Ficheros y acceso a datos

Índice

1. Ficheros tradicionales

El uso de ficheros tradicionales está permitido para los programas de Android, aunque como veremos en el resto de capítulos, hay otras tecnologías mucho más avanzadas, como es la base de datos y, para el caso de preferencias de las aplicaciones, las `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());
}
```

Para probarlo en el emulador necesitamos tener creada una tarjeta SD, que se puede crear con el comando que viene con las herramientas del Android SDK:

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

Para copiar fuera del emulador el archivo que hemos grabado en la darjeta de memoria podemos utilizar el comando

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

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 las `SharedPreferences` que almacenan pares de valores: clave y valor.

Aparte de eso, las actividades de Android utilizan eventos y manejadores especializados para guardar el estado de la interfaz de usuario cuando la aplicación pasa a segundo plano. El manejador `onSaveInstanceState` está diseñado para guardar el estado de la actividad cuando ésta debe ser terminada (por parte del sistema operativo) por falta de recursos.

2.1. Crear, guardar y leer preferencias

Para guardar preferencias comunes 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","Boyan");
        editor.commit();

        //Lectura de las preferencias:
        boolean esCierto = pref.getBoolean("esCierto",false);
        String  nombre   = pref.getString("Nombre", "sin nombre");
```

En la lectura de las preferencias se indica un valor por defecto porque, si la preferencia indicada en la clave no existe, será creada y se devolverá el valor por defecto.

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. Intefraz de usuario para las preferencias

Para crear una pantalla de modificación de preferencias, empezamos definiéndola en un XML de Android. Al indicar que se trata de un XML de preferencias, el plugin para Eclipse nos lo guarda en la carpeta `res/xml/`. Por ejemplo, el siguiente `preferencias.xml` define una pantalla de preferencias:

```
<?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 coincidir con las claves que después leamos en las `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.

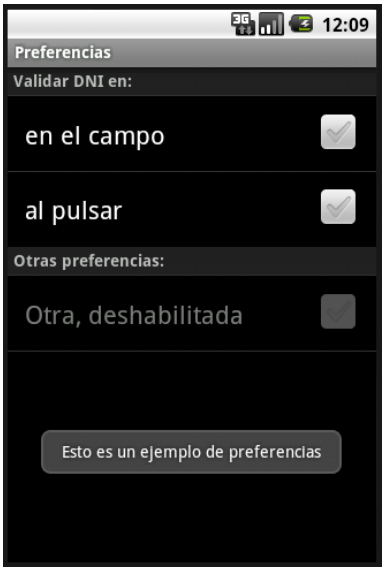
Para que el XML se cargue en una actividad nueva hay que crear una clase que herede de `PreferenceActivity`:

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

Para mostrar esta actividad utilizamos 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 es de código abierto, cumple con los estándares, y es extremadamente ligero. Además 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 en un 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, facilitando además la mantenibilidad de nuestros programas.

Para ilustrar el uso de SQLite en Android vamos a ver una clase adaptador de ejemplo. En esta clase (`MiDBAdapter`) abriremos la base de datos de una manera estándar: mediante un `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.ajdm.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 {

   private static final String DATABASE_NAME = "mibasededatos.db";
   private static final int DATABASE_VERSION = 1;
   private static final String TABLE_NAME = "ciudades";
   private static final String[] COLUMNAS =
{"_id","nombre","habitantes"};
   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)";

   private Context context;
   private SQLiteDatabase db;
   private SQLiteStatement insertStatement;

   public DataHelper(Context context) {
      this.context = context;
      MiOpenHelper openHelper = new MiOpenHelper(this.context);
      this.db = openHelper.getWritableDatabase();
      this.insertStatement = this.db.compileStatement(INSERT);
   }

   public long insert(String name, long number) {
      this.insertStatement.bindString(1, name);
      this.insertStatement.bindLong(1, number);
      return this.insertStatement.executeInsert();
   }

   public int deleteAll() {
      return db.delete(TABLE_NAME, null, null);
   }

   public List<String> selectAllNombres() {
      List<String> list = new ArrayList<String>();
      Cursor cursor = db.query(TABLE_NAME, COLUMNAS,
        null, null, null, null, null);
      if (cursor.moveToFirst()) {
         do {
            list.add(cursor.getString(1));
         } while (cursor.moveToNext());
```

```
      }
      if (cursor != null && !cursor.isClosed()) {
         cursor.close();
      }
      return list;
   }

   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);
      }
   }
}
```

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`, y después el código que hace uso de los métodos del adaptador, tiene que recorrer 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 que los adapters 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. Entre estos proveedores de contenidos nativos nos encontramos el `Browser`, `CallLog`, `ContactsContract`, `MediaStore`, `Settings`, `UserDictionary`. Para ellos además hay que añadir los permisos en el `AndroidManifest.xml`. Por ejemplo, para acceder al listín telefónico:

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

Accederemos al cursor de los contactos con el método

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

Por ejemplo, para acceder a un cursor con la lista completa de contactos:

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

Nota:

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

Podemos mapear campos de un cursor con componentes GUI. En este caso podemos 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. Se consigue con el método:

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

Para asignar un adapter a una lista del GUI, concretamente una `ListView`, utilizamos el método `setAdapter(Adapter)`:

```
                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 views del layout que define cada fila de la `ListView`, como se puede ver en el archivo `textviewlayout.xml` que tendríamos que crear:

```
<?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 lista en sí (identificada por `R.id.ListView01` en el presente ejemplo) estaría en otro XML layout, por ejemplo en `main.xml`.

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 recorra los teléfonos, ya que cada persona puede tener varios números de teléfono.

4.2. Proveedores propios

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

La URI base la declararemos en una constante pública de nuestro proveedor y será de tipo `Uri`. Por ejemplo, la URI base podría ser:

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

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.ajdm.proveedores/ciudades/1
- Una fila: content://es.ua.jtech.ajdm.proveedores/ciudades/2/23

Para distinguir entre una URI y otra nos 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.ajdm.proveedores",
"ciudades", ALLROWS);
          uriMatcher.addURI("es.ua.jtech.ajdm.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 anterior ejemplo es el del método `getType(Uri)` que 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 muchas filas y en .item si se trata de una sóla. 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 de `application`.

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

Veamos la clase `CiudadesProvider` completa en el siguiente listado de código. Observaremos muchas similitudes con el adapter para la base de datos, 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.ajdm.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.ajdm.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.ajdm.proveedores",
"ciudades", ALLROWS);
          uriMatcher.addURI("es.ua.jtech.ajdm.proveedores",
"ciudades/#", SINGLE_ROW);
        }

        public static final String DATABASE_NAME =
"mibasededatos.db";
        public static final int DATABASE_VERSION = 2;
        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)";

        private Context context;
        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.