**Sesión 3-01 Clase del 28 de noviembre**

En esta sesión vamos a ver porqué son necesarias las corrutinas y cómo se trabaja

*Hasta ahora, en todas las aplicaciones que hemos realizado corre síncronamente en un único hilo* ***Main Thread****. La UI (pintar componentes y realizar acciones de entrada y salida en pantalla) se ejecuta sólo sobre Main Thread. Ejecutar síncronamente significa que para comenzar a ejecutar una tarea tiene que haberse terminado la inmediatamente anterior.*

*Si hay un proceso que requiere bastante tiempo, mientras se está realizando, bloquearía la UI, es decir, el usuario no tiene interacción.*

*Actualmente, los procesos que pueden bloquear la UI se ejecutan mediante corrutinas, antes se realizaban con otras herramientas (AsyncTask). Una corrutina permite ejecutar un proceso en un hilo independiente al Main Thread.*

1.- Inicia un nuevo proyecto y copia este código para la pantalla **HomeView()** que se lanzará desde **MainActivity.**

@Composable  
fun HomeView() {  
 // variable de estado para el color de fondo de la pantalla  
 var backgroundColor by remember **{** *mutableStateOf*(Color.Blue) **}** // variable de estado del texto informativo sobre la descarga  
 var downloadText by remember **{** *mutableStateOf*("Descarga no realizada") **}** // variable de estado del texto informativo sobre petición API  
 var apiText by remember **{** *mutableStateOf*("sin consultar API") **}** Box(  
 modifier = Modifier.*fillMaxSize*().*background*(backgroundColor),  
 contentAlignment = Alignment.Center  
 ) **{** Column(  
 horizontalAlignment = Alignment.CenterHorizontally,  
 verticalArrangement = Arrangement.spacedBy(16.*dp*)  
 ) **{** // Botón Cambiar Color  
 Button(onClick = **{** backgroundColor =   
 if (backgroundColor == Color.Blue) Color.Red else Color.Blue  
 **}**) **{** Text("Cambiar Color")  
 **}** // Botón DESCARGAR  
 Button(onClick = **{ }**)  
 **{** Text("DESCARGAR")  
 **}** //Texto sobre estado de la descarga  
 Text(text = downloadText)  
 // Botón PETICIÓN API  
 Button(onClick = **{ }**)  
 **{** Text("PETICIÓN API")  
 **}** // Texto relacionado con estado de la petición API  
 Text(text = apiText)  
 **}  
 }**}

Hay dos variables de estado que se usan en los componentes Text para mostrar estado de descarga y consumo de API. Ahora no tienen utilidad ya que no cambian, pero las usaremos posteriormente.

El botón **Cambiar Color** hace que se cambie el color de fondo de la pantalla usando una variable de estado. Ejecuta para comprobar la funcionalidad.

2.- Vamos a hacer una simulación de que, al clicar en el botón **Descargar** se inicia la descarga de un archivo que requiere bastante tiempo (pero no vamos a hacer la descarga como tal).

Para ello programamos el **onClick** del botón **Descargar** de la siguiente forma:

downloadText="Descargando ……"

Thread.sleep(5000)

downloadText="Terminada la descarga"

Si ejecutamos ahora veremos que:

* Inicialmente el botón **Cambiar Color** funciona correctamente.
* Al clicar en **Descargar**, el botón **Cambiar Color** deja de funcionar (al menos durante 5 segundos). Se ha bloqueado la UI hasta que termina **sleep(5000).**
* Al terminar los 5 segundos, se pinta el mensaje de descarga finalizada y la UI deja de estar bloqueada.
* Después se muestra un mensaje debido a una excepción debida a que la UI no está respondiendo. Ello es debido a que el sistema detectó que la UI estuvo bloqueada.

Si incrementamos el tiempo de retardo llegará a mostrarse el mensaje de que la UI no responde antes de que termine el retardo.

3.- La solución para que un proceso pesado no bloquee la UI es realizar estos procesos dentro de una corrutina (que hará que no se .

//el scope de la corrutina llama al constructor launch  
// también se usa para otros casos async  
coroutineScope.*launch* **{** // Cambiar el texto a "Descargando video"  
 downloadText = "Descargando video"  
 // Simular una espera de 10 segundos  
 delay(10000L)  
 // Cambiar el texto a "Finalizada la descarga"  
 downloadText = "Finalizada la descarga"  
**}**

En este código se está usando una variable de estado **coroutineScope** que se ha declarado anteriormente.

// variable de estado para el scope de una corrutina  
val coroutineScope = rememberCoroutineScope()

**Probamos ahora la ejecución**. Todo se estará ejecutando dentro del hilo principal de la app pero, cuando se llega a **delay** dentro del lanzador de la corrutina, se despacha ese proceso a otro hilo para que no se bloquee la UI. Tras terminar **delay**, la corrutina vuelve al hilo de la UI y permite escribir “Finalizada la descarga”.

Así **coroutineScope** seobtiene del dispatcher **Dispatchers.main** asociado al hilo en que se ejecuta la UI. Un scope de corrutina permite lanzar o construir una corrutina.

*El* ***dispatcher*** *determina en qué hilo o conjunto de hilos se ejecutará una corrutina:*

* ***Dispatchers.Main*** *en el que utiliza el hilo principal (main thread), el de la UI.*
* ***Dispatchers.IO*** *es el diseñado para tareas* ***intensivas de entrada/salida (I/O)*** *como leer o escribir archivos.*
* ***Dispatchers.Default*** *es el diseñado para operaciones* ***intensivas en CPU****, como cálculos matemáticos complejos.*

*Un* ***scope (ámbito)*** *define el* ***ciclo de vida*** *y el* ***contexto*** *de una o varias corrutinas. Sirve para: Controlar* ***cuándo empiezan*** *y* ***cuándo terminan*** *las corrutinas, asociar corrutinas a un Dispatcher, cancelar corrutinas. Hay varios tipos de scope:*

* ***GlobalScope*** *(para procesos con la duración de la aplicación).*
* ***CoroutineScope*** *(puede asociarse a cualquier componente).*
* ***rememberCoritineScope*** *(específico de JetPack Compose, está asociado a un composable, si desaparece el composable, desaparecen las corrutinas que lance).*
* ***viewModelScope*** *(para ser usado en ViewModel).*

4.- Si en la anterior corrutina, modificamos código para escribir un Logs de “iniciada nueva descarga” y “finalizada descarga” podemos comprobar que puede estar ejecutándose simultáneamente la misma corrutina varias veces.

Si quisiéramos evitar esto, tendríamos que deshabilitar el botón de **Descargar** cuando se va a iniciar el retardo y habilitarlo cuando ha finalizado.

5.- Establece que cuando se clique el botón **Petición API** se genere un retardo de 5 segundo que simule que se está realizando una petición a una API. Comprueba que se pueden estar ejecutando simultáneamente los retardos producidos por **Descargar y Petición API** y que no se bloquea la UI.

6.- Agregamos a la pantalla un botón CONTADOR y un Texto con letra grande que escribirá un contador decremental de 60 a 0 que se modifica por cada segundo transcurrido.

Este es el código que debes incluir y comprender:

// Botón CONTADOR  
Button(onClick = **{** coroutineScope.*launch* **{** timerValue = 60 // Iniciar el temporizador desde 60  
 while (timerValue > 0) {  
 delay(1000L) // Esperar 1 segundo  
 timerValue-- // Decrementar en 1  
 }  
 **}  
}**) **{** Text("CONTADOR")  
**}**Text(  
 text = timerValue.toString(),  
 fontSize = 48.*sp*,  
 fontWeight = FontWeight.Bold  
)

Como ves, se está usando una variable **timerValue.** Debes declararla como variable de estado para contener **Int**.

7.- Por último, vamos a ver como se puede cancelar (detener y finalizar) una corrutina. Lo vamos a hacer clicando un nuevo botón DETENER.

// Botón DETENER  
Button(onClick = **{**/\* codigo para detener \*/**}**)   
**{** Text("DETENER")  
**}**

*Para que se pueda detener o cancelar una corrutina, hay que ejecutarla dentro de un Job. Un Job es una unidad de trabajo contenedor de corrutinas que permite controlar su ejecución. Mediante un Job podemos controlar si sus corrutinas hijas están en los posibles estados* ***isActive, isCompleted, isCancelled.*** *También podemos hacer otras operaciones sobre las corrutinas hijas como* ***cancelar****.*

Para poder usar un Job hay que declararlo como variable de estado:

// variable de stado para almacenar el Job de una corrutina  
var timerJob by remember **{** *mutableStateOf*<Job?>(null) **}**

Programamos ahora **onClick** del botón **Contador** para que la corrutina del temporizador se controle dentro del **Job timerJob:**

// Cancelar la corrutina previa si existe  
timerJob?.cancel()  
timerJob = coroutineScope.*launch* **{** timerValue = 60 // Iniciar el temporizador desde 60  
 while (timerValue > 0) {  
 delay(1000L) // Esperar 1 segundo  
 timerValue-- // Decrementar en 1  
 }  
**}**

*La función* ***timerJob?.cancel()*** *en principio no está realizando la cancelación de la corrutina la primera vez que se clica en CONTADOR, ya que no está activa en ese momento. La función que tiene ahí es que, si hemos clicado CONTADOR y, sin terminar el temporizador volvemos a clicar , se cancela la corrutina activa y se reinicia el temporizador a 60.*

Para detener o cancelar la corrutina clicando el botón DETENER haya que programar en el **onClick** de ese botón:

timerJob?.cancel() // Cancelar la corrutina hija  
timerJob = null // Limpiar el Job

8.- Como RETO te propongo implementar una pantalla en la que se va mostrando cada 5 segundo una imagen de una lista de imágenes. Cuando se llega a la última imagen de la lista, se vuelve a empezar con la primera.

*Próximamente usaremos corrutinas para acceder a bases de datos locales, descargar ficheros, consumir APIs y otros muchos procesos. En esos casos, las corrutinas las desarrollaremos dentro de un* ***ViewModel****, como es recomendable hacerlo*.