Threading

Por cada app que se lanza, Android crea un thread llamado **main**, que se encarga de procesar eventos de entrada (touch) y de dibujar componentes en pantalla.

Cuando requerimos un trabajo intensivo este thread no nos alcanza porque en caso contrario estaríamos bloqueándolo impidiendo dibujar en pantalla o procesar más eventos de entrada.

//Esto introduce el concepto de *tareas asincrónicas*.

la UI de Android no maneja la UI desde otro thread que no sea el *main.*

Entonces tenemos 2 reglas que cumplir:

* No bloquear el *main-thread* (Si lo hacemos por más **de 5 segundos**, obtendremos un **ANR** )
* No acceder a la UI desde otro thread que no sea el *main (****caso contrario excepción****)*

Las soluciones que existen son :

AsyncTask ***( ESTA DEPRECADO*** )

* Era la solución oficial al problema de tareas asincrónicas
* Está incluido en el SDK de Android
* Para implementarlo, necesitábamos extender de la clase AsyncTask<A,B,C>

*RxJava*

* Introduce la programación reactiva a la solución de tareas asincrónicas.
* Combina ideas del patrón *Observer*, el patrón *Iterator* y la programación funcional.
* Es una librería disponible en varios lenguajes: RxJava, RxJS, RxSwift, etc

*Lo MALO de rxJAVA :*

* *Es difícil de aprender*
* *El mal uso de los operadores puede conducir a un código muy difícil de comprender*
* *Es difícil de debuggear*

*lo BUENO de RXJAVA :*

* *Es una solución robusta y ampliamente utilizada, que soluciona muchos de los problemas de AsyncTask ( esta deprecado el asyntask)*
* *Elimina los callbacks*
* *Ofrece gran flexibilidad a través de sus operadores*
* *Integraciones con librerías populares (LiveData, Room, Retrofit)*

*La MEJOR SOLUCION ES*

***Coroutines***

Son esencialmente light-weight threads (hilos ligeros) que **nos permiten** escribir **código asincrónico de una manera secuencial**

**//POSEEN = suspending functions , Scope , *Jobs y Dispatcher***

**Son mas livianas que los threads**

* Su programación es más sencilla, de manera secuencial (¡¡no más callbacks!! 🎉)
* Son una implementación a nivel lenguaje => El SO no las conoce
* Permiten ejecutar suspending functions

**suspending functions**

* *Son funciones que se pueden suspender y continuar más adelante*
* *Se ejecutan dentro de una coroutine o dentro de otra suspending function*
* *Deben ser seguras, capaces de ser llamadas* ***sin bloquear el thread que las invoca***
* *Internamente, el compilador traduce el modificador suspend en una función con* ***callbacks*** *utilizando una* ***máquina de estados finitos***

*//* *La función get(“...“) es también una suspending function*

**Scope**

* *define el* ***contexto*** *sobre el que van a ejecutarse las coroutines*
* ***Las coroutines lanzadas por el scope*** *heredan su contexto*
* *Toda aplicación tiene al menos un scope asociado, llamado GlobalScope, que permite ejecutar coroutines durante todo el tiempo de vida de la aplicación*
* ***Los 2 más comunes son launch { } y async { }***

*launch { }: Crea una nueva coroutine sin bloquear el thread actual y devuelve una referencia a la coroutine como un Job*

*async { }: Crea una nueva coroutine sin bloquear el thread actual y nos devuelve una referencia a un Deferred value (que es un Job también). Se utiliza para obtener un valor como resultado de una ejecución, mediante un llamado a .await().*

***Jobs***

* Un Job representa a una coroutine
* Es el encargado del ciclo de vida, cancelación y relación parent-child de las coroutines

***Dispatcher***

* ***Determina que thread o threads va a utilizar la coroutine para su ejecución***
* ***Existen varios dispatchers que podemos utilizar:***
* **Dispatchers.Default: Es el utilizado por defecto si no se especifica otro**
* **Dispatchers.Main: Está relacionado al Main Thread que opera con objetos de UI. Depende de la plataforma, hay que agregar la dependencia correspondiente.**
* **Dispatchers.Unconfined: No limita a las coroutines a ningún thread en particular. Continúa en el thread de quien llamó a su resume().**
* **Dispatchers.IO: Utilizado para realizar operaciones de Entrada/Salida mediante un pool de threads compartido.**