PERSISTENCIA DE DATOS

Pág. 165 y sgts. del Manual de SGOliver

1. INTRODUCCION

- En Android existen tres formas de almacenar información de forma permanente:
  - Mediante lo que se llaman "**preferencias compartidas**", lo que permite almacenar conjuntos sencillos de datos en forma de pares clave/valor.
  - Sistemas tradicionales de archivos.
  - **Sistema de base de datos relacional (base de datos SQLite**).

2. PREFERENCIAS COMPARTIDAS

Más información en https://developer.android.com/guide/topics/data/data-storage.html#pref

- Android proporciona la clase ***SharedPreferences*** que permite guardar datos sencillos de una forma simple.
- Los datos se guardan en **parejas *clave/valor***, es decir, cada dato estará compuesto por un **identificador** único y un **valor** asociado a dicho identificador, de forma similar a como se hacía con los objetos de tipo ***Bundle***, salvo que en este caso se almacenan de forma permanente.
- Podemos crear múltiples objetos ***SharedPreferences*** pero disponemos de uno por defecto al que accederemos mediante el método ***getDefaultSharedPreferences()*** de la clase ***PreferenceManager.*** Este método recibe como parámetro el contexto de la actividad.

`static SharedPreferences`	`getDefaultSharedPreferences(Context context)` Gets a SharedPreferences instance that points to the default file that is used by the preference framework in the given context.

```
SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(MainActivity.this);
```

- No podemos modificar el contenido del objeto ***SharedPreferences*** directamente sino que tenemos que asociar dicho objeto a un editor de tipo ***SharedPreferences.Editor***, mediante el método ***edit()***.

abstract SharedPreferences.Editor	edit() Create a new Editor for these preferences, through which you can make modifications to the data in the preferences and atomically commit those changes back to the SharedPreferences object.

```
SharedPreferences.Editor editor = prefs.edit();
```

- Una vez obtenida la referencia al editor, utilizaremos los métodos ***put()*** correspondientes al tipo de datos de cada preferencia (por ejemplo, ***putString()*** o ***putInt()***) para insertar o actualizar su valor.

```
editor.putString("email", "yo@email.com");
editor.putString("nombre", "Yo");
```

- Finalmente, una vez completados todos los datos necesarios, ejecutaremos el método ***apply()*** para almacenarlos (también se puede usar ***commit()***).

abstract void	apply() Commit your preferences changes back from this Editor to the SharedPreferences object it is editing.
abstract SharedPreferences.Editor	clear() Mark in the editor to remove *all* values from the preferences.
abstract boolean	commit() Commit your preferences changes back from this Editor to the SharedPreferences object it is editing.

```
editor.apply();
```

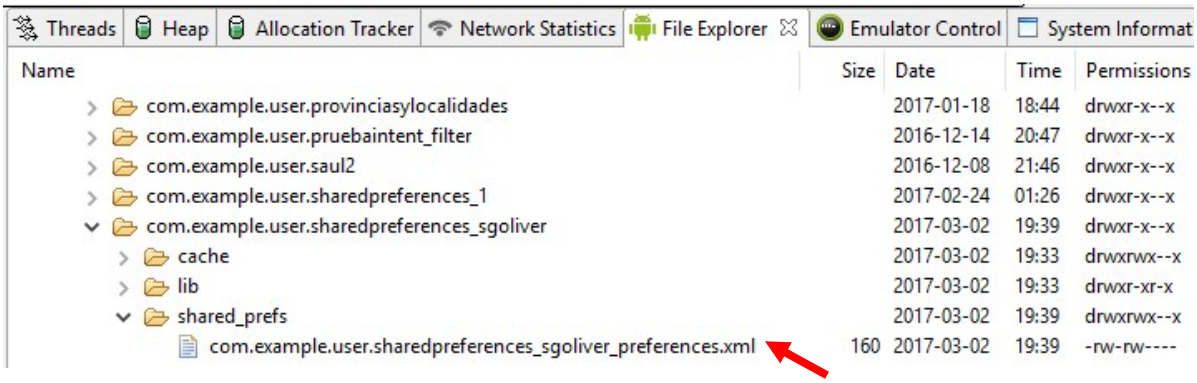
- Para leer el estado no es necesario el editor, sino que basta con usar el método ***get()*** (de la clase ***SharedPreferences***) correspondiente al tipo de dato que queramos leer, por ejemplo ***getString()***. El método ***get()*** recibe dos parámetros: la clave de la preferencia que queremos recuperar y un valor por defecto que será devuelto por el método en caso de que no se encuentre el dato con esa clave.

```
String correo=prefs.getString("email", "email_defectivo@email.com");
```

- Las preferencias compartidas no se almacenan en archivos binarios (como por ejemplo las bases de datos SQLite), sino en archivos XML. Estos ficheros XML se guardan en una ruta que sigue el siguiente patrón:

  /data/data/nombre_del_paquete/shared_prefs/nombre_preferencias.xml

  Podemos comprobarlo accediendo al explorador de ficheros con la herramienta ***Android Device Monitor***, como se muestra en la captura siguiente:




```
1  <?xml version='1.0' encoding='utf-8' standalone='yes' ?>
2  <map>
3  <string name="nombre">Luly Vázquez</string>
4  <string name="email">luly@email.com</string>
5  </map>
```

- Podemos probar esto en el **Ejercicio1**.

3. PREFERENCIAS COMPARTIDAS CON LA CLASE PreferenceActivity

- Las preferencias compartidas nos pueden permitir gestionar fácilmente las opciones de una aplicación creando los objetos necesarios y añadiendo o recuperando los valores de dichas opciones a través de los métodos correspondientes (***putString()*** – ***getString()***; ***putInt()*** – ***getInt()***…).

- Sin embargo, Android dispone de una forma alternativa para definir un conjunto de opciones para una aplicación y crea por nosotros las pantallas necesarias para permitir al usuario que modifique dichas opciones.

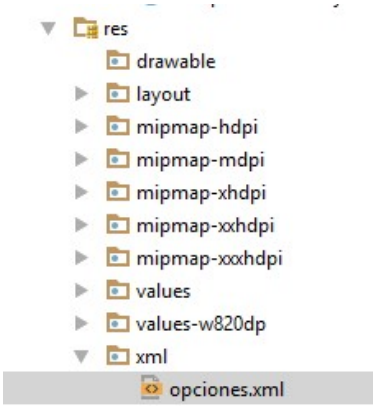
- Para ello, creamos una pantalla que va a contener las preferencias de nuestra aplicación. Esto se puede hacer usando fragmentos o, como vamos a hacer nosotros, creando una **Activity que hereda de la clase *PreferenceActivity***. Dicha Activity sobreescribe el método onCreate() incluyendo una llamada al método ***addPreferencesFromResource()***, que tiene como parámetro el fichero XML en el que hemos definido la pantalla de opciones.

```
public class OpcionesActivity extends PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.opciones);
    }
}
```

- Esta actividad, al heredar de ***PreferenceActivity***, se encargará por nosotros de crear la interfaz gráfica de nuestra lista de opciones según la hayamos definido en el XML

y también de mostrar, modificar y guardar las opciones cuando sea necesario tras la acción del usuario.

- El fichero XML que contiene la vista de nuestras preferencias debe ir en la carpeta ***res/xml***.



- En primer lugar, el elemento raíz contenedor de nuestras preferencias será ***<PreferenceScreen>***. Este elemento representará a la pantalla de opciones en sí y, a su vez, puede incorporar directamente las preferencias que queramos utilizar o podremos dividirlas en diferentes categorías. Para utilizar distintas categorías necesitamos añadir un elemento ***<PreferenceCategory>*** por cada categoría que queramos definir. Para cada categoría podemos indicar un texto descriptivo mediante el atributo ***android:title***.

- Dentro de cada categoría podemos añadir cualquier número de opciones, las cuales pueden ser de diferentes tipos:

  - **CheckBoxPreference**. CheckBox para habilitar/deshabilitar. Sus propiedades principales son:
    - **android:key**, valor interno de la preferencia.
    - **android:title**, nombre de la preferencia a mostrar.
    - **android:sumary**, breve descripción de la preferencia.

```
<CheckBoxPreference
    android:key="opcion1"
    android:title="Preferencia 1"
    android:summary="Descripción de la preferencia 1" />
```

  - **EditTextPreference**. Permite introducir valores de texto. Inicialmente muestra el nombre de la preferencia y una descripción. Al pulsar sobre ella se abre un cuadro de diálogo en el que aparece un *EditText* para que el usuario escriba el valor a almacenar. Además de los tres atributos indicados en el caso anterior, también existe **android:dialogTitle** para indicar el texto a mostrar en el cuadro de diálogo.

```
<EditTextPreference
    android:key="opcion2"
    android:title="Preferencia 2"
    android:summary="Descripción de la preferencia 2"
    android:dialogTitle="Introduce valor" />
```

  - **ListPreference**. Al pulsar sobre una opción de este tipo se mostrará la lista de valores posibles y el usuario podrá seleccionar sólo uno de ellos. Además de

los cuatro atributos comentados para los casos anteriores, para el elemento **<ListPreference>** existen dos más:

- **android:entries**, cada uno de los valores a visualizar en la lista.
- **android:entryValues**, valores internos correspondientes a cada uno de los valores de la lista que se muestra al usuario.

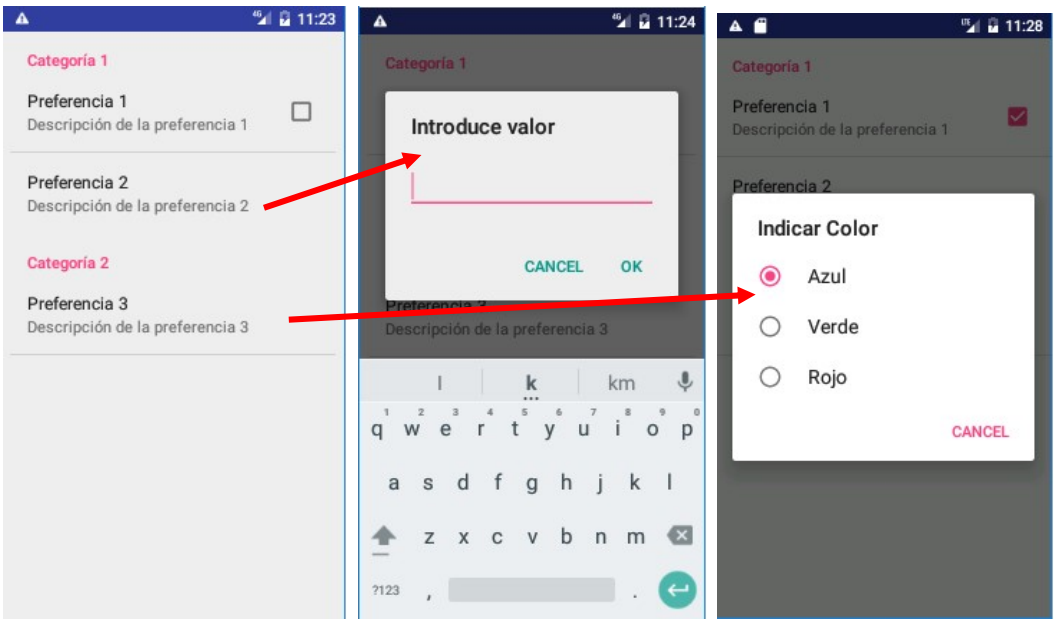
Estas listas de valores también serán ficheros XML dentro de la carpeta ***res.***

```
<ListPreference
    android:key="opcion3"
    android:title="Preferencia 3"
    android:summary="Descripción de la preferencia 3"
    android:dialogTitle="Indicar color"
    android:entries="@array/colores"
    android:entryValues="@array/codigocolores" />
```

```
<resources>
    <string-array name="colores">
        <item>Azul</item>
        <item>Verde</item>
        <item>Rojo</item>
    </string-array>
    <string-array name="codigocolores">
        <item>1</item>
        <item>2</item>
        <item>3</item>
    </string-array>
</resources>
```

- **MultiSelectListPreference**. Similar a la anterior, pero se trata de una lista de valores de selección múltiple.

- **Ejemplo:**



- Por último, habría que añadir a nuestra aplicación algún mecanismo para mostrar esta ventana de preferencias.

- Podemos probar esto como **Ejercicio 3.**

4. BASE DE DATOS SQLITE

- **SQLite** s un motor de bases de datos que se caracteriza por ser de **código libre**, ocupar **muy poco espacio**, **no necesitar servidor**, **no necesitar configuración**, y **permitir** hacer **transacciones**.

- Android incorpora todas las herramientas necesarias para la creación y gestión de bases de datos SQLite.

- La base de datos que se crea para una aplicación sólo es accesible para esta aplicación. Si se necesitase compartir datos entre diferentes aplicaciones se necesitaría trabajar con lo que se llaman "**proveedores de contenidos**" o "**Content Provider**".

- Una base de datos SQLite que se crea por medio de programación siempre se almacena en la memoria del teléfono, en la carpeta ***/data/data/nombre_del_paquete/databases***.

  Por ejemplo, si creamos una BD de nombre *miBD*, su ruta completa sería:

  /data/data/nombre_del_paquete/databases/miBD

- La forma más aconsejable de crear una BD es mediante la clase abstracta ***SQLiteOpenHelper***. Es decir, debemos definir una clase nuestra que derive de ***SQLiteOpenHelper***, y personalizarla para adaptarnos a las características de nuestra aplicación.

```
public class MiClaseParaBD extends SQLiteOpenHelper {…}
```

- La gran ventaja de utilizar esta clase es que ella se encargará de **abrir** la base de datos si existe o de **crearla** si no existe. Incluso de **actualizar** la versión si decidimos crear una nueva estructura de la base de datos. Además, esta clase tiene dos métodos: ***getReadableDatabase()*** y ***getWritableDatabase()*** que abren la base de datos en modo sólo lectura o lectura y escritura.

- La clase ***SQLiteOpenHelper*** dispone de:

  - Un constructor.
  - El método abstracto ***onCreate()***, que personalizaremos para realizar la creación de nuestra base de datos. El método *onCreate()* **será ejecutado automáticamente cuando** sea necesaria la creación de **la base de datos**, es decir, cuando **aún no exista**. Las tareas típicas que deben hacerse en este método serán la creación de todas las tablas necesarias y la inserción de los registros iniciales si fuese preciso.
  - El método abstracto ***onUpgrade()***, que permite el mantenimiento de la estructura de la base de datos en caso de que se quieran añadir o quitar tanto tablas como campos. El método *onUpgrade()* **será ejecutado automáticamente cuando** intentemos abrir **una versión** concreta de la base

de datos que **aún no exista**. Para ello, recibe como parámetros la **versión actual** de la base de datos en el sistema, y la **nueva versión** a la que se quiere convertir.

Public constructors

`SQLiteOpenHelper(Context context, String name, SQLiteDatabase.CursorFactory factory, int version)`
Create a helper object to create, open, and/or manage a database.

Public methods

`void`	`close()` Close any open database object.
`SQLiteDatabase`	`getReadableDatabase()` Create and/or open a database.
`SQLiteDatabase`	`getWritableDatabase()` Create and/or open a database that will be used for reading and writing.
`abstract void`	`onCreate(SQLiteDatabase db)` Called when the database is created for the first time.
`abstract void`	`onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` Called when the database needs to be upgraded.

- Podemos utilizar el generador de código para completar parte del contenido de nuestra nueva clase.

```
public class MiClaseParaBD extends SQLiteOpenHelper {

    public MiClaseParaBD(Context context, String name,
                         SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){
    }
}
```

- **Ejemplo**: vamos a crear una base de datos llamada "**BDUsuarios**", con una sola tabla llamada "**Usuarios**" que contenga dos campos: "**codigo**" y "**nombre**":

```
public class MiClaseParaBD extends SQLiteOpenHelper {

    //Sentencia SQL para crear la tabla de Usuarios con dos campos
    String sqlCreate = "CREATE TABLE Usuarios (codigo INTEGER PRIMARY KEY, nombre
TEXT)";

    public MiClaseParaBD(Context context, String name,
```

```
                            SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        //Se ejecuta la sentencia SQL de creación de la tabla
        db.execSQL(sqlCreate);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion){
        //Se elimina la versión anterior de la tabla
        db.execSQL("DROP TABLE IF EXISTS Usuarios");
        //Por ejemplo, se crearía la nueva versión de la tabla
        db.execSQL(sqlCreate);
    }
}
```

- Hemos utilizado el método ***execSQL()*** sobre un objeto de la clase ***SQLiteDatabase***.
- La clase ***SQLiteDatabase*** tiene métodos para crear, eliminar, ejecutar comandos SQL y realizar otras tareas comunes de administración de bases de datos.
- El método ***execSQL()*** ejecuta directamente el código SQL que le pasamos como parámetro. **No** puede ejecutar sentencias que devuelvan datos, como SELECT.

void	execSQL(String sql) Execute a single SQL statement that is NOT a SELECT or any other SQL statement that returns data.

- Una vez implementada la clase que hereda de SQLiteOpenHelper, podemos abrir la base de datos desde nuestra aplicación Android.

5. OPERACIONES SOBRE LA BASE DE DATOS

- Lo primero que debemos hacer es **crear un objeto de la clase** que hemos creado, (la **que extiende a SQLiteOpenHelper** y que, en nuestro ejemplo, hemos llamado MiClaseParaBD):

```
//crear objeto de nuestra clase, que hereda de SQLiteOpenHelper
MiClaseParaBD miClase = new MiClaseParaBD(this, "BDUsuarios", null, 1);
```

- Al crear dicho objeto le pasaremos el **contexto** de la aplicación, el nombre de la **base de datos**, un objeto ***CursorFactory*** (que normalmente tendrá el valor **null**) y, por último, la **versión** de la base de datos de nuestra aplicación. Al crear este objeto pueden ocurrir varias cosas:
  - Si la base de datos **no existe**, se llamará automáticamente al método ***onCreate()*** para crearla y conectarse a ella.

- Si la base de datos **ya existe** y su **versión actual coincide con la indicada**, se realizará la conexión con ella.
- Si la base de datos **existe** pero **su versión actual es anterior a la solicitada**, se llamará automáticamente al método ***onUpgrade()*** para convertir la base de datos a la nueva versión y, hecho eso, se conectará con la base de datos convertida.

- Una vez creada una referencia al objeto SQLiteOpenHelper, podemos invocar uno de los siguientes métodos:

  - ***getReadableDataBase()***: devuelve un objeto de tipo ***SQLiteDatabase*** sobre el que se pueden realizar las operaciones de consulta de datos.

  - ***getWriteableDatabase()***: lo mismo, para realizar también operaciones de escritura.

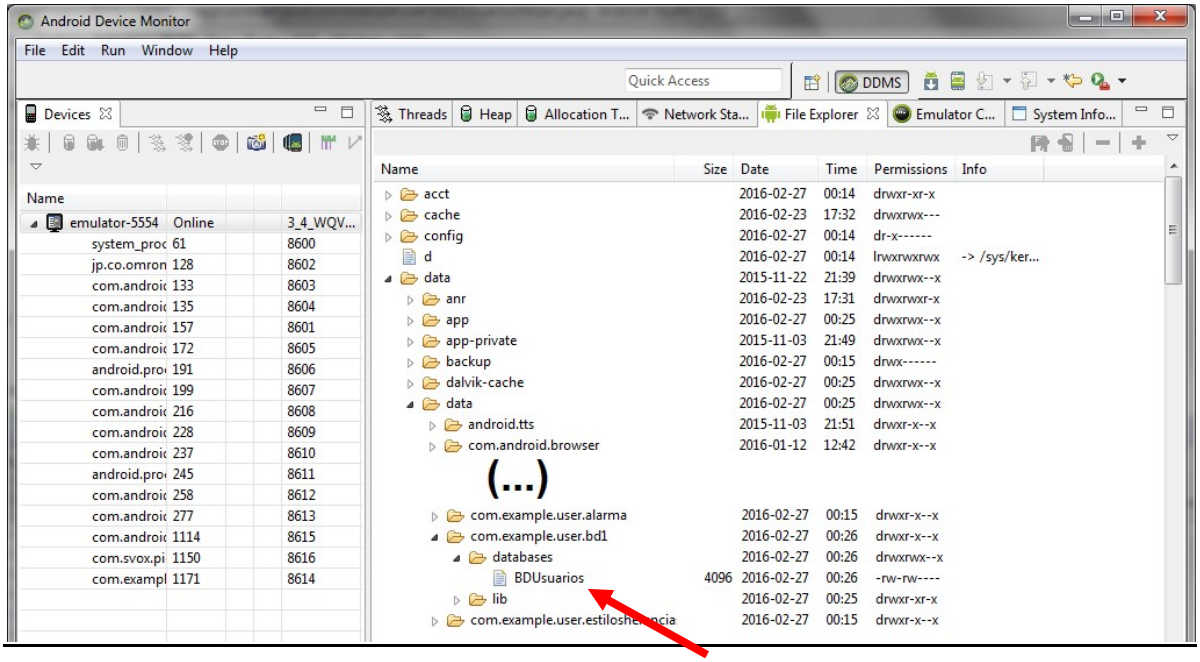
```
//Abrir la base de datos 'BDUsuarios' en modo escritura
SQLiteDatabase db = miClase.getWritableDatabase();
```

- Por último, cuando finalicemos las operaciones sobre la base de datos, debemos cerrar la conexión mediante la llamada al método ***close()***.

- **Comprobación:**

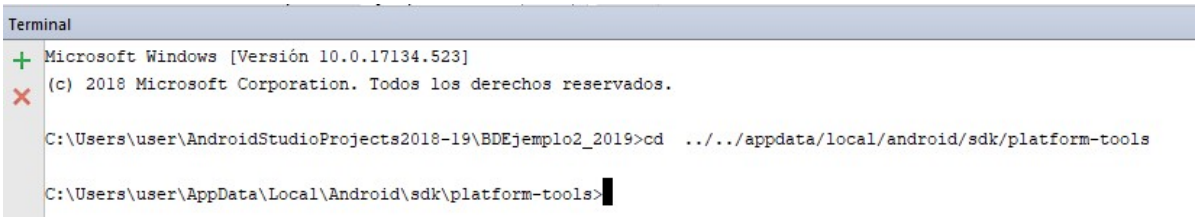
  Podemos comprobar la creación de la base de datos desde la utilidad "***DDMS***" (*Dalvik Debug Monitor Server*).

  En Android Studio (vs. anterior a la 3) podemos acceder a DDMS desde **Tools/Android/Android Device Monitor,** y en la solapa "***File Explorer***" podremos acceder al sistema de archivos del emulador para localizar la ruta donde se encuentra la base de datos.



Para comprobar la existencia de la tabla y su contenido, podemos hacer dos cosas:

- Acceder de forma remota al emulador a través de su **consola de comandos** (**shell**) y hacer uso de la utilidad **adb.exe** (Android Debug Bridge), que se distribuye con el Android SDK.
  - Esta utilidad está situada en la carpeta **platform-tools** del SDK de Android.



  - Podemos consultar los identificadores de todos los emuladores en ejecución mediante el comando "**adb devices**".

```
C:\Users\user\AppData\Local\Android\sdk\platform-tools>adb devices
List of devices attached
emulator-5554   device
```

  - Si solo tenemos un emulador abierto, podemos acceder a su shell mediante el comando "**adb shell**".

```
C:\Users\user\AppData\Local\Android\sdk\platform-tools>adb shell
root@generic_x86:/ #
```

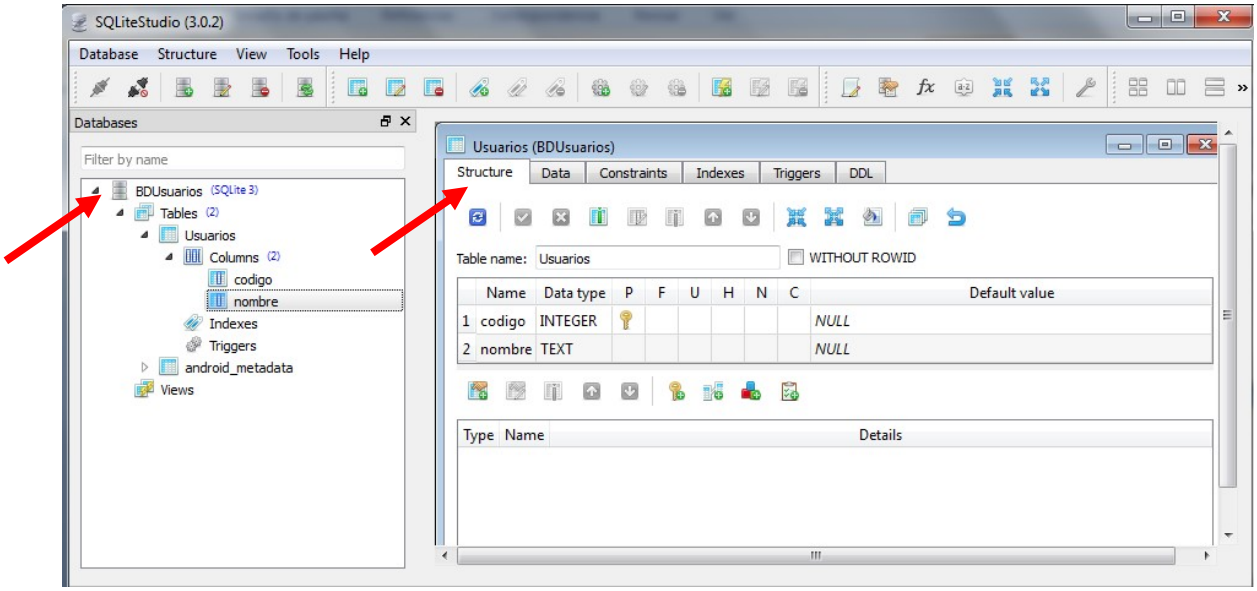
  - Una vez conectados, ya podemos acceder a nuestra base de datos utilizando el comando **sqlite3** pasándole la ruta del fichero:

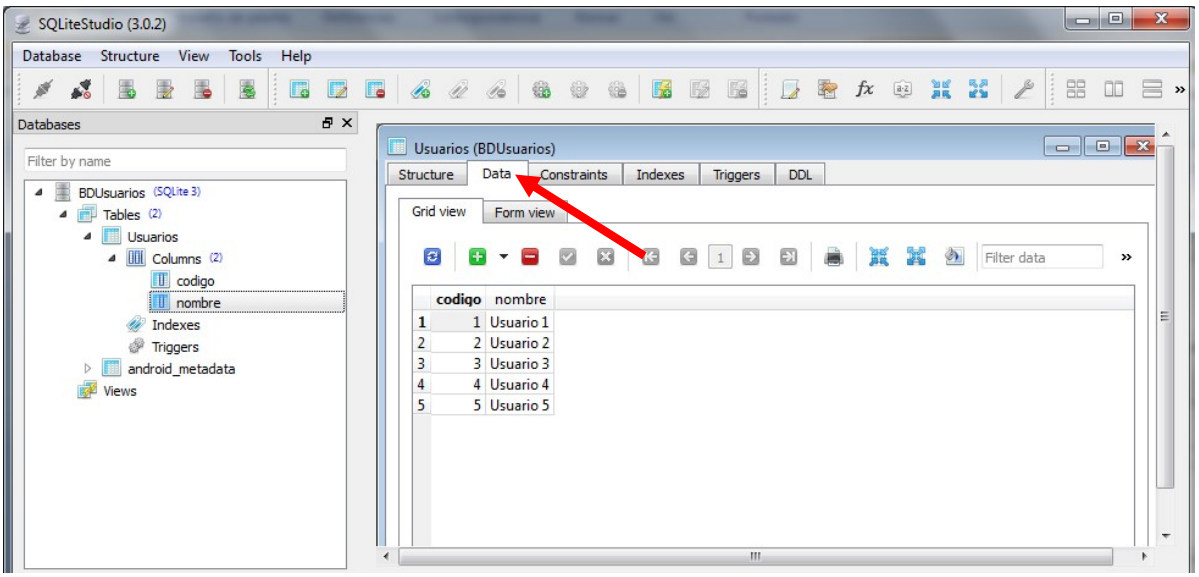
sqlite3 /data/data/nombre_del_paquete/databases/nombre_BD

  - A continuación del prompt de SQLite ya podemos escribir las consultas SQL sobre nuestra base de datos:

```
C:\Users\user\AppData\Local\Android\sdk\platform-tools>adb shell
root@generic_x86:/ # sqlite3  /data/data/com.example.user.bdejemplo1_2019/databases/BDUsuarios
SQLite version 3.8.6.1 2015-05-21 17:24:32
Enter ".help" for usage hints.
sqlite> select * from tUsuarios;
2|usuario_2
3|usuario_3
4|usuario_4
5|usuario nuevo
sqlite>
```

- Transferir la base de datos a nuestro PC y consultarla con cualquier administrador de bases de datos SQLite. Por ejemplo, este es el aspecto que muestra el gestor ***SQLite Studio*** con la base de datos creada en el ejemplo anterior





6. OPERACIONES SOBRE LOS REGISTROS DE UNA BASE DE DATOS

- Las operaciones básicas asociadas con las bases de datos son las operaciones de **inserción**, **actualización**, **borrado** y **consulta**.

- Las tres primeras operaciones no devuelven resultados (a diferencia de las consultas) y se pueden llevar a cabo con el método visto anteriormente: ***execSQL()*** de la clase **SQLiteDataBase**.

- El método ***execSQL()*** recibe como parámetro de entrada la cadena de texto correspondiente con la sentencia SQL que se quiere ejecutar contra la BD. Por ejemplo:

```
bd.execSQL("INSERT INTO tUsuarios (codigo, nombre) VALUES (4, 'usuario_4')");
```

- Otra forma de realizar las operaciones de inserción, borrado y modificación es mediante los métodos: ***insert()***, ***update()*** y ***delete()***, también de la clase **SQLiteDataBase**.

long	insert(String table, String nullColumnHack, ContentValues values) Convenience method for inserting a row into the database.
int	update(String table, ContentValues values, String whereClause, String[] whereArgs) Convenience method for updating rows in the database.
int	delete(String table, String whereClause, String[] whereArgs) Convenience method for deleting rows in the database.

6.1 INSERT

- Método ***insert()***: consta de **tres** parámetros:

  1. El **nombre de la tabla**.
  2. Normalmente será **null**.
  3. **Valores** del registro a insertar: Los valores a insertar los pasaremos como elementos de una colección de tipo **ContentValues**. En un objeto de tipo ContentValues se almacenan parejas "clave-valor", donde la clave será el nombre de cada campo y el valor será el dato correspondiente a insertar en dicho campo. Esto se hace con los métodos ***put…()***:

void	put(String key, Short value) Adds a value to the set.
void	put(String key, Long value) Adds a value to the set.
(...)	
void	put(String key, Integer value) Adds a value to the set.
void	put(String key, String value) Adds a value to the set.

- Ejemplo: insertar un registro con los valores (10, "usuario 10"):

```
ContentValues nuevoRegistro = new ContentValues();
nuevoRegistro.put("codigo", "10");
nuevoRegistro.put("nombre", "usuario 10");
bd.insert("tUsuarios", null, nuevoRegistro);
```

6.2 DELETE

- Método ***delete()***: consta de **tres** parámetros:

  1. El **nombre de la tabla**.
  2. La condición de la **cláusula WHERE** (si no hay cláusula WHERE se indicaría null, y se produciría el borrado de toda la tabla).
  3. Normalmente, null.

- Ejemplo: eliminar el registro con código 10 sería:

```
bd.delete("tUsuarios", "codigo=10", null);
```

6.3 UPDATE

- Método ***update()***: consta de **cuatro** parámetros:

  1. El **nombre de la tabla**.
  2. Los **nuevos valores**.
  3. La condición de la **cláusula WHERE.**
  4. Normalmente, null.

- Ejemplo: modificar el registro 3 para que el nombre pase a ser "usuario modificado":

```
ContentValues otroRegistro = new ContentValues();
otroRegistro.put("nombre", "usuario modificado");
bd.update("tUsuarios", otroRegistro, "codigo=3", null);
```

7. CONSULTAS

Pueden realizarse de dos formas:

- Utilizando el método ***rawQuery()*** de la clase **SQLiteDatabase**.

  - Este método recibe directamente como parámetro un comando SQL completo, donde indicamos los campos a recuperar y los criterios de selección.

`Cursor`	`rawQuery(String sql, String[] selectionArgs)` Runs the provided SQL and returns a `Cursor` over the result set.

  - El resultado de la consulta se obtiene en forma de **Cursor**, que posteriormente podremos recorrer para procesar los registros recuperados.

  - Podemos pensar en un **Cursor** como el puntero al resultado devuelto desde una consulta a una base de datos.

  - Para recorrer el cursor devuelto por una consulta, la clase **Cursor** dispone de varios métodos. Por ejmplo:
    - ***moveToFirst()***: mueve el puntero del cursor al primer registro devuelto.

- ***moveToNext()***: mueve el puntero del cursor al siguiente registro devuelto.

- Ambos métodos devuelven *TRUE* en caso de haber realizado el movimiento correspondiente del puntero sin errores, es decir, siempre que exista un primer registro o un registro siguiente, respectivamente.

- Una vez posicionados en cada registro podremos utilizar cualquiera de los métodos ***get<type>(índice_ columna)*** existentes para cada tipo de dato para recuperar el dato de cada campo del registro actual del cursor. Los índices empiezan en el valor 0.

- Ejemplo: recuperar el nombre del registro para un código determinado.

```
Cursor c = bd.rawQuery("SELECT nombre FROM tUsuarios WHERE codigo=10", null);
if (c.moveToFirst()) { //significa que se ha recuperado algo en la consulta
    String n = c.getString(0);
    Toast.makeText(this, "Nombre: " + n, Toast.LENGTH_LONG).show();
}
else{
    Toast.makeText(this, "Usuario inexistente", Toast.LENGTH_LONG).show();
}
```

- Utilizando el método ***query()*** de la clase **SQLiteDatabase**.

  - Es un método sobrecargado.

```
Cursor    query(String table, String[] columns, String
          selection, String[] selectionArgs, String groupBy,
          String having, String orderBy, String limit)

          Query the given table, returning a Cursor over the result set.
```

```
Cursor    query(String table, String[] columns, String
          selection, String[] selectionArgs, String groupBy,
          String having, String orderBy)

          Query the given table, returning a Cursor over the result set.
```

  - Este método recibe varios parámetros:

    1. el nombre de la **tabla**,
    2. un **array** con los nombres de **campos a recuperar**,
    3. la cláusula **WHERE**,
    4. un **array** con los argumentos variables incluidos en el WHERE (si los hay, null en caso contrario),
    5. la cláusula **GROUP BY** si existe,
    6. la cláusula **HAVING** si existe, y por último
    7. la cláusula **ORDER BY** si existe.
    8. Opcionalmente, se puede incluir un parámetro más al final, indicando el número máximo de registros que queremos que nos devuelva la consulta.

- Ejemplo: consultar todos los registros de la tabla de ejemplo:

```
String[] datosARecuperar={"codigo", "nombre"};
Cursor c = bd.query("tUsuarios", datosARecuperar, null,null,null,null,null,null);
if (c.moveToFirst()) { //significa que se ha recuperado algo en la consulta
    //recorremos el cursor hasta que no haya más registros
    do {
        int codigo = c.getInt(0);
        String nombre = c.getString(1);
        Toast.makeText(this, "Código: "+codigo+" Nombre: " + nombre, Toast.LENGTH_LONG).show();
    }while (c.moveToNext());
}
else
    Toast.makeText(this, "Usuario inexistente", Toast.LENGTH_LONG).show();
```