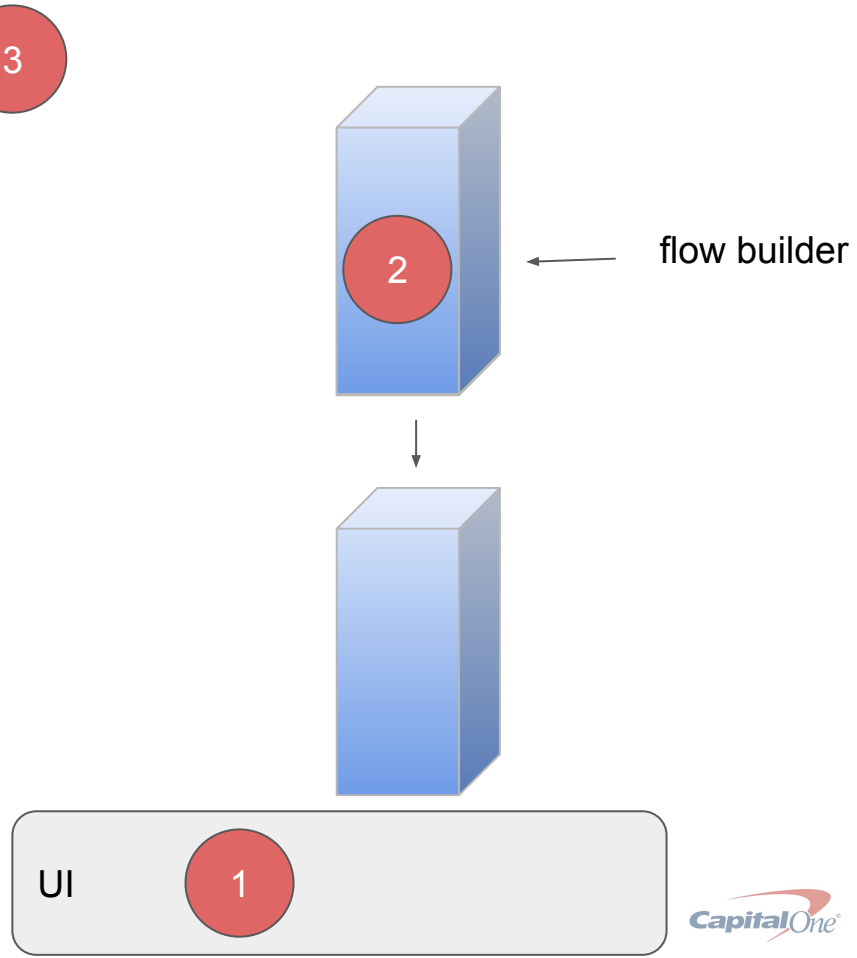
```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}
```

```
launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```



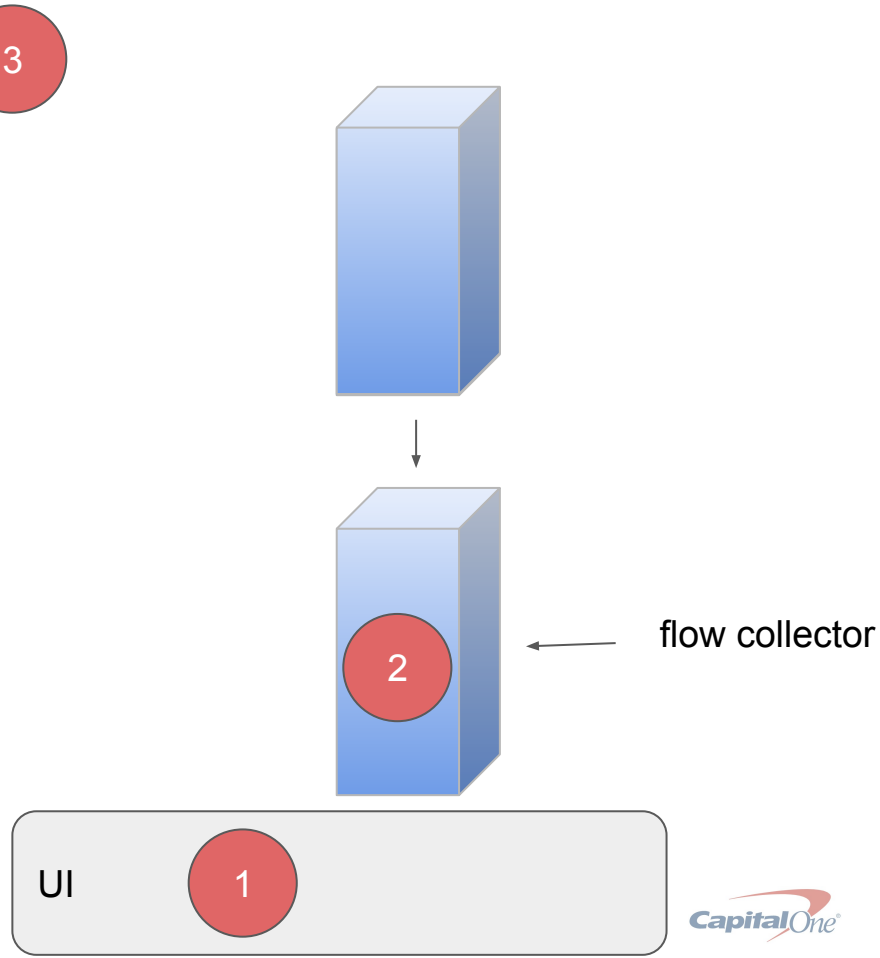
Simplest Flow Demo

```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}


launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```



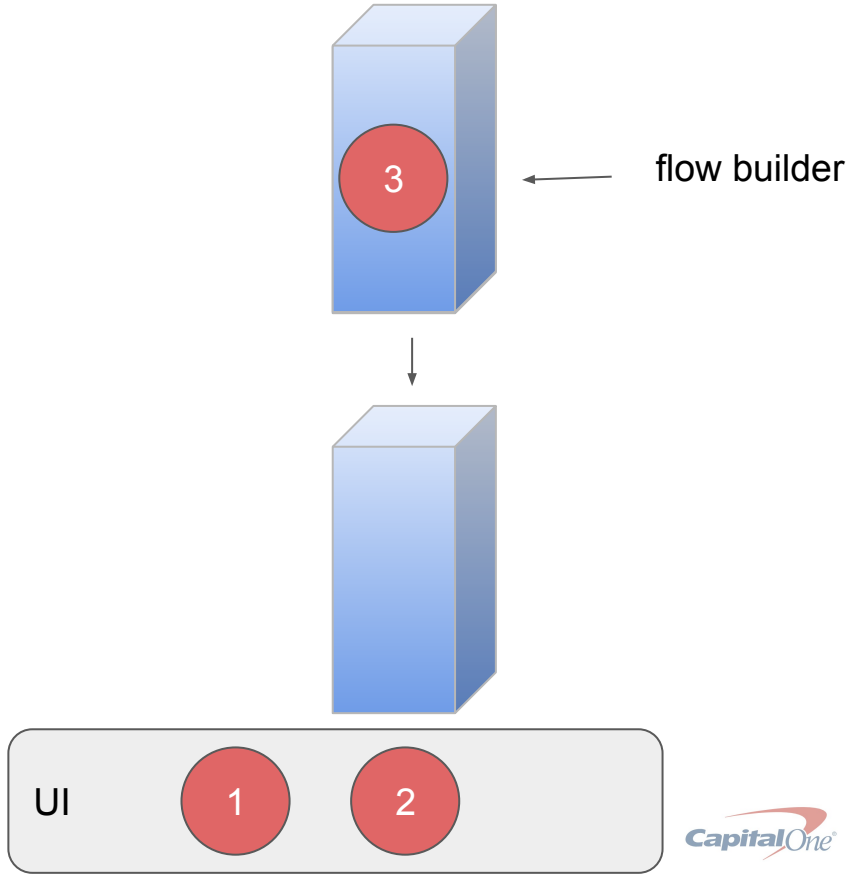
Simplest Flow Demo

```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}


launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```



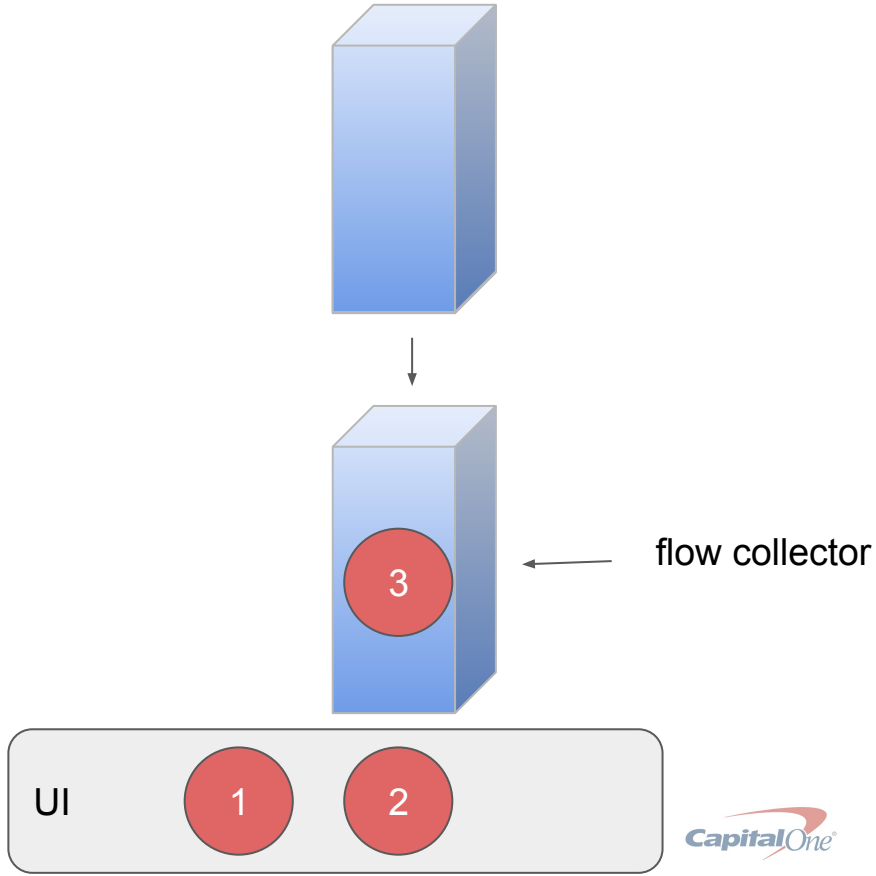
Simplest Flow Demo

```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}


launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```



Simplest Flow Demo

```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}


launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```

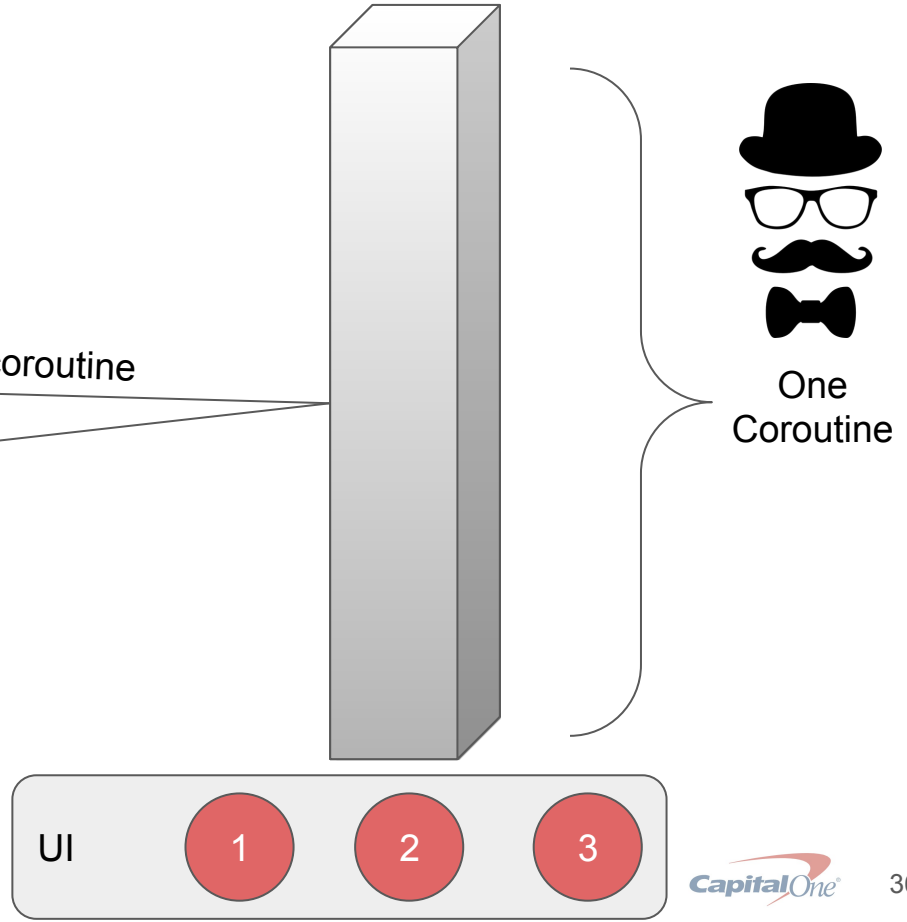


Simplest Flow Demo

```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach { number ->
        emit(number)
    }
}

launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```

Ends the flow and coroutine

The only coroutine



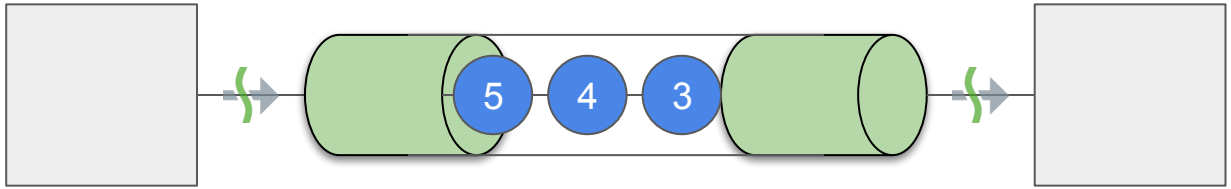
Flow

- One Coroutine

Buffered Flow

```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach {
        emit(number)
    }
}


launch {
    numbers
        .buffer()
        .collect { number ->
            updateUI(number)
    }
}
```

Buffered Channel



Flow

- One Coroutine
- Buffered Channel

```
val numbers: Flow<Int> = flow {
    listOf(1,2,3).forEach {
        emit(number)
    }
}
```

Anything can be a flow

```
listOf(1,2,3).asFlow()

flowOf(1, 2, 3)

((T) -> Int).asFlow()

(suspend (T) -> Int).asFlow()

LiveData<T>.asFlow()
```

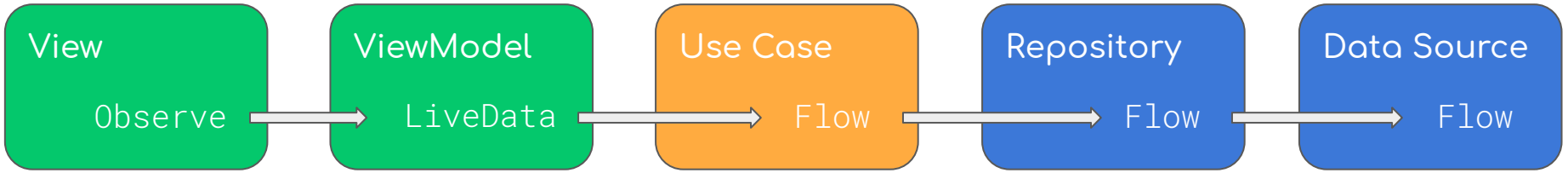
```
launch {
    numbers.collect { number ->
        updateUI(number)
    }
}
```

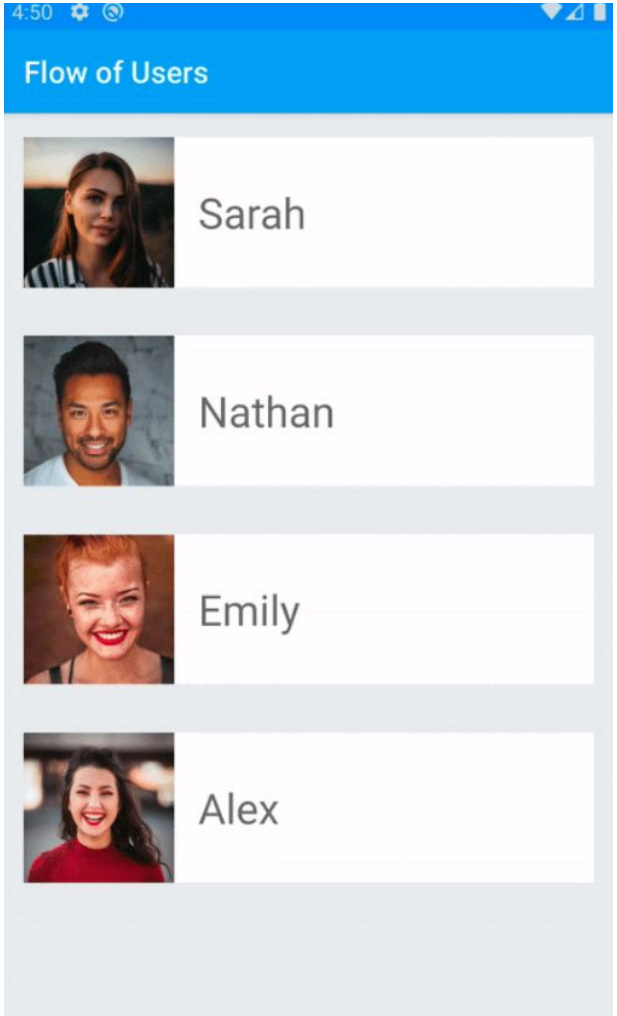
Flow

- One Coroutine
- Buffered Channel
… and so much more!

Flows and Channels Android Demo

Android Recommended Architecture





Example App

```
data class User(
    val name: String,
    var photo: Photo
)

data class Photo(
    var name: String,
    var image: Drawable?
)
```

Data Source

Flow

Room

Retrofit

```
class DataSource {
    companion object {
        val names: Flow<String>
            = listOf("Sarah", "Nathan", "Emily", "Alex")
                .asFlow()
    }
}
```

Repository

Flow

Common Operations

```
class PhotoRepository(names: Flow<String>) {

        val flowOfPhotos: Flow<Photo>

                = names

                    .map { it.getPhoto() }

                    .onEach { delay(1000L) }

                    .flowOn(Dispatchers.Default)
}
```

Business Logic

```
class UseCase(repository: PhotoRepository, names: Flow<String>) {

    val userFlow: Flow<User>
        = names
            .zip( repository.flowOfPhotos ) { name, photo ->
                User(name, photo)
            }
}
```

ViewModel

LiveData

```
class FlowViewModel(useCase: UseCase) : ViewModel() {

    val userLiveData: LiveData<User> =

            useCase.userFlow.asLiveData()

}
```

View

Observe

Observing ViewModel LiveData

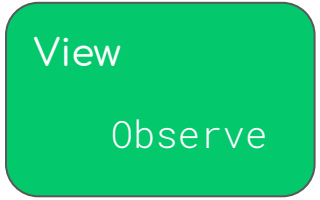
```
override fun onCreate(savedInstanceState: Bundle?) {
    // setup repository, useCase, viewModel...
    // setup listView and listAdapter...

    viewModel.userLiveData.observe(this, Observer { user ->
        adapter.addItem(user)
    })

}
```

View

Observe

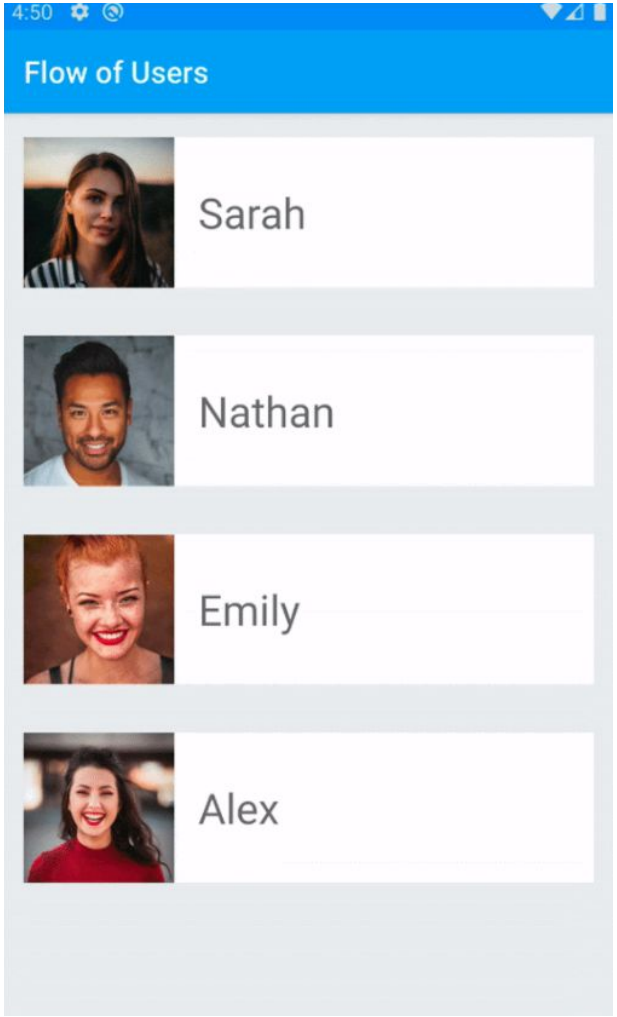
Flow is Reactive!!!

```
override fun onCreate(savedInstanceState: Bundle?) {
    // setup repository, useCase, viewModel...
    // setup listView and listAdapter...
    lifecycleScope.launchWhenCreated {
        viewModel.userFlow.collect { user ->
            adapter.addItem(user)
        }
    }
}
```



Flow is Reactive!!!

```
override fun onCreate(savedInstanceState: Bundle?) {
    // setup repository, useCase, viewModel...
    // setup listView and listAdapter...


    viewModel.userFlow
        .onEach { adapter.addItem(it) }
        .launchIn(lifecycleScope)

}
```



Example App

Thank you!

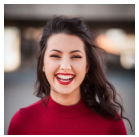
Photo Credits:

Štefan Štefančík
@cikstefan

Gabriel Silvério
@gabrielsilverio

Karen Arnold

Joseph Gonzalez
@miracletwentyone



Michael Dam
@michaeldam

Appendix

Exception handling in Flows

```
val numbers = (1..3).asFlow().map {
    check(it == 4) { "Must be 4" }
    it
}
```

```
try {
    numbers.collect()
} catch (e: Throwable) {
    print(e)
}
```

```
java.lang.IllegalStateException:
Must be 4
```

Exception handling in Flows

```
val numbers = (1..3).asFlow().map {
    check(it == 4) { "Must be 4" }
    it
}
```

```
numbers
    .catch { print(it) }
    .collect()
```

```
java.lang.IllegalStateException:
Must be 4
```