ROOM [KOTLIN]

Room es una librería de Android que forma parte de **Android Jetpack**. Está diseñada para trabajar con bases de datos **SQLite** de una forma más sencilla y eficiente. A diferencia del 'método tradicional', Room no requiere de sentencias SQL complicadas ni clases de acceso complejas. Utiliza **anotaciones** y **clases Kotlin** para definir tablas y consultas. Esto reduce errores, mejora el rendimiento y hace el código más legible y mantenible.

Para usar **Room** lo primero es añadir las dependencias en el **Gradle**. Esto se hace a nivel de módulo, añadiendo lo siguiente. Nótese que estoy utilizando Kotlin DSL, no Groovy.

Añadimos el plugin que habilita las anotaciones:

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)

    // Enables support for annotation processors
    id("kotlin-kapt")
}
```

Y después, las dependencias.

```
    // ROOM
    implementation(libs.androidx.room.runtime)
    // Annotation processor
    kapt(libs.androidx.room.compiler)
    // Kotlin Extensions
    implementation(libs.androidx.room.ktx)
```

Nótese también que estoy utilizando el fichero **lib.versions.toml** para centralizar el tema de las dependencias, por lo que habría que añadir en él:

```
[versions]
roomRuntime = "2.6.1"

[libraries]
androidx-room-ktx = { module = "androidx.room:room-ktx", version.ref = "roomRuntime" }
androidx-room-compiler = { module = "androidx.room:room-compiler", version.ref = "roomRuntime" }
androidx-room-runtime = { module = "androidx.room:room-runtime", version.ref = "roomRuntime" }
```

Room tiene tres componentes (clases) principales:

- **Entity**: Representa una tabla dentro de la base de datos
- **DAO (Data Access Object)**: Interfaz con las operaciones (métodos) sobre la BBDD.
- **RoomDatabase**: Clase Abstracta que relaciona las Entidades y los DAO.

Para crear una Base de Datos con Room se siguen estos pasos:

1) Creas las entidades (@Entity), una por cada tabla
2) Define los DAOs (@Dao), uno por cada tabla como norma general
3) Configura la base de datos (@Database)

Es habitual que la clase de base de datos siga el Patrón Singleton para garantizarnos que en nuestra App sólo va a haber una instancia de la misma.

Entidades

Room utiliza el patrón ORM para mapear las tablas de la base de datos en Kotlin. Gracias a esto, solo tendremos que definir una **data class** y marcarla como **@Entiy**. Esta **anotación** identifica a la data class como una entidad y, por tanto, como una tabla en la BBDD.

Cuando se genere la tabla, Room se encargará automática y de forma transparente de transformar las entidades en tablas, sus atributos en columnas, los tipos de datos, etc. Cada una de estas anotaciones en realidad es código que es leído por las librerías de Room, y cada una tiene un significado o función diferente.

```kotlin
@Entity(tableName = "t_enterprises")
data class Enterprise(
    @PrimaryKey (autoGenerate = true)
    var id: Long = 0,
    var nombre: String,
    var localizacion: String
)
```

En el ejemplo, indicamos con la anotación **@Entity** el nombre de la tabla que será creada a partir de esta data class. Mediante **@PrimaryKey** anotamos la columna id como la clave primaria de la tabla, y además, le indicamos que es auto incremental.

Existen otras anotaciones para añadir. Algunas requieren parámetros adicionales, otras no. A modo de resumen:

- **@ColumnInfo**: define atributos adicionales de una columna de la tabla, como asignarle un nombre diferente, un índice o un valor por defecto.
- **@Ignore**: previene que el atributo se almacene en la base de datos.
- **@Index**: genera un índice sobre la columna
- **@ForeignKey**: Define una clave externa sobre una clave primaria de otra Entidad.

A modo de ejemplo:

```
@Entity(tableName = "t_enterprises")
data class Enterprise(
    @PrimaryKey (autoGenerate = true)
    var id: Long = 0,
    var nombre: String,
    @ColumnInfo( name ="area")
    var localizacion: String,
    @Ignore
    var picture: Bitmap?
)
```

DAO

En general, lo más sencillo es crear un DAO por cada entidad; pero no tiene por qué ser así. Un DAO será una interfaz en la que especificaremos los métodos / consultas que podremos hacer sobre una tabla. De nuevo, la clase y los métodos van a tener que anotarse para poder usarlos después.

```
@Dao
interface EnterpriseDao {

    // Put here all the methods you want to

    @Query ( value = "select * from t_enterprises order by id")
    fun getAll () : List <Enterprise>
```

Anotamos la interfaz mediante **@DAO**. A continuación, definimos el método que queremos lanzar. En el ejemplo, getAll () es una función que nos retornará una lista de Empresas. En este caso, hemos decidido especificar la sentencia SQL que tiene que ejecutar Room mediante la anotación **@Query**. Por tanto, cuando invoquemos, getAll (), se va a ejecutar dicha sentencia sobre la tabla correspondiente.

```
    @Insert
    fun insertAll(vararg enterprises: Enterprise)

    @Delete
    fun delete(enterprise: Enterprise)
```

De nuevo, hay toda una serie de anotaciones para utilizar en los DAO. Por ejemplo, **@Query**, **@Insert**, **@Update** y **@Delete**. No existe limitación en cuanto al número de veces que puedas emplear cada anotación, pero un mismo método no puede tener diferentes anotaciones.

Database

Una vez definidas las Entidades y los DAO, sólo falta definir la base de datos que los relacione y permita realizar consultas contra la Base de Datos. Por simplicidad, te muestro aquí la clase **MyRoomDatabase** con el Patrón Singleton ya implementado:

```
@Database (entities = [Enterprise::class], version = 1)
abstract class MyRoomDatabase : RoomDatabase() {

    // Singleton pattern. We want ONLY one instance to the ROOM Data Base
    companion object {  11 Usages

        @Volatile  3 Usages
        private var instance : MyRoomDatabase? = null

        private val LOCK = Any()  1 Usage

        operator fun invoke (context:Context) = instance?: synchronized( lock = LOCK){
            instance?:buildDatabase (context).also { instance = it}
        }

        private fun buildDatabase (context: Context) = Room.databaseBuilder(context,
         klass = MyRoomDatabase::class.java,
         name = "myDataBase")
         .build()
    }

    // How many tables/Dao you have? Add another row here for each one
    abstract fun getEnterpriseDao () : EnterpriseDao  10 Usages  1 Implementation

}
```

La anotación **@Database** define de cuántas entidades se compone la base de datos. En el ejemplo, solamente tenemos una: Enterprise. Por último, al final del todo, se debe crear un método abstracto por cada DAO que vayamos a utilizar. Y ya estaría hecha la configuración de ROOM. Sólo queda aprender a usarla.

El motivo por el que usamos Singleton es porque instanciar esta clase es costoso. Y, dado que solo necesitamos una instancia mientras la App esté en funcionamiento, usamos este patrón. No obstante, puedes prescindir de él.

Usando ROOM

Para poder usar ROOM dentro de una Activity, bastaría con instanciar MyRoomDatabase, escoger el DAO correspondiente y después el método que necesitemos. Pero, no es tan sencillo: el acceso a base de datos con Room se tiene que hacer siempre como una tarea asíncrona**.**

Para ello, cada vez que lancemos la llamada a un método de un DAO, necesitamos utilizar la propiedad **lifecycleScope**. Se utiliza para **lanzar corrutinas** que se gestionan automáticamente según el ciclo de vida de un componente (la Actividad). Una **corrutina** es por tanto una forma **ligera y eficiente** de ejecutar tareas **asíncronas** en Kotlin.

```
val db = MyRoomDatabase( context = this)

lifecycleScope.launch( context = Dispatchers.IO) {
    // We get the EnterpriseDao and then call to getAll method
    val list = db.getEnterpriseDao().getAll()
    if (list.isEmpty()) {

        // Empty, so we add a few...
        db.getEnterpriseDao().insertAll(
            ...enterprises = Enterprise(nombre = "Elorrieta Inc.", localizacion = "Education"),
            Enterprise(nombre = "TechNova", localizacion = "Software")
        )
    }
}
```

En este ejemplo, estamos lanzando el método getAll () del DAO para que nos retorne una lista de todas las empresas; y si no hay ninguna, carga dos nuevas en su tabla correspondiente. Esta técnica es muy usada para precargar con información una base de datos al iniciarse una App (onCreate ()).

En este otro ejemplo, además, actualizamos el **adapter** de la **RecyclerView** de forma similar para que, una vez retornado el listado de todas las empresas, se le envíe una señal que permita al RecyclerView volver a cargarse con los cambios de la BBDD.

```
lifecycleScope.launch( context = Dispatchers.IO) {
    // We get the EnterpriseDao and then call to getAll method
    val list = db.getEnterpriseDao().getAll()

    // We do a Mutable List
    val enterpriseList = list.toMutableList()

    // Again, Coroutine stuff. This way, we can update the adapter
    launch( context = Dispatchers.Main) {
        // We update the adapter
        enterprisesAdapter.update( newEnterprises = enterpriseList)
    }
}
```

Dispones de varios ejemplos más y un adapter en el fichero: **UD6 - RoomEmpresas**

Práctica 41

[Kotlin] Realiza una app que sea capaz de almacenar sus canciones favoritas. Se guardará el título de la canción, el autor, y la URL de YouTube donde se puede escuchar. La app debe permitir listar todas las canciones, añadir una canción, borrar una canción y modificar sus datos. Las canciones deberán aparecer en formato lista.

Práctica 42

[Kotlin] Lo mismo que antes, pero al pulsar una canción de la lista se hará un *intent* y se reproducirá la canción en el navegador.

Práctica 43

[Java] Lo mismo que antes, pero al pulsar una canción de la lista se hará un *intent* y se reproducirá la canción en otro activity que tendrá un componente **Media Player**. Este ejercicio propuesto es para que investigues un rato si ya has terminado otras tareas. Esto lo veremos más adelante. Que te diviertas.