97. Corrutinas con Android Kotlin

Terminos :

- Corutines
- CoroutineScope
- Context
- dispatcher
- launch
- async
- await y join

Se utilizan en la programación de actividades paralelas, asíncronas. Son más ligeras que los threads.

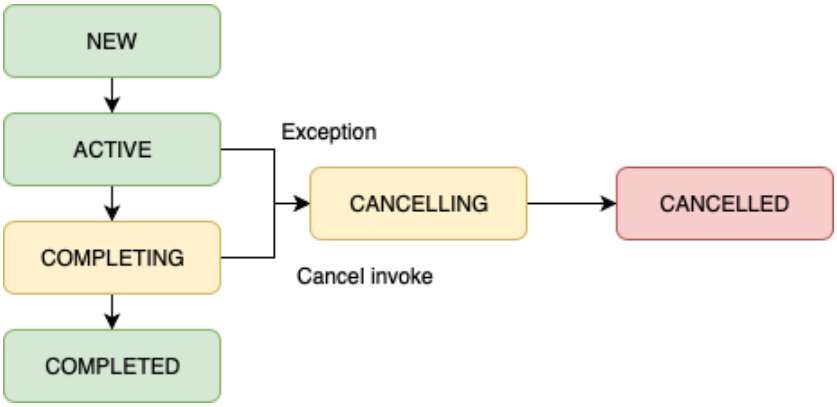
Podéis consultar: introducción a las corrutinas y algo básico de práctica en el playground de kotlin

Para poder utilizar las corrutinas se debe añadir a Gradle la dependencia:

```
dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")
}
```

97.1 Básico de corrutinas

Ciclo de vida de las corrutinas:



Ejemplo simple de corrutina en Kotlin:

```
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is delayed
```

```
}
// salida:
// Hello
// World
```

Es importante entender que una función de suspensión como `delay()` solo se puede llamar desde otra función de suspension. Al usar runblocking tenemos es función de suspensión.

Elementos utilizados en este ejemplo:

- `launch` es un `coroutine builder` , lanza una nueva corrutina concurrente con el resto del código
- `delay` puncion **suspend** especial
- `runBlocking` Un `coroutine builder` nos permite llamar a funciones de suspension

Las corrutinas siguen los principios de la concurrencia estructurada.

Las corrutinas solamente pueden ejecutar en un ámbito `CoroutineScope`

El ejemplo anterior podemos sacar el código que consume cpu en una función externa:

```
fun main() = runBlocking { // this: CoroutineScope
    launch { doWorld() }
    println("Hello")
}

// this is your first suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

La función ejecutada por launch se marca con `suspend` que son las funciones que se ejecutan dentro de subrutinas y pueden usar funciones de suspensión como `delay`

97.2 Scope Builders

Los usuarios pueden crear sus propios ambitos con coroutineScope. Los coroutineSope sirven para aseguar que una rutina no acaba hasta que se completen las hijas.

```
Los constructores runBlocking y coroutineScope esperan hata que su codigo y la de sus
hijos se completa. La diferencia princiapal es que runBlocking bloquea el Hilo actual
mientras espera mientras que coroutineScope suspende el Hilo subyacente para dedicarlo a
otras actividades. Por este motivo runBlocking es una función regular y coroutineScope
una función de suspensión
```

Se puede usar coroutineScope desde cualquier función suspendible.

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

suspend fun doWorld() = coroutineScope {  // this: CoroutineScope
    launch {
        delay(1000L)
        println("World!")
    }
```

```
    println("Hello")
}
```

97.3 ScopeBuilder y concurrencia

Un `coroutineScope` se puede usar en funciones suspendibles para lanzar multiples corutinas:

```
// Sequentially executes doWorld followed by "Done"
fun main() = runBlocking {
    doWorld()
    println("Done")
}

// Concurrently executes both sections
suspend fun doWorld() = coroutineScope { // this: CoroutineScope
    launch {
        delay(2000L)
        println("World 2")
    }
    launch {
        delay(1000L)
        println("World 1")
    }
    println("Hello")
}
```

97.4 Job explicito

El constructor de corrutinas `launch` devuelve un objeto `Job` que se usa para controlar las corrutinas y esperar explícitamente el final de la corrutina .

```
fun main() = runBlocking {
val job = launch { // launch a new coroutine and keep a reference to its Job
    delay(1000L)
    println("World!")
}
println("Hello")
job.join() // wait until child coroutine completes
println("Done")

}
```

97.5 Las corrutinas son ligeras en consumo de recursos

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    repeat(50_000) { // launch a lot of coroutines
        launch {
            delay(5000L)
            print(".")
        }
    }
}
```

97.6 Cancelaciones y timeout

Se puede cancelar cualquier corrutina llamando al método `.cancel()`

```
...

suspend fun getWeatherReport() = coroutineScope {
    val forecast = async { getForecast() }
    val temperature = async { getTemperature() }

    delay(200)
    temperature.cancel()

    "${forecast.await()}"
}

...
```

97.7 Tratamiento de excepciones en corrutinas.

El tratamiento de excepciones en corrutinas es similar al general pero con matices.

De forma muy resumida:
En un caso donde usamos corrutinas para obtener datos de un Rest api(productor) y los mostramos (consumidor), se aconseja capturar la excepción en el productor sin que afecete al consumidor. Ver esta parte del codelab

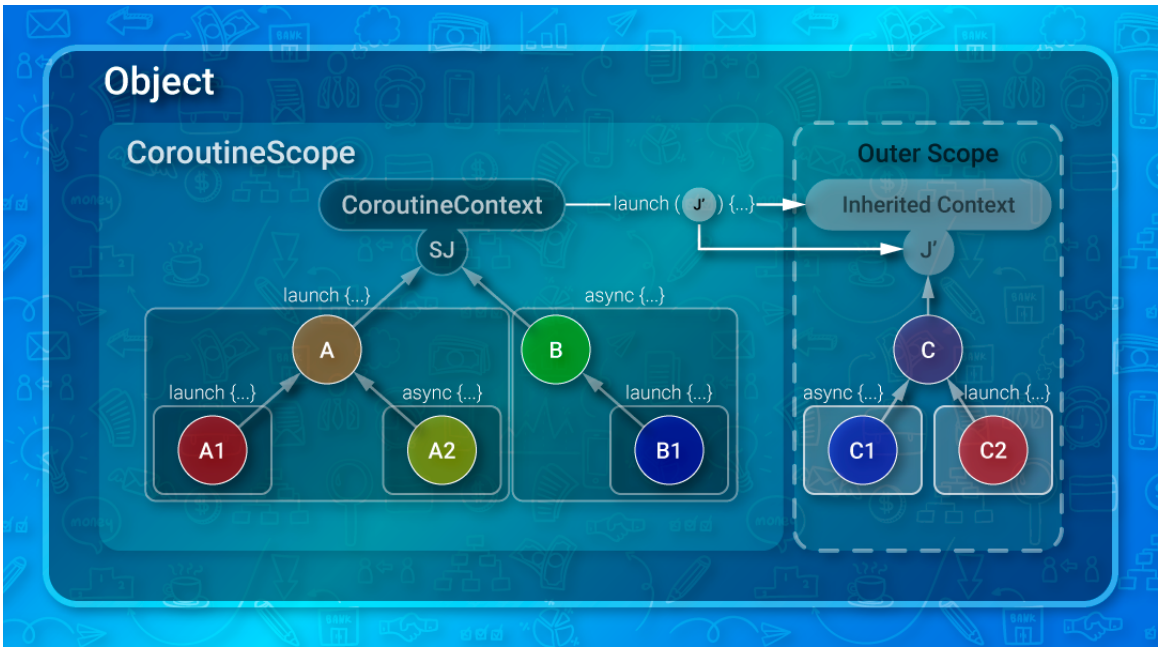
A tener en cuenta:
Las excepciones se propagan de manera diferente en el caso de las corrutinas que se inician con `launch` () en comparación con `async()`. Dentro de una corrutina iniciada por launch(), se arroja una excepción de inmediato para que puedas encerrar el código con un bloque try-catch si se espera que arroje una excepción. Consulta el ejemplo.

97.8 Resumen

*CorroutineScope * Dispatcher * CoroutineContex

97.8.1 Concurrencia estructurada

Las corrutinas se crean a partir de otras corrutinas formando un árbol:



97.8.2 CoroutineScope en Android

Mantiene control del ciclo de vida cualquier rutina creada en su scope con `launch` o `async`. En Android dispnemos de algunas cooutineScope KTX definidas en las librerías de Jetpack:

- viewModelScope
- lifecycleScope

Android proporciona compatibilidad con el alcance de corrutinas en entidades que tienen un ciclo de vida bien definido, como Activity (lifecycleScope) y ViewModel (viewModelScope). Las corrutinas que se inician dentro de estos alcances cumplirán con el ciclo de vida de la entidad correspondiente, como Activity o ViewModel.

Por ejemplo, supongamos que inicias una corrutina en un Activity con el alcance de corrutinas proporcionado que se denomina lifecycleScope. Si se destruye la actividad, se cancelará lifecycleScope, y también se cancelarán automáticamente todas sus corrutinas secundarias. Solo debes decidir si la corrutina que sigue al ciclo de vida de Activity es el comportamiento que deseas.

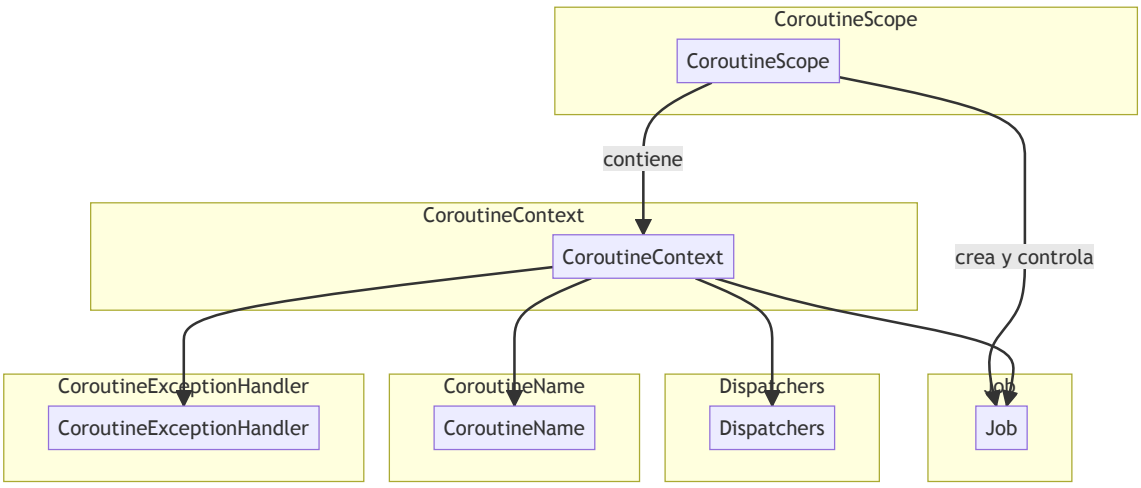
97.8.3 Dispatchers

97.8.4 CoroutineContext

CoroutineContext es un indice de la colección de cuatro elementos:

1. CoroutineDispatcher
2. Dispatchers.IO
3. Dispatchers.Main
4. Dispatchers.Default
5. CoroutineExceptionHandler (opcional)

6. CoroutineName
7. Job



Ejemplo:

```
fun ejemplo(){
    val job = viewModelScope.launch{
        withContext(Dispatchers.IO)
    }
}
```

Dentro de un coroutineContext podemos cambiar de contesto con la función de suspensión:

```
  withContext(Dispatchers.IO)
```

97.8.5 Job

La corrutina del ejemplo anterior podemos cancelarla usando el objeto job, por ejemplo si ocurre una excepción.

```
fun ejemplo(){
    val job = viewModelScope.launch{
        withContext(Dispatchers.IO)
    }
    try{
        // codigo operaciones
    } catch(e: Throwable){
        job.cancel()
    }
}
```

97.8.6 launch y async

En Kotlin, tanto launch como async se utilizan para iniciar corrutinas, pero tienen propósitos y comportamientos diferentes:

launch:

- launch se utiliza para iniciar una corrutina que no devuelve un resultado (o devuelve Unit).
- Se utiliza generalmente para tareas de disparar y olvidar, donde no necesitas el resultado de la tarea.
- launch devuelve una referencia a Job, que se puede utilizar para cancelar la ejecución de la corrutina.
- Un ejemplo común de su uso sería una corrutina que realiza una tarea de actualización de UI o alguna tarea de fondo que no necesita devolver datos. async:

async
* Se utiliza para iniciar una corrutina que devuelve un **resultado**. Es similar a launch, pero está diseñado para tareas que producen un valor. * async devuelve una referencia a `Deferred`, que es una promesa de un resultado futuro. Puedes obtener el resultado llamando a `await()` en el objeto Deferred. * Si una corrutina iniciada con async falla, la excepción se almacena en el objeto Deferred y se lanza cuando se llama a .await(). * Un uso común de async sería para realizar una operación de red o calcular un valor que se necesitará más adelante.

97.8.7 Conceptos trabajando con corrutinas

Ver aqui

97.8.7.1 Job

Cuando inicias una corrutina con la función launch(), se muestra una instancia de Job. Dicha instancia contiene un controlador o referencia de la corrutina para que puedas administrar su ciclo de vida, como por ejemplo :

```
job.cancel()
```

97.9 Apendice

Enlaces:
* Introducción a las corrutinas * Kotlin corutines Documento bien organizado sobre corutinas. * Corrutinas y retrofit codelab MarsFoto * Run a race * Corrutinas en playground de Kotlin

Versión 0.5 12-12-23 Versión 0.8 8-1-24