Capítulo 7

Video, Audio y Base de Datos

Práctica 7.1.- Reproducción de Audio.

Android proporciona en su API multimedia la clase MediaPlayer para manejar contenidos multimedia desde la aplicación.

Fichero manifests

Debemos incluir en el fichero manifest las declaraciones de permisos de uso. Hay dos posibles:

- Permiso de internet: si se va a manejar un contenido que se va a manejar por internet se debe solicitar permiso de acceso a la Red.

```
<uses-permission android:name="android.permission.INTERNET"  />
```

- Permiso de desbloqueo: si la aplicacion necesita poner ek dispositivo móvil en estado de desbloqueo (porque está con la pantalla en oscuro o el procesador está en modo sleeping) es necesario solicitar el permiso de desbloqueo.

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
```

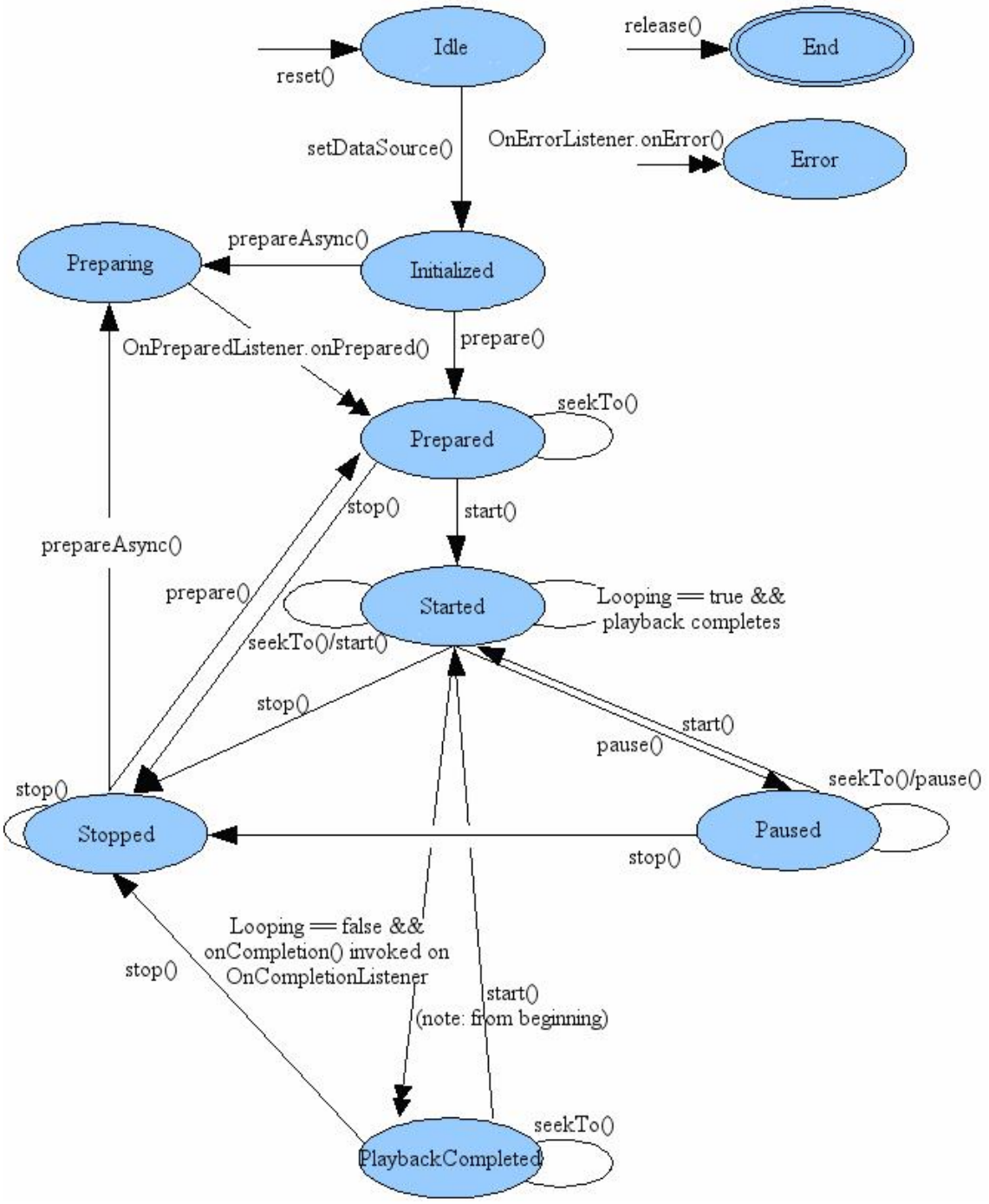
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.sonido"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="8"
        android:targetSdkVersion="16" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.sonido.MainSonido"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />

            </intent-filter>
        </activity>
    </application>
 <uses-permission android:name="android.permission.WAKE_LOCK" />
 <uses-permission android:name="android.permission.INTERNET"  />
</manifest>
```

MediaPlayer

Ciclo de vida:

Flechas de una punta llaman a métodos síncronos.

Flechas de 2 puntas llaman a métodos asíncronos.



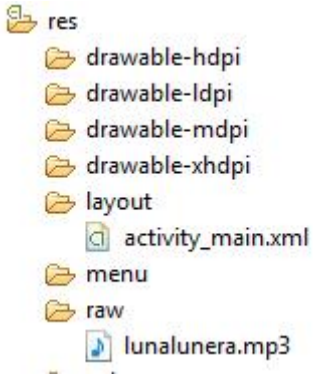
Media Player es una clase del paquete android.media que puede manejar vídeo y audio desde diferentes fuentes.

- **Desde la Web**: el contenido está ubicado en una fuente accesible por una URL de Internet.

```
static final String AUDIO_PATH =
"http://sayedhashimi.com/downloads/android/play.mp3";
mediaPlayer.setDataSource(AUDIO_PATH);
mediaPlayer .prepare();
mediaPlayer.start();
```

- **Desde un fichero local**: el contenido está ubicado en un fichero que es empaquetado como parte del fichero .apk de la aplicación. El fichero multimedia debe ser incluido como una fuente de la carpeta /res/raw o en /assets.

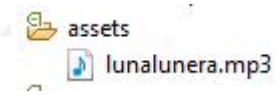
En la carpeta /res/raw




```
mediaPlayer = new MediaPlayer();
mediaPlayer = MediaPlayer.create(this, R.raw.lunaLunera);
mediaPlayer.start();
```

```
package com.prumus;

import java.io.FileDescriptor;
import java.io.IOException;

import android.media.MediaPlayer;
import android.os.Bundle;
import android.app.Activity;
import android.content.res.AssetFileDescriptor;
import android.view.Menu;

public class MainActivity extends Activity {

    public MediaPlayer mediaPlayer;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        try{
            play();
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }

    void play() throws Exception{
        mediaPlayer = new MediaPlayer();
        mediaPlayer = MediaPlayer.create(this, R.raw.lunalunera);
        mediaPlayer.start();
    }
}
```

En la carpeta assets



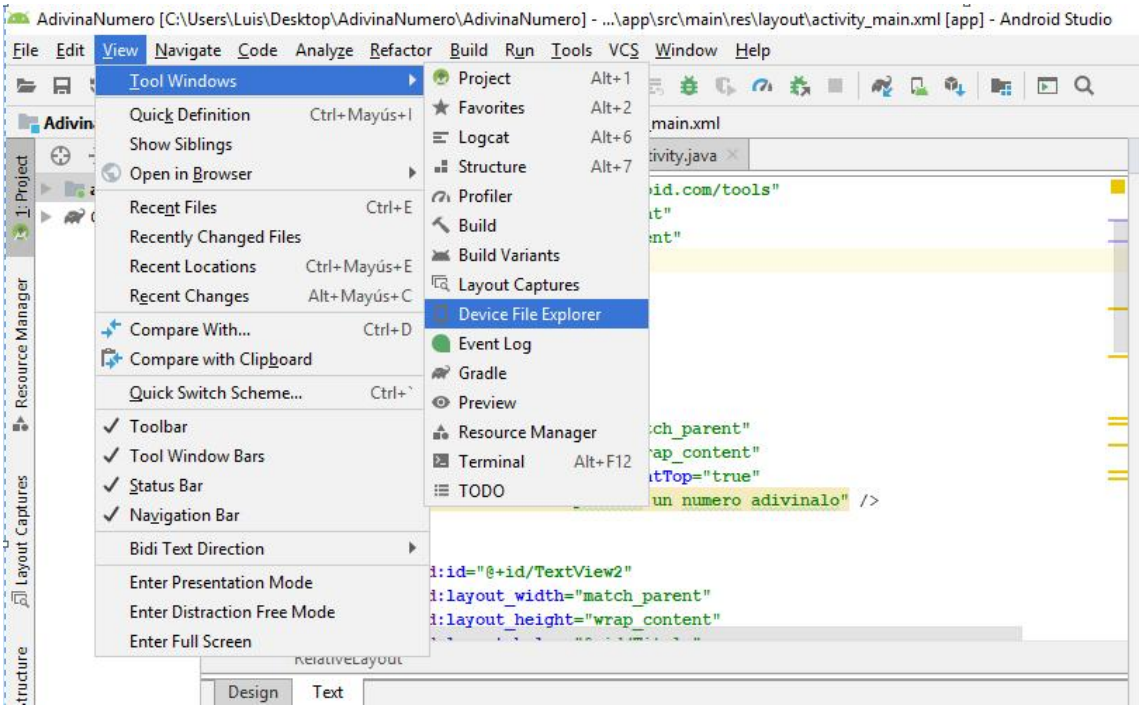

```
        mediaPlayer = new MediaPlayer();
        AssetFileDescriptor afd = getAssets().openFd("lunalunera.mp3");
        mediaPlayer.reset();
        mediaPlayer.setDataSource(afd.getFileDescriptor());
        mediaPlayer.prepare();
        mediaPlayer.start();
```

```
package com.prumus;

import java.io.FileDescriptor;

public class MainActivity extends Activity {

    public MediaPlayer mediaPlayer;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mediaPlayer = new MediaPlayer();

        try{
            play();
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.activity_main, menu);
        return true;
    }

    void play() throws Exception{
        mediaPlayer = new MediaPlayer();
        AssetFileDescriptor afd = getAssets().openFd("lunalunera.mp3");
        mediaPlayer.reset();
        mediaPlayer.setDataSource(afd.getFileDescriptor());
        mediaPlayer.prepare();
        mediaPlayer.start();
    }
}
```

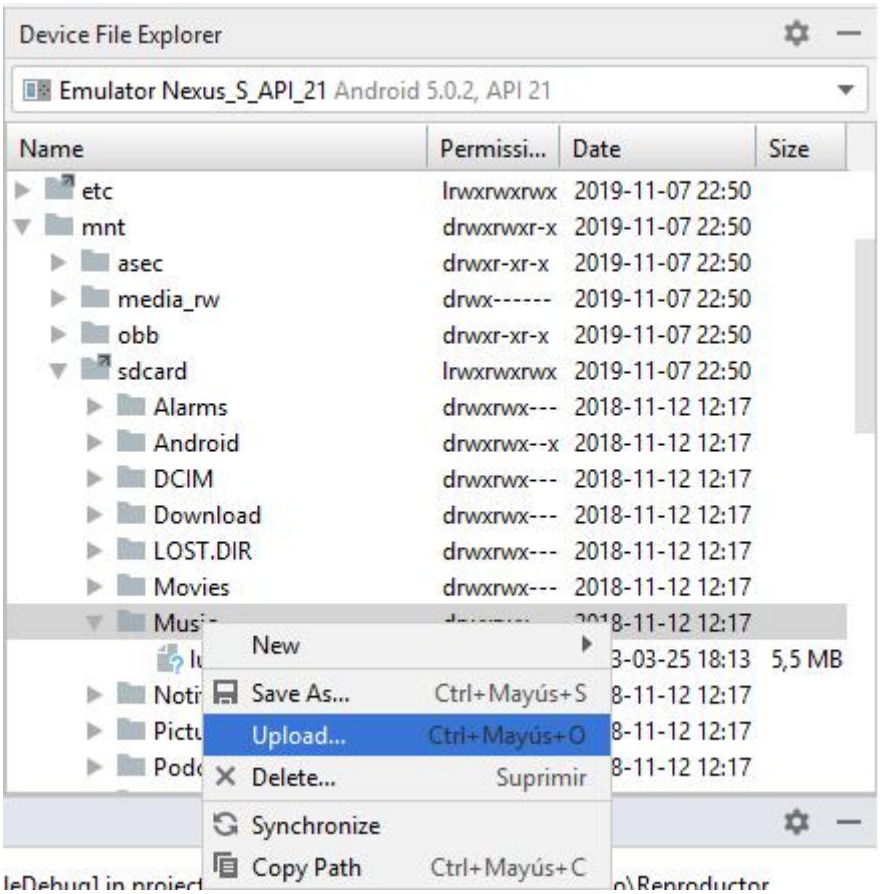
- **Desde una tarjeta SD (Secure Digital):** el contenido reside en una tarjeta DS en el móvil.

Esto solamente funciona con API menores de la 23 (no incluida)

/View/Tool Windows/Device File Explorer



En la carpeta mnt/sdcard/Music o mejor en la carpeta sdcard/Music sin mnt debemos subir el archivo mp3 (debemos seleccionar esta carpeta con el botón derecho del ratón). Pulsamos Upload y seleccionamos el .mp3, luego de esto tenemos el archivo montado en la tarjeta SD:



Cuando esté copiado el fichero en la carpeta /mnt/dscard/Music, escribimos el siguiente código.

```
//Ejecutamos este método al pulsar el botón definido en main.xml de layout
public void ejecutar(View v) {
        Uri datos = Uri.parse(Environment.getExternalStorageDirectory()
                .getPath() + "/Music/quebranto.mp3");
        MediaPlayer mp = MediaPlayer.create(this, datos);
        mp.start();
    }
```

Declaración del botón

```
<Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:onClick="ejecutar"
        android:text="Reproducción de archivo mp3 localizado en tarjeta SD"
/>
```

Otro paso importante antes de ejecutar la aplicación es agregar el permiso de leer la tarjeta SD, para ello modificamos el archivo AndroidManifest.xml agregando este permiso:

```
<uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>
```

- **Importante**

Si no funciona debemos dar permisos en la app del móvil o emulador.

Práctica 7.2 Reproducción de Video

Se suele utilizar VideoView con un objeto MediaController para manejar la reproducción de vídeo. La forma de "enlazar" estos objetos, VideoView y MediaContoller, es mediante el método setMediaController de VideoView:

public void setMediaController (MediaController controller)

Con el método setViewURI se establece la fuente que contiene el vídeo a reproducir. Este método requiere que se le pase como parámetro un objeto Uri:

Public void setVideoURI(Uri uri)

La clase MediaController es una clase que proporciona los botones típicos de play/pausa, rebobinado y avance rápido. En tiempo de ejecución, la ventana MediaController desaparece cuando hay una inactividad de tres segundos y vuelve a reaparecer cuando el usuario toca el control asociado por el método setAnchorView().

En el código XML contiene un VideoView para visualizar el vídeo.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
```

```
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Mi Reproductor"
        android:layout_centerHorizontal="true"
        android:layout_above="@+id/LinearLayout1"/>
    <LinearLayout
        android:id="@+id/LinearLayout1"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:layout_width="wrap_content" >
        <VideoView
            android:id="@+id/videoView1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
    </LinearLayout>

</RelativeLayout>
```

En el código fuente se incluyen los widgets MediaController y VideoView. En la clase Main se recupera la referencia al control VideoView (videoView1) y se asigna al objeto miVideoView el cual realiza la reproducción del video.

```
VideoView miVideoView = (VideoView) findViewById(R.id.videoView1);
```

A continuación se crea el objeto MediaController

```
MediaController mController = new MediaController(this);
```

Asociamos el objeto VideoView con el objeto MediaController.

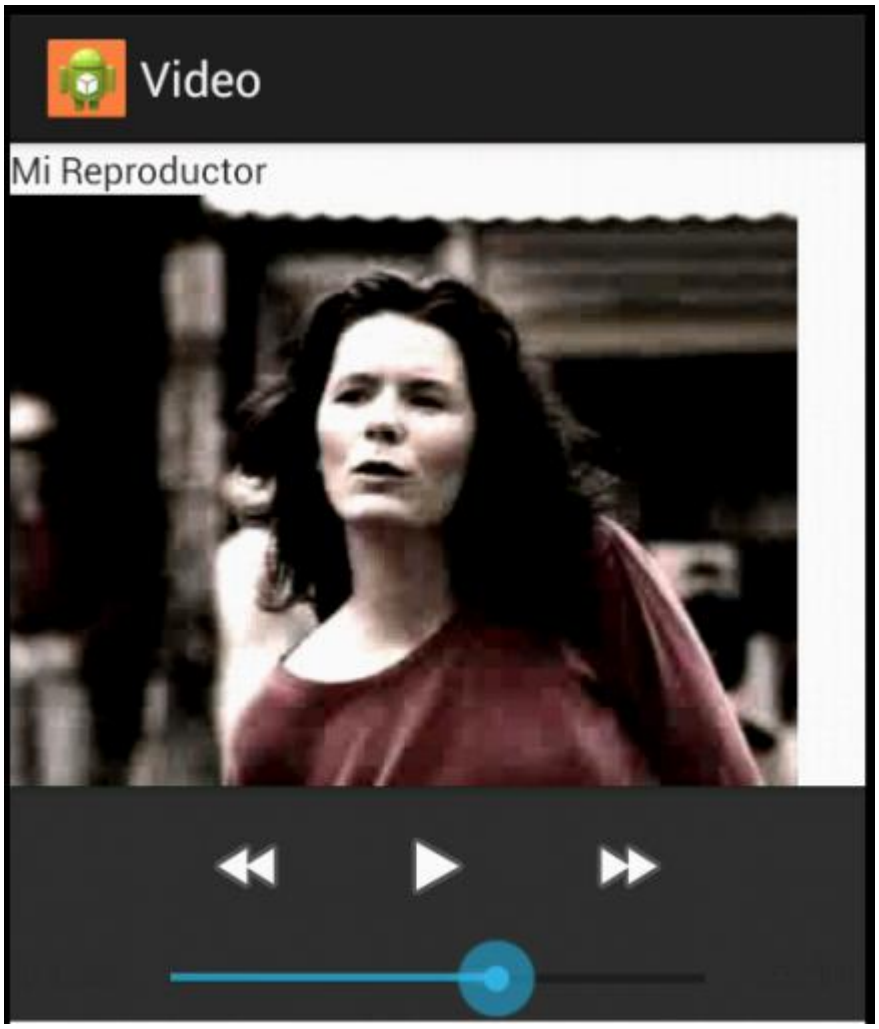
```
miVideoView.setMediaController(mController);
```

Indicamos la fuente del vídeo a reproducir mediante el método setVideoURI. El método public static Uri.parse(String uriString) crea un objeto Uri a partir del string uristring que se le pasa como URL. getPackgeName() obtiene el nombre de la aplicación para poder formar el path completo.

```
miVideoView.setVideoURI(Uri.parse("android.resource://" + getPackageName()
+"/"+R.raw.goodtime));
```

Por último, se ejecuta el método public final boolean requestFocus(), de esta forma queda dispuesto para que el usuario inicie la reproducción.

```
miVideoView.requestFocus();
```




```
public class MainVideo extends Activity {

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.main_video);

		VideoView miVideoView = (VideoView)
indViewById(R.id.videoView1);
		MediaController mController = new MediaController(this);
		miVideoView.setMediaController(mController);
		miVideoView.setVideoURI(Uri.parse("android.resource://" +
getPackageName() +"/"+R.raw.goodtime));
		miVideoView.requestFocus();
}
```

Práctica 7.3 Bases de Datos (Sqlite)

Para trabajar con SQLite es necesario crear una subclase SQLiteOpenHelper, la cual tiene la lógica para crear y actualizar una base de datos. Esta subclase necesita tres métodos:

- **Constructor**, Toma el contexto (por ejemplo una Activity en este caso la actividad principal), el nombre de la base de datos, un cursor opcional (generalmente null) y un entero que representa la versión del esquema de la base de datos que estamos utilizando.

```
public UsuariosSQLiteHelper(Context contexto, String nombre,
                CursorFactory factory, int version) {
        super(contexto, nombre, factory, version);
}
```

  - Si la base de datos ya existe y su versión actual coincide con la solicitada simplemente 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 se conectará con la base de datos convertida.
  - Si la base de datos no existe, se llamará automáticamente al método onCreate() para crearla y se conectará con la base de datos creada.

- **OnCreate(),** el cual pasa el objeto SQLiteDataBase que necesita rellenar con las tablas y datos iniciales apropiados.

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

- **OnUpgrade(),** el cual permite pasar de una versión de un esquema a otra. Dentro de este método se realiza la edición de los datos de las tablas hasta encajar con la estructura de la nueva versión.

```
@Override
public void onUpgrade(SQLiteDatabase db, int versionAnterior, int
versionNueva) {

      //Se elimina la versión anterior de la tabla
      db.execSQL("DROP TABLE IF EXISTS Usuarios");

      //Se crea la nueva versión de la tabla
      db.execSQL(sqlCreate);
      }
```

Vamos a crear una base de datos muy sencilla llamada ***BDContactos***, con una sóla tabla llamada ***Contactos*** que contendrá sólo dos campos: *nombre y telefono*. Para ellos, vamos a crear una clase derivada de SQLiteOpenHelper que llamaremos UsuariosSQLiteHelper, donde sobrescribiremos los métodos onCreate() y onUpgrade() para adaptarlos a la estructura de datos indicada:

```
package com.crearbasedatos;

import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;

public class UsuariosSQLiteHelper extends SQLiteOpenHelper {

       //Sentencia SQL para crear la tabla de Usuarios
      String sqlCreate = "CREATE TABLE Contactos (nombre NUMERIC, telefono
TEXT)";
      public UsuariosSQLiteHelper(Context contexto, String nombre,
                  CursorFactory factory, int version) {
            super(contexto, nombre, 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 versionAnterior, int
versionNueva) {
//NOTA: Por simplicidad del ejemplo aquí utilizamos directamente la opción de
// eliminar la tabla anterior y crearla de nuevo vacía con el nuevo formato.
//   Sin embargo lo normal será que haya que migrar datos de la tabla antigua
//    a la nueva, por lo que este método debería ser más elaborado.

//Se elimina la versión anterior de la tabla
            db.execSQL("DROP TABLE IF EXISTS Usuarios");

//Se crea la nueva versión de la tabla
            db.execSQL(sqlCreate);
            }

      }
```

Ejemplo de declaración de datos:

```
CREATE TABLE tabla1(
campo1    INTEGER PRIMARY KEY,
campo2    TEXT,
campo3    NUMERIC,
campo4    INTEGER,
campo5    REAL
);
```

Los tipos de datos declarados al crear una tabla serán meramente informativos y no impedirán que posteriormente se inserte cualquier valor de cualquier tipo en estos campos, siempre con las siguientes matizaciones:

- Si en un campo TEXT se inserta un valor numérico este valor se convertirá a texto antes de ser almacenado.
- Si en un campo NUMERIC se inserta un valor de tipo texto éste se intentará convertir a un valor numérico entero o real antes de su almacenamiento y si no es posible la conversión se almacenará directamente como texto.
- Los campos de tipo INTEGER se comportan de igual forma que NUMERIC salvo que si se inserta un valor real sin decimales o un valor textual que pueda convertirse a éste último se almacenará como entero.
- Los campos de tipo REAL se comportan de igual forma que NUMERIC salvo que se almacenará el dato siempre como valor de tipo real.
- Los campos INTEGER PRIMARY KEY servirán para crear un campo autonumérico o autoincremental (el campo se autoincrementa al realizar inserciones en la tabla con este campo nulo).

La subclase SQLiteOpenHelper proporciona el método getReadableDatabase() el cual devuelve un objeto de tipo SQLiteDataBase sobre el cual podemos realizar las operaciones de consultas de datos.

Si lo que queremos es escribir en la base de datos, tendremos que llamar al método *getWritableDataBase*()

Ahora que ya hemos conseguido una referencia a la base de datos (objeto de tipo *SQLiteDatabase*) ya podemos realizar todas las acciones que queramos sobre ella. Para nuestro ejemplo nos limitaremos a insertar 5 registros de prueba, utilizando para ello el método ya comentado *execSQL*() con las sentencias *INSERT* correspondientes. Por último cerramos la conexión con la base de datos llamando al método *close*().

```
package com.crearbasedatos;

import android.os.Bundle;
import android.app.Activity;
import android.database.sqlite.SQLiteDatabase;
import android.view.Menu;

public class MainCrearBD extends Activity {



	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.main_crear_bd);

		//Abrimos la base de datos 'DBContactos' en modo escritura
		UsuariosSQLiteHelper usdbh =
				new UsuariosSQLiteHelper(this, "DBContactos", null,
1);
		SQLiteDatabase db = usdbh.getWritableDatabase();
```

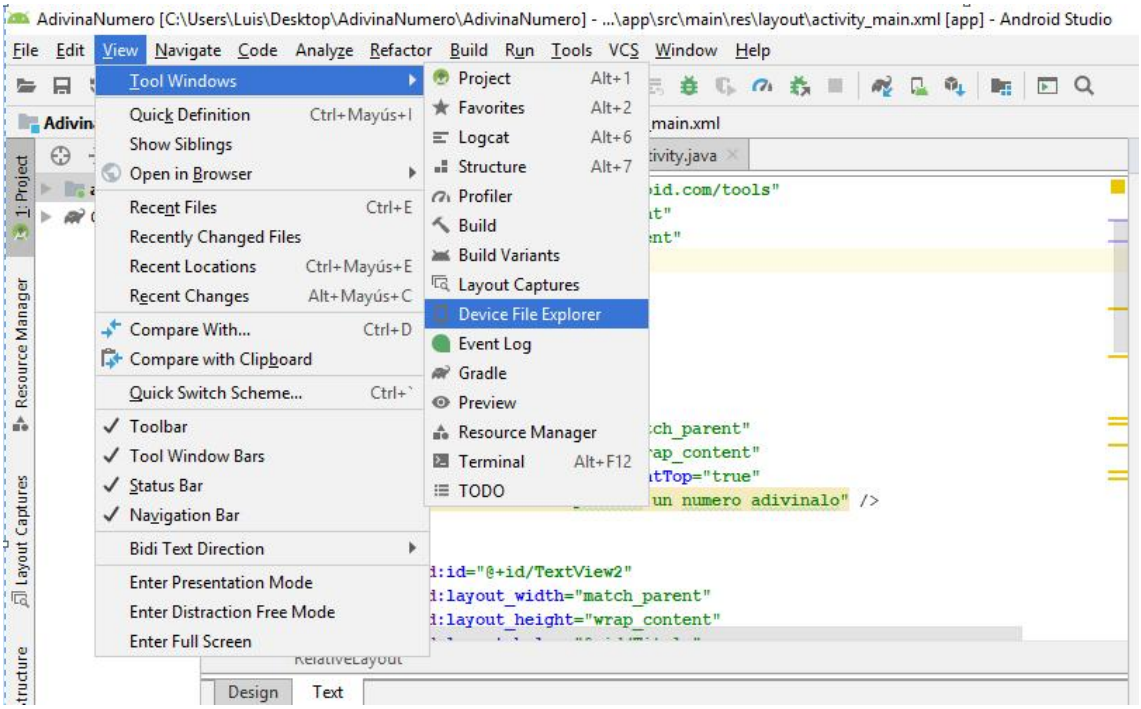
```
            //Si hemos abierto correctamente la base de datos
            if(db != null) {
                  //Insertamos 5 usuarios de ejemplo
                  for(int i=1; i<=5; i++) {
                        //Generamos los datos
                        int telefono = 11111111+i;
                        String nombre = "Usuario" + i;
                        //Insertamos los datos en la tabla Usuarios
                        db.execSQL("INSERT INTO Contactos (nombre,
telefono) " +
                        "VALUES ('" + nombre +"', " + telefono + " )");
}
                        //Cerramos la base de datos
                  db.close();
                  }

            }


      @Override
      public boolean onCreateOptionsMenu(Menu menu) {
            // Inflate the menu; this adds items to the action bar if it is
present.
            getMenuInflater().inflate(R.menu.main_crear_bd, menu);
            return true;
      }

}
```

Comprobación de la base de datos creada.

Todas las bases de datos SQLite creadas por aplicaciones Android se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón:

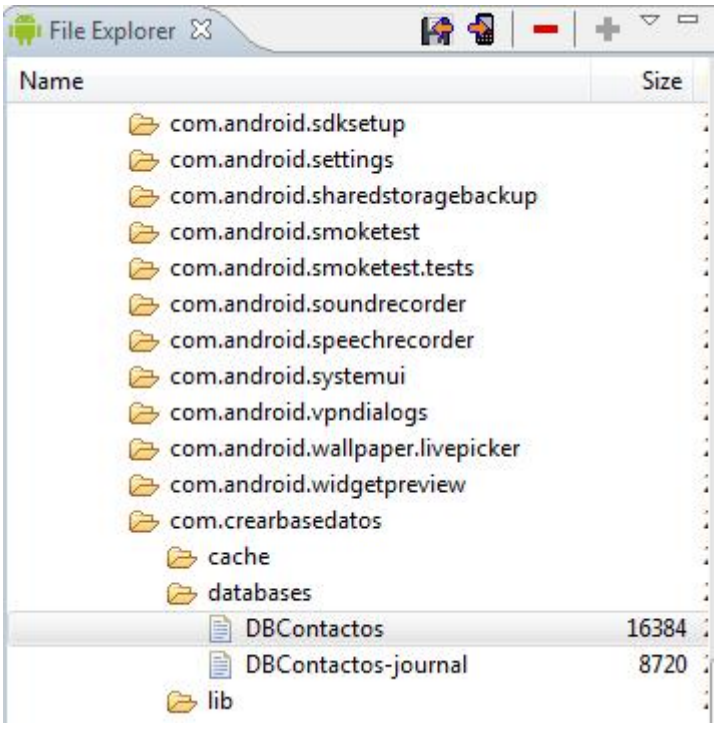
`/data/data/paquete.java.de.la.aplicacion/databases/nombre_base_datos`

En el caso de nuestro ejemplo, la base de datos se almacenaría por tanto en la ruta siguiente:
`/data/data/dm2e.creabasedatos/databases/DBContactos`

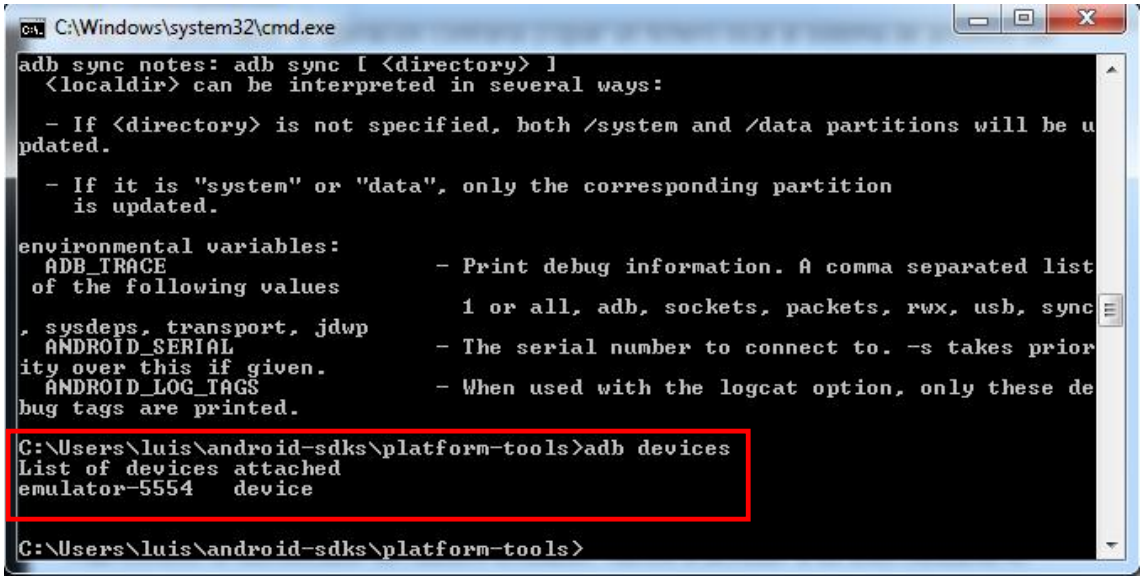


Pulsamos en View/Tool Windows/Device File Explorer

Seleccionamos File Explorer



Buscamos la Base de Datos.

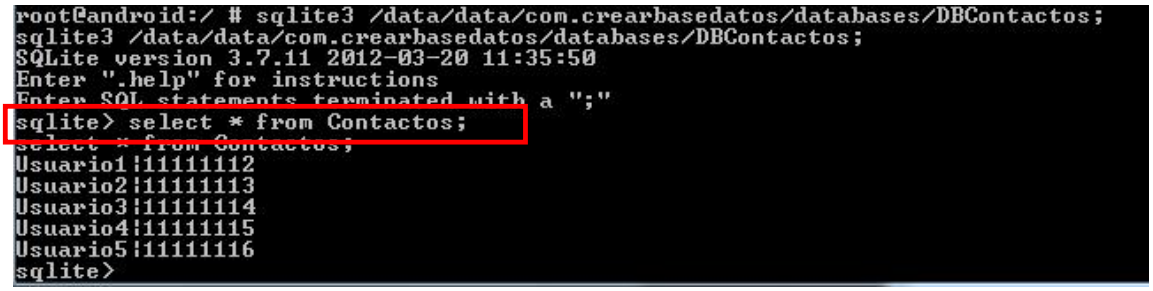


Ejecutamos el comando “adb devices”

Ejecutamos el comando “adb -s nombre_del_emulador Shell”.

o

adb -e shell "si solamente tenemos un emulador activo"



Ejecutamos la sentencia “sqlite3 /data/data/com.crearbasedatos/databases/DBContactos;”.

Consultamos la tabla contactos “select * from Contactos”.

NOTA

Algunas veces da un error al crear las tablas “E/SQLiteLog: (1) no such table: Contactos”

Debemos desinstalar la aplicación del emulador y volver a ejecutarla.

Práctica 7.4 Inserción, Actualización y Borrado

La API de SQLite de Android proporciona dos alternativas para realizar operaciones sobre la base de datos que no devuelven resultados (entre ellas la inserción/actualización/eliminación de registros, pero también la creación de tablas, de índices, etc).

El primero de ellos, que ya comentamos brevemente en el artículo anterior, es el método *execSQL*() de la clase *SQLiteDatabase*. Este método permite ejecutar cualquier sentencia SQL sobre la base de datos, siempre que ésta no devuelva resultados. Para ello, simplemente aportaremos como parámetro de entrada de este método la cadena de texto correspondiente con la sentencia SQL.

```
//Insertar un registro
db.execSQL("INSERT INTO Usuarios (usuario,email) VALUES
('usu1','usu1@email.com') ");
//Eliminar un registro
db.execSQL("DELETE FROM Usuarios WHERE usuario='usu1' ");
//Actualizar un registro
db.execSQL("UPDATE Usuarios SET email='nuevo@email.com' WHERE
usuario='usu1' ");
```

La segunda de las alternativas disponibles en la API de Android es utilizar los métodos *insert*(), *update*() y *delete*() proporcionados también con la clase *SQLiteDatabase*. Estos métodos permiten realizar las tareas de inserción, actualización y eliminación de registros de una forma algo más paramétrica que *execSQL*(), separando tablas, valores y condiciones en parámetros independientes de estos métodos.

Empecemos por el método *insert*() para insertar nuevos registros en la base de datos. Este método recibe tres parámetros:

- El primero de ellos será el nombre de la tabla,
- El tercero serán los valores del registro a insertar, y
- El segundo lo obviaremos por el momento ya que tan sólo se hace necesario en casos muy puntuales (por ejemplo para poder insertar registros completamente vacíos), en cualquier otro caso pasaremos con valor null este segundo parámetro.

Los valores a insertar los pasaremos como elementos de una colección de tipo *ContentValues*. Esta colección es de tipo diccionario, donde almacenaremos parejas de clave-valor, donde la clave será el nombre de cada campo y el valor será el dato correspondiente a insertar en dicho campo. Veamos un ejemplo:

```
//Creamos el registro a insertar como objeto ContentValues
      ContentValues nuevoRegistro = new ContentValues();
      nuevoRegistro.put("nombre", "usu10");
      nuevoRegistro.put("telefono","11111121");
```

```
//Insertamos el registro en la base de datos
      db.insert("Contactos", null, nuevoRegistro);
```

Los métodos *update*() y *delete*() se utilizarán de forma muy parecida a ésta, con la salvedad de que recibirán un parámetro adicional con la condición *WHERE* de la sentencia SQL. Por ejemplo, para actualizar el teléfono del usuario de nombre '*usu1*' haríamos lo siguiente:

```
//Establecemos los campos-valores a actualizar
ContentValues valores = new ContentValues();
valores.put("telefono","211111112");
//Actualizamos el registro en la base de datos
db.update("Contactos", valores, "nombre='usu1'");
```

Como podemos ver, como tercer parámetro del método *update*() pasamos directamente la condición del *UPDATE* tal como lo haríamos en la cláusula *WHERE* en una sentencia SQL normal.

El método *delete*() se utilizaría de forma análoga. Por ejemplo para eliminar el registro del usuario '*usu2*' haríamos lo siguiente:

```
//Eliminamos el registro del usuario 'usu2'
db.delete("Usuarios", "usuario='Usuario2'");
```

Como vemos, volvemos a pasar como primer parámetro el nombre de la tabla y en segundo lugar la condición *WHERE*. Por supuesto, si no necesitáramos ninguna condición, podríamos dejar como *null* en este parámetro.

Un último detalle sobre estos métodos. Tanto en el caso de *execSQL*() como en los casos de *update*() o *delete*() podemos utilizar argumentos dentro de las condiciones de la sentencia SQL. Esto no son más que partes variables de la sentencia SQL que aportaremos en un array de valores aparte, lo que nos evitará pasar por la situación típica en la que tenemos que construir una sentencia SQL concatenando cadenas de texto y variables para formar el comando SQL final. Estos argumentos SQL se indicarán con el símbolo '?', y los valores de dichos argumentos deben pasarse en el array en el mismo orden que aparecen en la sentencia SQL. Así, por ejemplo, podemos escribir instrucciones como la siguiente:

```
//Eliminar un registro con execSQL(), utilizando argumentos
String[] args = new String[]{"usu1"};
db.execSQL("DELETE FROM Contactos WHERE nombre=?", args);
//Actualizar dos registros con update(), utilizando argumentos
ContentValues valores = new ContentValues();
valores.put("telefono","111111222");
String[] args = new String[]{"usu1", "usu2"};
db.update("Contactos", valores, "nombre=? OR nombre=?", args);
```

Practica 7.5 Consultar, Recuperar registros

Vamos a tener dos opciones principales para recuperar registros de una base de datos SQLite en Android:

- La primera de ellas utilizando directamente un comando de selección SQL.
- La segunda opción utilizando un método específico donde *parametrizaremos* la consulta a la base de datos.

Para la primera opción utilizaremos 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. El resultado de la consulta lo obtendremos en forma de cursor, que posteriormente podremos recorrer para procesar los registros recuperados.

```
Cursor c = db.rawQuery(" SELECT nombre,telefono FROM Contactos WHERE
nombre='usu1' ");
```

También podemos añadir a este método una lista de argumentos variables que hayamos indicado en el comando SQL con el símbolo '?', por ejemplo así:

```
String[] args = new String[] {"usu1"};
Cursor c = db.rawQuery(" SELECT usuario, email FROM Contactos WHERE
usuario=? ", args);
```

Como segunda opción para recuperar datos podemos utilizar el método `query()` de la clase `SQLiteDatabase`. Este método recibe varios parámetros:

- El nombre de la tabla.
- Un array con los nombre de campos a recuperar.
- La cláusula *WHERE*.
- Un array con los argumentos variables incluidos en el *WHERE* (si los hay, `null` en caso contrario).
- La cláusula *GROUP BY* si existe.
- La cláusula *HAVING* si existe.
- Por último la cláusula *ORDER BY* si existe.

Opcionalmente, se puede incluir un parámetro al final más indicando el número máximo de registros que queremos que nos devuelva la consulta.

```
String[] campos = new String[] {"nombre", "telefono"};
String[] args = new String[] {"usu1"};
Cursor c = db.query("Contactos", campos, "nombre=?", args, null, null,
null);
```

Como vemos, los resultados se devuelven nuevamente en un objeto `Cursor` que deberemos recorrer para procesar los datos obtenidos.

Para recorrer y manipular el cursor devuelto por cualquiera de los dos métodos mencionados tenemos a nuestra disposición varios métodos de la clase `Cursor`, entre los que destacamos dos de los dedicados a recorrer el cursor de forma secuencial y en orden natural:

- `moveToFirst()`: mueve el puntero del cursor al primer registro devuelto.
- `moveToNext()`: mueve el puntero del cursor al siguiente registro devuelto.

Los métodos `moveToFirst()` y `moveToNext()` 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 `getXXX(índice_columna)` existentes para cada tipo de dato para recuperar el dato de cada campo del registro actual del cursor. Así, si queremos recuperar por ejemplo la segunda columna del registro actual, y ésta contiene un campo alfanumérico, haremos la llamada `getString(1)` [NOTA: los índices comienzan por 0, por lo que la segunda columna tiene índice 1], en caso de contener un dato de tipo real llamaríamos a `getDouble(1)`, y de forma análoga para todos los tipos de datos existentes.

```
String[] campos = new String[] {"nombre", "telefono"};
String[] args = new String[] {"Usuario1"};
Cursor c = db.query("Contactos", campos, "usuario=?", args, null,
null, null);
//Nos aseguramos de que existe al menos un registro
if (c.moveToFirst()) {
//Recorremos el cursor hasta que no haya más registros
do {
String nombre = c.getString(0);
int telefono = c.getInt(1);
} while(c.moveToNext());
}
```

Además de los métodos comentados de la clase `Cursor` existen muchos más que nos pueden ser útiles en muchas ocasiones. Por ejemplo, `getCount()` te dirá el número total de registros devueltos en el cursor, `getColumnName(i)` devuelve el nombre de la columna con índice i, `moveToPosition(i)` mueve el puntero del cursor al registro con índice i, etc.