

Concurrencia en Kotlin.



Kotlin session 2

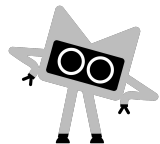


Contenido

- ¿Qué son las coroutines?
- Funciones suspend
- Coroutine Context y Dispatchers
- Builders y Jobs
- Scopes
- Exceptions
- Secuencias
- Flows
- StateFlow
- Share Flow
- Channel
- Callback Flow



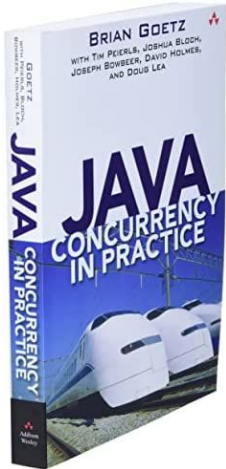
Técnicas de programación asíncrona



Durante décadas, como desarrolladores, nos enfrentamos a un problema que resolver: cómo evitar que nuestras aplicaciones se bloqueen. Ya sea que estemos desarrollando aplicaciones de escritorio, móviles o incluso del lado del servidor, queremos evitar que el usuario espere o, lo que es peor, cause cuellos de botella que impiden que una aplicación se escale.

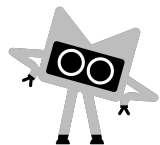
Ha habido muchos enfoques para resolver este problema, incluyendo:

- Threading
- Callbacks
- Futures, promises, and others
- Reactive Extensions
- Coroutines





¿Qué son las coroutines?



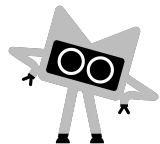
La programación asíncrona o sin bloqueo es una parte importante del panorama de desarrollo. Al crear aplicaciones móviles, de escritorio o del lado del servidor, es importante proporcionar una experiencia que no solo sea fluida desde la perspectiva del usuario, sino también escalable cuando sea necesario.

Kotlin resuelve este problema de manera flexible al brindar soporte de [Corrutinas](#) a nivel de lenguaje y delegar la mayor parte de la funcionalidad a las bibliotecas.

Además de abrir las puertas a la programación asíncrona, las corrutinas también brindan una gran cantidad de otras posibilidades, como la concurrencia y los actores.



¿Qué son las coroutines? (2)

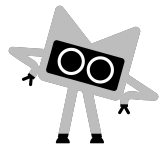


Una *corrutina* es una instancia de computación suspendible. Es conceptualmente similar a un subproceso, en el sentido de que necesita un bloque de código para ejecutarse que funciona simultáneamente con el resto del código. Sin embargo, una *corutina* no está vinculada a ningún hilo en particular. Puede suspender su ejecución en un hilo y reanudar en otro.

Las corrutinas pueden considerarse subprocesos livianos, pero hay una serie de diferencias importantes que hacen que su uso en la vida real sea muy diferente al de los subprocesos.



¿Qué son las coroutines? (3)

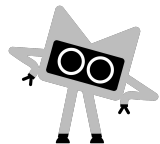


```
FirstCoroutine.kt

1 import kotlinx.coroutines.*
2
3 fun main() = runBlocking { // this: CoroutineScope
4     launch { // launch a new coroutine and continue
5         // non-blocking delay for 1 second (default time unit is ms)
6         delay(1000L)
7         println("World!") // print after delay
8     }
9     // main coroutine continues while a previous one is delayed
10    println("Hello")
11 }
12
```



Las Coroutine son livianas

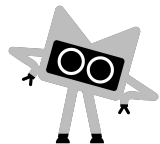


Las corrutinas consumen menos recursos que los subprocesos de JVM. El código que agota la memoria disponible de la JVM cuando se usan subprocesos se puede expresar mediante corrutinas sin alcanzar los límites de recursos.





Funciones suspend



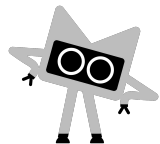
```
SuspendFunction.kt

1 import kotlinx.coroutines.*
2
3 fun main() = runBlocking { // this: CoroutineScope
4     launch { doWorld() }
5     println("Hello")
6 }
7
8 // this is your first suspending function
9 suspend fun doWorld() {
10     delay(1000L)
11     println("World!")
12 }
13
```





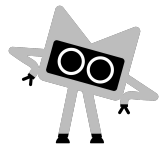
Funciones suspend (2)



Las funciones de suspensión se pueden usar dentro de rutinas como las funciones regulares, pero su característica adicional es que pueden, a su vez, usar otras funciones de suspensión (como `delay` en el ejemplo anterior) para suspender la ejecución de una corutina.



Secuencial por defecto



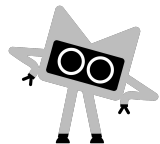
```
Composition.kt

1 suspend fun doSomethingUsefulOne(): Int {
2     delay(1000L) // pretend we are doing something useful here
3     return 13
4 }
5
6 suspend fun doSomethingUsefulTwo(): Int {
7     delay(1000L) // pretend we are doing something useful here, too
8     return 29
9 }
10
```





Uso simultáneo asíncrono



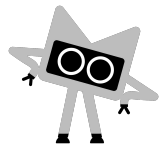
```
Composition.kt

1 import kotlinx.coroutines.*
2 import kotlin.system.*
3
4 fun main() = runBlocking<Unit> {
5
6     val time = measureTimeMillis {
7         val one = async { doSomethingUsefulOne() }
8         val two = async { doSomethingUsefulTwo() }
9         println("The answer is ${one.await() + two.await()}")
10    }
11    println("Completed in $time ms")
12
13 }
```





Coroutine Context y Dispatchers



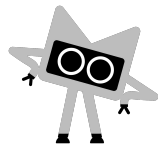
Las Coroutine siempre se ejecutan en algún contexto representado por un valor del tipo `CoroutineContext`, definido en la biblioteca estándar de Kotlin.

El contexto coroutine es un conjunto de varios elementos. Los elementos principales son el [Job](#) de la coroutine, su dispatcher.





Dispatchers & Threads



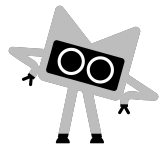
El contexto de coroutine incluye un **dispatcher** *de coroutine* ([CoroutineDispatcher](#)) que determina qué subproceso o subprocesos que utiliza la coroutine para su ejecución. El **dispatcher** de coroutine puede limitar la ejecución de coroutine a un subproceso específico, enviarlo a un grupo de subprocesos o dejar que se ejecute sin restricciones.

Todos los constructores de coroutine como [launch](#) y [async](#) aceptan un parámetro [CoroutineContext](#) opcional que se puede usar para especificar explícitamente el despachador para el nuevo coroutine y otros elementos de contexto.





Dispatchers & Threads (2)



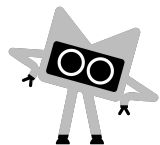
`Dispatchers.Unconfined` es un despachador especial que también parece ejecutarse en el subproceso main.

El despachador predeterminado se usa cuando no se especifica explícitamente a ningún otro despachador en el ámbito. Está representado por `Dispatchers.Default` y utiliza un grupo de subprocesos de fondo compartido.

`newSingleThreadContext` crea un hilo para que se ejecute la rutina. Un hilo dedicado es un recurso muy costoso. En una aplicación real, debe liberarse, cuando ya no se necesite, utilizando la función `de cierre`, o almacenarse en una variable de nivel superior y reutilizarse en toda la aplicación.



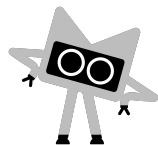
Dispatchers no confinado vs confinado



El despachador de rutina `Dispatchers.Unconfined` inicia una rutina en el subproceso desde donde se lo llama, pero solo hasta el primer punto de suspensión. Después de la suspensión, reanuda la rutina en el subproceso que está completamente determinado por la función de suspensión que se invocó. **El dispatcher no confinado es apropiado para corrutinas que no consumen tiempo de CPU ni actualizan datos compartidos (como la interfaz de usuario) confinados a un subproceso específico.**

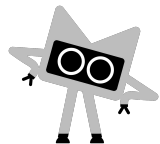
Por otro lado, el despachador se hereda del `CoroutineScope` externo de forma predeterminada. El despachador predeterminado para la corrutina `runBlocking`, en particular, se limita al subproceso del invocador, por lo que heredarlo tiene el efecto de limitar la ejecución a este subproceso con una programación FIFO predecible.

Builders y Jobs





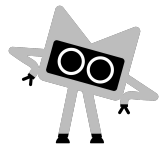
Async



Conceptualmente, `async` es como `launch` . Inicia una corrutina separada que es un subproceso liviano que funciona simultáneamente con todas las demás corrutinas. La diferencia es que `launch` devuelve un `job` y no tiene ningún valor resultante, mientras que `async` devuelve un `Deferred` , un futuro ligero sin bloqueo que representa una promesa de proporcionar un resultado más adelante. Puede usar `.await()` un valor diferido para obtener su resultado final, pero `Deferred` también es un `Job`, por lo que puede cancelarlo si es necesario.



Async (2) - Lazy mode



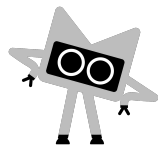
Opcionalmente, `async` se puede hacer perezoso configurando su start parámetro en `CoroutineStart.LAZY`. En este modo, solo inicia la corrutina cuando `await` requiere su resultado, o si se invoca su función Job de inicio.

Tenga en cuenta que si solo llamamos `await` sin `println` llamar primero a `start` en rutinas individuales, esto conducirá a un comportamiento secuencial, ya que `await` inicia la ejecución de la rutina y espera a que finalice, que no es el caso de uso previsto para la pereza. El caso de uso para **`async(start = CoroutineStart.LAZY)`** es un reemplazo de la lazy función estándar en los casos en que el cálculo del valor implica la suspensión de funciones.





Scopes



Define un ámbito para nuevas rutinas. Cada **constructor de rutinas** (como `launch` , `async` , etc.) es una extensión de `CoroutineScope` y hereda su `coroutineContext` para propagar automáticamente todos sus elementos y cancelarlos.

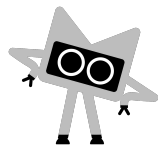
Las mejores formas de obtener una instancia independiente del alcance son las funciones **`CoroutineScope()`** y **`MainScope()`**, teniendo cuidado de cancelar estos alcances de corrutina cuando ya no sean necesarios.

Se pueden agregar elementos de contexto adicionales al alcance usando el operador `más` .



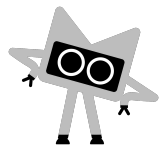


Convención para la concurrencia estructurada



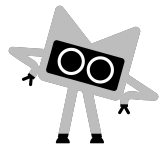
Cada coroutine builder (como `launch` , `async` y otros) y cada función de alcance (como `coroutineScope` y `withContext`) proporciona *su propio* alcance con su propia instancia `Job` en el bloque interno de código que ejecuta. Por convención, todos esperan a que se completen todas las corrutinas dentro de su bloque antes de completarse ellos mismos, lo que impone la concurrencia estructurada.

Charlas recomendadas





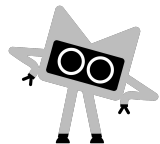
Exception Handling



Sabemos que una corrutina cancelada lanza `CancellationException` en los puntos de suspensión y que es ignorada por la maquinaria de las corrutinas. Aquí vemos lo que sucede si se lanza una excepción durante la cancelación o si varios elementos secundarios de la misma rutina lanzan una excepción.



Propagación de excepciones

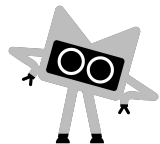


Los coroutine builders vienen en dos sabores: **propagar excepciones automáticamente** ([launch](#) y [actor](#)) o **exponerlas a los usuarios** ([async](#) y [produce](#)). Cuando estos constructores se utilizan para crear una corrutina *raíz* , que no es un *hijo* de otra corrutina, los primeros constructores tratan las excepciones como excepciones **no detectadas** , similar a las de Java `Thread.uncaughtExceptionHandler`, mientras que los últimos confían en que el usuario consuma la excepción final, por ejemplo a través de [await](#) o [receive](#) ([produce](#) y [receive](#) son parte de los [Channels](#)).





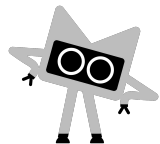
CoroutineExceptionHandler



Es posible personalizar el comportamiento predeterminado de imprimir excepciones **no detectadas**. El elemento de contexto `CoroutineExceptionHandler` en una corrutina *root* que se puede usar como un catch, bloque genérico para esta corrutina root y todos sus elementos secundarios donde puede tener lugar el manejo personalizado de excepciones. Es similar a `Thread.uncaughtExceptionHandler`. No puede recuperarse de la excepción en el archivo `CoroutineExceptionHandler`. La rutina ya se había completado con la excepción correspondiente cuando se llama al controlador. Normalmente, el controlador se usa para registrar la excepción, mostrar algún tipo de mensaje de error, finalizar y/o reiniciar la aplicación.



CoroutineExceptionHandler(2)

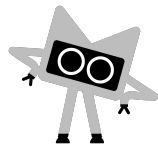


CoroutineExceptionHandler se invoca sólo en excepciones **no detectadas**, excepciones que no se manejaron de ninguna otra manera. En particular, todas las *corrutinas secundarias* (*corrutinas creadas en el contexto de otro [Job](#)*) delegan el manejo de sus excepciones a su corrutina principal, que también delega a la principal, y así sucesivamente hasta la raíz, por lo que CoroutineExceptionHandler nunca se usa la instalada en su contexto. Además de eso, el generador [async](#) siempre detecta todas las excepciones y las representa en el objeto [Deferred](#) resultante , por lo que CoroutineExceptionHandler tampoco tiene ningún efecto.





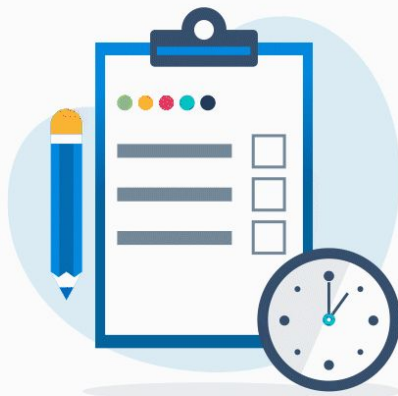
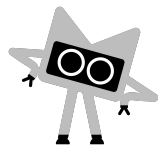
Cancelación y excepciones



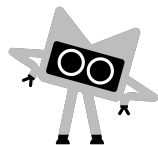
La cancelación está íntimamente relacionada con las excepciones. Las corrutinas se usan internamente **CancellationException** para la cancelación, estas excepciones son ignoradas por todos los controladores, por lo que deben usarse sólo como fuente de información de depuración adicional, que se puede obtener en el bloque catch. Cuando se cancela una corrutina usando `Job.cancel`, termina, pero no cancela su padre.



Supervisión




Charlas recomendadas




 KotlinConf'19

**Coroutines!
Gotta catch
'em all!**

Florina Muntenescu
Manuel Vivo



 **Cancelations
&
Exceptions**

 KotlinConf'19

**Error handling
strategies for
Kotlin programs**

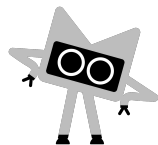
Nat Pryce
Duncan McGregor



KOTLIN



Secuencias



[Tipo de secuencia](#) que representa colecciones evaluadas de forma diferida. Funciones de nivel superior para crear instancias de secuencias y funciones de extensión para secuencias.

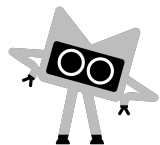
Las secuencias se diferencian del resto de colecciones en que, en vez de contener una serie de objetos ya disponibles desde el principio, esos objetos no se calculan hasta que no llega el momento de ser utilizados.

```
val seqOfElements = sequenceOf("first", "second", "third")
```





Secuencias: cuándo usarlas

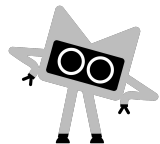


Como comentaba, la ventaja de una secuencia es que no tiene desde el primer minuto todos los objetos calculados, sino que se van creando según se necesitan. Esto nos trae dos ventajas:

- **Las secuencias pueden ser infinitas:** podemos definir una secuencia mediante un valor inicial y una operación (por ejemplo), y esa secuencia tendrá infinitos valores
- **Permiten evitar pasos intermedios:** a diferencia del resto de colecciones, cuando una secuencia realiza varias operaciones (de filtrado, transformación, etc), estas se aplican en cadena en los objetos uno a uno, en vez de necesitar crear una colección nueva a cada paso.



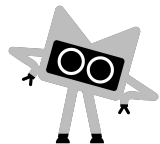
Operaciones: Stateless y Stateful



- **Stateless:** no necesitan ningún estado intermedio para procesarse, o una cantidad muy pequeña y constante. Algunas operaciones stateless son `map()`, `filter()`, `take()` o `drop()`
- **Stateful:** operaciones que requieren una gran cantidad de estado, normalmente proporcional al número de elementos de la secuencia. Algunos ejemplos serían todas las variantes de `sorted()`, `distinct()` o `chunked()`



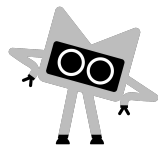
Operaciones: Intermediate y Terminal



- **Intermediate:** al aplicarlo, el resultado devuelto es otra secuencia, y por tanto no necesita aún calcular el resultado a partir de los valores de la secuencia. Todas las operaciones de las que hablábamos antes son intermedias.
- **Terminal:** necesita los valores de la secuencia, y por tanto, va a procesar toda la secuencia para obtener el resultado. Algunos ejemplos serían `toList()`, que devuelve una lista concreta, o `sum()` que calcula la suma de los valores de la secuencia.

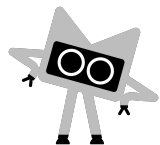


Flows



Un *flow* es un tipo que puede emitir varios valores de manera secuencial, en lugar de *suspender funciones* que muestran solo un valor único. Un flujo se puede usar, por ejemplo, para recibir actualizaciones en vivo de una base de datos.

Los flows se ejecutan sobre las corrutinas y pueden proporcionar varios valores. Un flujo es conceptualmente una *transmisión de datos* que se puede computar de forma asíncrona. Los valores emitidos deben ser del mismo tipo. Por ejemplo, un **Flow<Int>** es un flujo que emite valores enteros.

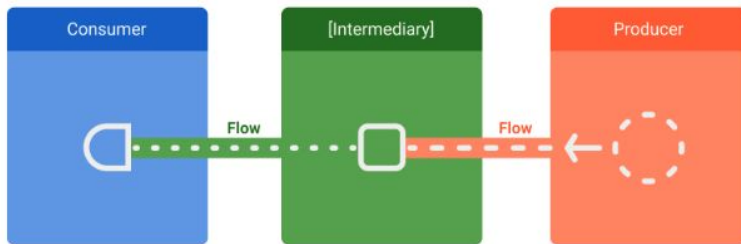


Flows (2)

Un flujo es muy similar a un Iterator que produce una secuencia de valores, pero usa funciones de suspensión para producir y consumir valores de forma asíncrona. Esto significa que, por ejemplo, el flujo puede enviar de forma segura una solicitud de red para producir el siguiente valor sin bloquear el subproceso principal.

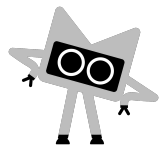
Hay tres entidades involucradas en transmisiones de datos:

- Un **productor** produce datos que se agregan al flujo. Gracias a las corrutinas, los flujos también pueden producir datos de forma asíncrona.
- Los **intermediarios (opcional)** pueden modificar cada valor emitido en el flujo, o bien el flujo mismo.
- Un **consumidor** consume los valores del flujo.





StateFlow



[StateFlow](#) es un flujo observable contenedor de estados que emite actualizaciones de estado actuales y nuevas a sus recopiladores. El valor de estado actual también se puede leer a través de su propiedad [value](#). Para actualizar el estado y enviarlo al flujo, asigna un nuevo valor a la propiedad [value](#) de la clase [MutableStateFlow](#).

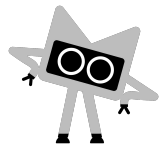
La clase responsable de **actualizar un MutableStateFlow es el productor**, mientras que todas las clases que se **recopilan de StateFlow son consumidores**. A diferencia de un flujo *frío* compilado con el compilador de flow, un StateFlow es *caliente*; recopilar datos del flujo no activa ningún código de productor. Un objeto StateFlow siempre se encuentra activo y en la memoria, y se vuelve apto para la recolección de elementos no utilizados solo cuando no hay otras referencias a él en la raíz de otra recolección.

Cuando un consumidor nuevo comienza a recopilarse desde el flujo, recibe el último estado del flujo y todos los estados posteriores. Puedes encontrar este comportamiento en otras clases observables, como [LiveData](#).





Cómo convertir flujos fríos en calientes con `shareIn`



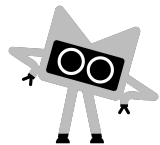
StateFlow es un flujo *caliente*, es decir, permanece en la memoria siempre que se recopile o mientras que una raíz de recolección de elementos no utilizados origine alguna referencia a él. Puedes usar el operador [`shareIn`](#) para cambiar los flujos fríos a calientes.

Si usas el callbackFlow creado en [`flows de Kotlin`](#) como ejemplo, en lugar de hacer que cada recopilador cree un flujo nuevo, puedes compartir los datos recuperados de Firestore entre recopiladores con **`shareIn`**. Debes pasar lo siguiente:

- Un **CoroutineScope** que se use para compartir el flujo (este alcance debe durar más que cualquier consumidor para mantener el flujo compartido activo el tiempo que sea necesario)
- La cantidad de elementos que se volverán a reproducir en cada recopilador nuevo
- La política de comportamiento de inicio.



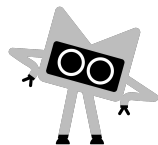
Cómo convertir flujos fríos en calientes con shareIn



```
1 class NewsRemoteDataSource(...,  
2     private val externalScope: CoroutineScope,  
3 ) {  
4     val latestNews: Flow<List<ArticleHeadline>> = flow {  
5         ...  
6     }.shareIn(  
7         externalScope,  
8         replay = 1,  
9         started = SharingStarted.WhileSubscribed()  
10    )  
11 }
```



Share Flow

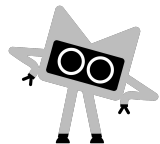


La función **shareIn** muestra un **SharedFlow**, un flujo caliente que emite valores para todos los consumidores que recopilan datos de él. Un **SharedFlow** es una generalización que admite una amplia configuración de **StateFlow**.

Puedes crear un **SharedFlow** sin usar **shareIn**. Como ejemplo, puedes usar un **SharedFlow** para enviar marcas al resto de la app, de modo que todo el contenido se actualice simultáneamente y de manera periódica. Al igual que con **StateFlow**, usa una propiedad de copia de seguridad de tipo **MutableSharedFlow** en una clase para enviar elementos al flujo.



Share Flow (2)



Puedes personalizar el comportamiento de `SharedFlow` de las siguientes maneras:

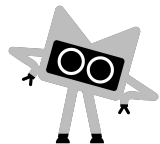
- **replay** te permite volver a enviar una serie de valores previamente emitidos para suscriptores nuevos.
- **onBufferOverflow** te permite especificar una política para las ocasiones en que el búfer está lleno de elementos que se enviarán. El valor predeterminado es **BufferOverflow.SUSPEND**, lo que provoca la suspensión del emisor. Otras opciones son **DROP_LATEST** o **DROP_OLDEST**.

MutableSharedFlow también tiene una propiedad **subscriptionCount**, que contiene la cantidad de recopiladores activos para que puedas optimizar tu lógica empresarial según corresponda. **MutableSharedFlow** también contiene una función **resetReplayCache** en caso de que no desees volver a reproducir la información más reciente enviada al flujo.





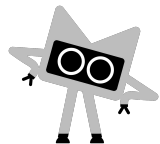
Channel



Un [canal](#) es conceptualmente muy similar a BlockingQueue. Una diferencia clave es que en lugar de una put operación de bloqueo, tiene un [send](#) de suspensión, y en lugar de una take operación de bloqueo, tiene una [receive](#) de suspensión .



Cierre e iteración sobre Channels

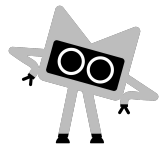


A diferencia de una cola, un canal se puede cerrar para indicar que no llegan más elementos. En el lado del receptor, es conveniente usar un `for` bucle regular para recibir elementos del canal.

Conceptualmente, un **cierre** es como enviar un token de cierre especial al canal. La iteración se detiene tan pronto como se recibe este token de cierre, por lo que hay una garantía de que se reciben todos los elementos enviados previamente antes del cierre:



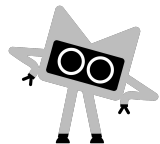
Callback Flow



Es un tipo de Flow que nos permite convertir cualquier API basada en callbacks (o en Listeners, que en Android pasa mucho) en un Flow que podemos recolectar, transformar y usar como hemos visto en el resto de artículos sobre este tema.



Callback Flow (2)

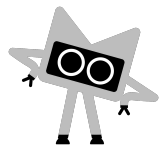


```
CallbackFlow.kt

1 val View.onClickEvents: Flow<View>
2     get() = callbackFlow {
3         val onClickListener = View.OnClickListener {
4             offer(it)
5         }
6         setOnClickListener(onClickListener)
7         awaitClose { setOnClickListener(null) }
8     }.conflate()
```



Callback Flow (3)



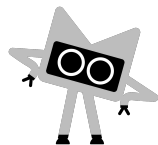
CallbackFlow.kt

```
1 val RecyclerView.lastVisibleEvents: Flow<Int>
2     get() = callbackFlow<Int> {
3         val lm = layoutManager as GridLayoutManager
4         val listener = object : RecyclerView.OnScrollListener() {
5             override fun onScrolled(recyclerView: RecyclerView, dx: Int, dy: Int) {
6                 offer(lm.findLastVisibleItemPosition())
7             }
8         }
9         addOnScrollListener(listener)
10        awaitClose { removeOnScrollListener(listener) }
11    }.conflate()
```





Charlas recomendadas



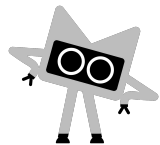
KotlinConf'19

Asynchronous Data Streams with Kotlin Flow

Roman Elizarov

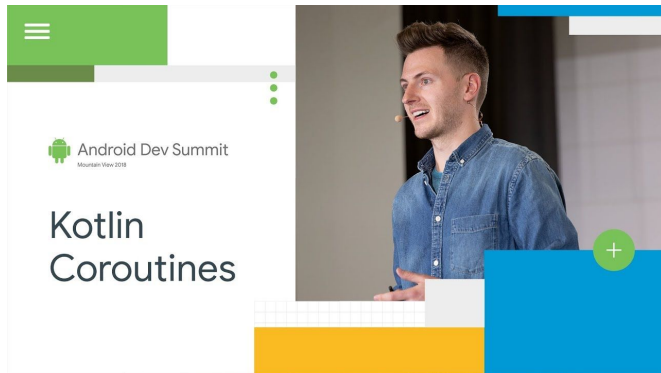


Charlas recomendadas - Android



Intermediate

Understand
Kotlin
Coroutines



Android conference
talks:

Kotlin Coroutines
101

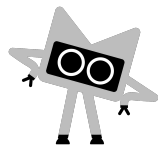


#AndroidDevSummit

LiveData
with
Coroutines
& Flow



Charlas recomendadas, si uso RxJava



KotlinConf'19

Migrating a library from RxJava To Coroutines

Mike Nakhimovich
Yiğit Boyar



Gracias

