Tema 3.7 - Corrutinas

Descargar estos apuntes pdf o html

Índice

Introducción

- Documentación oficial: **Lenguaje Kotlin**
- Documentación oficial: **Android**
- Vídeo: **Android Developers**
- Vídeo: **Martin Kiperszmid**
- Lista de reproducción: **Stevdza-San (Inglés)**
- Lista de reproducción: **Philipp Lackner (Inglés)**

En primer lugar comentar que las corrutinas no son un concepto exclusivo de Kotlin, ya que existen en otros lenguajes como C#, Python, Go, JavaScript, etc. En Kotlin, las corrutinas son una característica del lenguaje que implementa un patrón que **nos permite escribir código asíncrono de manera secuencial**. Esto significa que podemos escribir **código asíncrono como si fuera código síncrono**, sin tener que preocuparnos por los callbacks, los hilos, etc. Esto hace que el código sea más fácil de leer y de mantener.

¿Quiere decir esto que no podamos gestionar la concurrencia como lo hacemos en Java?. No, podemos seguir usando hilos tal cual lo hacemos en Java a través del paquete `kotlin.concurrent`. Sin embargo, las corrutinas nos permiten gestionar la concurrencia de una manera más sencilla y segura ya no están vinculada a ningún hilo en particular. Puede suspender su ejecución en un hilo y reanudarse en otro.

Serán útiles para tener:

- **Ligereza**: Puedes ejecutar muchas corrutinas de forma concurrente en un solo hilo debido a la compatibilidad con la suspensión, que no bloquea el hilo en el que se ejecuta la corrutina.
- **Menos fugas de memoria**: Puedo ejecutar un proceso dentro de un determinado '**scope**' o alcance, y cuando el alcance se cierra, todas las corrutinas que se ejecutan dentro de ese alcance se cancelan automáticamente. Esto significa que no hay fugas de memoria.
- **Compatibilidad con cancelación incorporada**: Se propaga automáticamente la cancelación a través de la jerarquía de corrutinas en ejecución.
- **Integración con Jetpack**: Muchas bibliotecas de Jetpack incluida Compose. Estas bibliotecas incluyen extensiones que proporcionan compatibilidad total con corrutinas.

Las corrutinas forman parte de la librería estándar de Kotlin. No obstante para usarlas necesitamos añadir como dependencia la librería de extensión (`kotlinx.coroutines`) donde se implementan en el fichero `build.gradle.kts` del módulo app:

Podemos ver las versión actual de corrutinas en el siguiente **repositorio de Kotlin**. **Además, debemos fijarnos cual es la versíon de Kotlin mínima asociada a la versión de corrutinas que queremos usar**.

Por ejemplo si usáramos un **programa de consola** tendríamos que añadir:

En el `libs.versions.toml` ...

```
[versions]
coroutines = "1.9.0"

[libraries]
coroutines-core = {module = "org.jetbrains.kotlinx:kotlinx-coroutines-core", version.ref = "coroutines" }
```

En el DSL definido en `build.gradle.kt` del módulo app...

```
dependencies {
    implementation(libs.coroutines.core)
}
```

En un **proyecto Android** posiblemente no necesitemos añadirla porque ya estará incluida la librería a través de alguna otra dependencia. Pero si no fuera así, tendríamos que añadir mínimo la implementación específica para Android:

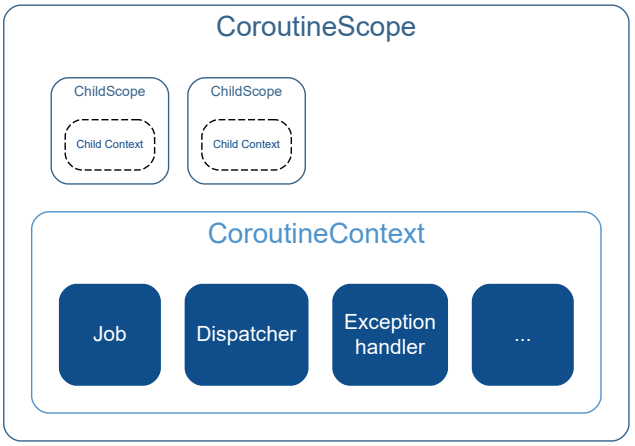
```
org.jetbrains.kotlinx:kotlinx-coroutines-android
```

Corrutinas en Kotlin

> **Nota**
>
> para probar el código del tema, puedes crear un **proyecto de consola** en Kotlin o usar el **playground**

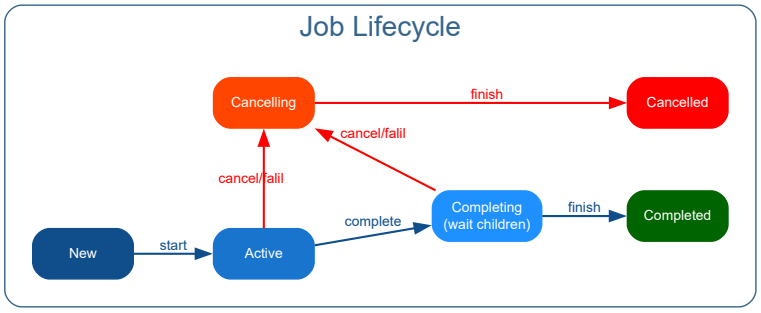
Anatomía de una corrutina

Estará formada por las siguientes partes del esquema:



Definiciones básicas:

- **CoroutinesScope**: Es el **ámbito** en el que se ejecuta la corrutina. Por ejemplo, si lanzamos una corrutina desde una actividad, el ámbito de la corrutina será la actividad. Cuando la actividad se destruya, la corrutina se cancelará automáticamente.
- **CoroutineContext**: Es el **contexto** en el que se ejecuta la corrutina y está forma por un conjunto de elementos que definen el comportamiento de la corrutina
  - **Job**: Objeto que representa la tarea concurrente que se está ejecutando. Podemos usar este objeto para controlar el estado de la corrutina, por ejemplo, para esperarla o cancelarla. El ciclo de vida del trabajo asociado a una corrutina lo puedes ver en el siguiente diagrama.



- **Dispatcher**: Podemos decir que es el **hilo** en el que se ejecuta la corrutina. Por defecto, las corrutinas se ejecutan en el hilo de la corrutina que las lanza. Pero **podemos cambiar el hilo de ejecución de una en el contexto de la misma**.
  - **CoroutineExceptionHandler**: Permite que la corrutina tenga un **manejo de excepciones personalizado en su contexto**.
- **Corrutinas '*Hijas*'**: Dentro de una corrutina podemos lanzar otras corrutinas. El ámbito de estas corrutinas secundarias (`ChildScope`) se ejecutarán dentro del ámbito o (`CoroutinesScope`) corrutina principal. Esto significa que si la corrutina principal se cancela, también se cancelarán las corrutinas secundarias. Es por eso que el Job o Tarea del contexto de la corrutina espera a que todas las corrutinas secundarias finalicen antes de finalizar su ciclo de vida.

Vamos a ir adentrándonos con ejemplos en las anteriores definiciones...

Primer ejemplo básico

Veamos el siguiente ejemplo extraído de la documentación oficial de Kotlin:

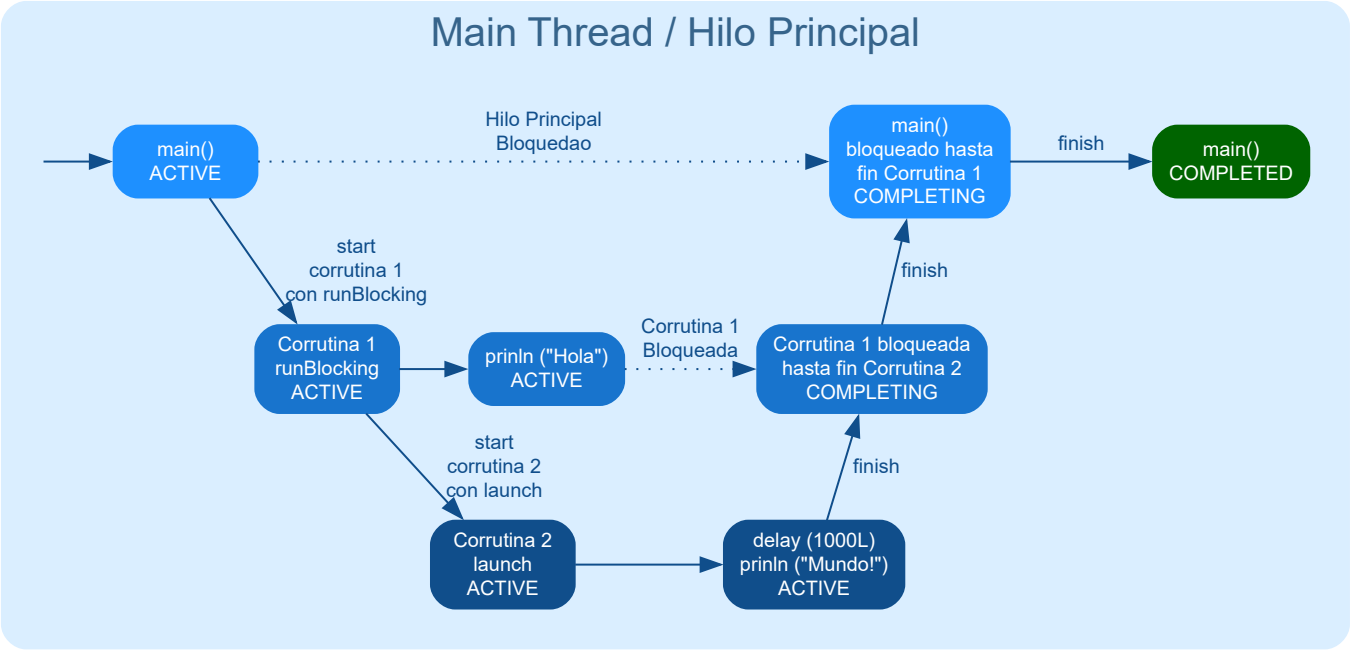
```
import kotlinx.coroutines.*

fun main() = runBlocking { // Corrutina 1
    launch {  // Corrutina 2
        delay(1000L)
        println("Mundo!")
    }
    println("Hola")
}
```

Mostrará por pantalla:

```
Hola
Mundo!
```

Vamos a desglosar lo que hace este código....

- `runBlocking` es un constructor de corrutinas, esto es, define un bloque de código que se ejecuta como una corrutina. En este caso, el bloque de código **se ejecuta en el hilo principal** (será el Dispatcher por defecto de su contexto si no especificamos otro). `runBlocking` es una función de suspensión que **bloquea el hilo donde se ejecuta** mientras se ejecuta el bloque de código. **Esto se hace para evitar que el programa finalice antes de que se ejecute la corrutina**.
- `launch` es un constructor de corrutinas. Lanza una nueva corrutina en paralelo con el resto del código, que continúa funcionando de forma independiente. Es por eso que se muestra primero '*Hola*'
- `delay` es una función de suspensión especial. Suspende la corrutina durante un tiempo específico.



Tanto el código de la **corrutina 1** (bloque `runBlocking`) y el código de la **corrutina 2** (bloque `launch`) **se ejecutan en el hilo principal de forma concurrente**.

1. Tras el primer `runBlocking`, el `main` se queda esperando a que termine la **corrutina 1**, por lo que se **bloquea el hilo principal**.
2. La **corrutina 1** lanza la **corrutina 2** y sigue ejecutando su código `println ("Hola")`.

3. La **corrutina 2** se suspende durante 1 segundo y luego imprime `println ("Mundo!")` pero la corrutina 1 ya habrá terminado y se habrá unido al hilo principal bloqueado que continúa con la ejecución del `main`.

Concurrencia estructurada

Las corrutinas siguen un principio de concurrencia estructurada, lo que significa que solo se pueden lanzar nuevas corrutinas en un `CoroutineScope` específico que **delimita la vida útil de la corrutina** como ya hemos comentado.

El ejemplo anterior muestra que `runBlocking` establece el alcance correspondiente y es por eso que el ejemplo anterior espera hasta que `Mundo!` se imprime después de un segundo de retraso y solo entonces sale.

En una aplicación real, lanzarás muchas corrutinas. **La concurrencia estructurada garantiza que no se pierdan**. Un ámbito externo no puede completarse hasta que se completen todas sus rutinas secundarias. La simultaneidad estructurada también garantiza que cualquier error en el código se informe correctamente y nunca se pierda.

Funciones de suspensión con `suspend`

El código de ejemplo está escrito en un solo bloque. **¿Cómo podríamos separar en diferentes funciones el bloque de las corrutinas?**.

Si extraemos el bloque de código interno de la **corrutina 2** `launch { ... }` a una función separada. Cuando se realiza la refactorización '*Extraer función*' en este código, obtiene una nueva función con el modificador `suspend`. Esta es nuestra primera **función de suspensión**.

```
fun main() = runBlocking {
    launch {
        doMundo()
    }
    println("Hello")
}

suspend fun doMundo() {
    delay(1000L)
    println("Mundo!")
}
```

Las **funciones de suspensión** se pueden usar dentro de las corrutinas al igual que las funciones normales, pero su característica adicional es que, a su vez, pueden usar otras funciones de suspensión (como `delay` este ejemplo) para suspender la ejecución de una corrutina.

Fíjate que en el código del editor, las funciones de suspensión se muestran con un icono de *'flecha suspendida'*. Esto es una ayuda visual para distinguir las funciones de suspensión de las funciones normales indicándome que el la corrutina se suspenderá en ese punto y continuará con la ejecución cuando finalice dicha función.

> **Resumen**
>
> Podemos resumir diciendo que una función modificada con `suspend` (función de suspensión) es una función que **solo puede ser ejecutada de forma asíncrona** dentro de una corrutina o dentro de otra función de suspensión.

Profundizando en el constructor de alcance `CoroutineScope`

Al usar `runBlocking` o `launch` dentro de un bloque de código, se crea un `CoroutineScope`. `CoroutineScope` es una interfaz que representa un ámbito de corrutina. **El ámbito de corrutina es responsable de cancelar todas las corrutinas que se ejecutan en él cuando se cierra**.

Además de los alcances definidos por los constructores que ya hemos definido. Es posible declarar nuestro propio alcance utilizando el constructor `coroutineScope`. Este crea un alcance de rutina y no se finaliza hasta que todos las corrutinas lanzadas dentro de él no finalicen.

```
fun main() = runBlocking {
    doMundo()
}

suspend fun doMundo() = coroutineScope {
    launch {
        delay(1000L)
        println("Mundo!")
    }
    println("Hola")
}
```

Los constructores `runBlocking` y `coroutineScope` pueden parecer similares porque ambos esperan a que se completen su bloque y todos sus corrutinas secundarias. La principal diferencia es que `runBlocking` **bloquea el hilo actual** en espera, mientras que `coroutineScope` suspende la corrutina actual liberando el hilo subyacente para otros usos. Por tanto, `coroutineScope` **es una función de suspensión** y solo puede ser llamado dentro de otra función de suspensión o dentro de una corrutina.

Además, todas las corrutinas que se ejecuten dentro del mismo `CorrutineScope` se ejecutarán de forma **concurrente** en el mismo hilo.

```
suspend fun doMundo() = coroutineScope {
    launch {
        delay(2000L)
        println("Mundo 2")
    }
    launch {
        delay(1000L)
        println("Mundo 1")
    }
    println("Hola")
}
```

Dispatchers e hilos

Como hemos comentado, el contexto de corrutina incluye un CoroutineDispatcher o *'despachador'* de corrutina que determina qué subproceso utiliza la corrutina correspondiente para su ejecución.

Todos los constructores de corrutinas, como `launch` y `async` , aceptan un parámetro `CoroutineContext` **opcional** que se puede usar para especificar explícitamente el despachador de la nueva rutina y otros elementos de contexto.

Hay 5 tipos de Dispatchers:

- `Dispatchers.Main` , este es el hilo principal. A diferencia de los demás, a veces tenemos que definirlo explícitamente (por ejemplo, en el entorno de prueba).
- `Dispatchers.IO` , esto para el proceso de Redes y Discos. Cualquier cosa que tenga que ver con la **extracción o envío de datos**.
- `Dispatchers.Default` , esto es para cualquier otro subproceso de trabajo que no sea principal (es decir, en segundo plano) y se asigna automáticamente.
- `Dispatchers.Unconfined` , este es un despachador especial que permite que la tarea cambie sus procesos específicos cuando suspende y reanuda su tarea.
- `newSingleThreadContext` , permite al usuario definir sus propios procesos.

Cuando `launch { ... }` se usa sin parámetros, hereda el contexto (y por lo tanto el despachador) del CoroutineScope desde el que se inicia. En este caso, hereda el contexto de la corrutina principal que se ejecuta en el hilo principal `Dispatchers.Main` .

Una forma de de indicar el hilo de ejecución es cuando lanzamos la corrutina `launch` o `async` . En el ejemplo anterior tanto `runBlocking` como `launch` usan el hilo principal `Dispatchers.Main` por heredar el contexto. Pero podemos cambiar el hilo de ejecución en `launch` de la siguiente manera...

```
fun main() = runBlocking {
    doMundo()
}

suspend fun doMundo() = coroutineScope {
    // Cambiamos el planificador de ejecución.
    launch(Dispatchers.Default) {
        delay(1000L)
        println("Mundo!")
    }
    println("Hola")
}
```

Cambiando el contexto y planificador de una corrutina con `withContext`

Podemos cambiar el contexto de una corrutina en cualquier momento usando la función de suspensión `withContext` . Esta **función de suspensión** crea un nuevo contexto de corrutina con el *'dispatcher'* especificado y ejecuta el bloque de código en él. Cuando el bloque de código finaliza, la corrutina vuelve al contexto anterior.

Normalmente, `withContext` se utiliza en situaciones en las que desea cambiar temporalmente a un contexto de ejecución diferente para realizar una operación que sea más apropiada para ese contexto, como realizar una solicitud de red, E/S de disco o cálculos que requieren un uso intensivo de la CPU.

Por ejemplo, **una forma más '*correcta*' de hacer el ejemplo anterior** sería la siguiente:

```
fun main() = runBlocking {
    launch { doMundo() }
    println("Hola")
}

suspend fun doMundo() = withContext(Dispatchers.Default) {
    delay(1000L)
    println("Mundo!")
}
```

Donde, es el método de suspensión según lo que tenga que hacer en que planificador se ejecuta cambiando su contexto.

Por tanto, este sería el esquema típico de uso de `withContext` ...

```
suspend fun peticionREST(): Dato
= withContext(Dispatchers.IO) { getDato() }

suspend fun realizaProcesoPesado(): Unit
= withContext(Dispatchers.Default) { proceso() }
```

Controlando el estado de una corrutina con `Job`

Cuando lanzamos una corrutina con `launch`, se devuelve un objeto `Job` que representa la tarea que se está ejecutando en el contexto de la corrutina. Como ya hemos comentado, podemos usar este objeto para controlar el estado de la corrutina, por ejemplo, para esperarla o cancelarla.

- `job.join()`: función de suspensión que espera a que la corrutina termine;

```
fun main() = runBlocking {
    val job : Job = launch {
        delay(1000L)
        println("Mundo!")
    }
    println("Hola")
    job.join()
    println("Fin")
}
```

- `job.cancelAndJoin()`: Notifica al trabajo su cancelación y espera a que termine ordenadamente. El trabajo internamente debe verificar periódicamente el estado de cancelación usando la propiedad de solo acceso de del `CorrutineScope` `isActive`. Además, las funciones de suspensión cancelables producen `CancellationException` en la cancelación, que se puede controlar de la manera habitual por ejemplo con un finally para realizar las acciones de limpieza necesarias.

```
fun main() = runBlocking {
    val job : Job = launch(Dispatchers.Default) {
        try {
            val startTime = System.currentTimeMillis()
            var nextPrintTime = startTime
            var i = 0
            while (isActive) { // cancellable computation loop
                // print a message twice a second
                if (System.currentTimeMillis() >= nextPrintTime) {
                    println("job: I'm sleeping ${i++} ...")
                    nextPrintTime += 500L
                }
            }
        } finally {
            println("job: I'm resuming the execution")
        }
    }
    delay(1300L) // Espera un rato
    println("main: I'm tired of waiting!")
    job.cancelAndJoin() // Cancela el trabajo y espera a que termine
    println("main: Now I can quit.")
}
```

  Mostrará ...

```
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
main: I'm tired of waiting!
job: I'm resuming the execution
main: Now I can quit.
```

- `withTimeout(timeout: Long) { }` Para generar un **contexto de corrutina** que expira después de un tiempo determinado generando un evento de cancelación y la excepción `TimeoutCancellationException`.
  `withTimeout` es una **función de suspensión**. Por ejemplo, podemos reescribir el código anterior de la siguiente forma:

```
suspend fun trabajoBloqueante(timeout: Long) = withTimeout(timeout) {
    val startTime = System.currentTimeMillis()
    try {
        var nextPrintTime = startTime
        var i = 0
        // Se ejecuta hasta que se cumple el timeout
        while (isActive) {
            if (System.currentTimeMillis() >= nextPrintTime) {
                println("job: I'm sleeping ${i++} ...")
                nextPrintTime += 500L
            }
        }
    } finally {
        println("job: I'm resuming the execution")
    }
}

fun main() = runBlocking {
    println("main: Waiting for job 1.3 secs.")
    val job = launch {
        trabajoBloqueante(1300L)
    }
    job.join() // Espera a que termine o expire el trabajo
    println("main: Now I can quit.")
}
```

Mostrará ...

```
main: Waiting for job 1.3 secs.
job: I'm sleeping 0 ...
job: I'm sleeping 1 ...
job: I'm sleeping 2 ...
job: I'm resuming the execution
main: Now I can quit.
```

Jobs asíncronos que devuelven un resultado `async/await`

Hasta ahora hemos visto que las corrutinas se ejecutan de forma concurrente, pero no hemos visto cómo obtener un resultado de una corrutina. Para ello, podemos usar el constructor `async`. Este constructor es similar a `launch`, pero devuelve un objeto `Deferred` que representa un resultado futuro.

`Deferred` es una **subclase** de `Job` y, por lo tanto, también se puede cancelar. Además, tiene una función de suspensión `await()` que devuelve el resultado cuando está listo.

Veamos un ejemplo donde tenemos la función asíncrona `esPrimo(n: Long)` que devuelve un booleano indicando si el número pasado como parámetro es primo o no.

Puesto que `async` debe ejecutarse dentro de un `CoroutineScope`, lo lanzamos dentro del `GlobalScope`. `GlobalScope` es un **alcance global** que no está vinculado a ningún hilo en particular y, por lo tanto, no bloquea el hilo principal.

```
fun esPrimo(n: Long): Deferred<Boolean> = GlobalScope.async {
    n > 1 && (2 until n).none { n % it == 0L }
}

fun main() = runBlocking {
    val n = 1000000007L
    println("Viendo si es primo el número $n")
    val job : Deferred<Boolean> = esPrimo(n)

    print("Esperando cálculo ")
    // Vamos a mostrar un punto cada 100 ms mientras vemos
    // si el número es primo o no
    while (!job.isCompleted) {
        print(".")
        delay(100)
    }

    job.await().let {
        println("\nEl numero $n ${if (it) "es" else "no es"} primo")
    }
}
```

Mostrará ...

```
Viendo si es primo el número 1000000007
Esperando cálculo ...................
El numero 1000000007 es primo
```

> **Warning**
>
> Aunque en este ejemplo hemos usado `GlobalScope` para lanzar la corrutina, **no es recomendable usarlo** y solo se ha usado para simplificar el ejemplo. Puedes ver más información en https://kotlinlang.org/docs/composing-suspending-functions.html#async-style-functions

Hemos usado GlobalScope para lanzar la corrutina, porque si hiciéramos esto que es más '*correcto*'. `muestraProgreso` quedaría bloqueado al estar ejecutándose en el mismo hilo y mismo alcance ya que `esPrimo` es un proceso que demanda mucho tiempo de CPU. **Ya verás que en el próximo punto solucionaremos esto con `withContext` para cambiar de hilo**.

```
suspend fun esPrimo(n: Long) = coroutineScope {
    n > 1 && (2 until n).none { n % it == 0L }
}
suspend fun muestraProgreso(job : Deferred<Boolean>)
= withContext(Dispatchers.Default) {
    print("Esperando cálculo ")
    while (!job.isCompleted) {
        print(".")
        delay(100)
    }
}

fun main() = runBlocking {
    val n = 1000000007L
    println("Viendo si es primo el número $n")
    val job : Deferred<Boolean> = async {esPrimo(n) }
    muestraProgreso(job)
    job.await().let {
        println("\nEl numero $n ${if (it) "es" else "no es"} primo")
    }
}
```

Depurando contexto y planificador de una corrutina

Vamos a definir la siguiente función de extensión para obtener información de nuestro contexto:

```
fun CoroutineContext.info() =
"\nCorrutina: {\n\tContexto: ${this}, \n\tProceso: ${Thread.currentThread().name}\n}"
```

Si volvemos a ejecutar el ejemplo inicial...

```
fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Mundo! ${coroutineContext.info()}")
    }
    println("Hola ${coroutineContext.info()}")
}
```

Mostrará ...

```
Hola
Corrutina: {
          Contexto: [BlockingCoroutine{Active}@7d4793a8, BlockingEventLoop@449b2d27],
          Proceso: main
}
Mundo!
Corrutina: {
          Contexto: [StandaloneCoroutine{Active}@1ae369b7, BlockingEventLoop@449b2d27],
          Proceso: main
}
```

Ahora modificamos el contexto de la corrutina 2 para que se ejecute en el hilo `Dispatchers.Default` y además le asignamos un nombre ambas corrutinas para poder identificarlas:

```
fun main() = runBlocking(CoroutineName("CORRUTINA 1")) {
    launch (Dispatchers.Default + CoroutineName("CORRUTINA 2")) {
        delay(1000L)
        println("Mundo! ${coroutineContext.info()}")
    }
    println("Hola ${coroutineContext.info()}")
}
```

Ahora la salida nos mostrará el nombre de la corrutina y el proceso en el que se ejecuta:

```
Hola
Corrutina: {
          Contexto: [CoroutineName(CORRUTINA 1), BlockingCoroutine{Active}@1bc6a36e, BlockingEventLoop@1ff8b8f],
          Proceso: main
}
Mundo!
Corrutina: {
          Contexto: [CoroutineName(CORRUTINA 2), StandaloneCoroutine{Active}@f4e4ec0, Dispatchers.Default],
          Proceso: DefaultDispatcher-worker-2
}
```

Ejemplo:

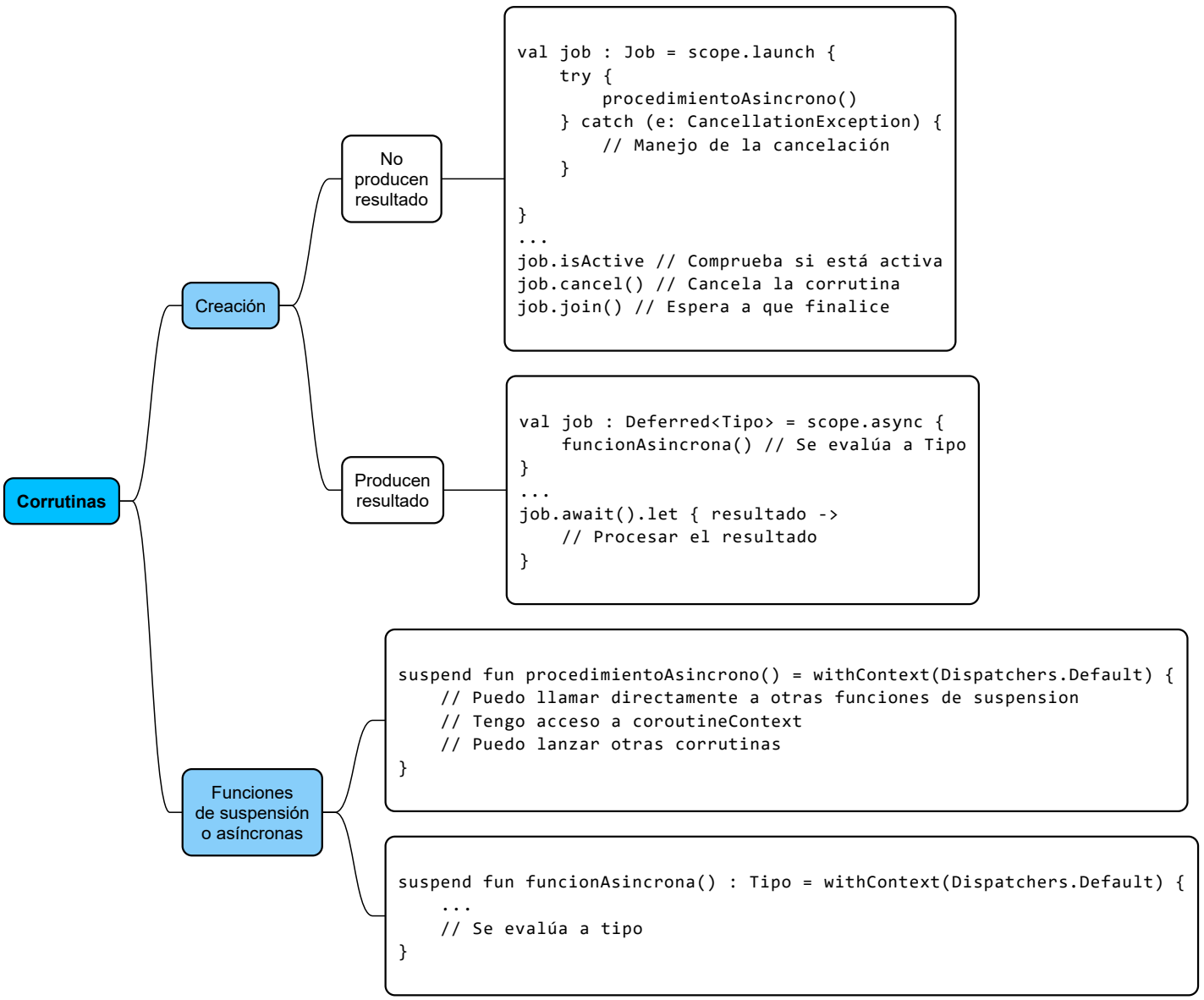
Volvamos al cálculo de números primos. Pero esta vez **vamos a implementarlo de forma más adecuada**, añadiendo logs de depuración, sin usar `GlobalScope`, definiendo funciones de suspensión y usando `withContext` para cambiar de hilo.

```
fun CoroutineContext.info() = "\nCorrutina: {\n\tContexto: ${this}, \n\tProceso: ${Thread.currentThread().name}\n}"

suspend fun esPirmo(n: Long): Boolean
= withContext(Dispatchers.Default + CoroutineName("ESPRIMO")) {
    println(coroutineContext.info())
    n > 1 && (2 until n).none { n % it == 0L }
}

suspend fun muestraProgreso()
= withContext(Dispatchers.Default + CoroutineName("PROGRESO")) {
    try {
        delay(200)
        println(coroutineContext.info())
        print("Esperando cálculo ")
        while (true) {
            print(".")
            delay(100)
        }
    } catch (e: CancellationException) {
        println("\nProgreso cancelado ${coroutineContext.info()}")
    }
}

fun main() = runBlocking {
    val n = 1000000007L
    println("Viendo si es primo el núumero $n")
    val jobPrimo = async { esPirmo(n) }
    val jobProgreso = launch { muestraProgreso() }
    println("En main esperando a esPrimio() ${coroutineContext.info()}")
    jobPrimo.await().let {
        println("\nEl numero $n ${if (it) "es" else "no es"} primo")
    }
    jobProgreso.cancel()
}
```

Mostrará ...

```
Viendo si es primo el núumero 1000000007
En main esperando a esPrimio()
Corrutina: {
         Contexto: [BlockingCoroutine{Active}@47fd17e3, BlockingEventLoop@7cdbc5d3],
         Proceso: main
}

Corrutina: {
         Contexto: [CoroutineName(ESPRIMO), DispatchedCoroutine{Active}@472bb070, Dispatchers.Default],
         Proceso: DefaultDispatcher-worker-2
}

Corrutina: {
         Contexto: [CoroutineName(PROGRESO), ScopeCoroutine{Active}@5931574c, Dispatchers.Default],
         Proceso: DefaultDispatcher-worker-1
}
Esperando cálculo ..................
El numero 1000000007 es primo

Progreso cancelado
Corrutina: {
         Contexto: [CoroutineName(PROGRESO), ScopeCoroutine{Cancelling}@5931574c, Dispatchers.Default],
         Proceso: DefaultDispatcher-worker-2
}
```

Esquema resumen conceptos básicos de corrutinas



Corrutinas en Android

Utilizaremos los conceptos ya vistos pero adaptados a Android. Alguna de las cosas que deberemos tener en cuenta son:

1. ❌ No debemos usar `runBlocking` en el hilo principal porque lo bloqueará.En su lugar, podemos usar `runBlocking` en un hilo secundario.
2. ❌ No deberíamos lanzar corrutinas en el **GlobalScope** porque no se cancelarán automáticamente cuando se destruya el componente de Android.
3. Las funciones de suspensión deberían ser seguras para su llamada desde el subproceso principal. Por ejemplo, cambiando el contexto de ejecución para realizar una solicitud de red con `withContext(Dispatchers.IO)`.
4. ❌ No debemos actualizar la UI desde una corrutina que se ejecute en un contexto diferente al del hilo principal. Deberemos usar `withContext(Dispatchers.Main)` para cambiar al contexto del hilo principal y actualizar la UI.
5. ❌ No debemos lanzar una corrutina que modifique un estado antes de finalizar la composición de la UI.
6. Si la aplicación entra en segundo plano. En aquellos alcances de corrutina que estén vinculados al ciclo de vida de un componente de Android, se cancelarán automáticamente solo cuando el componente se destruya. Pero si no es así, **deberemos cancelar manualmente** las corrutinas que se estén ejecutando en segundo plano si queremos que se paren.
7. ❌ No debemos exponer métodos de suspensión públicos en el ViewModel. Pero sí puede lanzar corrutinas en su alcance o mejor exponer flujos de datos que se puedan observar desde la UI.

CoroutineScopes más comunes en Android

Los alcances (`CoroutineScope`) de las corrutinas se pueden vincular a los ciclos de vida de los componentes de Android.

- `rememberCoroutineScope()` Vinculado al ciclo de vida de un componente de Android como un **Composable**. Se cancela cuando el componente se destruye.
  Lo vamos a usar solo cuando queramos ejecutar una acción sobre un componente tras un evento. Ejemplo extraído de **documentación oficial de Android**:

```
@Composable
fun MoviesScreen(scaffoldState: ScaffoldState = rememberScaffoldState()) {
    // Crea un CoroutineScope vinculado al ciclo de vida de MoviesScreen
    // este alcance es recordado en la recomposición.
    val scope = rememberCoroutineScope()

    Scaffold(scaffoldState = scaffoldState) {
        Column {
            /* ... */
            Button(
                onClick = {
                    // Crea una nueva corrutina en el manejador del
                    // evento del botón para mostrar el snackbar asociado al Scaffold
                    scope.launch {
                        // showSnackbar es una función de suspensión
                        scaffoldState.snackbarHostState.showSnackbar("Something happened!")
                    }
                }
            ) {
                Text("Press me")
            }
        }
    }
}
```

- `viewModelScope`: Vinculado al ciclo de vida del **ViewModel**. Se cancela cuando el ViewModel se destruye.
  Dentro de cualquier clase que herede de `ViewModel` dispondremos de la propiedad `viewModelScope` que será un `CoroutineScope` vinculado al ciclo de vida del mismo. Crear una corrutina de las formas que hemos visto es tan simple como hacer...

  ```
  viewModelScope.launch { ... }
  val deferredJob = viewModelScope.async { ... }
  ```

  Recuerda poara crear una corrutina hija dentro de otra corrutina, debemos usar el `CoroutineScope` de la corrutina padre. Por ejemplo, si queremos lanzar una corrutina desde un **ViewModel** que se ejecute en el hilo principal y que se cancele cuando el ViewModel se destruya, podemos hacerlo de la siguiente forma:

  ```
  viewModelScope.launch {
      ...
      // Corrutina hija en el alcance del padre, con su contexto específico.
      launch(Dispatchers.IO) {
          try {
          }
          catch (ce: CancellationException) {
              // Manejo de la cancelación
          }
      }
      // Esperará a que termine la corrutina hija
  }
  ```

  Para finaliar cualquier corrutina que se esté ejecutando en el `viewModelScope` podemos usar la función `viewModelScope.cancel()` o `viewModelScope.coroutineContext.cancelChildren()`. Por ejemplo...

  ```
  fun paraProceso() {
      // Cancelará las corrutinas hijas creados dentro de su contexto.
      // Provocará la excepción CancellationException en las corrutinas hijas
      viewModelScope.coroutineContext.cancelChildren()
  }
  ```

- `lifecycleScope`: Vinculado al ciclo de vida de un componente de Android como una **Activity**. Se cancela cuando el componente se destruye.
  Es idéntico a `viewModelScope` pero su alcance es una Actividad o Fragment.

Side-effects y corrutinas en Compose

Un efecto lateral o side-effect de Compose es un cambio en el estado de la app que ocurre fuera del alcance de una función de componibilidad. Puesto que no podemos realizar ningún proceso bloqueante durante la composición de la UI y además no tenemos ningún control sobre el momento, orden y finalización de la misma. En el ejemplo anterior donde hemos visto como podemos usar `rememberCoroutineScope()` para mostrar una notificación en un **Scaffold** tras un evento de click en un botón. Esto sería un efecto lateral fuera de la composición de la UI. **Por resumir, cuando un cambio de estado en la UI no lo controla la composición, sino que necesitemos que sea predecible y asíncrono, estaremos ante un efecto lateral**.

LaunchedEffect

Ejecuta funciones de suspensión en el alcance de un elemento componible.

Por ejemplo, para ejecutar una función de suspensión ante uno o varios cambios de estado.

```
@Composable
fun FlechaRotable() {
    var rotationZAnimationSatate = remember { Animatable(0f) }
    var rotado by remember { mutableStateOf(false) }

    // Indico el estado o los estados que provocarán la ejecución de
    // la o las funciones de suspensión
    LaunchedEffect(rotado) {
        // Estamos dentro de un CorourineScope
        // esto se ejecuta de forma asíncrona con la composición de la UI
        // por tanto las funcionen de suspensión no la bloquean.

        // Función de suspensión
        rotationZAnimationSatate.animateTo(
            targetValue = if (rotado) 180f else 0f,
            animationSpec = tween(durationMillis = 500, easing = FastOutSlowInEasing))
    }

    Icon(
        painter = rememberVectorPainter(image = Icons.Filled.ArrowCircleUp),
        contentDescription = null,
        modifier = Modifier
            .clickable { rotado = !rotado }
            .graphicsLayer { rotationZ = rotationZAnimationSatate.value }
            .size(200.dp)
    )
}
```

> 📌 **Nota**: Si especificamos `LaunchedEffect(Unit) { ... }` la corrutina se ejecutará solo en la primera composición del componente.

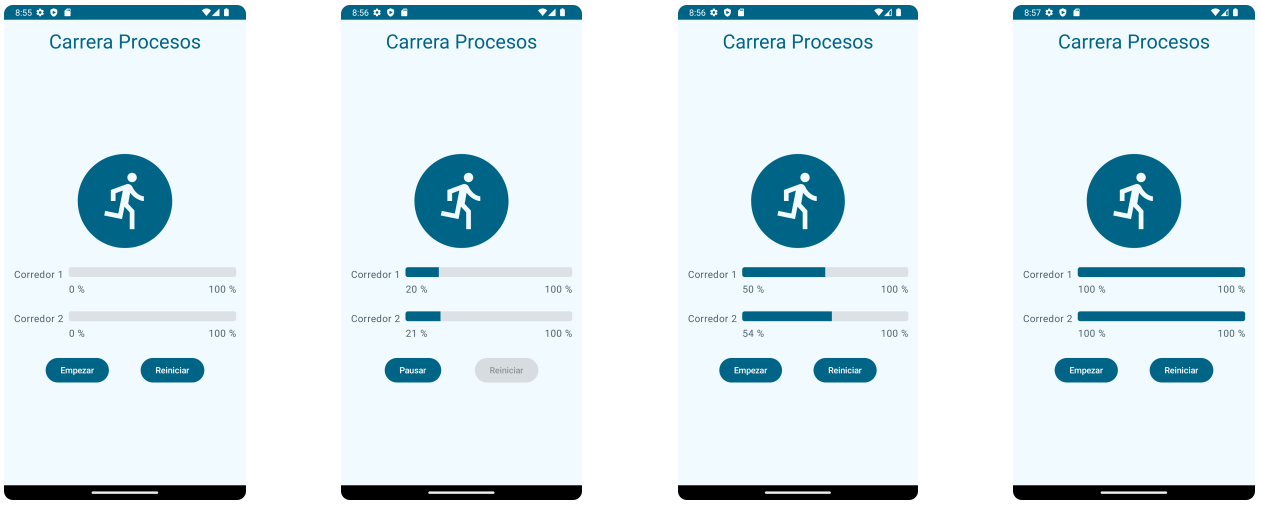
Podríamos reescribir el código de la función StateFul anterior gestionando el Side-effect en el manejador del evento con `rememberCoroutineScope()`.

```
@Composable
fun FlechaRotable() {
    var rotationZAnimationSatate = remember { Animatable(0f) }
    var rotado by remember { mutableStateOf(false) }
    var coroutineScope = rememberCoroutineScope()

    Icon(
        painter = rememberVectorPainter(image = Icons.Filled.ArrowCircleUp),
        contentDescription = null,
        modifier = Modifier
            .clickable {
                rotado = !rotado
                coroutineScope.launch {
                    rotationZAnimationSatate.animateTo(
                        targetValue = if (rotado) 180f else 0f,
                        animationSpec = tween(durationMillis = 500, easing = FastOutSlowInEasing))
                }
            }
            .graphicsLayer { rotationZ = rotationZAnimationSatate.value }
            .size(200.dp)
    )
}
```

Ejemplo de uso de corrutinas en Android

> 🔗 **Dercarga**: El proyecto con el código del ejemplo que vamos a ver lo puedes descargar de este **enlace**

Es la típica aplicación donde vamos a tener dos corrutinas a modo de corredores. Cada corredor se ejecutará en un hilo diferente y se mostrará su progreso en la UI. Además, tendremos un botón para **Empezar** y **Parar** la carrera y otro que **Reiniciará** la carrera y estará activo solo cuando la carrera esté parada. Además, al llegar ambos corredores al final del progreso la carrera parará automáticamente. También parará automáticamente si la aplicación pasa a segundo plano.



En primer lugar en el paquete `com.ejemplo_corrutinas.utilities` hemos definido el fuente CoroutinesDebug.kt que nos permitirá mostrar información de las corrutinas en el **Logcat**. Para ello, hemos definido la siguiente función de extensión a usar en un contexto de corrutina:

```
fun CoroutineContext.log(
    corrutina: String = "Corrutina",
    accion: String = "No especificada"
) {
    Log.println(
        Log.DEBUG, corrutina,
        "${corrutina}: {\n\tAccion: ${accion}, \n\tContexto: ${this}\n}"
    )
}
```

En el paquete `com.ejemplo_corrutinas.ui.features.seguimientocarrera` definiremos la pantalla de prueba de nuestra aplicación. El interfaz del componente de la pantalla será el siguiente:

```
@Composable
fun SeguimientoCarreraScreen(
    corredor1: CorredorUiState,
    corredor2: CorredorUiState,
    enCarrera: Boolean,
    onSeguimientoCarreraEvent: (SeguimientoCarreraEvent) -> Unit,
    modifier: Modifier = Modifier
)
```

Donde el estado de cada corredor se representará por la siguiente clase:

```
data class CorredorUiState(
    val nombre: String,
    val porcentajeProgreso : Int = 0
) {
    // Función de suspensión que simula el avance del corredor en la
    // carrera bloquerá la corrutina con una espera entre 5 y 300 ms
    suspend fun avanza(): CorredorUiState {
        delay((5L..300L).random())
        val nuevoPorcentaje = porcentajeProgreso + 1
        return copy(porcentajeProgreso = nuevoPorcentaje)
    }

    fun reinicia(): CorredorUiState = copy(porcentajeProgreso = 0)
}
```

Además, dispondremos de un *'State'* `enCarrera` que me indicará si se está corriendo la carrera o no. Y un `onSeguimientoCarreraEvent` que será el manejador de eventos de la pantalla y donde se controla la pulsación de ambos botones.

```
sealed interface SeguimientoCarreraEvent {
    data object OnEmpezarPararClick : SeguimientoCarreraEvent
    data object OnReiniciarClick : SeguimientoCarreraEvent
}
```

Podemos crear un `@Preview` de test similar a la carrera con corrutinas. Pero como se trata de un composable deberemos usar `LaunchesEffect` para ejecutar la simulación de la carrera en un `CoroutineScope` dentro de la composición de dicho *preview* de prueba

```
@Preview
@Composable
private fun SeguimientoCarreraScreenPreview() {

    // Creamos los estados a usar en la UI
    var corredor1 by remember {
        mutableStateOf(CorredorUiState(nombre = "Corredor 1"))
    }
    var corredor2 by remember {
        mutableStateOf(CorredorUiState(nombre = "Corredor 2"))
    }
    var enCarrera by remember { mutableStateOf(false) }

    ...
```

```
    ...
    // Cada vez que hay un cambio en uno de estos estados se recompone la UI
    // y se lanza el LaunchedEffect con las corrutinas de simulación de la carrera
    LaunchedEffect(
        key1 = enCarrera,
        key2 = corredor1.porcentajeProgreso,
        key3 = corredor2.porcentajeProgreso
    ) {
        // Cada corrutina hara que avance el corredor cambiando su estado
        // y por tanto recomponinedo la UI y relanzando el LaunchedEffect
        // Ambos procesos se ejecutan en un contexto de corrutina diferente
        // con Dispatchers.Default para que el sistema decida el hilo más adecuado para el preview
        val jobCorredor1 = launch(Dispatchers.Default) {
            if (enCarrera && corredor1.porcentajeProgreso < 100)
                corredor1 = corredor1.avanza()
        }
        val jobCorredor2 = launch(Dispatchers.Default) {
            if (enCarrera && corredor2.porcentajeProgreso < 100)
                corredor2 = corredor2.avanza()
        }
        // Esperamos a que los dos correodres avancen si lo tienen que hacer
        joinAll(jobCorredor1, jobCorredor2)
        // Si ambos corredores han llegado al 100% paramos la carrera
        if (corredor1.porcentajeProgreso == 100 && corredor2.porcentajeProgreso == 100)
            enCarrera = false
    }
    ...
```

Por último, definimos el componente de nuestra pantalla...

```
    ...
    EjemploCorrutinasTheme {
        Surface {
            SeguimientoCarreraScreen(
                corredor1 = corredor1,
                corredor2 = corredor2,
                enCarrera = enCarrera,
                onSeguimientoCarreraEvent = {
                    when (it) {
                        SeguimientoCarreraEvent.OnEmpezarPararClick -> {
                            enCarrera = !enCarrera
                        }
                        SeguimientoCarreraEvent.OnReiniciarClick -> {
                            corredor1 = corredor1.reinicia()
                            corredor2 = corredor2.reinicia()
                        }
                    }
                }
            )
        }
    }
}
```

Bien, ahora vamos a definir un **ViewModel** para nuestro Screen y realizar el proceso de la carrera en el mismo. La cosa va a cambiar '*un poco*' puesto que las corrutinas se lanzarán desde ahora desde un `CoroutineScope` distinto al de la composición.

```
class SeguimientoCarreraViewModel : ViewModel() {

    // Definimos los estados de la UI
    var corredor1 by mutableStateOf(
        CorredorUiState(nombre = "Corredor 1")
    )
        private set
    var corredor2 by mutableStateOf(
        CorredorUiState(nombre = "Corredor 2")
    )
        private set
    var enCarrera by mutableStateOf(false)
        private set

    // Gestión de los eventos del Screen
    fun onSeguimientoCarreraEvent(event: SeguimientoCarreraEvent) {
        when (event) {
            SeguimientoCarreraEvent.OnEmpezarPararClick -> {
                if (enCarrera)
                    // Cambiamos el estado de enCarrera = false
                    // y paramos las corrutinas que avanzan cambiando
                    // el valor del estado del progreso
                    pararDeCorrer()
                else
                    // Cambiamos el estado de enCarrera = false
                    // e iniciamos las corrutinas que avanzan cambiando
                    // el valor del estado del progreso
                    empezarACarrer()
            }
            SeguimientoCarreraEvent.OnReiniciarClick -> {
                // Ojo no debería llamar a Reiniciar si estoy en carrera
                // ya que una corrutina suspendida en delay puede actualizar
                // el estado de la UI tras el reset. Esto es porque guarda un
                // el valor de un estado inmutable de la UI anterior.
                corredor1 = corredor1.reinicia()
                corredor2 = corredor2.reinicia()
            }
        }
    }

    fun pararDeCorrer() {
        // Cancelo las corrutinas hijas que se estén ejecutando
        // en el contexto del viewModelScope que en mi caso son
        // los dos corredores.
        // Si tuviera más corrutinas, tendrá que guardarme los trabajos
        // como propiedades privadas en el ViewModel y bucarlos en el
        // contexto para cancelarlos de forma específica.
        viewModelScope.coroutineContext.log("CARRERA", "Parando corredores...")
        viewModelScope.coroutineContext.cancelChildren()
    }
    ...
```

```
    ...
    // Lanza una corrutina que simula el avance de un corredor
    // para ello recibe el estado inicial del corredor que se guarda en el VM
    // una función que actualiza el estado de la UI
    // Devolverá un Job que se puede usar para cancelar la corrutina o esperar a que termine
    private fun CoroutineScope.lanzaCorredor(
        estadoInicialCorredor: CorredorUiState,
        actualizaEstadoUi: (CorredorUiState) -> Unit
    ): Job = launch {
        // Guardamos el estado inicial del corredor en una variable mutable
        // donde se irá actualizando el estado del corredor de forma local
        // este estado servirá para actualizar el estado de la UI del corredor correspondiente
        var estadoCorredorLocal = estadoInicialCorredor
        try {
            coroutineContext.log("CARRERA", "Corredor ${estadoCorredorLocal.nombre} corriendo")
            while (estadoCorredorLocal.porcentajeProgreso < 100) {
                estadoCorredorLocal = estadoCorredorLocal.avanza() // SUSPEND FUN
                actualizaEstadoUi(estadoCorredorLocal)
            }
            coroutineContext.log("CARRERA", "${estadoCorredorLocal.nombre} está en meta")
        } catch (ce: CancellationException) {
            coroutineContext.log("CARRERA", "${estadoCorredorLocal.nombre} se para")
        }
    }

    fun empezarACarrer() =
        // Lanzo una corrutina con el alcance en el que se encuentra el ViewModel
        // y con el contexto de Dispatchers.Default
        viewModelScope.launch(Dispatchers.Default) {
            // Si alguno de los corredores no ha llegado a la meta, sigo en carrera
            if (corredor1.porcentajeProgreso != 100 || corredor2.porcentajeProgreso != 100) {
                enCarrera = true
                try {
                    val jobCorredor1 = lanzaCorredor(corredor1) { corredor1 = it }
                    val jobCorredor2 = lanzaCorredor(corredor2) { corredor2 = it }
                    // Espero a que terminen las dos corrutinas o bien
                    // porque alguno de los corredores ha llagado a la meta
                    // o bien porque se han cancelado las corrutinas
                    joinAll(jobCorredor1, jobCorredor2) // SUSPEND FUN
                    // Si los dos corredores han llegado a la meta, termino la carrera
                    if (corredor1.porcentajeProgreso == 100 && corredor2.porcentajeProgreso == 100)
                        enCarrera = false
                } catch (ce: CancellationException) {
                    enCarrera = false
                }
                viewModelScope.coroutineContext.log("CARRERA", "Corredores parados")
            }
        }
}
```

Por último, vamos a examinar el código en la MainActivity.

```
class MainActivity : ComponentActivity() {
    // Definimos el ViewModel como propiedad delegada y así usarla en
    // los métodos del ciclo de vida de la actividad
    private val vm by viewModels<SeguimientoCarreraViewModel>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            EjemploCorrutinasTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    SeguimientoCarreraScreen(
                        corredor1 = vm.corredor1,
                        corredor2 = vm.corredor2,
                        enCarrera = vm.enCarrera,
                        onSeguimientoCarreraEvent = vm::onSeguimientoCarreraEvent
                    )
                }
            }
        }
    }

    // Al pausar la actividad por cualquier motivo, paramos la carrera
    // a través del ViewModel que es quine gestiona las corrutinas de la misma
    // Ten en cuenta que el Accance del ViewModel sigue vivo aunque la actividad
    // esté pausada y por tanto los corredores seguirán corriendo
    override fun onPause() {
        super.onPause()
        vm.pararDeCorrer()
    }
}
```

Si quisiéramos que la carrera empezara nada más iniciarse la actividad, podríamos hacerlo en el método `onStart` de la actividad. Donde ya sabemos que se ha creado la UI y por tanto podemos modificar el estado de la misma.

```
override fun onStart() {
    super.onStart()
    vm.empezarACarrer()
}
```