



Persistencia de la información

Índice



Persistencia de la información

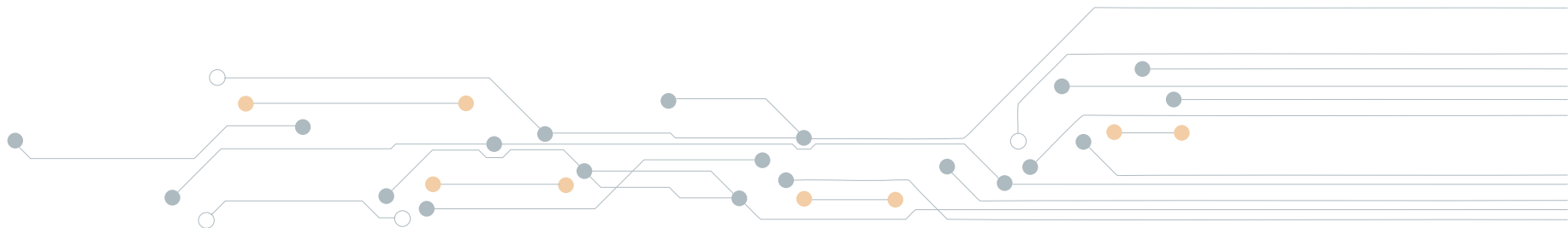
1 Introducción	3
2 Acceso a ficheros	4
3 Utilización de SharedPreferences	13
3.1 Almacenamiento de datos	13
3.2 Recuperación de datos	15
4 Acceso a bases de datos	16
4.1 Creación de la clase Helpe	17
4.2 La clase SQLiteDatabase	20
5 Proveedores de contenido	33
5.1 Las clases ContentResolver y ContentProvider	34
5.2 Permisos	38

1. Introducción

En todas las aplicaciones Android que hemos ido haciendo hasta el momento, la información manejada por el programa residía siempre en memoria, de modo que cuando la aplicación se detenía los datos utilizados se perdían.

La mayoría de aplicaciones Android trabajan con datos persistentes que son almacenados dentro del terminal. Dicha persistencia puede llevarse a cabo en un **fichero estándar** o en una **base de datos**. A lo largo de este apartado estudiaremos como almacenar y recuperar información, tanto con ficheros como con bases de datos.

Pero al mismo tiempo, existe una gran cantidad de información de aplicaciones del sistema a la que se puede acceder a través de unos componentes especiales conocidos como proveedores de contenido. A lo largo de este apartado te explicaremos también como utilizar estos componentes para acceder a esta valiosa información desde nuestras aplicaciones Android.



2. Acceso a ficheros

Desde un aplicación Android se puede almacenar y recuperar información en ficheros almacenados en el propio terminal. Para ello, utilizaremos las mismas clases que se emplean en las aplicaciones Java Estándar, y que podemos encontrar en el paquete **java.io**.

Para acceder al sistema de ficheros desde una aplicación Android, la interfaz Context, que como sabemos es implementada por Activity, nos proporciona los siguientes métodos:

- **openOutputStream(String name, int mode).** A partir del nombre del fichero devuelve un objeto FileOutputStream para escribir en el mismo. El fichero es local a la aplicación, se encuentra dentro de las carpetas de la misma, por lo que la ruta será siempre relativa. El parámetro mode hace referencia al modo de acceso al fichero. Los dos valores permitidos están definidos en dos constantes de Context:
 - **MODE_PRIVATE.** Si el fichero no existe se crea y, si existe, su contenido es eliminado.
 - **MODE_APPEND.** Si el fichero no existe se crea y, si existe, el nuevo contenido se añadirá al ya existente.
- **openInputStream(String name).** A partir del nombre del fichero, devuelve un objeto FileInputStream para leer del mismo. Si el fichero **no existe**, se producirá la excepción FileNotFoundException.

A partir de los objetos `FileInputStream` y `FileOutputStream`, se pueden crear objetos específicos para realizar las operaciones de escritura y lectura. Por ejemplo, el siguiente bloque de instrucciones escribiría el contenido de una cadena en el fichero `info.txt`:

```
FileOutputStream fos=  
    this.openOutputStream("info.txt",MODE_PRIVATE);  
PrintStream out=new PrintStream(fos);  
out.println("frase en fichero");  
out.flush();  
  
out.close(); //cierre del objeto de escritura
```

Para recuperar el contenido de dicho fichero y guardarlo en una variable, procederíamos del siguiente modo:

```
String contenido;  
FileInputStream fis=this.openInputStream("info.txt");  
BufferedReader bf=new BufferedReader(new  
    InputStreamReader(fis));  
contenido=bf.readLine();  
  
bf.close();
```

Ejercicio resuelto

Para poner en práctica la utilización de ficheros para el almacenamiento de información de la aplicación, vamos a desarrollar una nueva versión del ejercicio de las notas, solo que esta vez las notas las guardaremos en un fichero y de ahí serán recuperadas para la realización de los cálculos y su visualización en una lista. Añadiremos un nuevo botón en la actividad principal para limpiar el fichero.

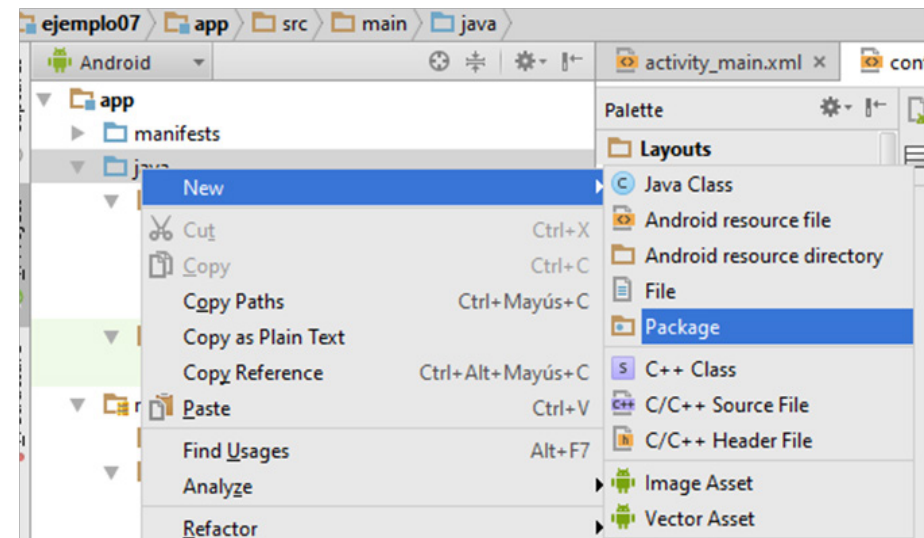
El aspecto de las tres actividades implicadas se muestra en la siguiente imagen:



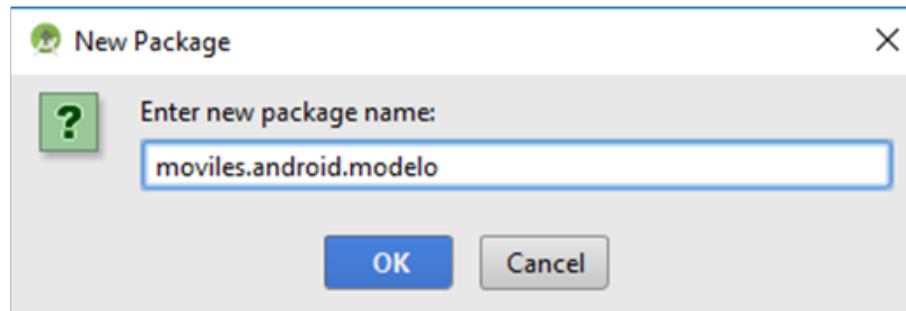
Lo primero que haremos, será crear una clase en la que vamos a encapsular el acceso a datos. Y es que, de cara a estructurar mejor la aplicación, toda la lógica de negocio de la misma, incluyendo el acceso a ficheros y bases de datos se encapsula en clases independientes. Esta forma de programar sigue los principios del patrón Modelo Vista Controlador, que define tres grandes bloques en toda aplicación:

- **Modelo.** Clases independientes donde se encapsula la lógica de negocio.
- **Controlador.** Componentes Android, como la actividad, donde se codifican los métodos de respuesta a los sucesos sobre la interfaz de usuario y del ciclo de vida de los componentes.
- **Vista.** Se encarga de generar la interfaz de usuario, en el caso de las aplicaciones Android, se trataría de las plantillas XML.

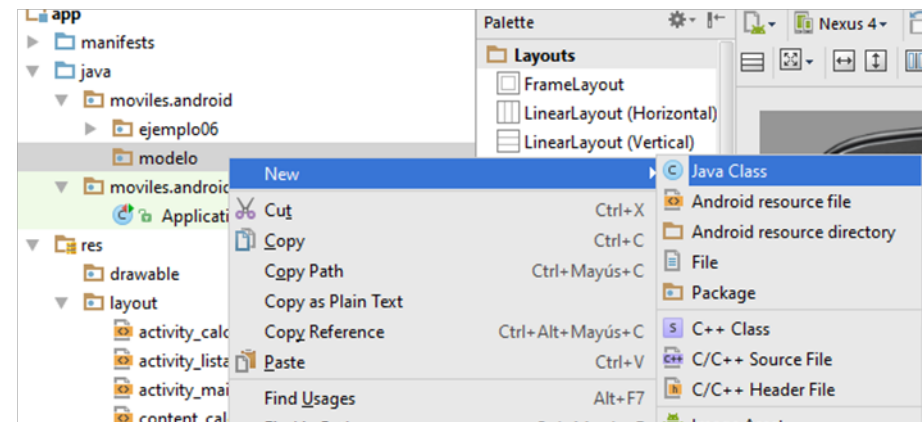
Pues bien, vamos a crear nuestra clase del modelo. Primeramente, vamos a crear un paquete donde colocar dicha clase, para ello, nos situamos sobre la carpeta java del proyecto y en el menú que aparece al pulsar el botón derecho del ratón elegimos New->Package:



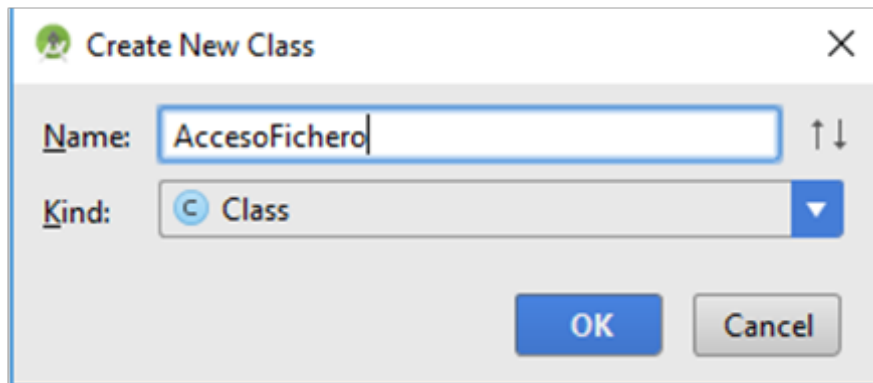
Seguidamente, introduciremos el nombre de nuestro nuevo paquete



Una vez creado el paquete, añadiremos la clase dentro del mismo. Nos situamos sobre él y en el menú del botón derecho del ratón elegimos New->Java class:



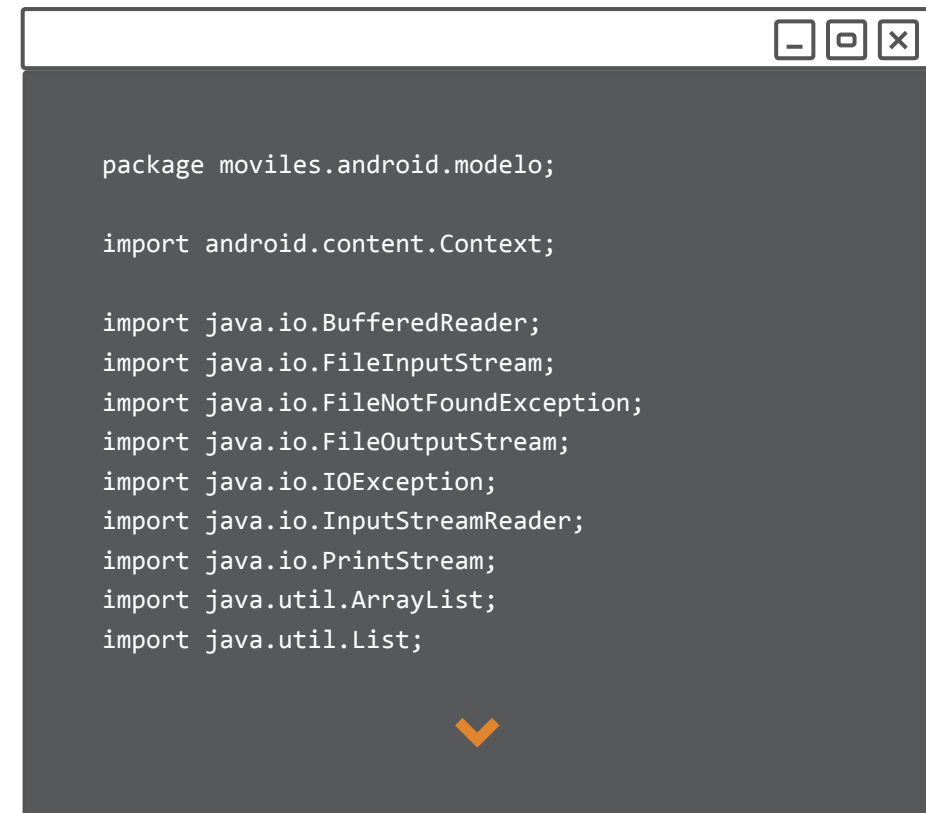
Le damos el nombre a la clase:




Y ya tenemos la nueva clase creada dentro del paquete *modelo*.

En esta clase añadiremos tres métodos encargados de hacer las tres operaciones requeridas, guardar una nota, recuperar todas las notas y limpiar el fichero. La clase no incluirá instrucciones relacionadas con la captura y presentación de los datos, solo con la manipulación del fichero. He aquí como quedaría dicha clase:

En esta clase añadiremos tres métodos encargados de hacer las tres operaciones requeridas, guardar una nota, recuperar todas las notas y limpiar el fichero. La clase no incluirá instrucciones relacionadas con la captura y presentación de los datos, solo con la manipulación del fichero. He aquí como quedaría dicha clase:







```

public class AccesoFichero {
    private Context contexto;
    private String nombre;
    //se proporciona el contexto y el nombre del fichero
    //durante la creación del objeto
    public AccesoFichero(Context contexto, String nombre){
        this.contexto=contexto;
        this.nombre=nombre;
    }
    public void guardarNota(double nota ){
        FileOutputStream fos= null;
        PrintStream out=null;
        try {
            //guardamos la nota en modo append para
            //añadirla a las ya existentes
            fos = contexto.openFileOutput(nombre,
                Context.MODE_APPEND);
            out=new PrintStream(fos);
            out.println(nota);
            out.flush();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        finally {
            if(out!=null){
                out.close();
            }
        }
    }
    public ArrayList<Double> recuperarNotas(){
        ArrayList<Double> notas=new ArrayList<>();
        FileInputStream fis=null;
        BufferedReader bf=null;
        try{

```

```

        pfis=contexto.openFileInput(nombre);
        bf=new BufferedReader(new InputStreamReader(fis));
        String nota;
        //recorremos linea por línea y extraemos las notas
        //como texto, por eso se convierten a Double
        while((nota=bf.readLine())!=null){
            notas.add(Double.parseDouble((nota)));
        }
    }
    catch(IOException ex){
        ex.printStackTrace();
    }
    return notas;
}
    public void limpiarFichero(){
        FileOutputStream fos= null;
        //para limpiar el fichero, basta con abrirlo
        //en modo sobrescritura y se eliminará todo
        //su contenido
        try {
            fos = contexto.openFileOutput(nombre,
                Context.MODE_PRIVATE);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}

```

Desde el controlador, o métodos de las actividades, se instanciaría dicha clase para poder llamar a los métodos requeridos en cada momento. El siguiente listado corresponde a la actividad principal, donde podemos ver como el método *guardar()* crea un objeto *AccesoFichero* para llamar al método *guardarNota()* y almacenar la nota en el fichero, en lugar de en memoria. En el método *limpiar()*, que responde al evento clic del nuevo botón, utilizamos también un objeto de esta clase para llamar al método que borra el contenido del fichero:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void guardar(View v){
        EditText edtNota=(EditText)this.findViewById(R.id.edtNota);
        //accede al fichero a través de AccesoFichero
        //para guardar la nota
        AccesoFichero af=new AccesoFichero(this, "notas.tt");
        af.guardarNota(Double.parseDouble(edtNota.getText().toString()));
        //borra el contenido del campo nota
        edtNota.setText("");
        //pasa el foco a dicho control
        edtNota.requestFocus();
    }
    public void calculos(View v){
        Intent intent=new Intent(this,CalculosActivity.class);

        this.startActivity(intent);
    }
    public void mostrarNotas(View v){
        Intent intent=new Intent(this,ListadoActivity.class);

        this.startActivity(intent);
    }
    public void limpiar(View v){
        AccesoFichero af=new AccesoFichero(this, "notas.tt");
        af.limpiarFichero();
    }
}
```

Podemos ver también como la llamada a las otras actividades ya no requiere el paso de ningún parámetro en el Intent.

En cuanto a la actividad ListadoActivity, así quedaría la nueva versión del método *onCreate()*:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_listado);
    AccesoFichero af=new AccesoFichero(this,"notas.tt");
    ArrayList<Double> notas=af.recuperarNotas();
    ListView lvNotas=(ListView)this.findViewById(R.id.lvNotas);
    ArrayAdapter<Double> adapter= new ArrayAdapter<Double>(this,
    android.R.layout.simple_list_item_1, notas);
    lvNotas.setAdapter(adapter);
}
```

En CalculosActivity también tendríamos que modificar el método *onCreate()* para que trabajase con los datos extraídos del fichero:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_calculos);
    AccesoFichero af=new AccesoFichero(this,"notas.tt");
    ArrayList<Double> notas=af.recuperarNotas();
    TextView tvMedia=(TextView)this.findViewById(R.id.tvMedia);
    TextView tvAprobados=(TextView)
        this.findViewById(R.id.tvAprobados);
    //llama a los métodos que hacen los cálculos y muestra
    //los resultados en los campos de texto
    tvMedia.setText("Nota media: "+media(notas));
    tvAprobados.setText("Aprobados: "+aprobados(notas));
}
```



3. Utilización de SharedPreferences

La utilización del objeto SharedPreferences de Android constituye un caso particular de almacenamiento de información en ficheros. A través de este objeto, podemos almacenar datos de aplicación en forma de parejas nombre-valor. Esta información se almacena internamente en un fichero xml privado para la aplicación, pero nosotros desde código no tendremos que preocuparnos de tratar con el fichero, solamente nos preocuparemos de llamar a los métodos de SharedPreferences para el almacenamiento y recuperación de información.

3.1 | Almacenamiento de datos

Lo primero que tenemos que saber es como obtener un objeto SharedPreferences. SharedPreferences es una interfaz que se encuentra en el paquete **android.content** y para obtener una implementación de ella tenemos que recurrir al método `getSharedPreferences()` de Context, al que tendremos que proporcionar el nombre del archivo de preferencias y el modo de acceso:

```
SharedPreferences sp=this.getSharedPreferences("temperaturas",  
Context.MODE_PRIVATE);
```

El modo de acceso a SharedPreferences siempre será en modo privado.

Una vez obtenido el objeto, necesitamos obtener un objeto Editor para guardar los datos. Editor es una interfaz interna a SharedPreferences:

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

La interfaz Editor dispone de una serie de métodos `putXxx()` para guardar datos en SharedPreferences.

Public Methods	
abstract void	apply() Commit your preferences changes back from this Editor to the <code>SharedPreferences</code> object it is editing.
abstract <code>SharedPreferences.Editor</code>	clear() Mark in the editor to remove <i>all</i> values from the preferences.
abstract boolean	commit() Commit your preferences changes back from this Editor to the <code>SharedPreferences</code> object it is editing.
abstract <code>SharedPreferences.Editor</code>	putBoolean(String key, boolean value) Set a boolean value in the preferences editor, to be written back once <code>commit()</code> or <code>apply()</code> are called.
abstract <code>SharedPreferences.Editor</code>	putFloat(String key, float value) Set a float value in the preferences editor, to be written back once <code>commit()</code> or <code>apply()</code> are called.
abstract <code>SharedPreferences.Editor</code>	putInt(String key, int value) Set an int value in the preferences editor, to be written back once <code>commit()</code> or <code>apply()</code> are called.
abstract <code>SharedPreferences.Editor</code>	putLong(String key, long value) Set a long value in the preferences editor, to be written back once <code>commit()</code> or <code>apply()</code> are called.
abstract <code>SharedPreferences.Editor</code>	putString(String key, String value) Set a String value in the preferences editor, to be written back once <code>commit()</code> or <code>apply()</code> are called.
abstract <code>SharedPreferences.Editor</code>	putStringSet(String key, Set<String> values) Set a set of String values in the preferences editor, to be written back once <code>commit()</code> or <code>apply()</code> is called.
abstract <code>SharedPreferences.Editor</code>	remove(String key) Mark in the editor that a preference value should be removed, which will be done in the actual preferences once

Todos estos métodos reciben como primer parámetro la clave o nombre del dato y como segundo parámetro el valor.

Por ejemplo, si queremos guardar dos datos, nombre y edad, sería:

```
editor.putString("nombre","Angel Perez");  
editor.putInt("edad",34);
```

Los datos que se almacenan en el editor quedan en memoria, para que la información se guarde en el fichero es necesario llamar al método `commit()` de Editor:

```
editor.commit();
```

Si en algún momento queremos eliminar todo el contenido del fichero, utilizamos el método `clear()` de Editor:

```
editor.clear();
```

3.2 | Recuperación de datos

Para recuperar datos del objeto `SharedPreferences` no necesitamos el objeto `Editor`, la propia interfaz `SharedPreferences` proporciona una serie de métodos *getXxx()* para recuperar los datos a partir de la clave. Estos métodos reciben un segundo parámetro que es el valor por defecto que se devolverá en caso de que la clave no exista en el fichero.

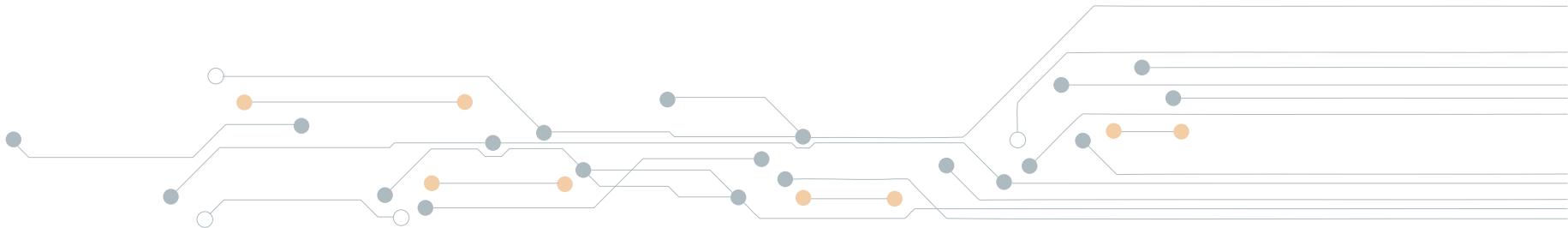
```
String n = sp.getString("nombre","");
```

En la instrucción anterior recuperamos el dato con clave "nombre" y si no existe, se devolverá una cadena vacía.

Además de estos métodos `get()` que nos permiten obtener datos de forma individual, `SharedPreferences` dispone del siguiente método:

```
Map<String, ?> getAll()
```

El método `getAll()` devuelve un objeto `Map` con todas las preferencias almacenadas. La clave de todos los datos es `String`, pero cada valor puede ser de tipo diferente, de ahí el comodín `?`.



4. Acceso a bases de datos

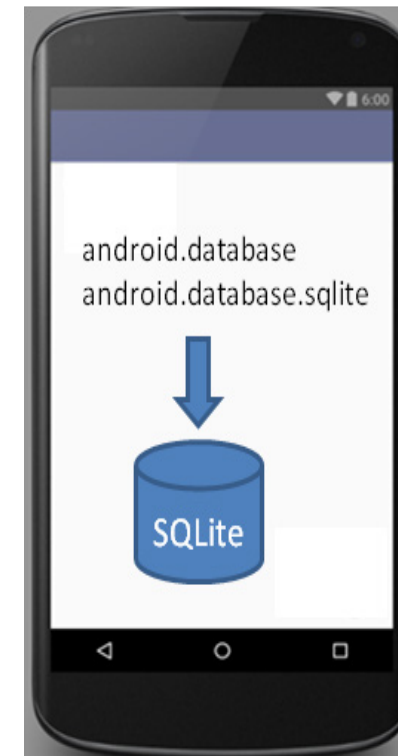
Los ficheros se utilizan para guardar datos sencillos. Si vamos a trabajar con conjuntos de datos más completos, incluso varios tipos de conjuntos relacionados entre sí, deberíamos emplear una base de datos relacionales.

Android incluye soporte para trabajar con bases de datos SQLite. SQLite es un mini sistema gestor de bases de datos, sencillo, potente y con gran optimización de recursos, lo que le hace apropiado para dispositivos con limitadas capacidades hardware como los teléfonos móviles.

Utilizando las librerías Android para SQLite, nuestras aplicaciones podrán crear y manipular bases de datos relacionales en el terminal. Estas bases de datos son privadas para la aplicación, lo que significa que otras aplicaciones no podrán acceder a ellas. Las aplicaciones Android crean estas bases de datos en la carpeta `data/data/nombre_paquete/databases` de la aplicación.

Los paquetes **android.database** y **android.database.sqlite** incluyen las clases e interfaces necesarias para poder manipular bases de datos SQLite.

Tal y como ya comentamos en el apartado anterior dedicado a ficheros, las instrucciones de acceso a datos las encapsularemos dentro de las clases independientes que formarán el modelo.



4.1 | Creación de la clase Helper

Lo primero que haremos cuando vamos a crear una aplicación que va a manipular bases de datos SQLite, es crear una clase helper o ayudante que gestione el ciclo de vida de la bases de datos y proporcione acceso a los métodos de apertura y cierra de la base de datos. Esta clase será una subclase de `android.database.sqlite.SQLiteOpenHelper`.

`SQLiteOpenHelper` contiene dos métodos abstractos del ciclo de vida que habrá que sobrescribir:

- **onCreate().** Es llamado durante la creación de la base de datos. Aprovecharemos este método para realizar la creación de las tablas.
- **onUpgrade().** Es llamado cuando se necesita realizar una actualización de la base de datos, algo que sucede, por ejemplo, cuando se produce un cambio de versión de la misma.

Veamos un ejemplo típico de creación de una clase Helper que posteriormente utilizaremos en una aplicación de ejemplo que vamos a realizar. Para ello, supongamos que vamos a trabajar con una base de datos llamada "libreria", en la que tendremos una tabla llamada "libros" con cuatro columnas: `_id`, título, autor y precio.

Esta sería la clase helper:

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

public class DatabaseHelper extends SQLiteOpenHelper{
    private static final String DATABASE_NAME="Libreria";
    private static final int DATABASE_VERSION=1;

    //sql de creación de tabla
    private static final String DATABASE_CREATE_LIBROS=
        "create table libros (_id integer primary key autoincrement,"+
        "titulo text not null, autor text not null, precio float not null)";
    //sql eliminación de tabla
    private static final String DATABASE_DELETE_LIBROS=
        "drop table if exists libros";
    public DatabaseHelper(Context context){
        super(context, DATABASE_NAME,null, DATABASE_VERSION);
    }

    //Se llama al crear la base de datos
    @Override
    public void onCreate(SQLiteDatabase db) {
        //creamos las tablas
        createTables(db);
    }
    //este método es llamado si a la hora de crear el DataBaseHelper
    //se pasa una versión superior a la ya existente
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion) {
        //reconstruimos la tabla
        deleteTables(db);
        createTables(db);
    }
    private void createTables(SQLiteDatabase db) {
        db.execSQL(DATABASE_CREATE_LIBROS);
    }

    private void deleteTables(SQLiteDatabase db) {
        db.execSQL(DATABASE_DELETE_LIBROS);
    }
}
```

Lo primero que vemos es la definición de una serie de constantes con el nombre y versión de la base de datos y las instrucciones SQL de creación y eliminación de tabla. A la hora de crear una tabla de SQLite, siempre incluiremos la columna `_id` de tipo autonumérico, que deberá ser siempre la clave primaria de la tabla.

En el constructor de nuestra clase debemos llamar al constructor de `SQLiteOpenHelper` que tendrá el siguiente formato:

```
SQLiteOpenHelper(Context context, String name,
    SQLiteDatabase.CursorFactory factory, int
    version)
```

Context es el contexto de aplicación, *name* es el nombre de la base de datos con la que queremos trabajar, *factory* es el objeto para la creación de cursores (más adelante veremos los cursores), pero como no lo vamos a utilizar pasamos *null*. El último parámetro es la versión de la base de datos con la que vamos a trabajar.

Además del constructor, realizamos la sobrescritura de los métodos abstractos comentados anteriormente. En el caso de *onCreate()*, este método es llamado por Android cuando se produce la creación de la base de datos, es decir, **si al crear el objeto helper la base de datos indicada no existe, esta se creará y a continuación se llamará a *onCreate()* sobre el helper**. Este método recibe un objeto `SQLiteDatabase` que representa la base de datos creada y que proporciona el método `execSQL` para lanzar instrucciones SQL sobre dicha base de datos. Aprovechamos por tanto el método `onCreate()` para crear las tablas de la base de datos.

En cuanto `onUpgrade()`, este método es llamado cuando al crear el objeto helper proporcionamos una versión de la base de datos diferente a la ya existente. En este método podemos no hacer nada o, como en este ejemplo, destruir la tabla existente y generar una nueva.

Además de la gestión del ciclo de vida de la base de datos, nuestra clase `DatabaseHelper` expondrá los siguientes métodos heredados de `SQLiteOpenHelper`:

- **`getWritableDatabase()`**. Devuelve un objeto `SQLiteDatabase` para realizar operaciones tanto de lectura como de escritura sobre la base de datos.
- **`close()`**. Cerrará cualquier base de datos abierta.

4.2 | La clase SQLiteDatabase

SQLiteDatabase expone una serie de métodos para operar contra una base de datos SQLite. Entre los más importantes están:

- **insert(String table, String nullColumnHack, ContentValues values).** Lanza una instrucción de tipo Insert a la base de datos para añadir una fila en la tabla indicada como primer parámetro. Los datos se proporcionan en el tercer parámetro a través de una colección ContentValues. El segundo parámetro está relacionado con el tratamiento de los valores nulos, habitualmente este parámetro será null.
- **query(String table, String[] columns, String selection, String[] selectionArgs, String groupBy, String having, String orderBy).** Envía una instrucción SQL de tipo Select a la base de datos, devolviendo un objeto Cursor para iterar sobre los resultados.

El significado de los parámetros es el siguiente:

- **table.** Nombre de la tabla de la que quieren recuperar los registros.
- **columns.** Nombres de las columnas que se quieren recuperar.
- **selection.** Criterio de selección. Sigue la misma sintaxis de las cláusulas where de SQL.
- **selectionArgs.** Si se ha especificado algún parámetro en el criterio de selección, selectionArgs contendrá los valores de cada uno de esos parámetros.
- **groupBy.** Criterio de agrupamiento de registros. Equivale a la cláusula SQL groupBy.
- **having.** Clausula having SQL
- **orderBy.** Clausula orderBy

- **delete(String table, String whereClause, String[] whereArgs).** Envía una instrucción SQL de tipo delete para eliminar los registros de la tabla indicada en el primer parámetro. El parámetro whereClause contiene la condición de eliminación, que sigue la sintaxis de la cláusula where SQL. El tercer parámetro representa los valores de los parámetros indicados en whereClause. Como resultado, el método delete() devuelve el número de filas eliminadas

- **update(String table, ContentValues values, String whereClause, String[] whereArgs).** Envía una instrucción SQL de tipo Update a la base de datos. Los nuevos valores son proporcionados en el segundo parámetro de tipo ContentValues. La llamada al método devuelve el número de filas afectadas.

- **execSql(String sql).** Ya vimos anteriormente este método, ejecuta una instrucción SQL sobre la base de datos, a excepción de las instrucciones Select. Si preferimos utilizar SQL para operar contra la base de datos, podemos usar este método en vez de insert(), delete() y update().

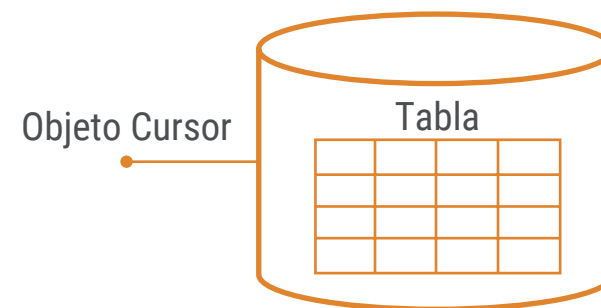
- **rawQuery(String sql, String[] selectionArgs).** Envía una instrucción Select a la base de datos, devolviendo un objeto Cursor para acceder al conjunto de registros seleccionados. Podemos usar este método en lugar de query().

Normalmente, todas las operaciones sobre la base de datos a través de SQLiteDatabase se encapsulan en una clase independiente (modelo) , que es utilizada desde los métodos de respuesta a eventos (controlador) para realizar las diferentes tareas requeridas por la aplicación.

La interfaz Cursor

Al presentar los métodos *query()* y *rawQuery()* hemos visto que estos devuelven un objeto de la interfaz Cursor. Un objeto Cursor nos permite iterar el conjunto de registros y acceder a la información contenida en cada fila.

Podemos imaginar al Cursor como un puntero que se puede mover por cada una de las filas del conjunto de registros. Este puntero se encuentra colocado inicialmente delante del primer registro del conjunto:



Para poder desplazarnos por el conjunto y acceder a la información contenida en cada celda, tendremos que recurrir a los métodos de la interfaz `Cursor`, entre los cuales están:

- **`moveToNext()`**. Desplaza el cursor al siguiente registro del conjunto y nos devolverá `true` si se trata de un registro válido o `false` si nos hemos salido del conjunto.

- **`getXxx()`**. Existe un conjunto de métodos `get()` por cada tipo de datos Java más `String`. Mediante estos métodos de `Cursor` podemos recuperar los datos de las diferentes columnas del registro actual a partir de la posición de cada columna, siendo la primera la que ocupa la posición 0.

Además de los métodos para acceso a la información, un `Cursor` se utiliza también para la creación de adaptadores de tipo `SimpleCursorAdapter` con los que poder vincular una lista a un conjunto de registros de la base de datos. En el siguiente ejercicio de ejemplo veremos su uso.



Aplicación de ejemplo

Como hemos indicado anteriormente, vamos a crear una aplicación de ejemplo que la que vamos a poner en práctica el acceso a base de datos SQLite desde una aplicación Android.

La aplicación en cuestión realizará una pequeña gestión de libros utilizando la base de datos de librería que creará la clase DatabaseHelper que presentamos anteriormente.

La aplicación que vamos a crear, tendrá una actividad principal con tres botones:



El botón “Nuevo libro” nos llevará a una actividad en la que se nos solicitarán los datos de un nuevo libro y se grabará en la base de datos.

El botón “Buscar”, nos llevará a una actividad en la que se nos solicitará el título de un libro. Si se encuentra, se mostrarán los datos del libro en un TextView y si no, se indicará un mensaje de libro no encontrado.

Por último, el botón “Listado” nos llevará a una actividad en la que se mostrarán los datos de todos los libros en un listview.

Bien, pues lo primero será crear el proyecto, al que llamaremos ejemplo09, con una actividad por defecto MainActivity que será la principal. Tras crear el proyecto, crearemos un subpaquete paquete modelo y dentro de éste la clase DatabaseHelper , cuyo código es el que te mostramos anteriormente.

Seguidamente, dentro del mismo paquete modelo, crearemos una nueva clase, a la que vamos a llamar DBLibros, donde vamos implementar una serie de métodos que encapsulen las operaciones sobre la base de datos a través de SQLiteDatabase. Esta nueva clase hará uso de la clase helper anterior.

Veamos el código de la clase DBLibros y seguidamente comentamos los aspectos más importantes del mismo:

```
import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;

public class DBLibros {
    private static final String TABLA="libros";
    //Atributos
    private SQLiteDatabase db=null;
    private DatabaseHelper dbhelper=null;

    //Contexto
    Context context;

    public DBLibros(Context ctx){
        this.context=ctx;
        //crea una instancia del helper
        dbhelper=new DatabaseHelper(context);
        //crea un objeto SQLiteDatabase para operar
        //contra la base de datos
        db=dbhelper.getWritableDatabase();
    }

    public void close(){
        dbhelper.close();
    }

    public long altaLibro(String titulo, String autor, double precio){
        //crea el contentvalues y añade una entrada
        //por cada dato del libro a guardar
        ContentValues initialValues=new ContentValues();
```




```

initialValues.put("titulo", titulo);
initialValues.put("autor", autor);
initialValues.put("precio", precio);
return db.insert(TABLA, null, initialValues);
}
public boolean borrarLibro(int id){
    //elimina el libro a partir del id
    return db.delete(TABLA, "_id="+id, null)>0;
}
public Libro recuperarLibroPorTitulo(String titulo){
    float valor=0.0f;
    Libro lib=null;
    Cursor c=db.query(TABLA, new String[]{"_id",
"autor","precio"},"titulo=?", new String[]{titulo},
null,null,null);
registro //el curso apunta a la posición anterior al primer
//debe desplazarlo al siguiente registro para
//apuntar al primero
if(c.moveToNext()){
    lib=new Libro(c.getInt(0),titulo,c.getString(1),c.
getDouble(2));
}
return lib;
}
public Cursor recuperarLibros(){
    //aunque no se utilice, se debe recuperar también
    // el campo _id
    return db.query(TABLA, new String[]{"_id","titulo",
"precio","autor"},null, null, null,null,null);
}
}

```

Lo primero que vemos es el constructor de DBLibros, en él, creamos el objeto de nuestra clase helper (DatabaseHelper) y lo guardamos en un atributo de la clase para su uso posterior. También realizamos la creación del objeto SQLiteDatabase llamando al método `getWritableDatabase()` del helper.

Seguidamente, tenemos el método `close()`, en el que realizamos el cierre del helper para cerrar la conexión con la base de datos y liberar recursos.

Después tenemos lo que sería el primer método de negocio, `altaLibro()`, que realiza la inserción de un nuevo registro en la tabla de libros con los datos recibidos como parámetros. Vemos como este método crear una colección `ContentValues` con los datos del libro, donde la clave es el nombre de la columna y el valor es el valor que se quiere dar a ese campo.

Aunque no se emplea en este ejercicio, definimos un método `borrarLibro()` que realiza la eliminación de un registro de la tabla de libros a partir de su id.

El método *recuperarLibroPorTitulo()* llama al método *query()* de *SQLiteDatabase*, indicando como condición que el valor de la columna *titulo* coincida con el dato recibido por el método. *Query* devuelve un objeto de la interfaz **android.database.Cursor** con el que podemos iterar por el conjunto de registros como explicamos anteriormente.

El método *recuperarLibroPorTitulo()* devuelve un objeto *JavaBean* de tipo *Libro* con los datos del libro localizado. En caso de no encontrarlo devolverá *null*.

Finalmente, tenemos el método *recuperarLibros()*, que devuelve un objeto *Cursor* para iterar sobre todos los registros de la tabla

En cuanto a las actividades, este sería el código de la actividad principal *MainActivity*, donde simplemente tenemos las llamadas a las restantes actividades desde los métodos escuchadores de evento:

```
public class MainActivity extends AppCompatActivity {

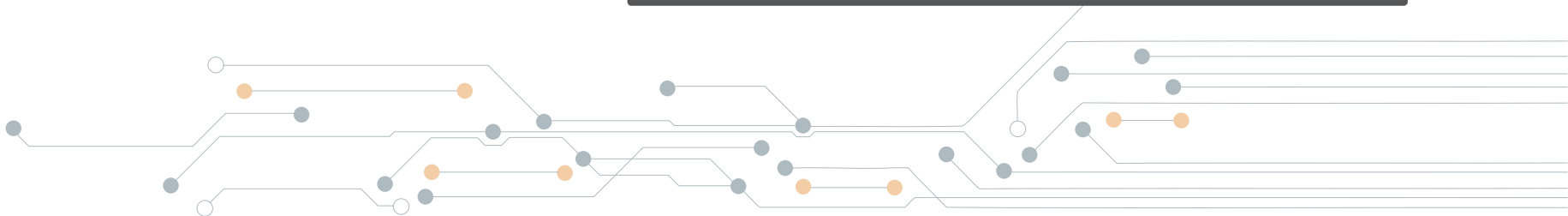
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    public void agregar(View v){
        Intent in=new Intent(this,AltaActivity.class);
        startActivity(in);
    }

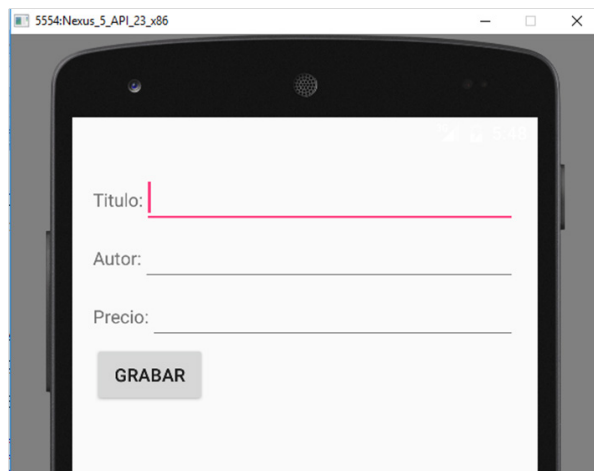
    public void mostrar(View v){
        Intent in=new Intent(this,ListadoActivity.class);
        startActivity(in);
    }

    public void buscar(View v){
        Intent in=new Intent(this,BuscadorActivity.class);
        startActivity(in);
    }

}
```



La actividad AltaActivity, es la que se encarga de realizar el alta de los libros. Su aspecto se muestra en la siguiente imagen:



A través de tres elementos EditText se recogen los datos del libro que se quiere dar de alta en la base de datos.

El código de la clase es el siguiente:

```
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import moviles.android.ejemplo08.datos.DBLibros;

public class AltaActivity extends Activity {

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

    public void grabar(View v){
        String titulo=
        ((EditText)findViewById(R.id.txtTitulo)).getText().toString();
        String autor=
        ((EditText)findViewById(R.id.txtAutor)).getText().toString();
        Double precio=
        Double.parseDouble(((EditText)findViewById(R.id.txtPrecio)).getText().
        toString());
        DBLibros adp=new DBLibros(this);
        adp.altaLibro(titulo, autor, precio);
        adp.close();
        finish();
    }
}
```

Como vemos, en el método manejador del evento clic del botón, se recogen los datos introducidos en los campos de texto y se pasan al método *altaLibro()* del objeto DBLibros.

Por su parte, la clase BuscadorActivity utiliza un EditText para recoger el título de búsqueda y muestra los datos del libro encontrado en un TextView. Este es el aspecto que tendrá esta actividad:



En cuanto al código de la misma, se muestra en el siguiente listado:

```
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;
import moviles.android.ejemplo08.datos.DBLibros;
import moviles.android.ejemplo08.datos.Libro;

public class BuscadorActivity extends Activity {

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

    public void buscar(View v){
        String titulo=((EditText)findViewById(R.id.textoTitulo)).
            getText().toString();
        DBLibros db=new DBLibros(this);
        Libro lib=db.recuperarLibroPorTitulo(titulo);
        String s="";
        //si se encuentra el libro se muestran sus datos
        //si no, un mensaje de aviso
        if(lib!=null) {
            s += "Titulo: " + lib.getTitulo() + " Autor: " +
                lib.getAutor() + " Precio: " + lib.getPrecio();
        }else{
            s="Libro no encontrado";
        }
        ((TextView)findViewById(R.id.resultado)).setText(s);
        db.close();
    }

    public void salir(View v){
        finish();
    }
}
```

Finalmente, la actividad ListadoActivity incluye un ListView para mostrar en cada fila los datos de cada libro:



Este sería el código de la actividad:

```
import android.app.Activity;
import android.database.Cursor;
import android.os.Bundle;
import android.support.v4.widget.CursorAdapter;

import android.view.View;
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;
import moviles.android.ejemplo08.datos.DBLibros;

public class ListadoActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_listado);
        DBLibros adp=new DBLibros(this);
        Cursor c=adp.recuperarLibros();
        String [] columns=new String[]{"titulo","autor","precio"};
        int[] views=new int[]{R.id.titulo,R.id.autor,R.id.precio};
        SimpleCursorAdapter sc= new SimpleCursorAdapter(
            this,
            R.layout.list_controls,
            c,
            columns,
            views,
            CursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER);
        ListView lista=(ListView)findViewById(R.id.listaLibros);
        lista.setAdapter(sc);
        adp.close();
    }

    public void salir(View v){
        finish();
    }

}
```

A diferencia del ejemplo que presentamos al estudiar el control ListView, en el que los datos de la lista se obtenían de un ArrayAdapter, en este caso se obtienen de otra subclase de BaseAdapter, llamada **SimpleCursorAdapter**, la cual permite crear un adaptador de datos para la lista a partir de un objeto Cursor.

Debido a su importancia, vamos a analizar con detalle la utilización de la clase SimpleCursorAdapter. En primer lugar, para crear un objeto de la misma, debemos utilizar el siguiente constructor:

```
SimpleCursorAdapter(Context context, int layout, Cursor c, String[] from, int[] to, int flags).
```

Vamos a comentar los parámetros requeridos por este constructor:

- **context.** Representa el contexto de aplicación. Como sabemos, puede ser la propia actividad.
- **layout.** Indica la plantilla utilizada para la presentación de los elementos de cada fila.

Dado que no es un simple texto el que hay que mostrar en esta lista, sino los valores de tres campos, debemos definir un archivo de plantilla personalizado en el que se indiquen los controles empleados dentro de la fila para mostrar cada campo y la organización de los mismos. El identificador de este archivo de plantilla es el valor que se pasará en este parámetro. En nuestro ejemplo, definiremos un archivo de plantilla llamado list_controls.xml, cuyo contenido será el que se indica a continuación:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center_vertical"
    android:orientation="horizontal" >
    <TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingRight="20dp"
        android:layout_weight="1"
        android:gravity="start"
        android:id="@+id/titulo"/>
    <TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingRight="20dp"
        android:layout_weight="2"
        android:gravity="right"
        android:id="@+id/autor"/>
    <TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="3"
        android:gravity="right"
        android:id="@+id/precio"/>
</LinearLayout>
```

En el vemos la utilización de un TextView para presentar cada campo, organizados en un LinearLayout horizontal.

- **c.** Es el objeto Cursor a partir del cual se obtendrán los datos del adaptador. En nuestro ejemplo, será el curso devuelto por el método obtenerLibros() de la clase DBLibros
- **from.** Array con los nombres de las columnas del cursor, cuyos datos vamos a presentar.
- **to.** Cada campo o columna del cursor se volcará en uno de los controles de la plantilla de presentación, en nuestro caso list_controls.xml. En este parámetro se suministra un array con los identificadores de dichos controles. Cada campo indicado en from se volcará en su correspondiente control indicado en este parámetro, en el mismo orden.
- **flags.** Determina el comportamiento del adaptador. El único valor aconsejado para este parámetro es la constante FLAG_REGISTER_CONTENT_OBSERVER, definida en la clase CursorAdapter. Dicho valor implica que las notificaciones en el Cursor serán notificadas a onContentChanged().

Te presentamos también el archivo de plantilla de esta actividad de listado para que puedas ver la organización de los componentes de la actividad, donde la fila de títulos se genera mediante un layout independiente colocado encima del ListView:

```

LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto" android:layout_width="match_
parent"
    android:layout_height="match_parent" android:paddingLeft="@dimen/activity_
horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="layout/activity_main" tools:context=".AltaActivity"
    android:orientation="vertical">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center_vertical"
        android:orientation="horizontal" >
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingRight="20dp"
            android:layout_weight="1"
            android:gravity="start"
            android:text="Titulo"/>
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:paddingRight="20dp"
            android:layout_weight="2"
            android:gravity="right"
            android:text="Autor"/>

```



```
<TextView android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="3"
    android:gravity="right"
    android:text="Precio"/>
```

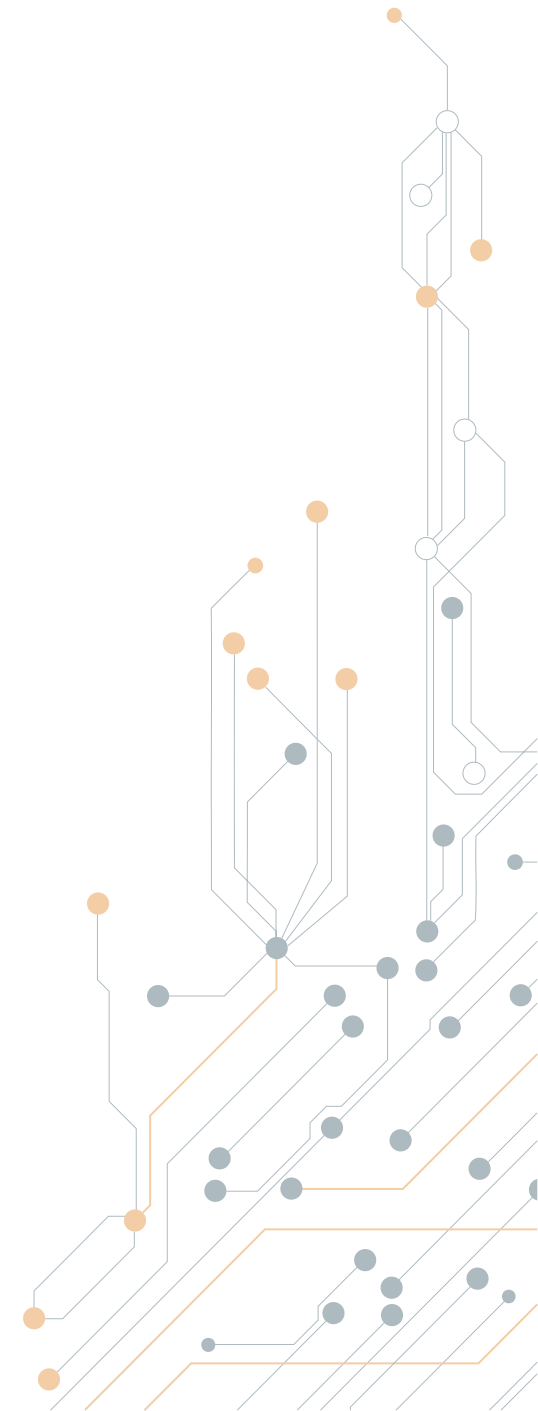
```
</LinearLayout>
```

```
<ListView
    android:id="@+id/listaLibros"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

```
<Button
```

```
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Salir"
    android:onClick="salir"/>
```

```
</LinearLayout>
```



5. Proveedores de contenido

Un proveedor de contenido es un tipo de componente Android que proporciona un mecanismo estandarizado y universal para acceder a cualquier fuente de datos.

Los proveedores de contenido proporcionan una interfaz universal que permite a las aplicaciones utilizar el mismo conjunto de métodos para manipular distintas fuentes de datos, independientemente del origen de estos y del medio en el que se encuentren almacenados.

Android proporciona gran cantidad de proveedores de contenido que permiten acceder a datos del sistema desde nuestras aplicaciones.

Este es el caso del **proveedor de contactos**, que nos da acceso a la lista de contactos del teléfono, o el proveedor de imágenes, con el que podemos acceder a las imágenes almacenadas en el terminal.

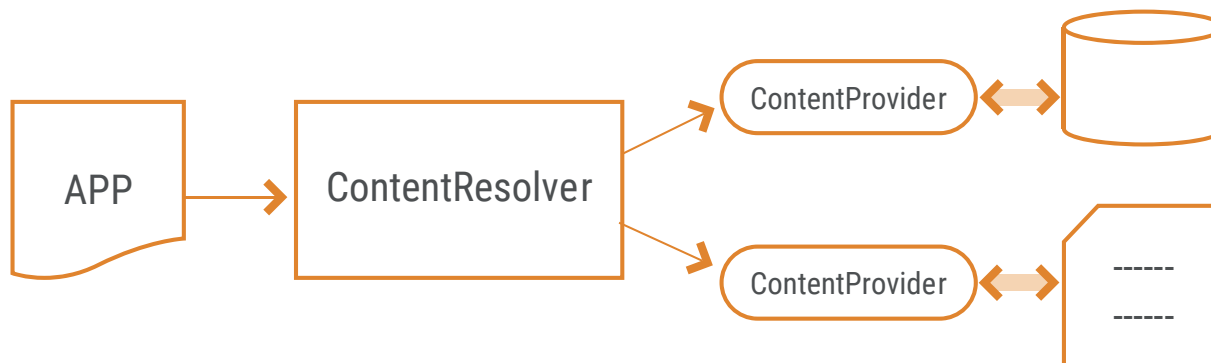
En este apartado aprenderemos a manipular un proveedor de contenidos y veremos cómo acceder a algunos de estos interesantes proveedores proporcionados por Android.



5.1 | Las clases ContentResolver y ContentProvider

La clase ContentResolver proporciona acceso a los proveedores de contenido registrados en el terminal. Todo proveedor de contenido hereda una clase llamada ContentProvider, que proporciona una serie de métodos abstractos que los proveedores de contenido deben implementar.

A través de un objeto ContentResolver asociado al contexto de aplicación, podemos comunicarnos con cualquier proveedor de contenidos. De hecho, esta clase expone el mismo juego de métodos que la clase ContentProvider, de modo que la llamada a los métodos del ContentResolver, será derivada al método del ContentProvider correspondiente:



Métodos de ContentResolver

Entre los principales métodos con los que cuenta la clase ContentResolver para acceder a los datos de un proveedor de contenido tenemos:

- **query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder).** Es el método que más utilizaremos, pues es el que nos permite recuperar el conjunto de datos proporcionados por un proveedor. Como vemos, este método recuerda mucho al método *query()* de un SQLiteDatabase y, al igual que este, nos devuelve un objeto Cursor para iterar sobre los datos. No obstante, vamos a comentar el significado de cada uno de los parámetros requeridos por el método:

- **uri.** Se trata del parámetro más importante de todos, pues a través de éste identificamos al proveedor con el que queremos conectarnos. Y es que todos los proveedores de contenidos se identifican con una cadena en forma de *uri* (identificador universal de recursos). El formato de la uri de un proveedor es el siguiente:

prefijo://autoridad/datos

El prefijo para todos los proveedores de contenido es content, mientras que la autoridad representa el nombre del proveedor de contenido. Un proveedor de contenido puede proporcionar diferentes datos, por lo que a continuación de la autoridad se indicarán los datos solicitados. Más adelante veremos un ejemplo de recuperación de datos de un proveedor, a continuación, te indicamos los identificadores de algunos de los proveedores más utilizados:

- content:// com.android.contacts/data. Cadena uri del proveedor que da acceso a la lista de contactos. El objeto Uri asociado a esta cadena se encuentra ya definido dentro de la constante android.provider.ContactsContract.Data.CONTENT_URI.

- content://call_log/calls. Cadena uri del proveedor de la pila de llamadas.

- content://media/internal/images. Cadena uri del proveedor de imágenes almacenadas en el terminal.

- **projection.** Array con los nombres de las columnas de datos que se quieren recuperar. Como veremos después en el ejemplo, la mayoría de los proveedores disponen de unas constantes en las que se encuentran definidos estos nombres.

▪ **selection.** Condición de selección, siguiendo el formato de la cláusula where de SQL

▪ **selectionArgs.** Valores de los argumentos para los parámetros definidos en la selección

▪ **sortOrder.** Criterio de ordenación.

▪ **insert(Uri url, ContentValues values).** Inserta una nueva fila de datos en el proveedor cuya uri se proporciona como primer parámetro. Los datos se proporcionan como un objeto ContentValues, al igual que en el método insert() de SQLiteDatabase

▪ **delete(Uri url, String where, String[] selectionArgs).** Realiza la eliminación de datos sobre el proveedor indicado en la Uri. El criterio de eliminación y los valores de los argumentos se proporcionan en el segundo y tercer parámetro, respectivamente.

▪ **update(Uri uri, ContentValues values, String where, String[] selectionArgs).** Actualiza sobre el proveedor indicado aquellas filas de datos que cumplen la condición where. Los nuevos datos se proporcionan en el objeto ContentValues.



Obtención de un objeto `ContentResolver`

La obtención de un objeto `ContentResolver` para poder contactar con un proveedor de contenidos es sumamente sencilla; el método `getContentResolver()` de la interfaz `Context` nos proporciona una implementación de `ContentResolver`. Dado que `Activity` hereda `Context`, podríamos obtener un `ContentResolver` desde el interior de una actividad con esta sencilla instrucción:

```
ContentResolver cr=this.getContentResolver();
```



5.2 | Permisos

Para poder utilizar determinados proveedores de contenidos, las aplicaciones requieren disponer de determinados permisos. Por ejemplo, para leer y modificar los contactos del terminal es necesario disponer de permisos de lectura y escritura en los contactos.

Los permisos se otorgan a las aplicaciones en el AndroidManifest.xml utilizando el elemento `<uses-permission>`, que se debe colocar delante de la etiqueta `<application>`. Cada permiso tiene un nombre definido en una constante **dentro de android.permission**, por ejemplo, el permiso de lectura de contactos es `android.permission.READ_CONTACTS`, mientras que el de lectura de ficheros del almacenamiento externo sería `android.permission.READ_EXTERNAL_STORAGE`.

El siguiente listado corresponde con el extracto de un AndroidManifest.xml en el que se han definido permisos de lectura y escritura en los contactos del teléfono:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="moviles.android.ejemplo09" >

    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportRtl="true"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
```



Ejercicio resuelto

A continuación vamos a realizar una aplicación en la que se hará uso de un proveedor de datos proporcionado por Android, concretamente, del proveedor de contactos.

Se trata de un programa bastante simple, consistente en una actividad que mostrará la lista de contactos del terminal. En ella aparecerá solamente el nombre y el teléfono de cada contacto.

Para apreciar mejor el resultado, te recomiendo que pruebes el ejercicio en tu teléfono, así podrás ver tus contactos reales. No obstante, es posible también crear contactos de prueba en el emulador a través del icono "Contacts".

En cuanto al archivo de plantilla de la actividad, constará únicamente de un TextView con el texto "Listado de contactos" y debajo de él un ListView en el que mostraremos la lista de nuestros contactos. Este sería el código del archivo content_main.xml:

```

RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto" android:layout_
width="match_parent"
    android:layout_height="match_parent" android:paddingLeft="@dimen/
activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:paddingBottom="@dimen/activity_vertical_margin"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:showIn="@layout/activity_main" tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Contactos del teléfono:"
        android:id="@+id/textView" />

    <ListView
        android:id="@+id/lvContactos"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true" />
</RelativeLayout>

```

En cuanto al código de la actividad, queremos que los contactos aparezcan desde el principio al cargarse la actividad, por lo que el código para acceder al proveedor, recuperar los contactos y mostrarlos en la lista se tendrá que ejecutar en el método *onCreate()* de la actividad. Lo que haremos en este ejercicio es codificar estas instrucciones en un método a parte que será llamado desde el *onCreate()*. Así es como quedará el código de la actividad:

```
import android.content.ContentResolver;
import android.database.Cursor;
import android.os.Bundle;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.provider.ContactsContract.Data;
import android.support.v4.widget.CursorAdapter;
import android.support.v7.app.AppCompatActivity;
import android.widget.ListView;
import android.widget.SimpleCursorAdapter;

public class MainActivity extends AppCompatActivity {

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

    public void cargarContactos(){
        //obtenemos el contentresolver para acceder a proveedores
        ContentResolver resolver=this.getContentResolver();
        //recuperamos todos los contactos
        //se establece una condición para que,
        //de todas las combinaciones obtenidas,
        // solamente se queda con aquellas
        //que corresponden al tipo teléfono
        Cursor c=resolver.query(Data.CONTENT_URI,
            null,
            Data.MIMETYPE+"='"+Phone.CONTENT_ITEM_TYPE+"'",
            null,
            null);
        //array con el nombre de las columnas a mostrar
```




```
String[] nombres={Phone.DISPLAY_NAME,Phone.NUMBER};
//nombres genéricos de los controles de tipo texto
//que serán utilizados para mostrar el contenido de
//cada fila
int[] ids={android.R.id.text1,android.R.id.text2};
//creamos el adaptador
SimpleCursorAdapter adp=new SimpleCursorAdapter(this,
    android.R.layout.two_line_list_item,
    c,
    nombres,
    ids,
    CursorAdapter.FLAG_REGISTER_CONTENT_OBSERVER);

    ListView lvContactos=(ListView)this.findViewById(R.
id.lvContactos);
    lvContactos.setAdapter(adp);

}

}
```

Como vemos, los nombres de las columnas los hemos obtenido de una serie de constantes definidas en la clase Phone, que se encuentra en android.provider.ContactsContract.CommonDataKinds. Concretamente se trata de las constantes DISPLAY_NAME y NUMBER, que representan el nombre del contacto y número de teléfono, respectivamente.

En cuanto a la uri del proveedor, se encuentra definida en la constante CONTENT_URI de la clase android.provider.ContactsContract.Data.

En la creación del objeto Cursor, debemos fijarnos en el parámetro que indica la condición de búsqueda de filas:

```
Data.MIMETYPE+"="+Phone.CONTENT_ITEM_TYPE+""
```

Y es que en la tabla de contactos encontramos filas con distinta información sobre el contacto, para quedarnos solamente con aquellas que contienen la información telefónica, debemos establecer la condición anterior, donde Data.MIMETYPE corresponde al tipo de dato de la fila y la constante Phone.CONTENT_ITEM_TYPE representa el tipo teléfono.

Observa también el uso del parámetro android.R.layout.two_line_list_item en la creación del SimpleCursorAdapter, el cual nos permite indicar la utilización de una plantilla consistente en la inclusión de dos elementos por cada fila .



Finalmente, habrá que incluir en el AndroidManifest.xml el permiso de lectura para la lista de contactos, tal y como indicamos anteriormente:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="moviles.android.ejemplo09" >

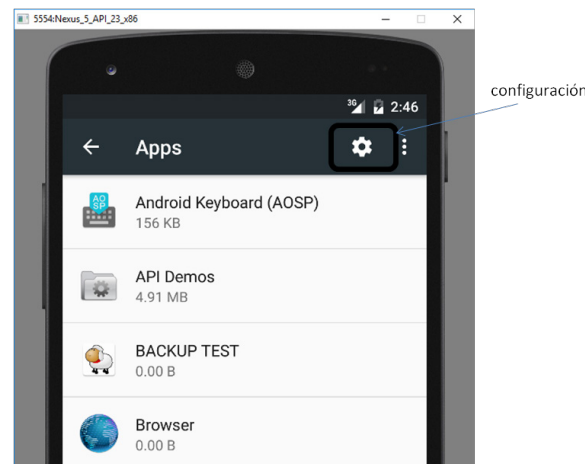
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>

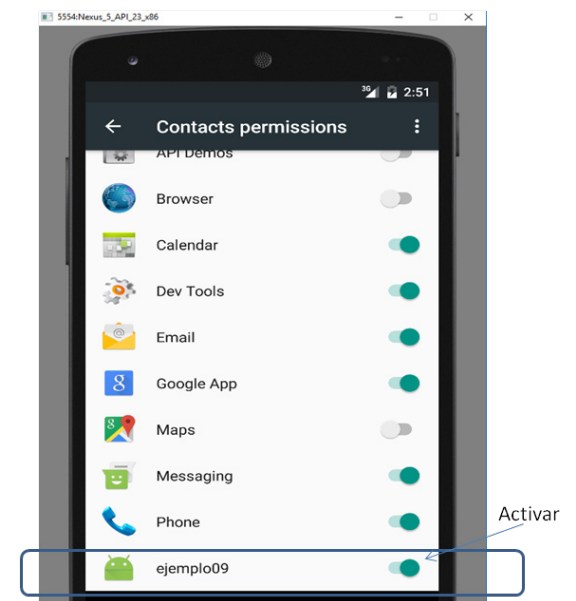
</manifest>
```

Dependiendo del emulador utilizado, es posible que la aplicación falle al ejecutarse, indicando un error de permisos. Y es que, aunque hayamos definido el permiso de lectura de contactos en el archivo de manifiesto, es necesario habilitar dicho permiso en el emulador.

Para hacerlo, entraremos en la configuración del emulador a través del icono *settings*, sección *Apps*. Dentro de esta ventana, pulsaremos el icono de la parte superior izquierda:



Dentro de la ventana que aparece a continuación, elegiremos *App permissions*. Nos aparecerá una lista con los objetos a los que se puede asignar permisos y elegiremos *contacts*. En la ventana que aparece a continuación, debemos activar el permiso para la aplicación *ejemplo09*:



Telefonica

EDUCACIÓN DIGITAL