Ficheros y acceso a datos

Índice

1 Ficheros tradicionales....2
2 Preferencias....2
2.1 Crear, guardar y leer preferencias....3
2.2 Interfaz de usuario para las preferencias....3
3 Base de datos SQLite....5
4 Proveedores de contenidos....8
4.1 Proveedores nativos....8
4.2 Proveedores propios....10

1. Ficheros tradicionales

El uso de ficheros tradicionales está permitido para los programas de Android, aunque como veremos más adelante hay otras tecnologías mucho más avanzadas, como es el uso de base de datos y los `SharedPreferences`. En Android un fichero se puede guardar y leer así:

```
FileOutputStream fos = openFileOutput("fichero.txt",
Context.MODE_PRIVATE);
FileInputStream  fis = openFileInput("fichero.txt");
```

El modo `Context.MODE_PRIVATE` hace que el fichero sea privado a la aplicación. Se crearía el fichero si éste no existiera. Para añadir al archivo se utiliza el modo `Context.MODE_APPEND`.

Para escribir en la tarjeta de memoria (SD card) utilizaremos el método `Environment.getExternalStorageDirectory()` y un `FileWriter` de la siguiente manera:

```
try {
    File raiz = Environment.getExternalStorageDirectory();
    if (raiz.canWrite()){
        File file = new File(raiz, "fichero.txt");
        BufferedWriter out = new BufferedWriter(new FileWriter(file));
        out.write("Mi texto escrito desde Android");
        out.close();
    }
} catch (IOException e) {
    Log.e("FILE I/O", "Error en la escritura de fichero: " +
e.getMessage());
}
```

Con tal de poder probar este ejemplo vamos a necesitar un emulador que disponga de una tarjeta SD emulada. Podemos crear un fichero `.iso` que represente dicha tarjeta en la línea de comandos utilizando lo siguiente:

```
mksdcard 512M sdcard.iso
```

Una vez creada la tarjeta SD, y con el emulador en funcionamiento, podemos utilizar el siguiente comando para extraer un fichero de dicha tarjeta y guardarlo en la máquina local:

```
adb pull /sdcard/fichero.txt fichero.txt
```

La operación inversa (copiar un fichero desde la máquina local a la tarjeta SD emulada) se realiza mediante el comando `adb push`.

2. Preferencias

Escribir en ficheros tradicionales puede llegar a ser muy tedioso y en el caso de

programas que tengan que guardar preferencias, acaba en código poco mantenible. Para facilitar el almacenamiento de opciones, android nos proporciona un mecanismo conocido como `SharedPreferences`, el cual se utiliza para guardar datos mediante pares de clave y valor.

Este mecanismo podría ser utilizado, por ejemplo, en el manejador `onSaveInstanceState`, el cual es invocado para guardar el estado de la actividad cuando ésta deba ser terminada por parte del sistema operativo por falta de recursos.

2.1. Crear, guardar y leer preferencias

Para guardar preferencias comunes de la aplicación hay que crear un editor de la siguiente manera:

```
SharedPreferences pref;
pref =
PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
SharedPreferences.Editor editor = pref.edit();
editor.putBoolean("esCierto",false);
editor.putString("Nombre","Pablo");
editor.commit();
```

Obsérvese como se hace uso de métodos de tipo `put` para asignar a una clave un valor determinado. Para obtener el valor asociado con estas claves desde otra parte de la actividad podríamos hacer uso del siguiente código:

```
boolean esCierto = pref.getBoolean("esCierto",false);
String  nombre   = pref.getString("Nombre", "sin nombre");
```

El segundo parámetro de los métodos `get` es un valor por defecto. Este valor es el devolverá el método en el caso en el que la clave indicada no exista.

Además puede ser interesante programar un Listener para que se ejecute cuando ocurra algún cambio en las preferencias:

```
prefValidacion.registerOnSharedPreferenceChangeListener(
        new OnSharedPreferenceChangeListener() {
                @Override
                public void onSharedPreferenceChanged(
                                SharedPreferences sharedPreferences,
                                String key)
                {
                        //Ha cambiado algo, emprender las acciones
necesarias.
                }
        });
```

2.2. Interfaz de usuario para las preferencias

Android permite de una manera sencilla la creación de actividades de preferencias. Cada campo de estas actividades se corresponderá con un par clave y valor de `SharedPreferences`. El primer paso para crear una actividad de modificación de

preferencias es definirla mediante un archivo XML en la carpeta `/res/xml/`, dentro de los recursos de la aplicación. Un ejemplo de archivo de este tipo podría ser el siguiente, al que podríamos llamar `preferencias.xml`:

```
  <?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
<PreferenceCategory android:title="Validar DNI en:">
        <CheckBoxPreference
        android:title="en el campo"
        android:summary="Validará la introducción de números y una letra"
        android:key="validacampo"></CheckBoxPreference>
        <CheckBoxPreference
        android:title="al pulsar"
        android:summary="Comprobará también que la letra sea la correcta"
        android:key="validaboton"></CheckBoxPreference>
</PreferenceCategory>
<PreferenceCategory android:title="Otras preferencias:">
        <CheckBoxPreference android:enabled="false"
        android:title="Otra, deshabilitada"
        android:key="otra"></CheckBoxPreference>
</PreferenceCategory>
</PreferenceScreen>
```

Las claves `android:key` deben coinciden con las claves que después podremos leer mediante `SharedPreferences`. No es necesario crear ningún método que lea de los campos definidos en el XML, al mostrarse la actividad de preferencias, éstas se mapean automáticamente con las preferencias compartidas utilizando las claves.

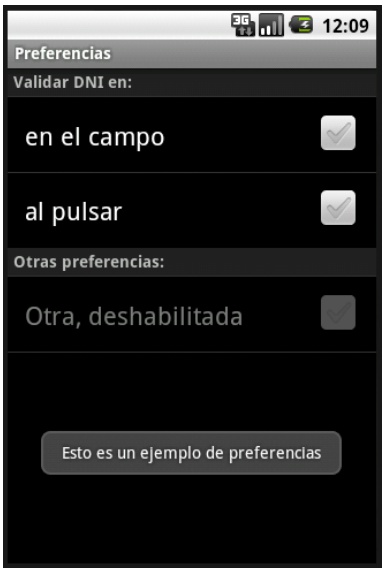
Para que el XML se cargue en una actividad nueva hay que crear una clase que herede de `PreferenceActivity`:

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

Para mostrar esta actividad utilizamos, como siempre, un Intent:

```
startActivity(new Intent(this, Preferencias.class);
```

El resultado se muestra en la imagen siguiente:



Menú de preferencias

3. Base de datos SQLite

SQLite es un gestor de bases de datos relacional y de código abierto, que cumple con los estándares, y además es extremadamente ligero. Otra de sus características es que guarda toda la base de datos en un único fichero. Es útil en aplicaciones pequeñas para que no requieran la instalación adicional de un gestor de bases de datos, así como para dispositivos embebidos con recursos limitados. Android incluye soporte a SQLite.

Una manera de separar el código que accede a la base de datos del resto del código es abstraerlo mediante un patrón adaptador que nos permita abrir la base de datos, leer, escribir, borrar, y otras operaciones que nuestro programa pueda requerir. Así, accedemos a métodos de este adaptador, y no tenemos que introducir código SQL en el resto del programa, haciendo además que el mantenimiento de nuestras aplicaciones sea más sencillo.

Para ilustrar el uso de SQLite en Android vamos a ver una clase adaptador de ejemplo. En esta clase (`DataHelper`) abriremos la base de datos de una manera estándar: mediante un patrón `SQLiteOpenHelper`. Esta clase abstracta nos obliga a implementar nuestro propio Helper de apertura de la base de datos, de una manera estándar. Básicamente sirve para que nuestro código esté obligado a saber qué hacer en caso de que la base de datos no exista (normalmente crearla), y cómo portar la base de datos en caso de detectarse un cambio de versión. La versión de la base de datos la indicamos nosotros. En este ejemplo el cambio de versión se implementa de una manera un poco drástica: borrar todos los contenidos y crear una nueva base de datos vacía, perdiendo todo lo anterior. Se trata de sólo un ejemplo, en cualquier aplicación real convendría portar los datos a las nuevas tablas.

```
package es.ua.jtech.daa.db;

import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteStatement;
import android.util.Log;
import java.util.ArrayList;
import java.util.List;

public class DataHelper {

   // Nombre del fichero en el que se guardará la base de datos
   private static final String DATABASE_NAME = "mibasededatos.db";
   // Versión de la base de datos, indicada por el programador. En el
   // caso de que introduzcamos algún cambio deberíamos modificar este
   // número de versión, con lo que el MiOpenHelper determinará qué hacer
   private static final int DATABASE_VERSION = 1;
   // Nuestra base de datos de ejemplo tiene una única tabla
   private static final String TABLE_NAME = "ciudades";
   // Y esta tabla tiene tres columnas, siendo la primera la clave
   // primaria
   private static final String[] COLUMNAS = {"_id","nombre","habitantes"};

   // Incluimos el código SQL como constantes
   private static final String INSERT = "insert into " + TABLE_NAME +
     "("+COLUMNAS[1]+","+COLUMNAS[2]+") values (?,?)";
   private static final String CREATE_DB = "CREATE TABLE " + TABLE_NAME +
         "("+COLUMNAS[0]+" INTEGER PRIMARY KEY, "
            +COLUMNAS[1]+" TEXT, "
            +COLUMNAS[2]+" NUMBER)";

   // El contexto de la aplicación
   private Context context;
   // La instancia de la base de datos que nos
   // proporcionará el Helper (ya sea abriendo una base de
   // datos ya existente, creándola si no existe, o actualizándola
   // en el caso de algún cambio de versión)
   private SQLiteDatabase db;
   // Este atributo se utilizará durante la inserción
   private SQLiteStatement insertStatement;

   public DataHelper(Context context) {
      this.context = context;
      // Obtenemos un puntero a una base de datos sobre la que poder
      // escribir mediante la clase MiOpenHelper, que es una clase
      // privada definida dentro de DataHelper
      MiOpenHelper openHelper = new MiOpenHelper(this.context);
      this.db = openHelper.getWritableDatabase();

      // La inserción se realizará mediante lo que se conoce mediante
      // una sentencia SQL compilada. Asociamos al objeto insertStatement
      // el código SQL definido en la constante INSERT. Obsérvese que
      // este código SQL se trata de una sentencia SQL genérica,
parametrizada
      // mediante el símbolo ?
      this.insertStatement = this.db.compileStatement(INSERT);
   }

   public long insert(String name, long number) {
      // Damos valor a los dos elementos genéricos (indicados por el
símbolo ?)
      // de la sentencia de inserción compilada mediante bind
      this.insertStatement.bindString(1, name);
```

```
      this.insertStatement.bindLong(1, number);
      // Y llevamos a cabo la inserción
      return this.insertStatement.executeInsert();
   }

   public int deleteAll() {
            // En este caso hacemos uso de un método de la instancia de
la base
            // de datos para realizar el borrado. Existen también métodos
            // para hacer queries y otras operaciones con la base de
datos
      return db.delete(TABLE_NAME, null, null);
   }

   public List<String> selectAllNombres() {
      List<String> list = new ArrayList<String>();
      // La siguiente instrucción almacena en un cursor todos los valores
      // de las columnas indicadas en COLUMNAS de la tabla TABLE_NAME
      Cursor cursor = db.query(TABLE_NAME, COLUMNAS,
        null, null, null, null, null);
      // El cursor es un iterador que nos permite ir recorriendo
      // los resultados devueltos secuencialmente
      if (cursor.moveToFirst()) {
         do {
                  // Añadimos a la lista que devolveremos como salida
                  // del método el nombre de la ciudad en la posición
actual
            list.add(cursor.getString(1));
         // El método moveToNext devolverá false en el caso de que se
         // haya llegado al final
         } while (cursor.moveToNext());
      }
      if (cursor != null && !cursor.isClosed()) {
         cursor.close();
      }
      return list;
   }

   // Esta clase privada del DataHelper se encarga de proporcionar una
instancia
   // de base de datos a DataHelper sobre la que poder trabajar.
   private static class MiOpenHelper extends SQLiteOpenHelper {
      MiOpenHelper(Context context) {
         super(context, DATABASE_NAME, null, DATABASE_VERSION);
      }

      // Este método se utilizará en el caso en el que la base de datos no
existiera
      @Override
      public void onCreate(SQLiteDatabase db) {
         db.execSQL(CREATE_DB);
      }

      // Este método se ejecutará en el caso en el que se cambie el valor
      // de la constante DATABASE_VERSION. En este caso se borra la base
      // de datos anterior antes de crear una nueva, pero lo ideal sería
      // transferir los datos desde la versión anterior a la nueva
      @Override
      public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
         Log.w("SQL","onUpgrade: eliminando tabla si existe y creándola de
nuevo");
         db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
         onCreate(db);
      }
```

```
    }
}
```

Obsérvese también el uso del `Cursor` y cómo éste se recorre para obtener los resultados. En este caso sólo se han recogido los nombres de las ciudades. También se podían haber obtenido los valores de los habitantes, pero hubiera hecho falta una estructura de datos más elaborada que mantuviera pares de nombre y número de habitantes. En muchas ocasiones los adaptadores no devuelven una lista tras hacer un select, sino directamente el `Cursor`, así que es tarea del programador que hace uso de dicho adaptador recorrer los resultados apuntados por el cursor.

4. Proveedores de contenidos

Los proveedores de contenidos o `ContentProvider` proporcionan una interfaz para publicar y consumir datos, identificando la fuente de datos con una dirección URI que empieza por `content://`. Son una forma más estándar en Android que los adaptadores a una Base de Datos de desacoplar la capa de aplicación de la capa de datos.

4.1. Proveedores nativos

Hay una serie de proveedores de contenidos que Android nos proporciona de manera nativa. Entre estos proveedores de contenidos nativos nos encontramos el `Browser`, `CallLog`, `ContactsContract`, `MediaStore`, `Settings`, `UserDictionary`. Para hacer uso de ellos es necesario añadir los permisos correspondientes en el fichero `AndroidManifest.xml`. Por ejemplo, para acceder al listín telefónico, añadiríamos el siguiente permiso:

```
    ...
        <uses-sdk android:minSdkVersion="8" />
        <uses-permission android:name="android.permission.READ_CONTACTS"/>
</manifest>
```

Para acceder a la información proporcionada por cualquier proveedor de contenidos hacemos uso de la clase `ContentResolver`; en concreto, deberemos utilizar su método `query`, que devuelve un `Cursor` apuntando a los datos solicitados:

```
ContentResolver.query(
                  Uri uri,
                  String[] projection,
                  String selection,
                  String[] selectionArgs,
                  String sortOrder)
```

Por ejemplo, para acceder la lista de contactos utilizaríamos el método `query` de la siguiente forma:

```
ContentResolver cr = getContentResolver();
Cursor cursor = cr.query(ContactsContract.Contacts.CONTENT_URI,
                null, null, null, null);
```

En el ejemplo anterior, la constante `CONTENT_URI` contiene la URI del proveedor de contenidos nativo correspondiente a la lista de contactos. Recuerda que se accede a los datos proporcionados por un proveedor de contenidos siempre a partir de una URI.

Nota:

En versiones anteriores a Android 2.0 las estructuras de datos de los contactos son diferentes y no se accede a esta URI.

Una ventaja de android es que es posible asociar directamente campos de un `Cursor` con componentes de la interfaz de la actividad. En ese caso podemos además hacer que cualquier cambio que ocurra en el cursor se refleje de manera automática (sin tener que programar nosotros el refresco) en el componente gráfico mediante el siguiente código:

```
cursor.setNotificationUri(cr, ContactsContract.Contacts.CONTENT_URI);
```

Por ejemplo, el siguiente código muestra como asignar los datos apuntados por un cursor a un elemento `ListView`. Para ello se le pasa como parámetro al método `setAdapter` del `ListView` una instancia de la clase `SimpleCursorAdapter`:

```
ListView lv = new (ListView)findViewById(R.id.ListView01);
SimpleCursorAdapter adapter = new SimpleCursorAdapter(
        getApplicationContext(),
        R.layout.textviewlayout,
        cursor,
        new String[]{
                ContactsContract.Contacts._ID,
                ContactsContract.Contacts.DISPLAY_NAME},
        new int[]{
                R.id.TextView1,
                R.id.TextView2});
        lv.setAdapter(adapter);
```

En este ejemplo los identificadores `R.id.TextView1` y `R.id.TextView2` se corresponden con vstas del layout que definen cada fila de la `ListView`, como se puede ver en el archivo de ejemplo de layout `textviewlayout.xml` que mostramos a continuación:

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
      android:orientation="horizontal"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content">
        <TextView android:id="@+id/TextView1"
          android:textStyle="bold"
          android:ems="2"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content">
        </TextView>
        <TextView android:id="@+id/TextView2"
          android:textStyle="bold"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content">
        </TextView>
</LinearLayout>
```

La vista de tipo `ListView` (identificada por `R.id.ListView01` en el presente ejemplo) estaría incluida en otro fichero XML de layout en el que se indicaran los componentes de la interfaz de la actividad que contiene dicha lista.

Nota:

Para obtener los números de teléfono de cada persona habría que recorrer el cursor del ejemplo anterior y, por cada persona, crear un nuevo cursor que recorriera los teléfonos, ya que cabe la posibilidad de que una persona posea más de un número de teléfono.

4.2. Proveedores propios

Para acceder a nuestras fuentes de datos propias siguiendo el estándar de diseño Android nos interesa implementar nuestros propios `ContentProvider`. Para ello heredaremos de esta clase y sobrecargaremos una serie de métodos, como `onCreate()`, `delete(...)`, `insert(...)`, `query(...)`, `update(...)`, etc. También debemos sobrecargar el método `getType(Uri)` que nos deolverá el tipo MIME de los contenidos, dependiendo de la URI.

En esta sección vamos a seguir los pasos necesarios para crear un proveedor de contenidos propios que nos permita acceder a nuestra Base de Datos de ciuidades, a partir del adaptador mostrado en la sección de SQLite. Dicho proveedor de contenidos se implementará por medio de la clase `CiudadesProvider`.

La URI base la declararemos en una constante pública de nuestro proveedor y será de tipo `Uri`. Recuerda que la URI es el mecanismo mediante el cual indicamos el origen de los datos. Un ejemplo de URI base podría ser la siguiente:

```
public static final Uri CONTENT_URI =
          Uri.parse("content://es.ua.jtech.daa.proveedores/ciudades");
```

Sobre esta URI base se pueden construir otras URIs que nos permitan especificar ciertos aspectos en cuanto a qué datos se desea acceder. Por ejemplo, para diferenciar entre el acceso a una única fila de datos o a múltiples filas, se pueden emplear números, por ejemplo, "1" si se accede a todas las filas, y "2" si se busca una concreta. En este último caso, también habría que especificar el ID de la fila que se busca.

- Todas las filas: `content://es.ua.jtech.daa.proveedores/ciudades/1`
- Una fila: `content://es.ua.jtech.daa.proveedores/ciudades/2/23`

Para distinguir entre una URI y otra declaramos un `UriMatcher` constante que inicializamos de forma estática en nuestra clase proveedora de contenidos.

```
private static final UriMatcher uriMatcher;
static{
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades",
ALLROWS);
        uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades/#",
```

```
SINGLE_ROW);
}
```

Así, dentro de cada función que sobrecarguemos primero comprobaremos qué resultado debemos devolver, usando una estructura `switch`:

```
@Override
public String getType(Uri uri) {
        switch(uriMatcher.match(uri)){
                case ALLROWS: return
"vnd.ua.cursor.dir/ciudadesprovidercontent";
                case SINGLE_ROW: return
"vnd.ua.cursor.item/ciudadesprovidercontent";
                default: throw new IllegalArgumentException("URI no
soportada: "+uri);
        }
}
```

El código anterior muestra un ejemplo de sobrecarga de `getType(Uri)`, el cual devuelve dos tipos MIME diferentes según el caso. La primera parte de la cadena, antes de la barra, debe terminar en ".dir" si se trata de más de una fila y en .item si se trata de una sola. La primera parte de la cadena debe comenzar por "vnd." y a continuación debe ir el nombre base del dominio sin extensión, en el caso de "www.jtech.ua.es", sería "ua". Este identificador debe ir seguido de ".cursor", ya que los datos se devolverán en un cursor. Por último, después de la barra se suele indicar nombre de la clase seguido de "content", en este caso es "ciudadesprovidercontent" porque la clase se llama `CiudadesProvider`.

Para poder usar el proveedor de contenidos propios, éste se debe declarar en el `AndroidManifest.xml`, dentro del elemento `application`.

```
<provider
        android:name="CiudadesProvider"
        android:authorities="es.ua.jtech.daa.proveedores" />
```

Veamos a continuación el código completo de la clase `CiudadesProvider`. Observaremos muchas similitudes con el adaptador para la base de datos de la sección sobre SQLite, ya que al fin y al cabo se trata de otra interfaz diferente (pero más estándar) para acceder a los mismos datos. También volvemos a utilizar el mismo Helper para abrir la base de datos.

```
package es.ua.jtech.daa.proveedores;

import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.net.Uri;
import android.text.TextUtils;
import android.util.Log;

public class CiudadesProvider extends ContentProvider {

        //Campos típicos de un ContentProvider:
```

```
        public static final Uri CONTENT_URI = Uri.parse(
                "content://es.ua.jtech.daa.proveedores/ciudades");
        private static final int ALLROWS = 1;
        private static final int SINGLE_ROW = 2;
        private static final UriMatcher uriMatcher;
        static{
          uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
          uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades",
ALLROWS);
          uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades/#",
SINGLE_ROW);
        }

        // Campos concretos de nuestro ContentProvider, haciendo
referencia a nuestra
        // base de datos
        public static final String DATABASE_NAME = "mibasededatos.db";
        public static final int DATABASE_VERSION = 1;
        public static final String TABLE_NAME = "ciudades";
        private static final String[] COLUMNAS =
{"_id","nombre","habitantes"};
        private static final String CREATE_DB = "CREATE TABLE " +
TABLE_NAME +
         "("+COLUMNAS[0]+" INTEGER PRIMARY KEY, "
            +COLUMNAS[1]+" TEXT, "
            +COLUMNAS[2]+" NUMBER)";

        // Contexto de la aplicación
        private Context context;
        // Instancia de la base de datos
        private SQLiteDatabase db;

        @Override
        public boolean onCreate() {
                this.context = getContext();
                MiOpenHelper openHelper = new MiOpenHelper(this.context);
                this.db = openHelper.getWritableDatabase();
                return true;
        }

        @Override
        public String getType(Uri uri) {
                switch(uriMatcher.match(uri)){
                case ALLROWS: return
"vnd.ua.cursor.dir/ciudadesprovidercontent";
                case SINGLE_ROW: return
"vnd.ua.cursor.item/ciudadesprovidercontent";
                default: throw new IllegalArgumentException("URI no
soportada: "+uri);
                }
        }

        @Override
        public int delete(Uri uri, String selection, String[]
selectionArgs) {
                int changes = 0;
                switch(uriMatcher.match(uri)){
                case ALLROWS:
                        changes = db.delete(TABLE_NAME, selection,
selectionArgs);
                        break;
                case SINGLE_ROW:
                        String id = uri.getPathSegments().get(1);
                        Log.i("SQL", "delete the id "+id);
                        changes =  db.delete(TABLE_NAME,
```

```
                                "_id = " + id +
                                        (!TextUtils.isEmpty(selection) ?
                                                " AND (" + selection + ')'
:
                                                ""),
                                selectionArgs);
                        break;
                default: throw new IllegalArgumentException("URI no
soportada: "+uri);
                }
                context.getContentResolver().notifyChange(uri, null);
                return changes;
        }

        @Override
        public Uri insert(Uri uri, ContentValues values) {
                long id = db.insert(TABLE_NAME, COLUMNAS[1], values);
                if(id > 0 ){
                        Uri uriInsertado =
ContentUris.withAppendedId(CONTENT_URI, id);
context.getContentResolver().notifyChange(uriInsertado, null);
                        return uriInsertado;
                }
                throw new android.database.SQLException(
                        "No se ha podido insertar en "+uri);
        }

        @Override
        public Cursor query(Uri uri, String[] projection, String
selection,
                        String[] selectionArgs, String sortOrder) {
                return db.query(TABLE_NAME, COLUMNAS, selection,
selectionArgs,
                        null, null, null);
        }

        @Override
        public int update(Uri uri, ContentValues values, String selection,
                        String[] selectionArgs) {
                int changes = 0;
                switch(uriMatcher.match(uri)){
                case ALLROWS:
                        changes =db.update(TABLE_NAME, values, selection,
selectionArgs);
                        break;
                case SINGLE_ROW:
                        String id = uri.getPathSegments().get(1);
                        Log.i("SQL", "delete the id "+id);
                        changes =  db.update(TABLE_NAME, values,
                                "_id = " + id +
                                        (!TextUtils.isEmpty(selection) ?
                                                " AND (" + selection + ')'
                                                : ""),
                                selectionArgs);
                        break;
                default: throw new IllegalArgumentException("URI no
soportada: "+uri);
                }
                context.getContentResolver().notifyChange(uri, null);
                return changes;
        }

        private static class MiOpenHelper extends SQLiteOpenHelper {

                MiOpenHelper(Context context) {
```

```
                        super(context, DATABASE_NAME, null,
DATABASE_VERSION);
                }

                @Override
                public void onCreate(SQLiteDatabase db) {
                        db.execSQL(CREATE_DB);
                }

                @Override
                public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion){
                        Log.w("SQL", "onUpgrade: eliminando tabla si ésta
existe,"+
                                " y creándola de nuevo");
                        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
                        onCreate(db);
                }
        }

}
```

Préstese también atención a las líneas

```
context.getContentResolver().notifyChange(uri, null);
```

Es necesario notificar los cambios al `ContentResolver` para así poder actualizar los cursores que lo hayan pedido a través del método `Cursor.setNotificationUri(...)`, como en el anterior ejemplo del listín telefónico que se mostraba en un componente de la interfaz gráfica.