Persistencia en Android: ficheros y SQLite

Índice

1. Introducción

En este módulo trataremos diferentes mecanismos existentes en los sistemas Android e iOS para el almacenamiento de datos, tanto en la memoria del dispositivo como en memoria externa. Comenzaremos en primer lugar tratando el manejo de ficheros en Android para a continuación explicar cómo utilizar el motor de base de datos SQLite en nuestras aplicaciones en dicho sistema. A continuación presentaremos los proveedores de contenidos, los cuales nos proporcionan un mecanismo estándar para el intercambio de información entre aplicaciones Android. Veremos cómo usar los proporcionados por el sistema y crear los nuestros propios.

En las sesiones 3 y 4 comentaremos los métodos más usados para almacenar datos dentro de nuestras aplicaciones iOS. Empezaremos explicando de forma detallada el uso de los ficheros de propiedades (*plist*), un tipo de ficheros con formato XML que es muy usado en sistemas iOS. Continuaremos con el manejo de bases de datos SQLite dentro de las aplicaciones iOS, explicando la libreria que disponemos dentro del SDK de iPhone. Por último, en la sesión 4 del módulo empezaremos explicando el método de almacenamiento mediante variables de tipo *User Defaults* para terminar con algo más complejo como es el uso de *Core Data*, forma de almacenamiento de datos muy extendida en estos últimos años dentro de los sistemas iOS.

2. Manejo de ficheros tradicionales en Android

El uso de ficheros tradicionales está permitido para los programas de Android, aunque como veremos en ésta y otras sesiones, hay otras tecnologías mucho más avanzadas, como el uso del motor de base de datos SQLite y, para el caso de preferencias de las aplicaciones, las `SharedPreferences`.

2.1. Apertura de ficheros

En Android podemos abrir un fichero para lectura o para escritura de la siguiente forma:

```
// Abrir un fichero de salida privado a la aplicación
FileOutputStream fos = openFileOutput("fichero.txt",
Context.MODE_PRIVATE);
// Abrir un fichero de entrada
FileInputStream  fis = openFileInput("fichero.txt");
```

Al abrir el fichero para salida, el modo `Context.MODE_PRIVATE` hace que éste sea privado a la aplicación. Si el fichero a abrir no existe, éste se crearía. Existe un parámetro para permitir compartir el fichero, pero lo adecuado en este caso sería hacer uso de proveedores de contenidos, por lo que no trataremos dicho parámetro.

Por otra parte, el parámetro `Context.MODE_APPEND` permite añadir contenido a un fichero que ya se hubiera creado previamente (o crear uno nuevo en el caso en el que éste no

existiera).

Estos métodos tan sólo permiten abrir ficheros en la carpeta de la aplicación, por lo que si se intenta añadir algún separador de ruta se producirá una excepción. El fichero abierto utilizando `OpenFileOutput` (usando cualquiera de los dos parámetros anteriores) se guardaría, dentro del árbol de directorios del dispositivo, en la carpeta `/data/data/[paquete]/files/`, donde `[paquete]` sería el nombre del paquete de la aplicación. Se trata de ficheros de texto normales, por los que se puede acceder a ellos utilizando el shell de adb y utilizando el comando `cat`. Para acceder al shell podemos hacer lo siguiente:

```
adb shell
```

Por supuesto no debemos olvidarnos de cerrar el fichero una vez vayamos a dejar de utilizarlo mediante la correspondiente llamada al método `close()`:

```
fos.close();
fis.close();
```

2.2. Ficheros como recursos

Es posible añadir ficheros como recursos de la aplicación. Para ello los almacenamos en la carpeta `/res/raw/` de nuestro proyecto. Estos ficheros serán de sólo lectura, y para acceder a ellos llamaremos al método `openRawResource` del objeto `Resource` de nuestra aplicación. Como resultado obtendremos un objeto de tipo `InputStream`. Un ejemplo sería el que se muestra a continuación:

```
Resources myResources = getResources();
InputStream myFile = myResources.openRawResource(R.raw.fichero);
```

Añadir ficheros a los recursos de la aplicación es un mecanismo para disponer de fuentes de datos estáticas de gran tamaño, como podría ser por ejemplo un diccionario. Esto será así en aquellos casos en los que no sea aconsejable (o deseable) almacenar esta información en una base de datos.

Como en el caso de cualquier otro recurso, sería posible guardar diferentes versiones de un mismo archivo para diferentes configuraciones del dispositivo. Podríamos tener por ejemplo un fichero de diccionario para diferentes idiomas.

2.3. Operar con ficheros

Una alternativa simple para escribir en un fichero o leer de él es utilizar las clases `DataInputStream` y `DataOutputStream`, que nos permiten leer o escribir a partir de diferentes tipos de datos, como enteros o cadenas. En el siguiente ejemplo vemos cómo es posible guardar una cadena en un fichero y a continuación leerla.

```
FileOutputStream fos = openFileOutput("fichero.txt",
Context.MODE_PRIVATE);
String cadenaOutput = new String("Contenido del fichero\n");
```

```
DataOutputStream dos = new DataOutputStream(fos);
dos.writeBytes(cadenaOutput);
fos.close();

FileInputStream fin = openFileInput("fichero.txt");
DataInputStream dis = new DataInputStream(fin);
String string = dis.readLine();
// Hacer algo con la cadena
fin.close();
```

2.4. Almacenar datos en la tarjeta de memoria

Veamos ahora qué pasos debemos seguir para poder escribir datos en la tarjeta de memoria (SD card). Para ello se requiere el uso del método `Environment.getExternalStorageDirectory()` y una instancia de la clase `FileWriter`, tal como se muestra en el siguiente ejemplo:

```
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\n");
                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, lo cual se puede conseguir mediante la herramienta del Android SDK `mksdcard`:

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

Podremos cargar esta tarjeta SD en nuestra terminal emulada seleccionando el archivo recién creado, tras seleccionar la opción adecuada, dentro del apartado SD card de la ventana de edición de las propiedades de una terminal emulada, la cual a su vez puede ser accedida mediante la opción Android SDK and AVD Manager bajo el menú Window. Para copiar fuera del emulador el archivo que hemos grabado en la tarjeta de memoria podemos utilizar el comando

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

Por último, no debemos olvidar añadir el permiso correspondiente en el archivo Manifest de nuestra aplicación para poder escribir datos en un dispositivo de almacenamiento externo. Para ello añadimos lo siguiente al elemento `manifest` del Manifest.xml (recuerda que esto no va ni dentro del elemento `application` ni dentro del elemento `activity`):

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

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. Su utilidad es evidente en el caso de aplicaciones de pequeño tamaño, pues éstas no requerirán la instalación adicional de ningún gestor de bases de datos. Esto también será así en el caso de dispositivos empotrados con recursos limitados. Android incluye soporte a SQLite, y en esta sección veremos cómo usarlo.

Por medio de SQLite podremos crear bases de datos relacionales independientes para nuestras aplicaciones. Con ellas podremos almacenar datos complejos y estructurados. Aunque el diseño de bases de datos quedará fuera del contenido de este curso, ya que necesitaríamos de más tiempo para poder tratarlo, es conveniente resaltar el hecho de que las mejores prácticas en este campo siguen siendo útiles en el caso de aplicaciones Android. En particular, la normalización de datos para evitar redundancia es un paso muy importante en el caso en el que estemos diseñando una base de datos para dispositivos con recursos limitados.

En el caso del sistema Android, SQLite se implementa como una librería C compacta, en lugar de ejecutarse en un proceso propio. Debido a ello, cualquier base de datos que creemos será integrada como parte de su correspondiente aplicación. Esto reduce dependencias externas, minimiza la latencia, y simplifica ciertos procesos como la sincronización.

Nota:

Las bases de datos de Android se almacenan en el directorio `/data/data/[PAQUETE]/databases/`, donde `[PAQUETE]` es el paquete correspondiente a la aplicación. Por defecto todas las bases de datos son privadas y tan sólo accesibles por la aplicación que las creó.

3.1. Content Values

Para insertar nuevas filas en tablas haremos uso de objetos del a clase `ContentValues`. Cada objeto de este tipo representa una única fila en una tabla y se encarga de emparejar nombres de columnas con valores concretos. Hay que tener en cuenta que SQLite difiere de muchos otros motores de bases de datos en cuanto al tipado de las columnas. La idea básicamente es que los valores para las columnas no tienen por qué ser de un único tipo; es decir, los datos en SQLite son débilmente tipados. La consecuencia de esto es que no es necesario comprobar tipos cuando se asignan o extraen valores de cada columna de una fila, con lo que se mejora la eficiencia.

3.2. Cursores

Los cursores son una herramienta fundamental en el tratamiento de información en Android, y también cuando se trabaja con bases de datos. Por lo tanto, en esta sección introducimos este concepto tan importante.

Cualquier consulta a una base de datos en Android devolverá un objeto de la clase `Cursor`. Los cursores no contienen una copia de todos los resultados de la consulta, sino que más bien se trata de punteros al conjunto de resultados. Estos objetos proporcionan un mecanismo para controlar nuestra posición (fila) en el conjunto de resultados obtenidos tras la consulta.

La clase `Cursor` incluye, entre otras, las siguientes funciones de navegación entre resultados:

- `moveToFirst`: desplaza el cursor a la primera fila de los resultados de la consulta.
- `moveToNext`: desplaza el cursor a la siguiente fila.
- `moveToPrevious`: desplaza el cursor a la fila anterior.
- `getCount`: devuelve el número de filas del conjunto de resultados de la consulta.
- `getColumnName`: devuelve el nombre de la columna especificada mediante un índice que se pasa como parámetro.
- `getColumnNames`: devuelve un array de cadenas conteniendo el nombre de todas las columnas en el cursor.
- `moveToPosition`: mueve el cursor a la fila especificada.
- `getPosition`: devuelve el índice de la posición apuntada actualmente por el cursor.

A lo largo de esta sesión aprenderemos cómo realizar consultas a una base de datos y como extraer valores o nombres de columnas concretos a partir de los cursores obtenidos por medio de dichas consultas.

3.3. Trabajar con bases de datos SQLite

Suele considerarse una buena práctica el crear una clase auxiliar que nos facilite la interacción con la base de datos. Se tratará de una clase de tipo adaptador, que creará una capa de abstracción que encapsulará dichas interacciones. De esta forma podremos añadir, eliminar o actualizar elementos en la base de datos sin necesidad de introducir ni una única instrucción en SQL en el resto del código, permitiendo además que se hagan las comprobaciones de tipos correspondientes.

Una clase de este tipo debería incluir métodos para crear, abrir o cerrar la base de datos. También puede ser un lugar conveniente para publicar constantes estáticas relacionadas con la base de datos, como por ejemplo constantes que almacenen el nombre de la tabla o las diferentes columnas.

A continuación se muestra el código de lo que podría ser un adaptador para una base de datos estándar. Incluye una extensión de la clase `SQLiteOpenHelper`, que será tratada en más detalle en la siguiente sección, y que se utiliza para simplificar la apertura, la creación y la actualización de la base de datos.

```
import android.content.Context;
import android.database.*;
import android.database.sqlite.*;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.util.Log;

public class MiAdaptadorBD {
        private static final String NOMBRE_BASE_DATOS =
"mibasededatos.db";
        private static final String TABLA_BASE_DATOS = "mainTable";
        private static final int VERSION_BASE_DATOS = 1;

        // Nombre de la columna de clave primaria
        public static final String CP_ID="_id";
        // El nombre e índice de cada columna en la base de datos
        public static final String COLUMNA_NOMBRE="nombre";
        public static final int COLUMNA_INDICE = 1;
        // PENDIENTE: crear campos adicionales para el resto de columnas
        // de la base de datos

        // Sentencia SQL para crear una nueva base de datos
        private static final String CREAR_BASE_DATOS = "create table " +
                TABLA_BASE_DATOS + " (" + CP_ID +
                " integer primary key autoincrement, " +
                COLUMNA_NOMBRE + " text not null);";

        // Variable para almacenar la instancia de la base de datos
        private SQLiteDatabase db;
        // Contexto de la aplicación que está usando la base de datos
        private final Context contexto;
        // Clase helper, usada para crear o actualizar la base de datos
        // (hablamos de ella en la siguiente sección)
        private miHelperBD dbHelper;

        // Crea la base de datos con ayuda del helper
        public MiAdaptadorBD(Context _contexto) {
                contexto = _contexto;
                dbHelper = new miHelperBD(contexto, NOMBRE_BASE_DATOS,
null,
                        VERSION_BASE_DATOS);
        }

        // Abre una base de datos ya existente
        public MiAdaptadorBD open() throws SQLException {
                db = dbHelper.getWritableDatabase();
                return this;
        }

        // Cierra la base de datos, cuando ésta ya no va a ser utilizada
        public void close() {
                db.close();
        }

        // Método para insertar un elemento en la tabla de la base de
datos
        public int insertar(MiObjeto _miObjeto) {
                // PENDIENTE: crear nuevos elementos ContentValues para
representar
                // al objeto e insertarlos en la base de datos

                // Devolvemos el índice del nuevo elemento en la base de
datos
                return index;
        }
```

```
        // Método para eliminar un elemento de la tabla de la base de
datos
        // Devuelve un valor booleano indicando el éxito o no de la
        // operación
        public boolean eliminar(long _indiceFila) {
                return db.delete(TABLA_BASE_DATOS, CP_ID + "=" +
_indiceFila, null) > 0;
        }

        // Método para obtener un Cursor que apunte a todas las filas
contenidas
        // en la tabla de la base de datos
        public Cursor obtenerTodos () {
                return db.query(TABLA_BASE_DATOS, new String[] {CP_ID,
COLUMNA_NOMBRE},
                                                null, null, null, null,
null);
        }

        // Método para obtener un elemento determinado de la base de datos
        // a partir de su índice de fila
        public MiObjeto getObjeto(long _indiceFila) {
                // PENDIENTE: obtener un cursor a una fila de la base
                // de datos y usar sus valores para inicializar una
                // instancia de MiObjeto

                return objectInstance;
        }

        // Método para actualizar un elemento de la tabla de la base de
datos
        public boolean actualizar(long _indiceFila, MiObjeto _miObjeto) {
                // PENDIENTE: crear nuevos ContentValues a partir del
objeto
                // y usarlos para actualizar una fila en la tabla
correspondiente

                return true;
        }

        // Clase privada auxiliar para ayudar en la creación y
actualización
        // de la base de datos
        private static class miHelperBD extends SQLiteOpenHelper {

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

                // Método llamado cuando la base de datos no existe en el
disco
                // y la clase Helper necesita crearla desde cero
                @Override
                public void onCreate(SQLiteDatabase _db) {
                        _db.execSQL(CREAR_BASE_DATOS);
                }

                // Método llamado cuando la versión de la base de
                // datos en disco es diferente a la indicada; en
                // este caso la base de datos en disco necesita ser
                // actualizada
                @Override
                public void onUpgrade(SQLiteDatabase _db, int
_versionAnterior,
```

```
                  int _versionNueva) {
                        // Mostramos en LogCat la operación
                        Log.w("AdaptadorBD", "Actualizando desde la
versión " +
                                _versionAnterior + " a la versión " +
                                _versionNueva + ", lo cual borrará los
datos
                                previamente almacenados");
                        // Actualizamos la base de datos para que se
adapte a la nueva versión.
                        // La forma más sencilla de llevar a cabo dicha
actualización es
                        // eliminar la tabla antigua y crear una nueva
                        _db.execSQL("DROP TABLE IF EXISTS " +
TABLA_BASE_DATOS);
                        onCreate(_db);
                  }
        }
}
```

3.4. La clase SQLiteOpenHelper

La clase `SQLiteOpenHelper` es una clase abstracta utilizada como medio para implementar un patrón de creación, apertura y actualización de una determinada base de datos. Al implementar una subclase de `SQLiteOpenHelper` podemos abstraernos de la lógica subyacente a la decisión de crear o actualizar una base de datos de manera previa a que ésta deba ser abierta.

En el ejemplo de código anterior se vio cómo crear una subclase de `SQLiteOpenHelper` por medio de la sobrecarga de sus métodos `onCreate` y `onUpgrade`, los cuales permiten crear una nueva base de datos o actualizar a una nueva versión, respectivamente.

Nota:

En el ejemplo anterior, el método `onUpgrade` simplemente eliminaba la tabla existente y la reemplazaba con la nueva definición de la misma. En la práctica la mejor solución sería migrar los datos ya existentes a la nueva tabla.

Mediante los métodos `getReadableDatabase` y `getWritableDatabase` podemos abrir y obtener una instancia de la base de datos de sólo lectura o de lectura y escritura respectivamente. La llamada a `getWritableDatabase` podría fallar debido a falta de espacio en disco o cuestiones relativas a permisos, por lo que es una buena práctica tener en cuenta esta posibilidad por medio del mecanismo de manejo de excepciones de Java. En caso de fallo la solución podría pasar por al menos proporcionar una copia de sólo lectura, tal como se muestra en el siguiente ejemplo:

```
dbHelper = new miHelperBD(contexto, NOMBRE_BASE_DATOS, null,
VERSION_BASE_DATOS);

SQLiteDatabase db;
try {
        db = dbHelper.getWritableDatabase();
} catch (SQLiteException ex) {
        db = dbHelper.getReadableDatabase();
```

```
}
```

Al hacer una llamada a cualquiera de los dos métodos anteriores se invocarán los manejadores de evento adecuados. Así pues, si por ejemplo la base de datos no existiera en disco, se ejecutaría el código de `onCreate`. Si por otra parte la versión de la base de datos se hubiera modificado, se dispararía el manejador `onUpgrade`. Lo que sucede entonces es que al hacer una llamada a `getWritableDatabase` o `getReadableDatabase` se devolverá una base de datos nueva, actualizada o ya existente, según el caso.

3.5. Crear una base de datos sin SQLiteHelper

También es posible crear y abrir bases de datos sin necesidad de utilizar una subclase de `SQLiteHelper`. Para ello haremos uso del método `openOrCreateDatabase`. En este caso será necesario también hacer uso del método `execSQL` sobre la instancia de la base de datos devuelta por el método anterior para ejecutar los comandos SQL que permitirán crear las tablas de la base de datos y establecer las relaciones entre ellas. El siguiente código muestra un ejemplo:

```
private static final String NOMBRE_BASE_DATOS = "mibasededatos.db";
private static final String TABLA_BASE_DATOS = "tablaPrincipal";

private static final String CREAR_BASE_DATOS =
        "create table " + TABLA_BASE_DATOS + " /_id integer primare key
autoincrement," +
        "columna_uno text not null);";

SQLiteDatabase db;

private void crearBaseDatos() {
        db = openOrCreateDatabase(NOMBRE_BASE_DATOS, Context.MODE_PRIVATE,
null);
        db.execSQL(CREAR_BASE_DATOS);
}
```

Nota:

Aunque no es necesariamente obligatorio, es recomendable que cada tabla incluya un campo de tipo clave primaria con autoincremento a modo de índice para cada fila. En el caso en el que se desee compartir la base de datos mediante un proveedor de contenidos, un campo único de este tipo es obligatorio.

3.6. Realizar una consulta

El resultado de cualquier consulta a una base de datos será un objeto de la clase `Cursor`. Esto permite a Android manejar los recursos de manera más eficiente, de tal forma que en lugar de devolver todos los resultados, éstos se van proporcionando conforme se necesitan.

Para realizar una consulta a una base de datos haremos uso del método `query` de la instancia correspondiente de la clase `SQLiteDatabase`. Los parámetros que requiere este

método son los siguientes:

- Un booleano opcional que permite indicar si el resultado debería contener tan sólo valores únicos.
- El nombre de la tabla sobre la que realizar la consulta.
- Un array de cadenas que contenta un listado de las columnas que deben incluirse en el resultado.
- Una cláusula *where* que defina las filas que se deben obtener de la tabla. Se puede utilizar el comodín *?*, el cual será reemplazado por los valores que se pasen a través del siguiente parámetro.
- Un array de cadenas correspondientes a argumentos de selección que reemplazarán los símbolos *?* en la cláusula *where*.
- Una cláusula *group by* que define cómo se agruparán las filas obtenidas como resultado.
- Un filtro *having* que indique que grupos incluir en el resultado en el caso en el que se haya hecho uso del parámetro anterior.
- Una cadena que describa la ordenación que se llevará a cabo sobre las filas obtenidas como resultado.
- Una cadena opcional para definir un límite en el número de resultados a devolver.

Nota:

Como se puede observar, la mayoría de estos parámetros hacen referencia al lenguaje SQL. Tratar este tema queda fuera de los objetivos de este curso, por lo que en nuestros ejemplos le daremos en la mayoría de las ocasiones un valor `null` a estos parámetros.

El siguiente código muestra un ejemplo de consultas utilizando diversos parámetros:

```
// Devolver los valores de las columnas uno y tres para todas las
// filas, sin duplicados
String[] columnas = new String[] {CLAVE_ID, CLAVE_COL1, CLAVE_COL3};

Cursor todasFilas = db.query(true, TABLA_BASE_DATOS, columnas, null, null,
null, null,
        null, null);

// Devolvemos los valores de todas las columnas para aquellas filas cuyo
valor de la
// columna 3 sea igual al de una determinada variable. Ordenamos las filas
según el
// valor de la columna 5
// Como queremos los valores de todas las columnas, le damos al parámetro
// correspondiente a las columnas a devolver el valor null
// En este caso no se ha hecho uso del primer parámetro booleano, que es
optativo
String where = CLAVE_COL3 + "=" + valor;
String orden = CLAVE_COL5;
Cursor miResultado = db.query(TABLA_BASE_DATOS, null, where, null, null,
null, orden);
```

3.7. Extraer resultados de un cursor

El primer paso para obtener resultados de un cursor será desplazarnos a la posición a partir de la cual queremos obtener los datos, usando cualquiera de los métodos especificados anteriormente (como por ejemplo `moveToFirst` o `moveToPosition`). Una vez hecho esto hacemos uso de métodos `get[TIPO]` pasando como parámetro el índice de la columna. Esto tendrá como resultado la devolución del valor para dicha columna de la fila actualmente apuntada por el cursor:

```
String valor = miResultado.getString(indiceColumna);
```

A continuación podemos ver un ejemplo más completo, en el que se va iterando a lo largo de los resultados apuntados por un cursor, extrayendo y sumando los valores de una columna de flotantes:

```
int COLUMNA_VALORES_FLOTANTES = 2;
Cursor miResultado = db.query("Tabla", null, null, null, null, null,
null);
float total = 0;

// Nos aseguramos de que se haya devuelto al menos una fila
if (miResultado.moveToFirst()) {
        // Iteramos sobre el cursor
        do {
                float valor =
miResultado.getFloat(COLUMNA_VALORES_FLOTANTES);
                total += valor;
        } while (miResultado.moveToNext());
}

float media = total / miResultado.getCount();
```

Nota:

Debido a que las columnas de las bases de datos en SQLite son debilmente tipadas, podemos realizar castings cuando sea necesario. Por ejemplo, los valores guardados como flotantes en la base de datos podrían leerse más adelante como cadenas.

3.8. Añadir, actualizar y borrar filas

La clase `SQLiteDatabase` proporciona los métodos `insert`, `delete` y `update`, que encapsulan las instrucciones SQL requeridas para llevar a cabo estas acciones. Además disponemos de la posibilidad de utilizar el método `execSQL`, el cual nos permite ejecutar cualquier sentencia SQL válida en nuestras tablas de la base de datos en el caso en el que queramos llevar a cabo estas (u otras) operaciones manualmente.

Nota:

Cada vez que modifiques los valores de la base de datos puedes actualizar un cursor que se pueda ver afectado por medio del método `refreshQuery` de la clase `Cursor`.

Para **insertar** una nueva fila es necesario construir un objeto de la clase `ContentValues` y usar su método `put` para proporcionar valor a cada una de sus columnas. Para insertar la

nueva fila pasaremos como parámetro este objeto del tipo `ContentValues` al método `insert` de la instancia de la base de datos, junto por supuesto con el nombre de la tabla. A continuación se muestra un ejemplo:

```
// Crear la nueva fila
ContentValues nuevaFila = new ContentValues();

// Asignamos valores a cada columna
nuevaFila.put(NOMBRE_COLUMNA, nuevoValor);
[.. Repetir para el resto de columnas ..]

// Insertar la nueva fila en la tabla
db.insert(TABLA_BASE_DATOS, null, nuevaFila);
```

Nota:

Los ficheros, como imágenes o ficheros de audio, no se suelen almacenar dentro de tablas en una base de datos. En estos casos se suele optar por almacenar en la base de datos una cadena con la ruta al fichero o una URI.

La operación de **actualizar** una fila también puede ser llevada a cabo por medio del uso de la clase `ContentValues`. En este caso deberemos llamar al método `update` de la instancia de la base de datos, pasando como parámetro el nombre de la tabla, el objeto `ContentValues` y una cláusula *where* que especifique la fila o filas a actualizar, tal como se puede ver en el siguiente ejemplo:

```
// Creamos el objeto utilizado para definir el contenido a actualizar
ContentValues valoresActualizar = new ContentValues();

// Asignamos valores a las columnas correspondientes
valoresActualizar.put(NOMBRE_COLUMNA, nuevoValor);
[.. Repetir para el resto de columnas a actualizar ..]

String where = CLAVE_ID + "=" + idFila;

// Actualizamos la fila con el índice especificado en la instrucción
anterior
db.update(TABLA_BASE_DATOS, valoresActualizar, where, null);
```

Por último, para **borrar** una fila, simplemente llamaremos al método `delete` de la instancia de la base de datos, especificando en este caso el nombre de la tabla que se verá afectada por la operación y una cláusula *where* que utilizaremos para especificar las filas que queremos borrar. Un ejemplo podría ser el siguiente:

```
db.delete(TABLA_BASE_DATOS, CLAVE_ID + "=" + idFila, null);
```