
GALAXY
TRAINING

ANDROID DESDE 0

Sesión 4

Ing. Marco Estrella
Instructor en Tecnologías Java y Android
Github @jmarkstar

TEMAS A TRATAR

SharedPreferences

SQLiteOpenHelper

SQLiteDatabase

Cursores

Alertas con Anko

Enviar objetos a otra pantalla con @Parcelize

SharedPreferences

Los SharedPreferences son almacenes de tipo clave valor incorporado en el app.

El almacén es un archivo XML.

El SharedPreferences no es seguro, por ende no se debería guardar datos sensibles.

```xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="KEY_STRING">valor de la cadena</string>
    <string name="KEY_ENTERO">1000</string>
    <string name="KEY_DECIMAL">500.50</string>
</map>
```

Crear y/o obtener un SharedPreferences

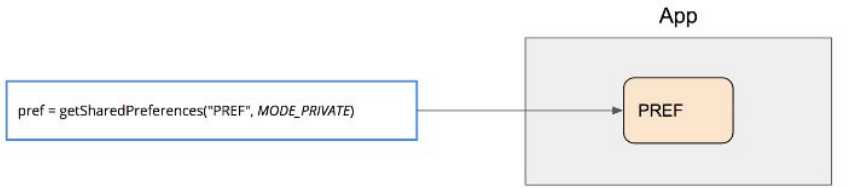
La función **getSharedPrefences()** del activity crea o obtiene un SharedPreferences.

Consta de 2 parámetros; name y privacidad.

name.- es el nombre del SharedPreferences, se podrá usar este nombre para poderlo usar en otro activity.

privacidad.- lo recomendable es que siempre sea PRIVATE

```
val pref = getSharedPreferences( name: "PREF1", Context.MODE_PRIVATE)
```

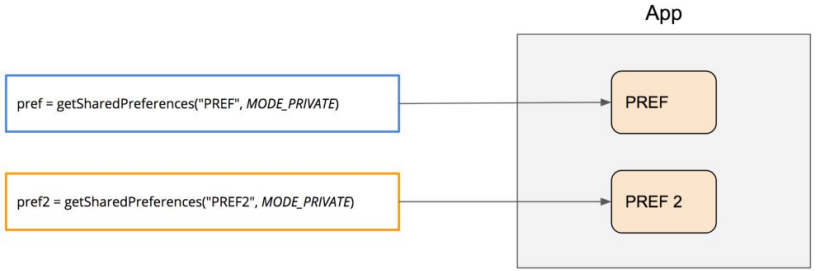


SharedPreferences

Se puede crear varios SharedPreferences en un solo app, solo hay que diferenciarlos por el nombre.

```
val pref = getSharedPreferences( name: "PREF1", Context.MODE_PRIVATE)

val pref2 = getSharedPreferences( name: "PREF2", Context.MODE_PRIVATE)
```

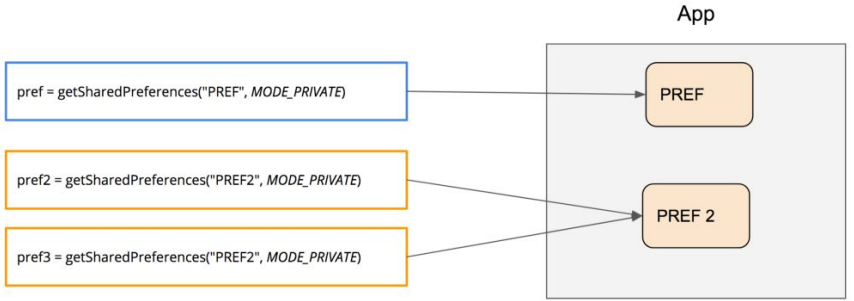


Si un SharedPreferences es creado en un activity, tambien podra ser usado en otros activities.

```
val pref = getSharedPreferences( name: "PREF1", Context.MODE_PRIVATE)

val pref2 = getSharedPreferences( name: "PREF2", Context.MODE_PRIVATE)

val pref3 = getSharedPreferences( name: "PREF2", Context.MODE_PRIVATE)
```



Estructura Interna

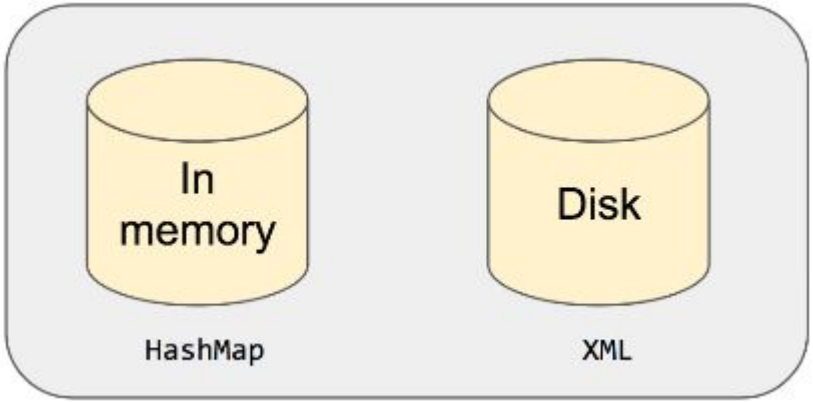
Un SharedPreferences internamente tiene un almacén en memoria ram y otro en el memoria rom. Cada operación que se realice(registrar, obtener, eliminar, actualizar) primero va a la memoria ram y luego al disco rom si es necesario.

El almacén en disco duro es un archivo XML.

```xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="KEY">value</string>
</map>
```

El almacén en memoria ram es un HashMap

```
                //<clave, valor>
val map = HashMap<String, Any>()
```



SharedPreferences.Editor

Registrar, Modificar, Eliminar items

La clase SharedPreferences tiene un método llamado **edit()**. Este método retorna la instancia del objeto **SharedPreferences.Editor**, esta clase es quien nos permite hacer aquellas operaciones con los siguientes métodos.

Estos métodos crean o editan un ítem.

putBoolean(clave: String, valor: Boolean)

putFloat(clave: String, valor: Float)

putInt(clave: String, valor: Int)

putLong(clave: String, valor: Long)

putString(clave: String, valor: String)

putStringSet(clave: String, valor: Set<String>)

Registrando un solo ítem.

```
pref.edit().putString("nombreProducto", nombreProducto).apply()
```

Registrando varios ítems.

```
with(pref.edit()){ this: SharedPreferences.Editor!
    putString("nombreProducto", nombreProducto)
    putInt("cantidad", cantidad.toInt())
    putFloat("precio", precio.toFloat())
    apply()
}
```

Para eliminar un ítem se usa la función **remove(clave: String)**.

```
with(pref.edit()){ this: SharedPreferences.Editor!
    remove( key: "precio")
    apply()
}
```

o

```
pref.edit().remove( key: "nombreProducto").apply()
```

Para eliminar todos los items de un SharedPreferences se tiene que usar la función clear()

```
pref.edit().clear().apply()
```

Función **commit()** permite guardar los cambios realizados de forma síncrona.

```
with(pref.edit()){ this: SharedPreferences.Editor!
    putString("nombreProducto", nombreProducto)
    val success:Boolean = commit()
}
```

Función **appy()** permite guardar los cambios realizados de forma asíncrona.

Referencia: https://developer.android.com/reference/android/content/SharedPreferences.Editor

Obtener los valores de los ítems guardados

La clase SharedPreferences tiene métodos para obtener los datos guardados según tipo y son los siguientes.

getBoolean(clave: String, defValor: Boolean)

getFloat(clave: String, defValor: Float)

getInt(clave: String, defValor: Int)

getLong(clave: String, defValor: Long)

getString(clave: String, defValor: String)

getStringSet(clave: String, defValor: Set<String>)

Nota: defValor es el valor por defecto que si en caso la clave no existe, el método retornada ese valor.

```
val nombreProducto = pref.getString( key: "nombreProducto",  defValue: "")
val cantidad = pref.getInt( key: "cantidad",  defValue: -1)
val precio = pref.getFloat( key: "precio",  defValue: -1.0f)
```

Función **contains**(clave: String)

retorna un booleano en caso exista o no clave en el SharedPreferences.

```
if(pref.contains("nombreProducto")){
    // código en caso si exista esa clave
}else{
    //código en caso no exista esa clave
}
```

Función **getAll**() o solo **all**

retorna un Map con todos los ítems con su clave y valor del SharedPreferences.

```
val map = pref.all

for ( (clave, valor) in map ){
    println("$clave - $valor \n")
}
```

android.database.sqlite

Fue iniciado en el año 2000.

Soporta tablas, índices, triggers y views.

Es un archivo que puedes copiar entre sistemas de 32-bit y 64-bit.

Pesa aproximadamente 500 kb.

Es transaccional(ACID).

Las transacciones ACID pueden ser interrumpidas por errores del app o del sistema y/o problemas de energía.

La librería SQLite es parte del core de la plataforma android. Las aplicaciones y content providers usan la librería **android.database.sqlite** para accederlo.

Es un paquete donde están las clases que permitirán administrar la base de datos privada para una Aplicación.

La versión de sqlite depende de la versión de android.

Android API	SQLite Version
API 27	3.19
API 26	3.18
API 24	3.9
API 21	3.8
API 11	3.7
API 8	3.6
API 3	3.5
API 1	3.4

Los apps pueden compartir su información privada Usando un Content Provider.

Sucede que a veces vamos a recibir información de otras apps. En este caso usaremos las clases del paquete **android.database** pero este tema no está en el alcance del curso.

Las clases más importantes de esta librería son SQLiteOpenHelper y SQLiteDatabase.

SQLiteOpenHelper

Es una clase de ayuda para administrar la creación de una base de datos y sus actualizacionaciones.

Si queremos una base de datos para nuestro app, esta es la clase por la cual debemos empezar a implementar.

Para ello, tenemos que crear una clase y extender de **SQLiteOpenHelper.**

Cuando extendamos nuestra clase. Inmediatamente android studio nos dirá que debemos **sobreescribir 2 métodos y son los siguientes:**

onCreate(db: SQLiteDatabase)

onUpdate(db SQLiteDatabase, versionAntigua: Int, nuevaVersion: Int)

SQLiteOpenHelper

Además tenemos que pasarle datos a la clase SQLiteOpenHelper por su constructor.

El resultado deberia ser asi:

```
class DBHelper(context: Context) : SQLiteOpenHelper(context,
                                                    name: "sqlite.db",
                                                    factory: null,
                                                    version: 1) {

    override fun onCreate(db: SQLiteDatabase) {
        //escribir codigo
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        //escribir codigo
    }
}
```

La clase se llama DBHelper.

EL constructor de mi clase DBHelper pide un argumento; el **Context**.

Se ha extendido de la clase SQLiteOpenHelper.

La clase **SQLiteOPenHelper** require 4 argumentos:

context: es el contexto que recibimos por DBHelper.
name: es el nombre de la base de datos.
factory: cursores personalizados(null por defecto).
version: es la versión de la base de datos.

Finalmente se implementó los métodos onCreate y onUpdate,

SQLiteOpenHelper

El método onCreate() se ejecuta la primera vez que la aplicación es usada; después de su instalación.

El método onUpdate() se ejecuta en la actualizaciones de la apps.

Vamos a crear una tabla en onCreate() y quedaria asi:

Ejecutamos una sentencia sql con el método execSQL de la clase SQLiteDatabase.

```
class DBHelper(context: Context) : SQLiteOpenHelper(context,
                                                    name: "sqlite.db",
                                                    factory: null,
                                                    version: 1) {

    private val SQL_CREAR_TABLA_USUARIO =
            "CREATE TABLE usuario (" +
                    "id INTEGER PRIMARY KEY AUTOINCREMENT," +
                    "nombre INTEGER," +
                    "genero TEXT," +
                    "email TEXT," +
                    "celular TEXT," +
                    "direccion TEXT)"

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(SQL_CREAR_TABLA_USUARIO)
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        //No es cubierto por el curso
    }
}
```

Después de haber creado una base de datos. La idea es usarlo en nuestro app, para poderlo abrir podríamos usar alguno de estos métodos.

```
private var db: SQLiteDatabase = DBHelper(context).readableDatabase
```

readableDatabase
Abre la base de datos en modo lectura; sólo funcionarán los select.

```
private var db: SQLiteDatabase = DBHelper(context).writableDatabase
```

writableDatabase
Abre la base de datos donde podemos hacer operaciones transaccionales.

Estos 2 métodos retornan un objeto de tipo **SQLiteDatabase.**

close() Cierra cualquier objeto de base de datos.

Expone métodos para administrar la base de datos.

Tiene métodos para crear, modificar y eliminar tablas.

Ejecutar comandos SQL.

Insertar, actualizar, eliminar y consultar registros.

El nombre de la base de datos, debe ser único dentro de la aplicación, no atreves de todas
Las apps en el equipo.

El método execSQL(sql: String) ejecuta una simple sentencia SQL que NO sea un SELECT o alguna sentencia que retorne data. Es ideal para crear, modificar y eliminar tablas, crear índices, etc.

Nota: Si ejecutamos algún INSERT o UPDATE de registros, funcionará pero no nos retornará la cantidad de registros afectados. Los registros afectados nos permite saber si nuestra sentencia SQL funcionó o no.

En resumen, no es ideal para SELECT/INSERT/UPDATE/DELETE.

Para insertar registros es recomendado usar el método **insert()** y para la actualización de datos el método **update()**.

Insert(nombreTabla : String, nullColumnHack : String , values: ContentValues) es el método conveniente para hacer inserciones de filas en la base de datos.

update(nombreTabla : String, values : ContentValues , sentenciaWhere : String , argumentosWhere Array<String>) es el método conveniente para hacer actualizaciones de datos en la base de datos.

delete(nombreTabla : String, sentenciaWhere : String , argumentosWhere Array<String>) es el método conveniente para eliminar filas en la base de datos.

isOpen() retorna true si la base de datos está actualmente abierta.

isReadOnly() retorna true si la base de datos está abierta en modo lectura.

ALERTAS CON ANKO

Como primer paso tenemos que importar la librería anko:

```
implementation "org.jetbrains.anko:anko-commons:0.10.5"
implementation "org.jetbrains.anko:anko-design:0.10.5"
```

Luego Agregamos el Alerta:

```
alert("Mensaje", "Titulo") {

    isCancelable = false

    positiveButton("ACEPTAR", {
        toast("click positiveButton")
    })

    negativeButton("CANCELAR", {
        longToast("click negativeButton")
    })
}.show()
```

PARCELABLE

Es una interface que tenemos que implementar en la clase que queremos pasar entre pantallas.

Si es necesario pasar un objeto entre pantallas y tienes que escoger entre Serializable y Parcelable, google recomienda usar Parcelable por que es mas optimo al serializar y deserializar los datos.

Al lado derecho esta un ejemplo de una clase implementando Parcelable.

Es mucho codigo verdad? hay formas de como ahorrar tiempo y código.

```
data class UsuarioModel(val id: Int? = null,
                        val nombre: String,
                        val genero: Int,
                        val email: String,
                        val celular: String,
                        val direccion: String) : Parcelable {

    constructor(source: Parcel) : this(
            source.readValue(Int::class.java.classLoader) as Int?,
            source.readString(),
            source.readInt(),
            source.readString(),
            source.readString(),
            source.readString()
    )

    override fun describeContents() = 0

    override fun writeToParcel(dest: Parcel, flags: Int) = with(dest) { this: Parcel
        writeValue(id)
        writeString(nombre)
        writeInt(genero)
        writeString(email)
        writeString(celular)
        writeString(direccion)
    }

    companion object {
        @JvmField
        val CREATOR: Parcelable.Creator<UsuarioModel> = object : Parcelable.Creator<UsuarioModel> {
            override fun createFromParcel(source: Parcel): UsuarioModel = UsuarioModel(source)
            override fun newArray(size: Int): Array<UsuarioModel?> = arrayOfNulls(size)
        }
    }
}
```

@Parcelize

Es una anotación que fue agregado en la versión 1.1.4 de kotlin.

Generar código automáticamente.

Lo que tenemos que hacer es lo sgte:

1 Habilitar las características experimentales de android extensions en el build.gradle del módulo.

sqlite | sharedpreferences | MaterialABSesion4

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {...}

androidExtensions {
    experimental = true
}

dependencies {...}
```

2 Agregar la anotación en la clase e implementar Parcelable.

```
@Parcelize
data class UsuarioModel(val id: Int? = null,
                        val nombre: String,
                        val genero: Int,
                        val email: String,
                        val celular: String,
                        val direccion: String) : Parcelable
```

Thank you!
Questions?