TEMA 7. PERSISTENCIA

1. INTRODUCCIÓN

Con el término persistencia, en relación a una aplicación, nos referimos a la capacidad de esta para que los datos por ella manipulados perduren en el tiempo tras la ejecución de la misma. Básicamente, consiste en almacenar información en un medio secundario, no volátil, para su posterior recuperación y utilización, independientemente en el tiempo del proceso que los creó.

Android ofrece varias opciones para asegurar los datos de una aplicación, como pueden ser las bases de datos, las preferencias compartidas, etc., utilizándose una u otra variante dependiendo de las necesidades específicas de la misma. Este sistema permite desde almacenar y compartir las preferencias de las aplicaciones, hasta realizar almacenamientos remotos utilizando recursos y protocolos de la red, pasando por persistencia local, en ficheros, en tarjetas de datos y bases de datos relacionales, entre otros.

2. PREFERENCIAS COMPARTIDAS

Las preferencias no son más que datos que una aplicación debe guardar para personalizar la experiencia del usuario, por ejemplo información personal, opciones de presentación, etc. Uno de los métodos disponibles en la plataforma Android para almacenar datos son las bases de datos SQLite. Las preferencias de una aplicación se podrían almacenar por supuesto utilizando este método, y no tendría nada de malo, pero Android proporciona otro método alternativo diseñado específicamente para administrar este tipo de datos: las **preferencias compartidas o *shared preferences***. Cada preferencia se almacenará en forma de clave-valor, es decir, cada una de ellas estará compuesta por un identificador único (por ejemplo, "*email*") y un valor asociado a dicho identificador (por ejemplo "*prueba@email.com*"). Además, y a diferencia de SQLite, los datos no se guardan en un fichero binario de base de datos, sino en ficheros XML.

La API para el manejo de estas preferencias es muy sencilla. Toda la gestión se centraliza en la clase *SharedPrefences*, que representará a una colección de preferencias. Una aplicación Android puede gestionar varias colecciones de preferencias, que se diferenciarán mediante un identificador único.

Para obtener una referencia a una colección determinada utilizaremos el método *getSharedPrefences()* al que pasaremos el identificador de la colección y un modo de acceso. El **modo de acceso** indicará qué aplicaciones tendrán acceso a la colección de preferencias y qué operaciones tendrán permitido realizar sobre ellas. Así, tendremos tres posibilidades principales:

- **MODE_PRIVATE:** sólo nuestra aplicación tiene acceso a estas preferencias.
- **MODE_WORLD_READABLE:** todas las aplicaciones pueden leer estas preferencias, pero sólo la nuestra puede modificarlas.
- **MODE_WORLD_WRITABLE:** todas las aplicaciones pueden leer y modificar estas preferencias.

Las dos últimas opciones son relativamente "peligrosas" por lo que en condiciones normales no deberían usarse. De hecho, se han declarado como obsoletas en la API 17 (Android 4.2).

Teniendo todo esto en cuenta, para **crear o para obtener** una referencia a una colección de preferencias llamada, y como modo de acceso exclusivo para nuestra aplicación haríamos lo siguiente:

```
SharedPreferences prefs = getSharedPreferences("Nombre_archivo",
                                                Context.MODE_PRIVATE);
```

Para **actualizar o insertar nuevas preferencias**, la actualización o inserción no la haremos directamente sobre el objeto *SharedPreferences*, sino sobre su objeto de edición *SharedPreferences.Editor*. A este último objeto accedemos mediante el método *edit()* de la clase *SharedPreferences*. Una vez obtenida la referencia al editor, utilizaremos los métodos *put* correspondientes al tipo de datos de cada preferencia para actualizar/insertar su valor; por ejemplo *putString(clave, valor)*, para actualizar una preferencia de tipo String. Tendremos disponibles métodos *put* para todos los tipos de datos básicos: *putInt()*, *putFloat()*, *putBoolean()*, etc. Finalmente, una vez actualizados/insertados todos los datos necesarios llamaremos al método *commit()* para confirmar los cambios.

```
SharedPreferences.Editor editor = prefs.edit();
editor.putString("Clave1","Valor1");
editor.putString("Clave2","Valor2");
editor.commit();
```

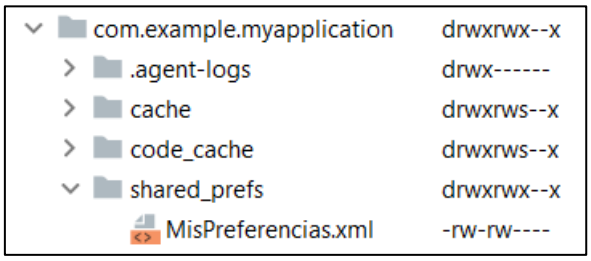
Por ejemplo:

```
SharedPreferences prefs = getSharedPreferences("MisPreferencias",
Context.MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
editor.putString("email","modificado@email.com");
editor.putString("nombre","Prueba");
editor.commit();
```

Estas preferencias se almacenarían en un fichero XML, ubicado dentro del dispositivo móvil en una ruta con el siguiente patrón.

```
/data/data/paquete.java/shared_prefs/nombre_coleccion.xml
```

Siguiendo la opción de menú en Android Studio *View>Tool>Windows>Device File Explorer*, podemos acceder al explorador de archivos en el dispositivo móvil. En la ruta antes mostrada, podemos encontrar el archivo de preferencias de la actividad realizada. En este caso, la ubicación y el contenido serían los siguientes:




```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="nombre">Prueba</string>
    <string name="email">modificado@email.com</string>
</map>
```

Para realizar la **lectura de los datos almacenados**, utilizaremos el método *get* correspondiente al tipo de dato que hemos almacenado. Así, usaremos *getString()*, *getFloat()*, *getBoolean()*, *getInt()*, etc.

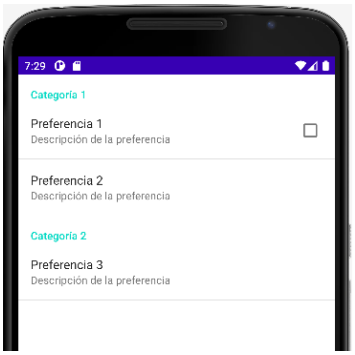
```
String valor = prefs.getString("clave","valor por defecto");
```

Por ejemplo:

```
SharedPreferences prefs = getSharedPreferences("MisPreferencias",
Context.MODE_PRIVATE);
String correo = prefs.getString("email","correo_por_defecto@gmail.com");
```

3. PREFERENCIAS COMPARTIDAS II: PREFERENCE ACTIVITY

Otra forma de almacenar las preferencias de una aplicación es con la clase PreferenceActivity. Esta nos permite definir las preferencias mediante un fichero XML, donde se pueden incluir las opciones de configuración de manera cómoda.



El fichero XML que define las preferencias de la aplicación debe ubicarse en la ruta res/xml, donde se pueden guardar tantos ficheros como necesite la aplicación.

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

        <!-- Preferencias definidas -->

</PreferenceScreen>
```

3.1. Categorías

En el fichero de preferencias podemos establecer distintas secciones que se presentarán en la interfaz de usuario como apartados distintos con título propio. Esto se consigue con el elemento *PreferenceCategory*, al que daremos un texto descriptivo utilizando su atributo *android:title*.

```
<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">

    <PreferenceCategory android:title="Categoría 1">
        <!-- Preferencias de esta categoría -->
    </PreferenceCategory>


    <PreferenceCategory android:title="Categoría 2">
        <!-- Preferencias de esta categoría -->
    </PreferenceCategory>

</PreferenceScreen>
```

Dentro de las categorías, podemos añadir elementos de los siguientes tipos:

- **CheckBoxPreference:** marca seleccionable.
- **EditTextPreference:** cadena simple de texto.
- **ListPreference:** lista de valores seleccionables (exclusiva).
- **MultiSelectListPreference:** lista de valores seleccionables (múltiple).

3.2. Elementos

3.2.1. CheckBoxPreference

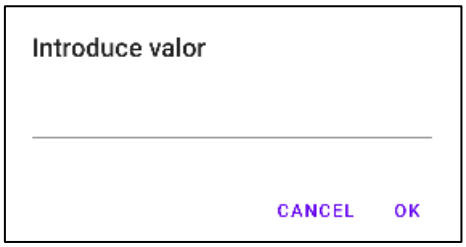
Representa un tipo de opción que sólo puede tomar dos valores distintos: activada o desactivada. Es el equivalente a un control de tipo checkbox. En este caso tan sólo tendremos que especificar los atributos:

- **android:key**: nombre interno de la opción (es decir, el identificador).
- **android:title**: texto a mostrar.
- **android:summary**: descripción de la opción.

```
<CheckBoxPreference
    android:key="opcion1"
    android:title="Preferencia 1"
    android:summary="Descripción de la preferencia"/>
```

3.2.2. EditTextPreference

Representa un tipo de opción que puede contener como valor una cadena de texto. Al pulsar sobre una opción de este tipo se mostrará un cuadro de diálogo sencillo que solicitará al usuario el texto a almacenar.



Para este tipo, además de los tres atributos comunes a todas las opciones (*key*, *title* y *summary*) también tendremos que indicar el texto a mostrar en el cuadro de diálogo, mediante el atributo *android:dialogTitle*.

```
<EditTextPreference
    android:key="opcion2"
    android:title="Preferencia 2"
    android:summary="Descripción de la preferencia"
    android:dialogTitle="Introduce valor"/>
```

3.2.3. ListPreference

Representa un tipo de opción que puede tomar como valor un elemento, y sólo uno, seleccionado por el usuario entre una lista de valores predefinida.

Además de los cuatro ya comentados (*key*, *title*, *summary* y *dialogTitle*) tendremos que añadir dos más, uno de ellos indicando la lista de valores a visualizar en la lista (*android:entries*) y el otro indicando los valores internos que utilizaremos para cada uno de los valores de la lista anterior (*android:entriesValues*). Estas listas de valores las definiremos también como ficheros XML dentro de la carpeta */res/xml*.

```
<ListPreference
    android:key="opcion3"
    android:title="Preferencia 3"
    android:summary="Descripción de la preferencia"
    android:dialogTitle="IndicarPais"
    android:entries="@array/pais"
    android:entryValues="@array/codigopais"/>
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="pais">
        <item>España</item>
        <item>Francia</item>
        <item>Alemania</item>
    </string-array>

    <string-array name="codigopais">
        <item>ESP</item>
        <item>FRA</item>
        <item>ALE</item>
    </string-array>

</resources>
```

3.2.4. MultiSelectListPreference

Las opciones de este tipo son muy similares a las ListPreference, con la diferencia de que el usuario puede seleccionar varias de las opciones de la lista de posibles valores. Los atributos a asignar son por tanto los mismos que para el tipo anterior.

```
<MultiSelectListPreference
    android:key="opcion4"
    android:title="Preferencia 4"
    android:summary="Descripción de la preferencia"
    android:dialogTitle="IndicarPais"
    android:entries="@array/pais"
    android:entryValues="@array/codigopais"/>
```

3.3. Acceso a las preferencias

Para mostrar las preferencias en pantalla, se debe crear una clase Java que extienda de PreferenceActivity y cuya función es la de mostrar el contenido del fichero de preferencias.

```
public class OpcionesActivity extends PreferenceActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.opciones);
    }
}
```

Para **lanzar** esta clase desde la actividad principal, se puede utilizar un Intent como respuesta a cualquier evento (un clic de botón, por ejemplo).

```
btnPreferencias.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        startActivity(new
Intent(MainActivity.this,OpcionesActivity.class));
    }
});
```

Para **obtener las preferencias marcadas** se utilizará la clase *SharedPreferences*, gestionada por *PreferenceManager*. Una instancia de esta permitirá obtener los valores de cada una de ellas a través de las claves asignadas, definiendo valores por defecto para aquellos casos que no se haya introducido ningún valor.

```
btnConsultar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        SharedPreferences pref =
PreferenceManager.getDefaultSharedPreferences(MainActivity.this);

        Log.i("","Opción 1: " + pref.getBoolean("opcion1",false));
        Log.i("","Opción 2: " + pref.getString("opcion2",""));
        Log.i("", "Opción 3:" + pref.getString("opcion3",""));
    }
});
```

3.4. Notas adicionales

Aunque esto continúa funcionando sin problemas en versiones recientes de Android, la API 11 trajo consigo una nueva forma de definir las pantallas de preferencias haciendo uso de fragmentos. Para ello, basta simplemente con definir la clase java del fragmento, que deberá extender de *PreferenceFragment* y añadir a su método *onCreate()* una llamada a *addPreferencesFromResource().*

```
public static class OpcionesFragment extends PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        addPreferencesFromResource(R.xml.opciones);
    }
}
```

Hecho esto, ya no será necesario que la clase de nuestra pantalla de preferencias extienda de *PreferenceActivity*, sino que podrá ser una actividad normal. Para mostrar el fragmento creado como contenido principal de la actividad *utilizaríamos el fragment manager para sustituir el contenido de la pantalla (android*.R.id.content) por el de nuestro fragmento de preferencias recién definido:

```
public class SettingsActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        getFragmentManager().beginTransaction()
                .replace(android.R.id.content, new OpcionesFragment())
                .commit();
    }
}
```

4. ALMACENAMIENTO EN LA MEMORIA INTERNA DEL DISPOSITIVO

Lo primero que hay que tener en cuenta es dónde queremos almacenar los ficheros y el tipo de acceso que queremos tener a ellos. Así, podremos leer y escribir ficheros localizados en:

1. La **memoria interna** del dispositivo.
2. La **tarjeta SD** externa, si existe.
3. La propia aplicación, en forma de **recurso**.

4.1. Almacenar información en la memoria interna

Cuando almacenamos ficheros en la memoria interna debemos tener en cuenta las limitaciones de espacio que tienen muchos dispositivos, por lo que no deberíamos abusar de este espacio utilizando ficheros de gran tamaño.

Para el **proceso de escritura** en memoria interna, Android proporciona el método *openFileOutput()*, que recibe como parámetros el nombre del fichero y el modo de acceso con el que queremos abrir el fichero. Este modo de acceso puede variar entre:

- **MODE_PRIVATE:** para acceso privado desde nuestra aplicación (crea el fichero o lo sobrescribe si ya existe).
- **MODE_APPEND:** para añadir datos a un fichero ya existente.
- **MODE_WORLD_READABLE:** para permitir a otras aplicaciones leer el fichero.
- **MODE_WORLD_WRITABLE**: para permitir a otras aplicaciones escribir sobre el fichero.

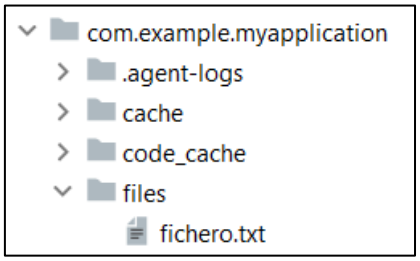
Los dos últimos no deberían utilizarse dada su peligrosidad. De hecho, han sido declarados como obsoletos (deprecated) en la API 17.

Este método devuelve una referencia al stream de salida asociado al fichero (en forma de objeto *FileOutputStream*), a partir del cual ya podremos utilizar los métodos de manipulación de ficheros tradicionales del lenguaje java (api java.io). Como ejemplo, convertiremos este stream a un OutputStreamWriter para escribir una cadena de texto al fichero.

```
try{
    OutputStreamWriter miFichero = new
OutputStreamWriter(openFileOutput("fichero.txt", Context.MODE_PRIVATE));
    miFichero.write("TEXTO DE PRUEBA");
    miFichero.close();

}catch (Exception ex){
    Log.e("Ficheros","Error al escribir fichero en memoria interna");
}
```

Los ficheros creados, Android los almacenará en la siguiente ruta:

```
/data/data/paquete.java/files/nombre_fichero
```



Para leer los datos almacenados, utilizaremos el método *openFileInput()* para abrir el fichero, y los métodos de lectura de java.io para leer el contenido (por ejemplo, con el método *BufferedReader*).

```
try {
    BufferedReader miFichero = new BufferedReader(new
InputStreamReader(openFileInput("fichero.txt")));
    String texto = miFichero.readLine();
    miFichero.close();
} catch (Exception e) {
    Log.e("Ficheros","Error al leer desde la memoria interna");
}
```

4.2. Almacenar información como recurso

La segunda forma de almacenar ficheros en la memoria interna del dispositivo es incluirlos como recurso en la propia aplicación. Aunque este método es útil en muchos casos, sólo debemos utilizarlo cuando no necesitemos realizar modificaciones sobre los ficheros, ya que tendremos limitado el **acceso** a **sólo lectura**.

Para incluir un fichero como recurso de la aplicación **debemos colocarlo en la carpeta *res/raw*** de la aplicación, **en formato *.txt***. Esta carpeta no existe por defecto, así que habrá que crearla.

El código para leer un fichero como recurso sería:

```
try{
    InputStream fraw = getResources().openRawResource(R.raw.prueba_raw);
    BufferedReader brin = new BufferedReader(new InputStreamReader(fraw));

    String linea = brin.readLine();
    fraw.close();
}
catch (Exception ex){
    Log.e("Ficheros","Error al leer fichero desde recurso raw");
}
```

4.3. Almacenar en memoria externa

Hasta ahora hemos visto como almacenar información en la memoria interna. Sin embargo, esta memoria suele ser relativamente limitada y no es aconsejable almacenar en ella ficheros de gran tamaño. La alternativa natural es utilizar para ello la memoria externa del dispositivo, constituida normalmente por una tarjeta de memoria SD, aunque también está presente en forma de almacenamiento no extraíble del dispositivo, aunque no por ello debe confundirse con la memoria interna. A diferencia de la memoria interna, el almacenamiento externo es público, es decir, todo lo que escribamos en él podrá ser leído por otras aplicaciones y por el usuario, por tanto hay que tener cierto cuidado a la hora de decidir lo que escribimos en memoria interna y externa.

Para tener acceso a la memoria externa tendremos que especificar en el fichero AndroidManifest.xml que nuestra aplicación necesita permiso de escritura en dicha memoria. Para añadir un nuevo permiso usaremos la siguiente clausula:

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

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

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

    <application
        android:allowBackup="true"
```

A diferencia de la memoria interna, la tarjeta de memoria no tiene por qué estar presente en el dispositivo, e incluso estándolo puede no estar reconocida por el sistema. Por tanto, el primer paso recomendado a la hora de trabajar con ficheros en memoria

externa es asegurarnos de que dicha memoria está presente y disponible para leer y/o escribir en ella.

Para esto la API de Android proporciona (como método estático de la clase *Environment*) el método *getExternalStorageStatus()*, que nos dice si la memoria externa está disponible y si se puede leer y escribir en ella. Este método devuelve una serie de valores que nos indicarán el estado de la memoria externa, siendo los más importantes los siguientes:

- **Environment.MEDIA_MOUNTED:** indica que la memoria externa está disponible y podemos tanto leer como escribir en ella.
- **Environment.MEDIA_MOUNTED_READ_ONLY:** indica que la memoria externa está disponible pero sólo podemos leer de ella.
- **Environment.MEDIA_UNMOUNTED:** la memoria está disponible, pero no está montada.
- **Environment.MEDIA_REMOVED:** indica que la memoria no está disponible.

Paso 1: Comprobar que la memoria externa está presente y disponible

```
boolean sdDisponible = false;
boolean sdAccesoEscritura = false;

// Comprobamos el estado de la memoria externa (SD)
String estado = Environment.getExternalStorageState();
if (estado.equals(Environment.MEDIA_MOUNTED)){
    sdDisponible = true;
    sdAccesoEscritura = true;
}
else if (estado.equals(Environment.MEDIA_MOUNTED_READ_ONLY)){
    sdDisponible = true;
    sdAccesoEscritura = false;
}
```

Paso 2: Leer o escribir de ella

Escribir

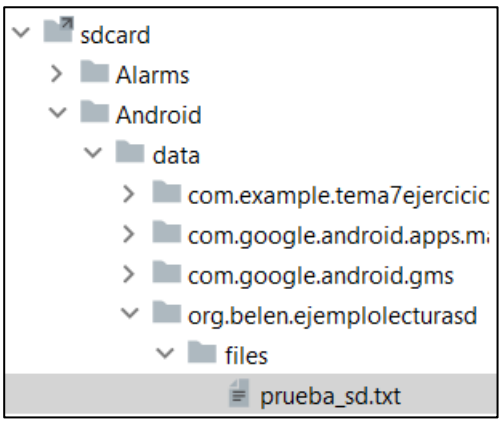
Una vez que se ha comprobado que la tarjeta de memoria existe y está montada, procedemos a escribir en ella, para lo que vamos a obtener la dirección del directorio raíz.

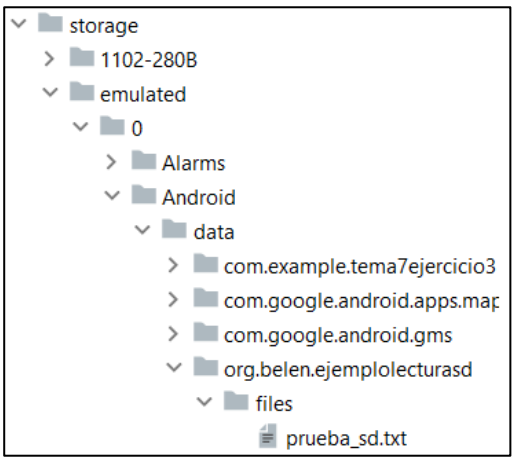
```
try {
    // Obtengo la ruta del directorio raíz
    File ruta_sd = Environment.getExternalStorageDirectory();

    File f = new File(ruta_sd.getAbsolutePath(),"prueba_sd.txt");

    OutputStreamWriter fout =
        new OutputStreamWriter(new FileOutputStream(f));

    fout.write("Texto de prueba");
    fout.close();
}
catch (Exception ex){
    Log.e("Ficheros","Error al escribir en la tarjeta SD");
}
```

El método *getExternalStorageDirectory* fue declarado obsoleto con la API 29. Por lo tanto, si lo utilizamos con esa versión o superiores, nos dará error al crear el fichero.

El código anterior funciona sin problemas pero escribirá el fichero directamente en la carpeta raíz de la memoria externa. Esto, aunque en ocasiones puede resultar necesario, no es una buena práctica. Lo correcto sería disponer de una **carpeta propia para nuestra aplicación**, lo que además tendrá la ventaja de que al desinstalar la aplicación también se liberará este espacio. Esto lo conseguimos utilizando el método *getExternalFilesDir(null)* en vez de *getExternalStorageDirectory()*. Si se define como null, el fichero de almacenamiento será creado directamente en la ruta de una carpeta específica para nuestra aplicación dentro de la memoria externa siguiendo el siguiente patrón:

<raíz_mem_ext>/Android/data/nuestro.paquete.java/files





Si en vez de null le indicamos como parámetro un tipo de datos determinado (DIRECTORY_MUSIC, DIRECTORY_PICTURES, DIRECTORY_MOVIES, DIRECTORY_RINGTONES, DIRECTORY_PODCASTS, DIRECTORY_ALARMS, DIRECTORY_NOTIFICATIONS) nos devolverá una subcarpeta dentro de la anterior con su nombre correspondiente. Así, por ejemplo, una llamada al método getExternalFilesDir(Environment.DIRECTORY_MUSIC) nos devolvería la siguiente carpeta:

<raíz_mem_ext>/Android/data/nuestro.paquete.java/files/Music

Esto último, además, ayuda a Android a saber qué tipo de contenidos hay en cada carpeta, de forma que puedan clasificarse correctamente por ejemplo en la galería multimedia.

```
try {

    // Ruta para directorios predefinidos
    File ruta_sd = getExternalFilesDir(null);


    File f = new File(ruta_sd.getAbsolutePath(),"prueba_sd.txt");

    OutputStreamWriter fout =
         new OutputStreamWriter(new FileOutputStream(f));

    fout.write("Texto de prueba");
    fout.close();
}
catch (Exception ex){
    Log.e("Ficheros","Error al escribir en la tarjeta SD");
}
```

Si ejecutamos ahora el código y nos vamos al explorador de archivos del DDMS podremos comprobar cómo se ha creado correctamente el fichero en el directorio raíz de nuestra SD.

Leer

Para leer los datos, actuamos de manera similar, pero creando el buffer en el sentido contrario (es decir, de lectura).

```
try {
    File ruta_sd = getExternalFilesDir(DIRECTORY_RINGTONES);
    File f = new File(ruta_sd.getAbsolutePath(),"prueba_sd.txt");

    BufferedReader fin =
            new BufferedReader(
                new InputStreamReader(
                    new FileInputStream(f)));
    String texto = fin.readLine();
    fin.close();
}
catch (Exception ex){
    Log.e("Ficheros","Error al leer en la tarjeta SD");
}
```

5. ALMACENAR INFORMACIÓN EN BASE DE DATOS (SQLITE)

Cuando el volumen de información dificulta el realizar su almacenamiento por alguno de los métodos vistos anteriormente, y deseamos asegurar la información en el propio dispositivo, puede ser una buena opción utilizar bases de datos.

De las posibles opciones que las compatibilidades de Android ofrecen, SQLite es una buena opción. SQLite es un motor de bases de datos de pequeño tamaño, no necesita servidor, precisa de poca configuración, es transaccional y de código libre. Guarda toda la base de datos en un único fichero y no requerirá la instalación adicional de ningún gestor de bases de datos. Como inconveniente de este método es que no mantiene la integridad de datos, le falta soporte Unicode y una interfaz gráfica de administración.

Android incorpora de serie todas las herramientas necesarias para la creación y gestión de bases de datos SQLite, y entre ellas una completa API para llevar a cabo de manera sencilla todas las tareas necesarias.

La base de datos que creemos se almacenará en **/data/data/paquete.java/databases/nombre_bbdd**. Por defecto, todas las bases de datos son privadas y solo accesibles por la aplicación creadora.

```
/data/data/paquete.java/databases/nombre_bbdd
```

5.1. Creación

Para gestionar la base de datos, deberemos usar la clase ***SQLiteOpenHelper***, para lo que se debe crear una clase Java que herede de esta y cuyo constructor reciba como parámetros el contexto, el nombre de la base de datos, el *CursorFactory* (suele ser *null* para utilizar el estándar de *SQLiteCursor*) y la versión del esquema de base de datos (para posibles migraciones).

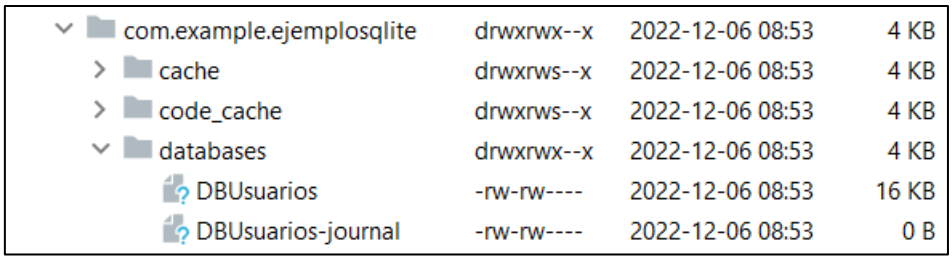
La clase *SQLiteOpenHelper* tiene tan sólo un constructor, que normalmente no necesitaremos sobrescribir, y dos métodos abstractos, ***onCreate()*** y ***onUpgrade()***, que deberemos personalizar con el código necesario para crear nuestra base de datos y para actualizar su estructura respectivamente.

```
public class UsuariosSQLiteHelper extends SQLiteOpenHelper {

    // Sentencia SQL para crear la tabla de usuarios
    String sqlCreate = "CREATE TABLE Usuarios(codigo INTEGER PRIMARY
KEY, nombre TEXT)";

    public UsuariosSQLiteHelper(Context context, String nombre,
SQLiteDatabase.CursorFactory factory, int version){
        super(context, nombre, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // Se ejecuta la sentencia SQL de creación de tabla
        db.execSQL(sqlCreate);
    }


    //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.
    @Override
    public void onUpgrade(SQLiteDatabase db, int i, int i1) {

        //Se elimina la versión anterior de la tabla
        db.execSQL("DROP TABLE IF EXISTS Usuarios");

        //Se crea la nueva versión de la tabla
        db.execSQL(sqlCreate);
    }
}
```

El método *onCreate()* será ejecutado automáticamente por la clase *UsuariosSQLiteHelper* 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 el caso del ejemplo, sólo se ha creado la tabla Usuarios. Para la creación de la tabla se utilizará la sentencia SQL ya definida y se ejecutará 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. El modo más sencillo de hacerlo es borrar el contenido y volver a crearlo, que es lo que se ha hecho en el ejemplo.

Para comprobar que realmente se ha creado la base de datos, podemos ir al explorador de archivos del dispositivo (*Device File Explorer*) y consultarlo en la ruta mencionada anteriormente (/data/data/paquete/dabases/nombre_base_datos)



Con esto ya comprobamos al menos que el fichero de nuestra base de datos se ha creado en la ruta correcta. Si queremos comprobar que tanto las tablas creadas como los datos insertados (más adelante veremos cómo hacerlo) también se han incluido correctamente en la base de datos, tenemos dos métodos distintos, que podemos consultar en el siguiente enlace.

5.2. Apertura

Para manejar la base de datos desde la actividad principal y realizar los procesos de mantenimiento de la misma, se ha de instanciar la clase Java, que extiende de *SQLiteOpenHelper*, e invocar el método que nos la abre para escritura/lectura. 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.

```
UsuariosSQLiteHelper usuariosBBDD = new UsuariosSQLiteHelper(this,
                                                "DBUsuarios", null,1);
SQLiteDatabase db = usuariosBBDD.getWritableDatabase();
```

5.3. Insertar registros

Para insertar registros se podrá utilizar el método execSQL, que permite ejecutar sentencias SQL, o usar objetos de la clase *ContentValues*.

En el **primer caso**, un ejemplo sería:

```
db.execSQL("INSERT INTO Usuarios(codigo,nombre) VALUES (5,'usuarioprueba')");
```

En el **segundo caso**, cada objeto de tipo *ContentValues* representa una fila única en la tabla y se encarga de emparejar nombre de columna con valores. Hay que tener en cuenta que, a diferencia de otros motores de bases de datos, los datos en SQLite son débilmente tipados, por lo que es necesario comprobar tipos cuando se asignan o extraen valores de cada columna de una fila.

```
ContentValues nuevoRegistro = new ContentValues();
nuevoRegistro.put("código",valor1);
nuevoRegistro.put("nombre",valor2);
db.insert("Usuarios",null,nuevoRegistro)
```

Un ejemplo completo sería:

```
public class MainActivity extends AppCompatActivity {

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

        // Abrimos la base de datos DBUsuarios en modo escritura
        UsuariosSQLiteHelper usuariosBBDD = new UsuariosSQLiteHelper(this,
"DBUsuarios", null,1);
        SQLiteDatabase db = usuariosBBDD.getWritableDatabase();

        // Comprobamos que se ha abierto correctamente la base de datos
        if (db!=null){
            // Insertamos 5 usuarios
            String usuario;
            for (int i=1; i<=5; i++){
                usuario = "Usuario" + i;
                db.execSQL("INSERT INTO Usuarios(codigo,nombre) VALUES (" + i + ",'"
+ usuario +"')");
            }
        }

        // Cerramos la base de datos
        db.close();;
    }
}
```

5.4. Eliminar registros

Al igual que en el caso anterior, habría dos opciones:

Opción 1

```
db.execSQL("DELETE FROM Usuarios WHERE codigo=5");
```

Opción 2

```
db.delete("Usuarios","codigo=5",null);
```

5.5. Actualizar Registros

Opción 1

```
db.execSQL("UPDATE Usuarios SET nombre='usunuevo' WHERE codigo=5");
```

Opción 2

```
ContentValues valores = new ContentValues();
valores.put("nombre","nuevovalor2");
db.update("Usuarios",valores,"codigo="+valor1,null);
```

5.6. Uso de argumentos en las operaciones

Tanto en el caso de *execSQL()* como en los casos de *update()* o *delete()* podemos utilizar argumentos dentro de las condiciones 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 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:

```
//Eliminar un registro con execSQL(), utilizando argumentos
String[] args = new String[]{"usuario1"};
db.execSQL("DELETE FROM Usuarios WHERE nombre=?", args);

//Actualizar dos registros con update(), utilizando argumentos
ContentValues valores = new ContentValues();
valores.put("nombre","usunuevo");

String[] args = new String[]{"usuario1", "usuario2"};
db.update("Usuarios", valores, "nombre=? OR nombre=?", args);
```

5.7. Consultar registros

Tememos dos opciones 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 clase ***SQLiteDatabase***. 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

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

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 '?':

```
String[] args = new String[] {"usu1"};
Cursor c = db.rawQuery(" SELECT codigo,nombre FROM Usuarios WHERE nombre=?
", args);
```

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():

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

Cursor c = db.query("Usuarios", campos, "usuario=?", args, null, null,
null);
```

Para recorrer y manipular el cursor devuelto tenemos a nuestra disposición varios métodos de la clase Cursor, entre los que se pueden destacar 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 métodos ***getXXX(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 a getDouble(1), y de forma análoga para todos los tipos de datos existentes.

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

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

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

c.close();
```

Al terminar el recorrido, se ha de llamar al método ***close()*** del objeto Cursor.

5.8. Movimientos del cursor

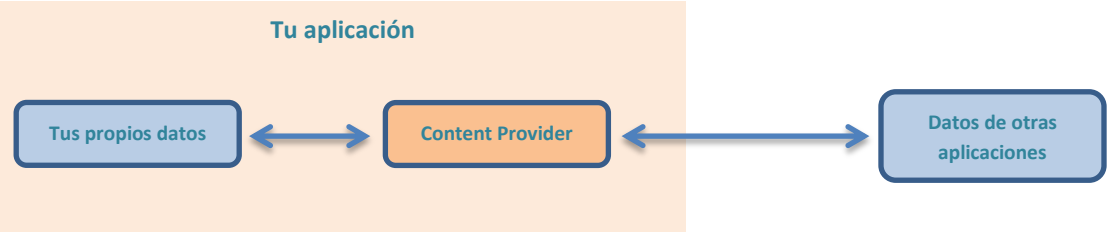
Las opciones de desplazamiento que nos ofrecen los cursores son:

- ***moveToFirst:*** mueve el cursor a la primera fila de los resultados que cumplen la consulta.
- ***moveToNext:*** mueve el cursor a la siguiente fila.
- ***moveToPrevious:*** mueve el cursor a la fila anterior.
- ***getCount:*** devuelve el número de filas que cumplen el resultado de la consulta.
- ***getColumnName:*** devuelve el nombre de la columna que tiene el índice indicado.
- ***getColumnNames:*** devuelve un array con el nombre de todas las columnas del cursor.
- ***moveToPosition:*** mueve el cursor a la fila indicada.
- ***getPosition:*** devuelve el índice de la posición donde se sitúa el cursor.

En los métodos moveToXXX, el método devolverá TRUE en caso de haber sido posible realizarlo sin errores.

6. CONTENT PROVIDERS

Un Proveedor de Contenidos (*Content Provider*) no es más que el mecanismo proporcionado por la plataforma Android para permitir compartir información entre aplicaciones. Una aplicación que desee que todo o parte de la información que almacena esté disponible de una forma controlada para el resto de aplicaciones del sistema deberá proporcionar un *content provider* a través del cual se pueda realizar el acceso a dicha información. Este mecanismo es utilizado por muchas de las aplicaciones estándar de un dispositivo Android, como por ejemplo la lista de contactos, la aplicación de SMS, o el calendario/agenda. Esto quiere decir que podríamos acceder a los datos gestionados por estas aplicaciones desde nuestras propias aplicaciones Android haciendo uso de los *content providers* correspondientes.



Son por tanto dos temas los que debemos tratar en este apartado, por un lado a construir nuevos *content providers* personalizados para nuestras aplicaciones, y por otro utilizar un *content provider* ya existente para acceder a los datos publicados por otras aplicaciones.

6.1. Crear un proveedor de contenidos (*Content Provider*)

Para añadir un *content provider* a nuestra aplicación tendremos que:

1. Crear una nueva clase que extienda a la clase android ***ContentProvider***.
2. Declarar el nuevo *content provider* en nuestro fichero ***AndroidManifest.xml***

Nuestra aplicación tendrá que contar previamente con algún método de almacenamiento interno para la información que queremos compartir. Lo más común será disponer de una base de datos SQLite, aunque también podríamos tener los datos almacenados de cualquier otra forma, por ejemplo en ficheros de texto, ficheros XML, etc. El *content provider* será el mecanismo que nos permita publicar esos datos a terceros de una forma homogénea y a través de una interfaz estandarizada.

Los registros de datos proporcionados por un *content provider* deben contar siempre con un campo llamado _ID que los identifique de forma unívoca del resto de registros.

Así, por ejemplo, los registros devueltos por un *content provider* de clientes podría tener este aspecto:

_ID	Cliente	Teléfono	Email
3	Antonio	900123456	email1@correo.com
7	Jose	900123123	email2@correo.com
9	Luis	900123987	email3@correo.com

Por lo tanto, es interesante que nuestros datos también cuenten internamente con este campo _ID (no tiene por qué llamarse igual) de forma que nos sea más sencillo después generar los resultados del *content provider*.

Visto esto, vamos a ver los pasos para crear el proveedor de contenidos.

Creamos la fuente de datos

```
public class ClientesSqliteHelper extends SQLiteOpenHelper {

    // Sentencia SQL  para crear la tabla clientes
    String sqlCreate = "CREATE TABLE Clientes " +
                       "(_id INTEGER PRIMARY KEY AUTOINCREMENT, " +
                       "nombre TEXT, " +
                       "telefono TEXT, " +
                       "email TEXT)";

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

    @Override
    public void onCreate(SQLiteDatabase db) {
        // Creamos la tabla
        db.execSQL(sqlCreate);

        // Por simplicidad del ejempo, insertamos clientes
        for (int i=1; i<10; i++){
            // Generamos los datos
            String nombre = "Cliente " + i;
            String telefono = "900-123-00" + i;
            String email = "email" + i + "@mail.com";

            // Insertamos los datos en la tabla Cliente
            db.execSQL("INSERT INTO Clientes(nombre, telefono, email) VALUES
('" + nombre + "','" + telefono + "','" + email + "')");

        }
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int versionAnterior, int
versionNueva) {
        db.execSQL("DROP TABLE IF EXISTS Clientes");
        db.execSQL(sqlCreate);
    }
}
```

Preparamos la URI para el Content Provider

Android realiza el acceso a los *content provi*der mediante una URI. Las direcciones URI de los *content providers* están formadas por 3 partes:

- El **prefijo «content://»**, que indica que dicho recurso deberá ser tratado por un content provider.
- El **identificador del *content provider***, también llamado *authority*. Dado que este dato debe ser único, es una buena práctica utilizar un *authority* de tipo «*nombre de clase java invertido*», por ejemplo, «org.iesbelen.android.contentproviders».
- La **entidad concreta a la que queremos acceder** dentro de los datos que proporciona el *content provider*.

En el caso del ejemplo que estamos haciendo será simplemente la tabla de "clientes", ya que será la única existente, pero dado que un *content provider* puede contener los datos de varias entidades distintas, en este último tramo de la URI habrá que especificarlo.

Si se quisiera, en una URI se podría hacer referencia directamente a un registro concreto de la entidad seleccionada. Esto se haría indicando al final de la URI el ID de dicho registro. Por ejemplo, para hacer referencia directamente al cliente con _ID=23 utlizaríamos la uri «content://org.iesbelen.android.contentproviders/clientes/23»

Creamos el *Content Provider*

El siguiente paso es crear una clase que extienda a la clase *ContentProvider*. Para ello, deberemos extender los siguientes métodos:

- onCreate()
- query()
- insert()
- update()
- delete()
- getType()

El primero de ellos nos servirá para inicializar todos los recursos necesarios para el funcionamiento del nuevo *content provider*. Los cuatro siguientes serán los métodos que permitirán acceder a los datos (consulta, inserción, modificación y eliminación, respectivamente) y por último, el método ***getType()*** permitirá conocer el tipo de datos devueltos por el *content provider*.

Definición de constantes

Definición de la Uri

Junto con los métodos, será necesario **definir** una serie de **constantes** necesarias para la creación del proveedor. Una de ellas será la Uri del *content provider*.

```
private static final String uri =
                    "content://como.example.android.ejemplocontentprovider/clientes";
private static final Uri CONTENT_URI = Uri.parse(uri);
```

Definición de un objeto UriMatcher

La primera tarea que el *content provider* deberá hacer cuando se acceda a él será **interpretar la URI utilizada**. Para facilitar esta tarea Android proporciona una clase llamada ***UriMatcher***, capaz de interpretar determinados patrones en una URI. Esto nos será útil para determinar por ejemplo **si una URI hace referencia a una tabla genérica o a un registro concreto a través de su ID**. Por ejemplo:

- «content://org.iesbelen.android.ejemplocontentproviders/clientes»:
  Acceso genérico a tabla de clientes.

- «content://org.iesbelen.android.ejemplocontentproviders/clientes/17»:
  Acceso directo al cliente con ID = 17

Para conseguir esto, definiremos un objeto *UriMatcher* y dos nuevas constantes que representen los dos tipos de URI que hemos indicado: acceso genérico a tabla o acceso a cliente por ID. A continuación inicializaremos el objeto *UriMatcher* indicándole el formato de ambos tipos de URI, de forma que pueda diferenciarlos y devolvernos su tipo (una de las dos constantes definidas).

```
public class ClientesProvider extends ContentProvider {

    // Definición del CONTENT_URI
    private static final String uri =
"content://com.example.android.ejemplocontentprovider/clientes";
    public static final Uri CONTENT_URI = Uri.parse(uri);

    // UriMatcher
    private static final int CLIENTES = 1; // Acceso genérico a tabla
    private static final int CLIENTES_ID = 2; // Acceso a una fila (acceso a clientes por ID)
    private static final UriMatcher uriMatcher; // Objeto UriMatcher

    // Inicializamos el UriMatcher
    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("com.exameple.android.ejemplocontentprovider","clientes",CLIENTES);
        uriMatcher.addURI("com.exameple.android.ejemplocontentprovider","clientes/#",CLIENTES_ID);
    }
```

Definición de las constantes de columna

A continuación, definimos las constantes de las columnas del *content provider*. Como ya dijimos antes existen columnas predefinidas que deben tener todos los *content providers*, por ejemplo la columna _ID. Estas columnas estándar están definidas en la clase *BaseColumns*, por lo que para añadir las nuevas columnas **definiremos una clase interna pública tomando como base la clase BaseColumns y añadiremos nuestras nuevas columnas**.

```
// Clase interna para declarar las constantes de columna
public static final class Clientes implements BaseColumns{
    private Clientes() {}

    // Nombres de columnas
    public static final String COL_NOMBRE = "nombre";
    public static final String COL_TELEFONO = "telefono";
    public static final String COL_EMAIL = "email";
}
```

Definición de atributos para almacenar el nombre y la versión de la BBDD.

Por último, vamos a definir varios atributos privados auxiliares para almacenar el nombre de la base de datos, la versión, y la tabla a la que accederá el *content provider*.

```
// Base de datos
private ClientesSqliteHelper clidbh;
private static final String BD_NOMBRE = "DBClientes";
private static final int BD_VERSION = 1;
private static final String TABLA_CLIENTES = "Clientes";
```

El código creado hasta el momento nos quedaría de la siguiente forma:

```
public class ClientesProvider extends ContentProvider {

    // Definición del CONTENT_URI
    private static final String uri =
           "content://com.example.android.ejemplocontentprovider/clientes";
    public static final Uri CONTENT_URI = Uri.parse(uri);

    // UriMatcher
    private static final int CLIENTES = 1; // Acceso genérico a tabla
    private static final int CLIENTES_ID = 2; // Acceso a una fila (acceso a clientes por ID)
    private static final UriMatcher uriMatcher; // Objeto UriMatcher

    // Inicializamos el UriMatcher
    static {
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
        uriMatcher.addURI("com.exameple.android.ejemplocontentprovider","clientes",CLIENTES);
        uriMatcher.addURI
                    ("com.exameple.android.ejemplocontentprovider","clientes/#",CLIENTES_ID);
    }

    // Clase interna para declarar las constantes de columna
    public static final class Clientes implements BaseColumns{
        private Clientes() {}

        // Nombres de columnas
        public static final String COL_NOMBRE = "nombre";
        public static final String COL_TELEFONO = "telefono";
        public static final String COL_EMAIL = "email";
    }

    // Base de datos
    private ClientesSqliteHelper clidbh;
    private static final String BD_NOMBRE = "DBClientes";
    private static final int BD_VERSION = 1;
    private static final String TABLA_CLIENTES = "Clientes";
```

Definición de los métodos

Una vez definidos todos los miembros necesarios para el nuevo *content provider*, tenemos que implementar los métodos comentados anteriormente.

El primero de ellos es ***onCreate()***. En este método nos limitaremos simplemente a inicializar nuestra base de datos, a través de su nombre y versión, y utilizando para ello la clase ClientesSqliteHelper que creamos al principio del artículo.

```
@Override
public boolean onCreate() {
    clidbh = new ClientesSqliteHelper(getContext(), BD_NOMBRE, null, BD_VERSION);

    return true;
}
```

El método ***query()*** deberá devolver los datos solicitados según la URI indicada y los criterios de selección y ordenación pasados como parámetro, ya sea sobre toda la tabla o sobre una fila de esta. Como vimos, esto será posible gracias al objeto uriMatcher, utilizando su método ***match()***.

```
@Override
public Cursor query(Uri uri, String[] projection,  String selection, String[]
selectionArgs, String sortOrder) {

    // Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if (uriMatcher.match(uri) == CLIENTES_ID){
        where = "_id=" + uri.getLastPathSegment();
    }
    SQLiteDatabase db = clidbh.getWritableDatabase();

    Cursor c = db.query(TABLA_CLIENTES, projection, where, selectionArgs, null, null,
                  sortOrder);

    return c;
}
```

Los métodos ***update()*** y ***delete()*** son análogos a este en su construcción, con la diferencia de que nos devuelven el número de registros en lugar de un cursor con los resultados:

```
@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {

    int cont;

    // Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if (uriMatcher.match(uri) == CLIENTES_ID){
        where = "_id=" + uri.getLastPathSegment();
    }
    SQLiteDatabase db = clidbh.getWritableDatabase();

    cont = db.delete(TABLA_CLIENTES, where, selectionArgs);

    return cont;
}
```

```
@Override
public int update(Uri uri, ContentValues values, String selection, String[]
selectionArgs) {

    int cont;

    // Si es una consulta a un ID concreto construimos el WHERE
    String where = selection;
    if (uriMatcher.match(uri) == CLIENTES_ID){
        where = "_id=" + uri.getLastPathSegment();
    }
    SQLiteDatabase db = clidbh.getWritableDatabase();

    cont = db.update(TABLA_CLIENTES, values, where, selectionArgs);

    return cont;
}
```

A continuación, implementaríamos el método ***insert()***, que debe devolver la URI que hace referencia al nuevo registro insertado.

```
@Override
public Uri insert(Uri uri, ContentValues values) {

    long regId = 1;

    SQLiteDatabase db = clidbh.getWritableDatabase();

    regId = db.insert(TABLA_CLIENTES, null, values);

    Uri newUri = ContentUris.withAppendedId(CONTENT_URI,regId);

    return newUri;
}
```

Por último, tan sólo nos queda implementar el método ***getType()***. Este método se utiliza para identificar el tipo de datos que devuelve el *content provider*. Este tipo de datos se expresará como un *MIME Type*, al igual que hacen los navegadores web para determinar el tipo de datos que están recibiendo tras una petición a un servidor. Identificar el tipo de datos que devuelve un *content provider* ayudará por ejemplo a Android a determinar qué aplicaciones son capaces de procesar dichos datos.

Existen dos tipos MIME distintos para cada entidad del *content provider*, uno de ellos destinado a cuando se devuelve una lista de registros como resultado, y otro para cuando se devuelve un registro único concreto. De esta forma, seguiremos los siguientes patrones para definir uno y otro tipo de datos:

- «**vnd.android.cursor.item**/vnd.xxxxxx» –> Registro único
- «**vnd.android.cursor.dir**/vnd.xxxxxx» –> Lista de registros

```
@Override
public String getType(Uri uri) {

    int match = uriMatcher.match(uri);

    switch (match){
        case CLIENTES:
            return "vnd.android.cursor.dir/vnd.iesbelen.cliente";

        case CLIENTES_ID:
            return "vnd.android.cursor.item/vnd.iesbelen.cliente";

        default:
            return null;
    }
}
```

Declaración del content provider en el Manifest

Por último, debemos declarar el *content provider* en el fichero *AndroidManifest.xml*, de forma que una vez instalada la aplicación en el dispositivo, Android conozca la existencia de dicho recurso.

Para ello, bastará insertar un nuevo elemento <*provider*> dentro de <*application*> indicando el nombre del *content provider* y su *authority*.

```
<application
          ...

    <provider
        android:authorities="org.iesbelen.android.contentproviders"
        android:name="ClientesProvider"/>

</application>
```

6.2. Esquema para la creación del proveedor de contenidos

1. Crear la fuente de datos (cliente SQLiteOpenHelper)
2. Preparar la Uri para el Content Provider
3. Crear el Content Provider
    a. Definición de constantes
        i. Definición de la Uri
        ii. Definición de un objeto UriMatcher
        iii. Definición de las constantes de columna
        iv. Definición de un objeto para guardar el nombre y la versión de la base de datos.
    b. Definición de los métodos
        i. onCreate()
        ii. query()
        iii. update()
        iv. delete()
        v. insert()
        vi. getType()
4. Definición del Content Provider en el *manifest*

6.3. Acceder a datos de otra aplicación (*Content Resolver*)

Este componente es el responsable de acceder a la información del *Content Provider*.

Para utilizar un *content provider*, debemos obtener una referencia a un *Content Resolver*, objeto a través del que realizaremos todas las acciones necesarias sobre el *content provider*. Para ello, usaremos el método *getContentResolver()* desde nuestra actividad para obtener la referencia indicada. Una vez obtenida la referencia al *content resolver*, podremos utilizar sus métodos *query(), update(), insert()* y *delete()* para realizar las acciones equivalentes sobre el *content provider*.

6.3.1. Realizar consultas

El proceso es muy similar a las consultas de bases de datos de SQLite. Habrá que definir un array con los nombres de las columnas de la tabla que queremos recuperar en los resultados de la consulta. Tras esto, obtendremos una referencia al *content resolver* y utilizaremos su método *query()* para obtener los resultados en forma de cursor.

```
// Columnas a recuperar
String[] columnas = new String[]{
        ClientesProvider.Clientes._ID,
        ClientesProvider.Clientes.COL_NOMBRE,
        ClientesProvider.Clientes.COL_TELEFONO,
        ClientesProvider.Clientes.COL_EMAIL};

Uri clientesUri = ClientesProvider.CONTENT_URI;

ContentResolver cr = getContentResolver();

// Hacemos la consulta
Cursor cur = cr.query(clientesUri,
        columnas, //Columnas a devolver)
        null,        // Condición de la query
        null,        // Argumentos variables de la query
        null);       // Orden de los resultados
```

Una vez obtenido el cursor, lo recorreremos para obtener los resultados.

```
if (cur.moveToFirst()){
    String nombre;
    String telefono;
    String email;

    int colNombre = cur.getColumnIndex(ClientesProvider.Clientes.COL_NOMBRE);
    int colTelefono = cur.getColumnIndex(ClientesProvider.Clientes.COL_TELEFONO);
    int colEmail = cur.getColumnIndex(ClientesProvider.Clientes.COL_EMAIL);

    txtResultados.setText("");

    do{
        nombre = cur.getString(colNombre);
        telefono = cur.getString(colTelefono);
        email = cur.getString(colEmail);

        txtResultados.append(nombre + " - " + telefono + " - " + email + "\n");

    }while (cur.moveToNext());
}
```

Este código dará error si el cursor está vacío. Para que no ocurra, antes de hacer la consulta habrá que preguntar si el cursor es distinto de null.

6.3.2. Insertar registros

Para insertar nuevos registros, el trabajo será también exactamente igual al que se hace al tratar directamente con bases de datos SQLite. Rellenaremos en primer lugar un objeto *ContentValues* con los datos del nuevo cliente y posteriormente utilizamos el método *insert()* pasándole la URI del *content provider* en primer lugar, y los datos del nuevo registro como segundo parámetro.

```
ContentValues values = new ContentValues();
values.put(ClientesProvider.Clientes.COL_NOMBRE,"Cliente nuevo");
values.put(ClientesProvider.Clientes.COL_TELEFONO,"999111222");
values.put(ClientesProvider.Clientes.COL_EMAIL,"nuevo@email.com");

ContentResolver cr = getContentResolver();

cr.insert(ClientesProvider.CONTENT_URI,values);
```

6.3.3. Borrar registros

Para borrar registros se utilizará directamente el método *delete()* del *Content Resolver*, al que habrá que indicar el criterio de localización del registro que se va a eliminar.

```
ContentResolver cr = getContentResolver();
cr.delete(ClientesProvider.CONTENT_URI, ClientesProvider.Clientes.COL_NOMBRE +
          "= 'Cliente nuevo'",null);
```

6.3.4. Acceso a Content Provider públicos

Este mismo mecanismo lo podemos utilizar para acceder a muchos datos de la propia plataforma Android. En la documentación oficial del paquete *android.provider* podemos consultar los datos que tenemos disponibles, entre ellos el historial de llamadas, la agenda de contactos y teléfonos, las bibliotecas multimedia (audio y video), o el historial y la lista de favoritos del navegador.

A modo de ejemplo, podemos mencionar la posibilidad de acceder a registro de llamadas situado en *android.provider.CallLog*. De él se pueden extraer diversos datos, como el tipo de llamada (*CallLog.Calls.TYPE*) y el número de dicha llamada (*CallLog.Calls.NUMBER*).

El proceso es idéntico al visto en puntos anteriores, obteniendo el URI, instanciando un *ContentResolver* y recuperando de él un cursor con unos criterios de búsqueda.

Mediante el tipo es posible conocer si se trata de una llamada recibida (Calls.INCOMING_TYPE), emitida (Calls.OUTGOING_TYPE) o perdida (Calls.MISSED_TYPE).

```
String[] columnas = new String[]{CallLog.Calls.TYPE, CallLog.Calls.NUMBER};
Uri llamadasUri = CallLog.Calls.CONTENT_URI;

ContentResolver cr = getContentResolver();
Cursor cur = cr.query(llamadasUri, columnas, null, null, null);

if (cur.moveToFirst()){
    int tipo;
    String tipoLlamada = "";
    String telefono;

    int colTipo = cur.getColumnIndex(CallLog.Calls.TYPE);
    int colTelefono = cur.getColumnIndex(CallLog.Calls.NUMBER);

    txtResultados.setText("");

    do{
        tipo = cur.getInt(colTipo);
        telefono = cur.getString(colTelefono);

        if (tipo == CallLog.Calls.INCOMING_TYPE)
            tipoLlamada = "ENTRADA";
        else if (tipo == CallLog.Calls.OUTGOING_TYPE)
            tipoLlamada = "SALIDA";
        else if (tipo == CallLog.Calls.MISSED_TYPE)
            tipoLlamada = "PERDIDA";

        txtResultados.append(tipoLlamada + " - " + telefono + "\n");
    }while (cur.moveToNext());
}
```

Para poder realizar estas acciones, es necesario autorizar la consulta en el AndroidManifext.xml:

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

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.READ_CONTACTS"></uses-
permission>
    <uses-permission android:name="android.permission.READ_CALL_LOG"></uses-
permission>

    <application
…
```

En el caso de **utilizar *Marshmallow* o versiones superiores**, gestionar los permisos en el propio dispositivo. En el siguiente enlace, se puede consultar cómo gestionar los permisos en tiempo de ejecución.

Grosso modo, y ajustándolo al ejemplo usado del registro de llamadas, seguiríamos los siguientes pasos:

Paso 1. Creación del MainActivity

Hacer que la clase *MainActivity* implemente la interfaz *ActivityCompat.OnRequestPermissionsResultCallback*, cuya tarea será solicitar en tiempo de ejecución los permisos necesarios para la lectura de los contactos y del número del dispositivo.

```
public class MainActivity extends AppCompatActivity implements
ActivityCompat.OnRequestPermissionsResultCallback{
```

Paso 2. Declaración de las constantes para los permisos

Se definirán constantes de tipo *int*, que permitirán definir identificadores únicos asociados a cada uno de los permisos que se solicitarán en tiempo de ejecución.

```
private static final int PERMISSION_REQUEST_CONTACTS= 0;
private static final int PERMISSION_REQUEST_CALL_LOG= 1;
```

Paso 3. Comprobar los permisos

Al pulsar el botón para obtener el listado de llamadas, se comprobarán los permisos disponibles para la tarea a realizar, solicitándolos en caso de que no estén asignados:

```
// Compruebo si los permisos para la llamada han sido concedidos
int permisoLecturaHistorial = ContextCompat.checkSelfPermission(MainActivity.this,
Manifest.permission.READ_CALL_LOG);
if (permisoLecturaHistorial!= PackageManager.PERMISSION_GRANTED) {
    // Pido permisos
    peticionPermisos(Manifest.permission.READ_CALL_LOG, new
String[]{Manifest.permission.READ_CALL_LOG},PERMISSION_REQUEST_CALL_LOG, "a los
contactos");
}
else{
    // Los permisos han sido concedidos, compruebo las llamadas
```

El método *peticionPermisos* que se ve en el código anterior, quedaría implementado de la siguiente manera:

```
private void peticionPermisos(String permiso, final String[] manifest, final int id,
String tipo){
    if (ActivityCompat.shouldShowRequestPermissionRationale(this, permiso)){
        Snackbar.make(vista, "Es ncesario el acceso " + tipo + " para su gestión de
la app", Snackbar.LENGTH_INDEFINITE).setAction("ACEPTAR", new View.OnClickListener()
{
            @Override
            public void onClick(View v) {
                ActivityCompat.requestPermissions(MainActivity.this, manifest,id);
            }
        }).show();
    }
    else{
        ActivityCompat.requestPermissions(MainActivity.this,manifest,id);
    }
}
```

Para la descripción de los permisos solicitados, se utiliza el tipo de notificación *Snackbar*, y que invocando al método *make*, permitirá mostrar un mensaje emergente desde la parte inferior de la pantalla, recibiendo como parámetros de entrada la vista donde se mostrará la notificación (la variable vista hace referencia al *id* del layout principal), el texto a mostrar, y la duración del mensaje (es posible utilizar la constante *Snackbar.LENGTH_INDEFINITE*, que mantendrá el mensaje en pantalla hasta que el usuario pulse el botón definido). Para este último caso, será necesario invocar al método *setAction*, donde se muestra el texto a pulsar por el usuario, y el evento *onClick* que contiene la acción a realizar (en este caso la solicitud de permisos en tiempo de ejecución).

Paso 4. Sobreescribir el método *onRequestPermissionResult*

A continuación, tendremos que sobreescribir el método *onRequestPermissionsResult*, que será llamado cuando el usuario acepte el permiso requerido. Dicho método será implementado por la interfaz *ActivityCompat.OnRequestPermissionsResultCallback*.

```
@Override
public void onRequestPermissionsResult(int requestCode, @NonNull String[]
permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    switch (requestCode){
        case PERMISSION_REQUEST_CALL_LOG:
            if (grantResults.length>0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED){
                Snackbar.make(vista, "Permiso de lectura de historial
establecido",Snackbar.LENGTH_LONG).show();

            }
            else {
                Snackbar.make(vista,"Permiso de lectura de contactos denegado",
Snackbar.LENGTH_LONG).show();
            }
            return;

    }
}
```

Paso 5. Declaración de permisos en el *Manifest*

Por último recordar que será necesario declarar dentro del AndroidManifest.xml los diferentes permisos necesarios.

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

El código completo nos quedaría de la siguiente manera:

```
public class MainActivity extends AppCompatActivity implements
ActivityCompat.OnRequestPermissionsResultCallback{

    private static final int PERMISSION_REQUEST_CONTACTS= 0;
    private static final int PERMISSION_REQUEST_CALL_LOG= 1;

    private Button btnLlamadas;
    private TextView txtResultados;
    private View vista;

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

        txtResultados = findViewById(R.id.TxtResultados);
        btnLlamadas = findViewById(R.id.BtnLlamadas);

        vista = findViewById(R.id.main_layout);


        //-----------------------------------------
        // ACCEDO A CONTENT PROVIDER PÚBLICO - LLAMADA
        btnLlamadas.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                realizarLlamada();
            }
        });
    }

    private void realizarLlamada(){

        // Compruebo si los permisos para la llamada han sido concedidos
        int permisoLecturaHistorial =
ContextCompat.checkSelfPermission(MainActivity.this, Manifest.permission.READ_CALL_LOG);
        if (permisoLecturaHistorial!= PackageManager.PERMISSION_GRANTED) {
            // Pido permisos
            peticionPermisos(Manifest.permission.READ_CALL_LOG, new
String[]{Manifest.permission.READ_CALL_LOG},PERMISSION_REQUEST_CALL_LOG, "a los
contactos");
        }
        else{
            // Los permisos han sido concedidos, compruebo las llamadas
            String[] columnas = new String[]{CallLog.Calls.TYPE, CallLog.Calls.NUMBER};
            Uri llamadasUri = CallLog.Calls.CONTENT_URI;

            ContentResolver cr = getContentResolver();
            Cursor cur = cr.query(llamadasUri, columnas, null, null, null);

            if (cur.moveToFirst()){
                int tipo;
                String tipoLlamada = "";
                String telefono;

                int colTipo = cur.getColumnIndex(CallLog.Calls.TYPE);
                int colTelefono = cur.getColumnIndex(CallLog.Calls.NUMBER);

                txtResultados.setText("");

                do{
                    tipo = cur.getInt(colTipo);
                    telefono = cur.getString(colTelefono);

                    if (tipo == CallLog.Calls.INCOMING_TYPE)
                        tipoLlamada = "ENTRADA";
                    else if (tipo == CallLog.Calls.OUTGOING_TYPE)
                        tipoLlamada = "SALIDA";
                    else if (tipo == CallLog.Calls.MISSED_TYPE)
                        tipoLlamada = "PERDIDA";

                    txtResultados.append(tipoLlamada + " - " + telefono + "\n");
                }while (cur.moveToNext());
            }
        }
    }
```

```
    private void peticionPermisos(String permiso, final String[] manifest, final int id,
String tipo){
        if (ActivityCompat.shouldShowRequestPermissionRationale(this, permiso)){
            Snackbar.make(vista, "Es ncesario el acceso " + tipo + " para su gestión de
la app", Snackbar.LENGTH_INDEFINITE).setAction("ACEPTAR", new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    ActivityCompat.requestPermissions(MainActivity.this, manifest,id);
                }
            }).show();
        }
        else{
            ActivityCompat.requestPermissions(MainActivity.this,manifest,id);
        }
    }



    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[]
permissions, @NonNull int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        switch (requestCode){
            case PERMISSION_REQUEST_CALL_LOG:
                if (grantResults.length>0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED){
                    Snackbar.make(vista, "Permiso de lectura de historial
establecido",Snackbar.LENGTH_LONG).show();

                }
                else {
                    Snackbar.make(vista,"Permiso de lectura de contactos denegado",
Snackbar.LENGTH_LONG).show();
                }
                return;

        }
    }
}
```

7. ALMACENAMIENTO EN LA RED

En este caso, el almacenamiento recae en la parte servidora, mostrándose en el dispositivo móvil aquella información que lee del servidor, por lo que cada acceso requiere una consulta, con alta latencia y sensibilidad a las desconexiones. Uniendo las técnicas vistas anteriormente, se puede lograr una persistencia mixta en la que se tiene copia local de parte de la información, realizando esporádicas comunicaciones de sincronización con el servidor.

- ✓ A fin de realizar operaciones de red en tu aplicación, el manifiesto debe incluir los siguientes permisos:

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

- ✓ Además, es aconsejable comprobar que tenemos conectividad disponible, para lo cual utilizamos *android.permission.ACCESS_NETWORK_STATE*, y mediante las clases **ConnectivityManager** y **NetworkInfo** saber si tenemos el servicio conectado (***isConnected()***) y disponible (***isAvailable()***)

7.1. Conexión HTTP

Este tipo de conexiones siguen el estándar *Hypertext Transfer Protocol* o HTTP. En la comunicación se realiza una petición de envío que suele ir acompañada de datos del cliente (sistema operativo, navegador, archivo solicitado). Una petición puede tener el objetivo de recuperar datos (GET) o de publicar datos (POST). Android nos ofrece, entre otras, dos posibilidades para interactuar con servidores HTTP/HTPPS:

- *Apache HttpClient*: fácil de usar, especialmente para servicios web REST, pero que a partir de Android 5.1 (Api 22) fue marcada como obsoleta.
- *Clase HttpUrlConnection*: basada en la API estándar de Java e incluida en el paquete java.net.*, permite desarrollar aplicaciones que contengan clientes HTTP ligeros. Con ella es posible enviar y recibir información a través de la Web.

Lo primero que vamos a hacer es configurar que se permitan accesos a la red desde el hilo principal:

```
StrictMode.setThreadPolicy(new
StrictMode.ThreadPolicy.Builder().permitNetwork().build());
```

A continuación, se deberá abrir la conexión hacia el recurso al que queremos acceder en el servidor. Para ello, tras instanciar la URL, se usa el método *openConnection()*. Al resultado obtenido se le debe hacer un casting a *HttpUrlConnection* para instanciar el cliente.

```
URL miUrl = new URL("https://www.google.com/humans.txt");
HttpURLConnection conexion = (HttpURLConnection) miUrl.openConnection();
```

7.1.1. Obtener información HTTP

Para recuperar datos de una URL se utiliza el método *getInputStream()* para obtener el flujo de datos asociado al recurso buscado, habiendo previamente establecido el tipo de solicitud que haremos mediante el método *setRequestMethod()*. Todo ello dentro de un bloque try/catch, y dependiendo de la naturaleza de la información descargada puede ser aconsejable hacerlo en un hilo secundario.

```
try{
    // Abrimos la conexión hacia el recurso al que queremos acceder en el
servidor
    URL miUrl = new URL("http://servidor.es/persistencia");
    HttpURLConnection conexion = (HttpURLConnection) miUrl.openConnection();

    conexion.setRequestMethod("GET");
    InputStream miEntrada = conexion.getInputStream();

    InputStreamReader miLector = new InputStreamReader(miEntrada);
    BufferedReader miBufferLector = new BufferedReader(miLector);
    String linea = miBufferLector.readLine();

    // Código a ejecutar

    conexion.disconnect();
```

Es importante que, al finalizar la transmisión, se libere la instancia y, con ello, la memoria a ella asociada, mediante el método *disconnect()*.

En algunos casos, dependiendo del tipo de servidor, se debe configurar la conexión mediante el método *setRequestProperty*, donde se puede definir el tipo de contenido, cliente, codificación…

También la clase *HttpURLConnection* nos permite establecer los tiempos de caducidad de conexión, en caso de que esta no se establezca, y tiempos de caducidad al finalizar la lectura. Asimismo, el método *getRespondeCode()* nos codifica la respuesta dada por el servidor.

```
conexion.setConnectTimeout(20000);
conexion.setReadTimeout(5000);
```

7.1.2. Enviar información HTTP

También debe realizarse en un bloque try/catch.

En este caso, tras abrir la conexión, se establece el método *setRequestMethod()* como POST, y se autoriza la salida de datos con el método setDoOutput.

```
conexion.setRequestMethod("POST");
conexion.setDoOutput(true);
```

El siguiente paso consistirá en enviar la información, para lo que se utiliza la clase abstracta *Outputstream*, superclase de todas las que representan flujo de salida. Posteriormente, se abre sobre ella un canal de salida en el que implementar un buffer de escritura.

```
OutputStream miSalida = conexion.getOutputStream();
OutputStreamWriter miEscritor = new OutputStreamWriter(miSalida);
BufferedWriter miBufferEscritor = new BufferedWriter(miEscritor);
miBufferEscritor.write("Texto a enviar");

miSalida.close();
```

O lo que es lo mismo:

```
BufferedWriter miBufferEScritor = new BufferedWriter(
                    new OutputStreamWriter(conexion.getOutputStream()));
miBufferEscritor.write("Texto a enviar");
miSalida.close();
```

O bien enviar directamente el flujo de bytes:

```
OutputStream miSalida2 =
                    new BufferedOutputStream(conexion.getOutputStream());
String texto = "Texto a enviar";
miSalida2.write(texto.getBytes());
miSalida2.flush();
miSalida2.close();
```

Si se espera respuesta, se deberá reabrir la entrada para obtener la respuesta del servidor, cerrando la conexión al finalizar con *disconnect();*.