Introducción

El framework Android le permite almacenar los datos necesarios para el buen funcionamiento de una aplicación de varias formas. Todo depende de las características de almacenamiento que necesite.

Podrá elegir entre los siguientes tipos de almacenamiento:

- **SharedPreferences**: permite almacenar datos en forma de pares clave/valor.
- **Archivos**: permite almacenar datos en archivos (creados en la memoria interna o externa del dispositivo).
- **Almacenamiento en base de datos**: permite crear una base de datos SQLite para almacenar los datos necesarios para el funcionamiento de la aplicación.

SharedPreferences

La clase **SharedPreferences** proporciona un conjunto de métodos que permiten almacenar y recuperar muy fácilmente un par **clave/valor**, con la particularidad de que sólo contiene datos primitivos (float, int, string, Parcelable...).

Estos datos se mantienen hasta que se desinstala la aplicación.

El primer paso para el uso de datos **SharedPreferences** consiste en recuperar una instancia de esta clase. Para ello, dispone de dos métodos:

- **getPreferences(int mode)**: utilice este método si necesita un único archivo de preferencias en su aplicación, ya que este método no le permite especificar el nombre del archivo donde se almacenarán los datos **SharedPreferences**.
- **getSharedPreferences(String name, int mode)**: utilice este método si necesita varios archivos de preferencias que podrá identificar por su nombre.

El parámetro **mode** puede adquirir uno de los siguientes valores:

- **MODE_PRIVATE**: el archivo es privado y por lo tanto, está reservado para su aplicación (valor por defecto).

- **MODE_MULTI_PROCESS**: permite que varios procesos puedan usar el archivo simultáneamente (disponible a partir de Gingerbread - Android 2.3).

```
SharedPreferences myPref = getPreferences(MODE_PRIVATE);

boolean myValue = myPref.getBoolean(SHARED_KEY, false);
```

Una vez se ha obtenido la instancia de la clase **SharedPreferences**, puede utilizar los diferentes métodos disponibles para obtener los datos almacenados. Esta recuperación se realiza mediante una clave que sirve de identificador del dato solicitado. También puede especificar un valor por defecto si el dato no existiera en el **SharedPreferences** consultado.

Para poder añadir datos a un **SharedPreferences**, hay que obtener una instancia de la clase **Editor** (clase que permite modificar un SharedPreferences). Para ello, utilice el método **edit**, disponible a través la instancia **SharedPreferences**.

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

A continuación, puede utilizar los distintos métodos disponibles para agregar valores a la colección de tipo **SharedPreferences**.

```
editor.putBoolean(SHARED_KEY, true) ;
```

Su par **clave/valor** no se agregará de manera efectiva hasta la llamada al método **apply**.

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

Puede añadir varios valores en una colección SharedPreferences antes de invocar al método apply (buena práctica).

Puede actualizar valores ya almacenados en la colección **SharedPreferences** de una aplicación de forma sencilla gracias a la clave que identifica a los datos.

```
SharedPreferences myPref = getPreferences(MODE_PRIVATE);

SharedPreferences.Editor editor = myPref.edit();

editor.putBoolean(SHARED_KEY, false);

editor.commit();
```

Para eliminar un valor de una colección **SharedPreferences**, utilice el método **remove** presente en la clase **Editor**.

```
editor.remove(SHARED_KEY);
```

También puede hacer que una aplicación se suscriba a las modificaciones realizadas en una colección **SharedPreferences** utilizando un listener en la colección. Para ello, utilice el método **registerOnSharedPreferenceChangeListener**.

```
myPref.registerOnSharedPreferenceChangeListener(new

OnSharedPreferenceChangeListener() {

   @Override

   public void onSharedPreferenceChanged(SharedPreferences

sharedPreferences, String key) {

      //Verificar la colección SharedPreferences y la clave

usada

para realizar la modificación

   }

});
```

Utilice el método **unregisterOnSharedPreferenceChangeListener** para anular la suscripción a un **SharedPreferences**.

Almacenamiento interno

Puede guardar archivos directamente en la memoria interna del teléfono. Por defecto, los archivos guardados por una aplicación no son accesibles para el resto de aplicaciones.

→Si el usuario desinstala una aplicación, los archivos correspondientes, albergados en el almacenamiento interno, se eliminarán.

1. Escritura de un archivo

A continuación tenemos un ejemplo de creación de un archivo en el almacenamiento interno del teléfono.

```
String FILENAME = "miArchivo.txt";

String str = "Esto es un ejemplo de almacenamiento interno";

 FileOutputStream fos;

try {

    fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);

    fos.write(str.getBytes());

} catch (FileNotFoundException e) {

   e.printStackTrace();

} catch (IOException e) {

   e.printStackTrace();

} finally {

     fos.close();

}
```

La creación de un archivo en el almacenamiento interno se realiza siguiendo los siguientes pasos:

- Invoque al método **openFileOutput** con el nombre del archivo y el modo de apertura del archivo (privado, público en modo lectura, público en modo escritura, en adición).
- Utilice el método **write** para escribir los datos en el archivo.
- No se olvide de cerrar el archivo para las operaciones de escritura.

Estas operaciones pueden producir dos excepciones: si no se puede encontrar el archivo o si se producen problemas en la escritura (un problema de permisos, por ejemplo).

2. Lectura de un archivo

La lectura de un archivo se realiza del mismo modo. La diferencia está en el uso de los métodos **openFileInput** y **read**.

```
FileInputStream fis;

try {

   fis = openFileInput(FILENAME);

   byte[] buffer = new byte[1024];

   StringBuilder content = new StringBuilder();

   while ((fis.read(buffer)) != -1) {

           content.append(new String(buffer));

   }

} catch (FileNotFoundException e) {

   e.printStackTrace();

} catch (IOException e) {

   e.printStackTrace();

} finally {

    fis.close()

}
```

- La variable **content** alberga el contenido del archivo.
- El buffer se va rellenando a medida que se van haciendo lecturas.
- El método **read** devuelve -1 cuando acaba la lectura del archivo.

Estas operaciones pueden producir dos tipos de excepciones: si no se puede encontrar el archivo o si se producen problemas de lectura.

Los tratamientos de lectura/escritura en un archivo deben realizarse en un Thread diferente al del UI Thread (véase el capítulo Tratamiento en tareas en segundo plano).

3. Utilización de archivos en caché

Si desea almacenar datos temporales sin tener por ello que guardarlos en archivos persistentes, utilice el método **getCacheDir()** para abrir una carpeta de caché.

Alojada en la memoria del teléfono, esta carpeta representa la ubicación que servirá para guardar los archivos temporales de su aplicación.

Si el dispositivo tiene problemas de espacio, Android podrá eliminar estos archivos para obtener más espacio.

Sus archivos de caché deben ocupar un espacio acotado en la memoria del teléfono (1 MB como máximo, por ejemplo).

Recuerde que, en la desinstalación de la aplicación, la carpeta de caché se eliminará.

Almacenamiento externo

1. Comprobar la disponibilidad del almacenamiento externo

Para evitar problemas con la memoria disponible, puede almacenar los archivos de una aplicación en la memoria externa del dispositivo.

Los tratamientos de lectura/escritura en un archivo deben realizarse en un Thread diferente al UI Thread.

Hay dos tipos de memoria externa:

- Almacenamiento externo desmontable (tarjeta SD, por ejemplo).
- Almacenamiento externo no desmontable (alojado en la memoria física del teléfono).

Los archivos creados en el almacenamiento externo se abren en modo lectura para el usuario y el resto de aplicaciones. Además, el usuario podrá eliminar estos archivos sin desinstalar la aplicación.

En un dispositivo Android, el usuario puede utilizar su memoria externa para almacenar sus canciones, fotos y vídeos. Por lo tanto, esta memoria es susceptible de no estar disponible (lectura/escritura) mientras se usa una aplicación. Por este motivo, es necesario comprobar la disponibilidad del almacenamiento externo antes de realizar cualquier tipo de petición de escritura o lectura. El método **getExternalStorageState** le permite comprobar dicha disponibilidad.

```
final String storageState =

Environment.getExternalStorageState();

if (storageState.equals(Environment.MEDIA_MOUNTED)) {

   //Puede leer y escribir en el almacenamiento externo

} else if

(storageState.equals(Environment.MEDIA_MOUNTED_READ_ONLY)) {

   //Solamente puede leer del almacenamiento externo
```

```
} else if (storageState.equals(Environment.MEDIA_REMOVED)) {

   //El almacenamiento externo no está disponible

} else {

   //Resto de casos

}
```

En este ejemplo, puede comprobar el estado del almacenamiento externo entre los diferentes valores disponibles en la clase**Environment**.

A continuación se muestran los diferentes estados posibles:

- **MEDIA_BAD_REMOVAL**: se ha desconectado el almacenamiento sin haber sido desmontado.
- **MEDIA_CHECKING**: se está comprobado el almacenamiento.
- **MEDIA_MOUNTED**: el almacenamiento está disponible en modo lectura/escritura.
- **MEDIA_MOUNTED_READ_ONLY**: el almacenamiento está disponible en modo sólo lectura.
- **MEDIA_NOFS**: el almacenamiento está disponible pero utiliza un formato no soportado.
- **MEDIA_REMOVED**: el almacenamiento no está presente.
- **MEDIA_SHARED**: el almacenamiento se usa en USB (conectado al PC).
- **MEDIA_UNMOUNTABLE**: el almacenamiento está presente pero no puede ser montado.
- **MEDIA_UNMOUNTED**: el almacenamiento está disponible pero no está montado.

→ La escritura en el almacenamiento externo requiere el *permiso* **WRITE_EXTERNAL_STORAGE**.

2. Acceder a los archivos de una aplicación

Puede acceder a los archivos guardados por su aplicación en el almacenamiento externo del dispositivo:

- A partir de la API 8 (Android 2.2), utilice el método **getExternalFileDir(String type)**, que permite obtener una instancia de la clase **File** que apunta al directorio en el que debe guardar sus archivos (no visible para el usuario, como la carpeta de medios).
- El método recibe como parámetro una cadena de caracteres que representa el tipo de la carpeta que se abrirá. Este parámetro puede adquirir los siguientes valores:
    - **DIRECTORY_ALARMS**: ubicación de sonidos que pueden usarse como alarma.
    - **DIRECTORY_DCIM**: ubicación de imágenes y fotos.
    - **DIRECTORY_DOWNLOADS**: ubicación de archivos descargados.
    - **DIRECTORY_MOVIES**: ubicación de películas, series y otros vídeos.
    - **DIRECTORY_MUSIC**: ubicación de música.
    - **DIRECTORY_NOTIFICATION**: ubicación de sonidos usados para las notificaciones.
    - **DIRECTORY_PICTURE**: ubicación de imágenes accesibles para el usuario.
    - **DIRECTORY_PODCAST**: ubicación de archivos de audio que representan podcasts.
    - **DIRECTORY_RINGTONES**: ubicación de sonidos utilizados como tonos de llamada.
    - **null**: devuelve la carpeta raíz de su aplicación.

```
File outFile =

Environment.getExternalFilesDir(Environment.DIRECTORY_DCIM);
```

Desde la versión de Jelly Bean de Android, el almacenamiento externo (en modo lectura) requiere el permiso**READ_EXTERNAL_STORAGE**.

→ Estos archivos se eliminarán cuando se desinstale su aplicación.

3. Acceder a archivos compartidos

Los archivos compartidos se guardan en un directorio accesible para el usuario y el resto de aplicaciones. Estos archivos no se eliminarán cuando se desinstale su aplicación, ya que potencialmente pueden utilizarse en otras aplicaciones.

→Utilice este método solamente para archivos que se compartan con otras aplicaciones.

Estos archivos se encuentran en la raíz del almacenamiento externo y puede acceder a ellos, utilice el método**getExternalStoragePublicDirectory(String type)**. El argumento **type** puede tener los mismos valores que los enumerados en el apartado anterior.

```
File sharedFile =

Environment.getExternalStoragePublicDirectory(Environment.

DIRECTORY_DCIM);
```

Almacenamiento en base de datos

Cada dispositivo Android incluye una base de datos **SQLite** que puede utilizar en sus aplicaciones para almacenar diferentes tipos de datos.

SQLite es una base de datos relacional open source; sólo necesita una pequeña cantidad de memoria en su ejecución.

La sintaxis de **SQLite** se parece a la de SQL y los tipos utilizados son los mismos (INTEGER, TEXT...).

Una base de datos **SQLite** es privada y sólo se puede acceder directamente a ella a través de la aplicación que la ha creado. Se encuentra en una ubicación específica del dispositivo: **data/data/package_de_la_aplicación/databases**

Las operaciones de lectura/escritura no deben realizarse desde el UI thread, sino que hay que externalizarlas en otro **Thread** o en una **AsyncTask**, por ejemplo (véase el capítulo Tratamiento en tareas en segundo plano).

Para la creación de una base de datos, necesitará varios elementos:

- **SQLiteOpenHelper**: se utiliza para facilitar la creación y la gestión de las distintas versiones de una base de datos.
- **SQLiteDatabase**: sirve para exponer los métodos necesarios para la interacción con una base de datos (creación, borrado, actualización…).
- **Cursor**: representa el resultado de una consulta. Este resultado puede contener 0 o más elementos.

SQLite es un motor de bases de datos muy popular en la actualidad por ofrecer características tan interesantes como su pequeño tamaño, no necesitar servidor, precisar poca configuración, ser transaccional y por supuesto ser de código libre.

En Android, la forma típica para crear, actualizar, y conectar con una base de datos SQLite será a través de una clase auxiliar llamada SQLiteOpenHelper, o para ser más exactos, de una clase propia que derive de ella y que debemos personalizar para adaptarnos a las necesidades concretas de nuestra aplicación.

La clase SQLiteOpenHelper tiene tan sólo un constructor, que normalmente no necesitaremos sobrescribir, y dos métodos abstractos, onCreate() y onUpgrade(), que deberemos personalizar con el código necesario para crear nuestra base de datos y para actualizar su estructura respectivamente.

Como ejemplo, nosotros vamos a crear una base de datos muy sencilla llamada BDUsuarios, con una sola tabla llamada Usuarios que contendrá sólo los campos: id, nombre, apellidos, y pass. 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:

```
public class UsuariosSQLiteHelper extends SQLiteOpenHelper {

String sql="CREATE TABLE Usuarios(_id INTEGER PRIMARY KEY
AUTOINCREMENT,nombre TEXT,apellidos TEXT,pass TEXT)";
    public UsuariosSQLiteHelper(Context context, String name,
SQLiteDatabase.CursorFactory factory, int version) {
        super(context, name, factory, version);
    }

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

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

```
            Log.e("BASE DE DATOS","elimino si existe");
            //Se elimina la versión anterior de la tabla
            db.execSQL("DROP TABLE IF EXISTS Usuarios");
            Log.e("BASE DE DATOS","CREO la tabla");
            //Se crea la nueva versión de la tabla
            db.execSQL(sql);
        }
}
```

Lo primero que hacemos es definir una variable llamado sqlCreate donde almacenamos la sentencia SQL para crear una tabla llamada Usuarios con el campo numérico autoincrementado id y alfanuméricos nombre e apellido y password. **NOTA:** No es objetivo de este tutorial describir la sintaxis del lenguaje SQL ni las particularidades del motor de base de datos SQLite. Para más información sobre SQLite puedes consultar la documentación oficial .

El método onCreate() será ejecutado automáticamente por nuestra clase UsuariosDBHelper 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 datos iniciales si son necesarios. En nuestro caso, sólo vamos a crear la tabla Usuarios descrita anteriomente. Para la creación de la tabla utilizaremos la sentencia SQL ya definida y la ejecutaremos contra la base de datos utilizando el método más sencillo de los disponibles en la API de SQLite proporcionada por Android, llamado execSQL(). Este método se limita a ejecutar directamente el código SQL que le pasemos como parámetro.

Por su parte, el método onUpgrade() se lanzará automáticamente cuando sea necesaria una actualización de la estructura de la base de datos o una conversión de los datos. Un ejemplo práctico: imaginemos que publicamos una aplicación que utiliza una tabla con los campos usuario e email (llamémoslo versión 1 de la base de datos). Más adelante, ampliamos la

funcionalidad de nuestra aplicación y necesitamos que la tabla también incluya un campo adicional como por ejemplo con la edad del usuario (versión 2 de nuestra base de datos). Pues bien, para que todo funcione correctamente, la primera vez que ejecutemos la versión ampliada de la aplicación necesitaremos modificar la estructura de la tabla Usuarios para añadir el nuevo campo edad. Pues este tipo de cosas son las que se encargará de hacer automáticamente el método onUpgrade() cuando intentemos abrir una versión concreta de la base de datos que aún no exista. Para ello, como parámetros recibe la versión actual de la base de datos en el sistema, y la nueva versión a la que se quiere convertir. En función de esta pareja de datos necesitaremos realizar unas acciones u otras. En nuestro caso de ejemplo optamos por la opción más sencilla: borrar la tabla actual y volver a crearla con la nueva estructura, pero como se indica en los comentarios del código, lo habitual será que necesitemos algo más de lógica para convertir la base de datos de una versión a otra y por supuesto para conservar los datos registrados hasta el momento.

Una vez definida nuestra clase *helper*, la apertura de la base de datos desde nuestra aplicación resulta ser algo de lo más sencillo. Lo primero será crear un objeto de la claseUsuariosSQLiteHelper al que pasaremos el contexto de la aplicación (en el ejemplo una referencia a la actividad principal), el nombre de la base de datos, un objeto CursorFactor y que típicamente no será necesario (en ese caso pasaremos el valor null), y por último la versión de la base de datos que necesitamos. La simple creación de este objeto puede tener varios efectos:

- 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.

Una vez tenemos una referencia al objeto UsuariosSQLiteHelper, llamaremos a su método getReadableDatabase() o getWritableDatabase() para obtener una referencia a la base de datos, dependiendo si sólo necesitamos consultar los datos o también necesitamos realizar modificaciones, respectivamente.

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().

```
1   package net.sgoliver.android.bd;
2
3   import android.app.Activity;
4   import android.database.sqlite.SQLiteDatabase;
5   import android.os.Bundle;
6
7   public class AndroidBaseDatos extends Activity
8   {
9       @Override
10      public void onCreate(Bundle savedInstanceState)
11      {
12          super.onCreate(savedInstanceState);
            setContentView(R.layout.main);
```

```
13
14              //Abrimos la base de datos 'DBUsuarios' en modo escritura
15              UsuariosSQLiteHelper usdbh =
16                  new UsuariosSQLiteHelper(this, "DBUsuarios", null, 1);
17
18              SQLiteDatabase db = usdbh.getWritableDatabase();
19
20              //Si hemos abierto correctamente la base de datos
21              if(db != null)
22              {
23                  //Insertamos 5 usuarios de ejemplo
24                  for(int i=1; i<=5; i++)
25                  {
26                      //Generamos los datos
27                      int codigo = i;
28                      String nombre = "Usuario" + i;
29                      String apellido = "Apellido" + i;
30                      String pass = "pass" + i;
31
32                      //Insertamos los datos en la tabla Usuarios
33  db.execSQL("INSERT INTO Usuarios (nombre,apellidos,pass) " +
34              "VALUES ('" + nombre + "','" + apellidos + "','" + pass + "')");
35                  }
36
37                  //Cerramos la base de datos
38                  db.close();
39              }
        }
    }
```

Vale, ¿y ahora qué? ¿dónde está la base de datos que acabamos de crear? ¿cómo podemos comprobar que todo ha ido bien y que los registros se han insertado correctamente? Vayamos por partes.

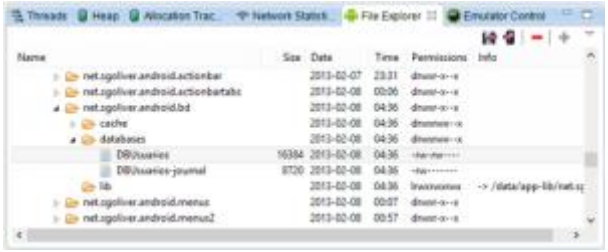
En primer lugar veamos dónde se ha creado nuestra base de datos. Todas las bases de datos SQLite creadas por aplicaciones Android utilizando este método se almacenan en la memoria del teléfono en un fichero con el mismo nombre de la base de datos situado en una ruta que sigue el siguiente patrón:

/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/net.sgoliver.android.bd/databases/DBUsuarios

Para comprobar esto podemos hacer lo siguiente. Una vez ejecutada por primera vez desde Eclipse la aplicación de ejemplo sobre el emulador de Android (y por supuesto antes de cerrarlo) podemos ir a la perspectiva "DDMS" (*Dalvik Debug Monitor Server*) de Eclipse y en la solapa "*File Explorer*" podremos acceder al sistema de archivos del emulador, donde podremos buscar la ruta indicada de la base de datos. Podemos ver esto en la siguiente imagen (click para ampliar):



Con esto ya comprobamos al menos que el fichero de nuestra base de datos se ha creado en la ruta correcta. Ya sólo nos queda comprobar que tanto las tablas creadas como los datos insertados también se han incluido correctamente en la base de datos. Para ello podemos recurrir a dos posibles métodos:

1. Transferir la base de datos a nuestro PC y consultarla con cualquier administrador de bases de datos SQLite.
2. Acceder directamente a la consola de comandos del emulador de Android y utilizar los comandos existentes para acceder y consultar la base de datos SQLite.

El primero de los métodos es sencillo. El fichero de la base de datos podemos transferirlo a nuestro PC utilizando el botón de descarga situado en la esquina superior derecha del explorador de archivos (remarcado en rojo en la imagen anterior). Junto a este botón aparecen otros dos para hacer la operación contraria (copiar un fichero local al sistema de archivos del emulador) y para eliminar ficheros del emulador. Una vez descargado el fichero a nuestro sistema local, podemos utilizar cualquier administrador de SQLite para abrir y consultar la base de datos, por ejemplo SQLite Administrator (freeware).

El segundo método utiliza una estrategia diferente. En vez de descargar la base de datos a nuestro sistema local, somos nosotros los que accedemos de forma remota al emulador a través de su consola de comandos (*shell*). Para ello, con el emulador de Android aún abierto, debemos abrir una consola de MS-DOS y utilizar la utilidad adb.exe (*Android Debug Bridge*) situada en la carpeta platform-tools del SDK de en particular, con las herramientas **adb** y **sqlite3** (únicamente en el emulador o en el dispositivo rooteado).

Para empezar, compruebe que el emulador en cuestión está correctamente conectado y que ha sido correctamente detectado por **adb devices**.

A continuación, conéctese al dispositivo mediante el siguiente comando:

```
adb shell
```

Ya puede acceder a la carpeta que contiene la base de datos creada por la aplicación y enumerar los archivos albergados:

```
# cd /data/data/com.eni.android.database/databases

cd /data/data/com.eni.android.database/databases

# ls

ls

usuarios

usuarios-journal
```

A continuación, acceda a la base de datos (**usuarios**) mediante el comando **sqlite3**, siga las siguientes instrucciones:

```
# sqlite3 usuarios

sqlite3 usuarios

SQLite version 3.7.4

Enter ".help" for instructions

Enter SQL statements terminated with a ";"

sqlite>
```

A continuación, es fácil consultar los datos de una tabla.

```
select * from usuarios;

1|La plataforma Android|Presentación y cronología de

la plataforma Android

2|Entorno de desarrollo|Presentación e instalación

del entorno de desarrollo Android

3|Principios de programación|Presentación de las
características
```

```
específicas deldesarrollo de Android
```

Para más información, consulte la ayuda (comando **.help**).

Actualizar, Eliminar e insertar

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 métodoexecSQL() 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. Cuando creamos la base de datos en el post anteriorya vimos algún ejemplo de esto para insertar los registros de prueba. Otros ejemplos podrían ser los siguientes:

```
1    //Insertar un registro
2    db.execSQL("INSERT INTO Usuarios (codigo,nombre) VALUES (6,'usuariopru') ");
3
4    //Eliminar un registro
5    db.execSQL("DELETE FROM Usuarios WHERE codigo=6 ");
6
7    //Actualizar un registro
8    db.execSQL("UPDATE Usuarios SET nombre='usunuevo' WHERE codigo=6 ");
```

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 tipoContentValues. 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:

```
1    //Creamos el registro a insertar como objeto ContentValues
2    ContentValues nuevoRegistro = new ContentValues();
3    nuevoRegistro.put("codigo", "6");
4    nuevoRegistro.put("nombre","usuariopru");
5
6    //Insertamos el registro en la base de datos
7    db.insert("Usuarios", 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 nombre del usuario con código ‘6’ haríamos lo siguiente:

```
1    //Establecemos los campos-valores a actualizar
2    ContentValues valores = new ContentValues();
3    valores.put("nombre","usunuevo");
4
5    //Actualizamos el registro en la base de datos
6    db.update("Usuarios", valores, "codigo=6", null);
```

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 con código '6' haríamos lo siguiente:

```
1    //Eliminamos el registro del usuario '6'
2    db.delete("Usuarios", "codigo=6", null);
```

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 (lo que eliminaría todos los registros de la tabla).

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 condiones de la sentencia SQL. Éstos 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:

```
1       //Eliminar un registro con execSQL(), utilizando argumentos
2       String[] args = new String[]{"usuario1"};
3       db.execSQL("DELETE FROM Usuarios WHERE nombre=?", args);
4
5       //Actualizar dos registros con update(), utilizando argumentos
6       ContentValues valores = new ContentValues();
7       valores.put("nombre","usunuevo");
8
9       String[] args = new String[]{"usuario1", "usuario2"};
10      db.update("Usuarios", valores, "nombre=? OR nombre=?", args);
```

Esta forma de pasar a la sentencia SQL determinados datos variables puede ayudarnos además a escribir código más limpio y evitar posibles errores.

Para este apartado he continuado con la aplicación de ejemplo del apartado anterior, a la que he añadido dos cuadros de texto para poder introducir el código y nombre de un usuario y tres botones para insertar, actualizar o eliminar dicha información.

Consultar/Recuperar registros

De forma análoga a lo que vimos para las sentencias de modificación de datos, 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, y como 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 claseSQLiteDatabase. 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. Sirva la siguiente consulta a modo de ejemplo:

```
1    Cursor c = db.rawQuery(" SELECT codigo,nombre FROM Usuarios WHERE nombre='usu1'
```

Como en el caso de los métodos de modificación de datos, también podemos añadir a este método una lista de argumentos variables que hayamos indicado en el comando SQL con el símbolo '?', por ejemplo así:

```
1    String[] args = new String[] {"usu1"};

2    Cursor c = db.rawQuery(" SELECT codigo,nombre FROM Usuarios WHERE nombre=? ", ar
```

Más adelante en este artículo veremos cómo podemos manipular el objeto Cursor para recuperar los datos obtenidos.

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, y 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. Veamos el mismo ejemplo anterior utilizando el método query():

```
1    String[] campos = new String[] {"codigo", "nombre"};

2    String[] args = new String[] {"usu1"};

3

4    Cursor c = db.query("Usuarios", campos, "usuario=?", 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 métodosgetXXX(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 (cero), por lo que la segunda columna tiene índice 1], en caso de contener un dato de tipo real llamaríamos agetDouble(1), y de forma análoga para todos los tipos de datos existentes.

Con todo esto en cuenta, veamos cómo podríamos recorrer el cursor devuelto por el ejemplo anterior:

```
1
      String[] campos = new String[] {"codigo", "nombre"};
2
      String[] args = new String[] {"usu1"};
3

4
      Cursor c = db.query("Usuarios", campos, "nombre=?", args, null, null, null);
5

6     //Nos aseguramos de que existe al menos un registro
7     if (c.moveToFirst()) {
8          //Recorremos el cursor hasta que no haya más registros
9          do {
10              String codigo= c.getString(0);
11              String nombre = c.getString(1);
          } while(c.moveToNext());
12    }
13
```

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. Podéis consultar la lista completa de métodos disponibles en la clase Cursor en la documentación oficial de Android.

En este apartado he seguido ampliando la aplicación de ejemplo anterior para añadir la posibilidad de recuperar todos los registros de la tabla Usuarios pulsando un nuevo botón de consulta.