

## Caso práctico

**María y Juan** ya tienen claro como diseñar las interfaces de usuario de una aplicación Android. Es hora de aprender a manejar la información que la aplicación puede generar o solicitar de otras aplicaciones.

- ✓ ¿Se pueden guardar ficheros en un dispositivo móvil?
- ✓ ¿Dónde y cómo se almacenarían?
- ✓ ¿La lista de contactos de nuestro dispositivo puede ser accesible desde nuestra aplicación?

En definitiva: ¿cómo se gestionan los datos en una aplicación Android? Éstas y muchas otras preguntas que ahora se plantean se irán dando respuesta conforme avancen en los contenidos.

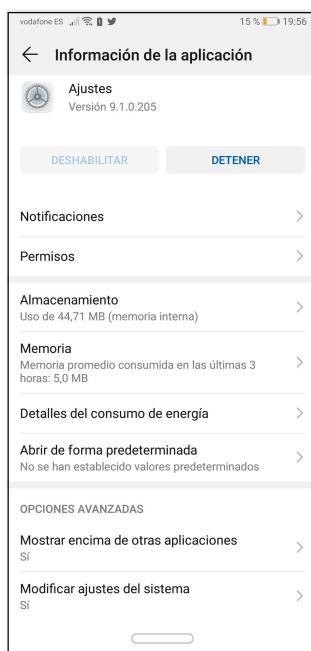


# 1.- Preferencias

## Caso práctico

Ada ha reunido a su equipo de desarrolladores Android para proponerles una tarea inicial. Tienen que consultar y utilizar varias aplicaciones Android ya existentes en el mercado y deben anotar todo aquello que haga referencia al tratamiento de los datos:

- ¿Qué tipo de información se solicita al usuario?
- ¿Debe el usuario introducir siempre la misma información en una aplicación o ésta recuerda el último estado de la misma?
- ¿Existen similitudes en la interfaz de usuario de aplicaciones distintas cuando el usuario configura sus preferencias?



Captura del IDE Android Studio. [Apache 2.0](#)

Las **pantallas de configuración** en las aplicaciones Android son muy útiles y prácticamente todas las aplicaciones las incluyen. También se conocen como **pantallas de preferencias**, porque el usuario las utiliza para introducir la información que desea a la hora de utilizar cada una de las aplicaciones.

Cuando el usuario modifica una preferencia, este cambio se guarda en un fichero en forma de par **clave-valor** hasta que se modifique de nuevo, de forma que siempre se guardan las preferencias del usuario.

Si eres usuario de Android, habrás comprobado cómo este tipo de pantallas es muy común en casi todas las aplicaciones. Aunque esta norma no es reconocida oficialmente, la opción de preferencias o **Settings** en Android Studio forma parte del menú principal y nunca es visible (`app:showAsAction="never"`) y normalmente es la última opción (`android:orderInCategory=100`).

En este apartado vamos a aprender a diseñar este tipo de interfaz gráfica para que posteriormente el usuario de la aplicación pueda configurar sus preferencias guardando y recuperando dicha información.

Para implementar las preferencias tendrás que comprobar en primer lugar qué librería de soporte estás utilizando. La forma que recomienda Google es utilizar la librería Preference de **AndroidX**. No obstante, se indicarán las diferencias con respecto a la librería anterior `android.preference` que en la versión Android 11 ha quedado obsoleta.

## Citas Para Pensar

“

*No nos hace falta valor para emprender ciertas cosas porque sean difíciles, sino que son difíciles porque nos falta valor para emprenderlas.*

Séneca

# 1.1.- Diseño de la pantalla de preferencias I

Como veíamos en la unidad de **Programación en Android: Interfaz de usuario**, el diseño de pantalla de preferencias se basa en ficheros XML. Android dispone de un mecanismo que hace muy fácil la gestión de preferencias de una aplicación para el desarrollador.

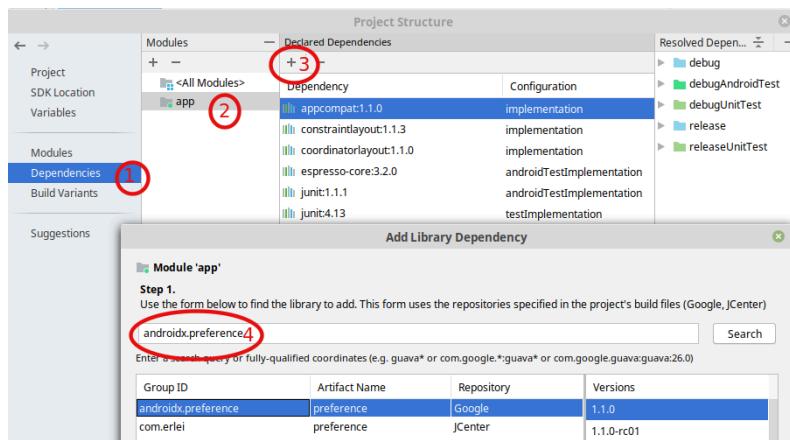
## Crear recurso XML de las preferencias

Para crear este nuevo fichero XML de preferencias donde poder definir la interfaz gráfica podemos seguir los siguientes pasos desde el IDE Android Studio:

- 1.- En primer lugar creamos una carpeta nueva llamada /xml/ dentro de la carpeta /res/. Para ello, pulsamos botón derecho sobre la carpeta /res/ y seleccionamos New - Android resource directory.
- 2.- En la pantalla que se abre seleccionamos el tipo de recurso XML.
- 3.- Sobre la carpeta que acabamos de crear pulsamos botón derecho y seleccionamos New - XML resource file.
- 4.- En la siguiente ventana del asistente ponemos un nombre al fichero (Ej: **settings.xml**) que automáticamente lo almacenará con el tipo XML.
- 5.- Pulsamos en el botón Finish y ya podemos comenzar con el diseño de la pantalla. Podemos asegurarnos viendo el código XML de la plantilla creada que tiene como layout principal PreferenceScreen y en vista diseño podremos incorporar los elementos de esta pantalla que necesitemos.

**Si nos sale el siguiente error "android.preference.PreferenceScreen is deprecated" es que tenemos que añadir al proyecto la librería específica AndroidX.**

Para ello vamos a **File -> Projects Structure**. Se selecciona el módulo **app**, y se da al icono ->**Library Dependency** y en el campo de búsqueda se pone androidx.preference.



Captura del IDE Android Studio. [Apache 2.0](#)

Recordemos que para diseñar una interfaz de usuario debemos tener en cuenta el contenedor donde se organizan los objetos o controles. En el caso de la pantalla de preferencias, este contenedor está formado por las etiquetas:

```
<PreferenceScreen> ..... </PreferenceScreen>
```

Además los objetos también pueden estar a su vez agrupados u organizados en categorías de forma que se muestra el título de categoría y añade una línea divisoria al final de la categoría separando visualmente los grupos de preferencias. En este caso se va a crear una categoría que muestre los "**Datos del usuario**" y dentro de ella agruparemos el resto de preferencias:

```
<PreferenceCategory android:title="@string/category_user_information">
..... <br /></PreferenceCategory>
```

## Tipos de preferencias

Para que el usuario de la aplicación pueda configurar sus preferencias utilizaremos una serie de controles u objetos entre los que destacamos los siguientes por ser los más utilizados:

- Preference: muestra un elemento de texto simple. Hay que definir al menos tres atributos:

- clave (key): esta clave se utiliza para hacer referencia al valor que el usuario introduce en el fichero de preferencias. Es la misma clave que se utiliza para recuperar el valor de nuestras preferencias.
- título (title): el título de la preferencia.
- resumen (summary): es una descripción que se muestra debajo del título con un tamaño de texto menor.
- ícono (icon): se puede utilizar un ícono que represente a la preferencia.

```
        android:key="preference_key"
        android:title="@string/preference_user"
        android:summary="@string/preference_user_summary"
        android:icon="@drawable/ic_action_user" >
```

- CheckBoxPreference: marca que el usuario puede activar o desactivar. El valor que devuelve esta preferencia es si está habilitado o deshabilitado. Esta opción suele estar en desuso, ya que se suele utilizar SwitchPreferenceCompat.
- SwitchPreferenceCompat: guarda dos estados al igual que CheckBoxPreference pero se muestra como un pequeño interruptor.

```
<SwitchPreference
    android:defaultValue="true"
    android:key="preference_saveuser_key"
    android:title="@string/preference_saveuser" />
```

- EditTextPreference: aparece un cuadro de diálogo donde el usuario puede introducir una cadena de texto sencilla. A veces es recomendable que el valor introducido por el usuario aparezca como valor resumen. Para habilitar este comportamiento se tiene que añadir el atributo app:useSimpleSummaryProvider="true" dentro de la preferencia en XML. En el caso de que no esté definido aparece **Not Set**.

```
<EditTextPreference
    android:dialogTitle="@string/dialog_title_name_avatar"
    android:key="preference_name_avatar_key"
    android:title="@string/preference_name_avatar"
    app:useSimpleSummaryProvider="true"
    android:icon="@drawable/ic_action_name_avatar"/>
```

- SeekBarPreference: esta preferencia muestra un SeekBar para introducir un valor entero. En este ejemplo se introduce 18 como valor inicial mediante android:defaultValue. Además se puede establecer un valor máximo con el atributo android:max

```
<SeekBarPreference
    android:key="preference_age_key"
    android:title="@string/preference_age"
    android:defaultValue="18"/>
```

## Autoevaluación

**El atributo android:key se utiliza en los controles de preferencias:**

MultiSelectListPreference

CheckBoxPreference

EditTextPreference

ListPreference

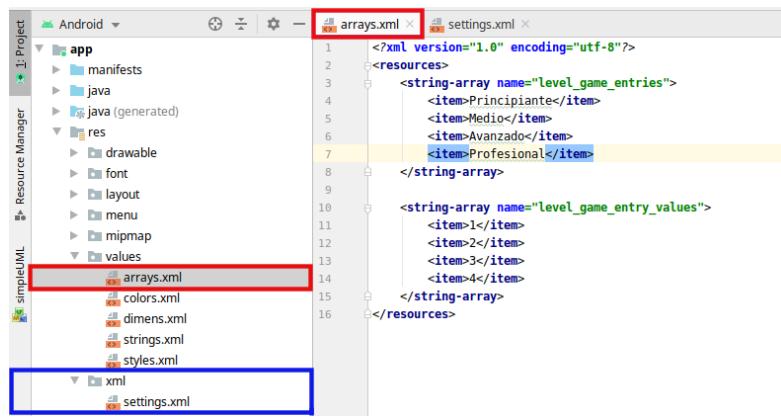
## Solución

1. Correcto
2. Correcto
3. Correcto
4. Correcto

## 1.2.- Diseño de la pantalla de preferencias II

En este segundo apartado vamos a implementar una segunda categoría llamada "**Opciones del juego**" donde veremos cómo implementar una preferencia que muestra una lista de opciones. Estas preferencias deben tener dos atributos android:entries que contiene el texto de los elementos de la lista y android:entryValues que define los valores asociados con cada elemento de la lista.

Los valores de ambos atributos se deben definir como <string-array>, y se crea un fichero XML llamado /res/values/arrays.xml para ello:



Captura del IDE, Android Studio. Apache 2.0

Una vez definido los atributos, veamos qué tipos de preferencias se pueden crear:

- **DropDownPreference**: muestra un despegable de opciones.
- **ListPreference**: lista de opciones donde el usuario sólo puede seleccionar una de ellas. Con esta preferencia si se puede usar app:useSimpleSummaryProvider para que en el resumen aparezca la opción que se ha seleccionado.

```
<ListPreference  
    android:defaultValue="1"  
    android:dialogTitle="@string/dialog_title_level"  
    android:entries="@array/level_game_entries"  
    android:entryValues="@array/level_game_entry_values"  
    android:key="preference_level"  
    android:title="@string/preference_level"  
    app:useSimpleSummaryProvider="true" />
```

- **MultiSelectListPreference**: lista de opciones donde el usuario puede seleccionar más de una. Con esta preferencia se guardará un array con todas las opciones. En este caso no se puede utilizar el atributo app:useSimpleSummaryProvider.

```
<MultiSelectListPreference  
    android:key="preference_options_key"  
    android:title="@string/preference_options"  
    android:summary="@string/preference_options_summary"  
    android:entries="@array/options_entries"  
    android:entryValues="@array/options_entry_values"  
    android:dialogTitle="@string/dialog_title_options"/>
```

### Autoevaluación

En una preferencia MultiSelectListPreference se puede usar el atributo app:useSimpleSummaryProvider de forma que se actualice el resumen de la preferencia al valor seleccionado por el usuario

- Verdadero  Falso

Falso

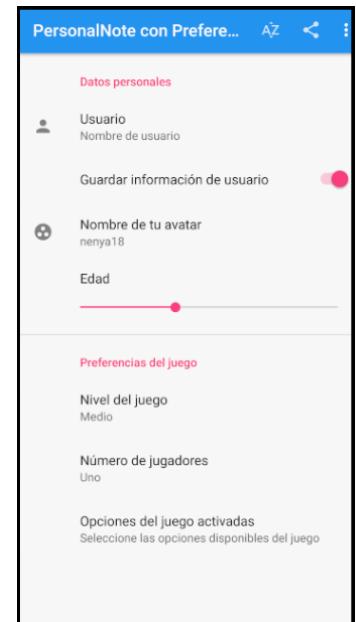
La preferencia MultiSelectListPreference no admite el atributo app:useSimpleSummaryProvider porque el usuario puede seleccionar más de un valor. Para actualizar el valor resumen se tendría que hacer mediante código.

# 1.3.- Mostrar la pantalla de preferencias

¿Recuerdas cómo se muestra la pantalla de una actividad? En efecto, utilizando código Java.

Las nuevas versiones de Android van cambiando la implementación de algunas funcionalidades que se quedan obsoletas, y aunque siguen funcionando, es preferible utilizar las nuevas técnicas si queremos desarrollar aplicaciones para las versiones más utilizadas de forma que nuestra aplicación funcione en la mayor parte de los dispositivos móviles de forma correcta. En este caso, desde la API 11 se ha implementado una nueva forma de mostrar las preferencias y aunque sigue funcionando la anterior, la nueva técnica se desarrolla utilizando **Fragments**. No obstante vamos a explicar las dos formas de mostrar la interfaz gráfica.

1.- Mediante la clase PreferenceActivity (para versiones anteriores a la API 11 aunque sigue funcionando en las versiones actuales). De la misma forma que para mostrar una pantalla o actividad necesitamos crear una clase Java que extiende de la clase Activity, para mostrar la interfaz gráfica de preferencias debemos crear una clase Java que en esta ocasión extenderá de la clase PreferenceActivity, la cual hereda de la superclase Activity. En esta clase se sobrescribe el método onCreate() de la misma forma que lo hacemos en una actividad, la única diferencia es que debemos llamar al método addPreferencesFromResource() para cargar el fichero XML que contiene el diseño de la pantalla de preferencias.



Captura del IDE Android Studio. [Apache 2.0](#)

No olvides registrar la actividad que muestra las preferencias en el fichero de manifiesto.

2.- Mediante la clase PreferenceFragment para versiones a partir de la API 11. Como ya se ha comentado, en esta ocasión se implementa utilizando fragmentos y por tanto debemos crear una nueva clase que extenderá de la clase PreferenceFragment. En esta clase es donde se sobrescribe el método onCreate() y se llama al método setPreferencesFromResource() para cargar el fichero XML que contiene el diseño de la pantalla de preferencias como veíamos en la técnica anterior. Tal y como se muestra en la siguiente imagen.

Ahora tan sólo nos queda crear la clase Java para mostrar el fragmento que acabamos de implementar, para ello esta clase sólo extenderá de la clase Activity y debemos utilizar FragmentManager para reemplazar el contenido de la pantalla (R.id.fragmentcontainer) por el de nuestro fragmento. Como puedes ver en el código cuando el usuario pulsa sobre la opción settings del menú de la barra de acción se muestra las preferencias de la aplicación.

```
3 import android.os.Bundle;
4 import android.view.View;
5
6 import androidx.preference.PreferenceFragmentCompat;
7
8 public class SettingsFragment extends PreferenceFragmentCompat {
9
10     public static final String TAG = "SettingsFragment";
11
12     public SettingsFragment() {
13         // Constructor Por Defecto
14     }
15
16     @Override
17     public void onCreatePreferences(Bundle savedInstanceState, String rootKey) {
18         setPreferencesFromResource(R.xml.settings,rootKey);
19     }
20
21 }
22
```

Captura del IDE Android Studio. [Apache 2.0](#)

Recuerda que R.id.fragmentcontainer es un FrameLayout que ocupa toda la pantalla y es donde se visualizan los fragmentos.

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        .....
        case R.id.action_settings:
            showSettingsFragment();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}

public void showSettingsFragment() {
    FragmentManager fm = getSupportFragmentManager();
    settingsFragment = (SettingsFragment) fm.findFragmentByTag(SettingsFragment.TAG);
    if (settingsFragment == null) {
        settingsFragment = new SettingsFragment();
    }
    FragmentTransaction ft = fm.beginTransaction();
    ft.replace(R.id.fragmentcontainer, settingsFragment, SettingsFragment.TAG);
    ft.addToBackStack(null);
    ft.commit();
}
```

## Reflexiona

Y ahora bien, ¿desde dónde puede acceder el usuario a esta pantalla en nuestra aplicación?

[Mostrar retroalimentación](#)

Si sueles utilizar aplicaciones Android, sabrás que se suele acceder desde el botón que abre el menú de la aplicación o desde el ActionBar.

## Debes conocer

En este enlace encontrarás un proyecto base de ejemplo de preferencias implementado mediante la clase PreferenceActivity.

[Código del proyecto Ejemplo Preferences](#) [1.3 MB]

En este otro enlace encontrarás el proyecto GamePreferences donde se ha implementado las preferencias utilizando PreferenceFragment.

[Código del proyecto GamePreferences](#) (zip - 15.6 MB).

## Autoevaluación

**Siempre es necesario registrar en el archivo de manifiesto la clase de la actividad que lanza la pantalla de preferencias.**

- Verdadero  Falso

**Verdadero**

Es cierto porque toda actividad necesita registrarse previamente en el fichero de manifiesto de la aplicación.

## 1.4.- Recuperar las preferencias almacenadas

Como acabamos de ver el usuario selecciona e introduce una serie de datos en la pantalla de preferencias que hemos diseñado y mostrado en el dispositivo móvil para configurar la aplicación. Pero, ¿dónde se almacenan los valores de estas opciones? ¿Cómo se pueden recuperar? ¿Se eliminan cuando salimos de la aplicación o apagamos nuestro dispositivo móvil?

De nada serviría una pantalla de preferencias donde el usuario ha elegido una configuración determinada si después no se almacena y no se pueden recuperar los datos. En realidad, este tipo de gestión de datos es transparente incluso para el programador. Android almacena internamente estas preferencias en forma de clave-valor, es decir, cada una de ellas está compuesta por una clave o identificador único y un valor asociado a ese identificador. Como desarrollador nos interesa conocer cómo manejar esas preferencias compartidas en la aplicación. Por tanto debemos utilizar la clase `androidx.preference.PreferenceManager` y `android.content.SharedPreferences`.

Primero obtenemos una instancia de la clase `SharedPreferences` mediante la clase `PreferenceManager`:

```
SharedPreferences <span>sharedPreferences</span> =  
    PreferenceManager.getDefaultSharedPreferences(activity_context)
```



Megan Rexzin (Licencia Pixabay)

Después recuperamos los valores de preferencias dependiendo del tipo de dato almacenado en cada una de ellas. Así podremos utilizar los siguientes métodos:

```
<span>sharedPreferences</span>.getString(key, default_value)  
<span>sharedPreferences</span>.getInt(key, default_value)  
sharedPreferences.getFloat(key, default_value)  
sharedPreferences.getBoolean(key, default_value)  
...  
...
```

Si queremos recuperar el valor de un `SwitchPreference` utilizaremos `getBoolean(key,default_value)` donde `key` es la clave de la preferencia especificada mediante el atributo `android:key` y el parámetro `default_value` es el valor por defecto en el caso que el usuario no lo haya especificado.

```
Boolean saveuser = sharedPreferences.getBoolean("preference_saveuser_key", true);
```

Evidentemente Android hace todo esto por nosotros de forma automática y transparente, puesto que al salir de la pantalla de preferencias e incluso cerrar la aplicación posteriormente cuando volvemos a abrir esta pantalla tendremos los valores que se introdujeron por última vez.

### Reflexiona

Entonces, ¿para qué puede servirnos recuperar estos datos si ya lo hace Android por nosotros?

[Mostrar retroalimentación](#)

En este caso, cuando entramos de nuevo en la pantalla de preferencias, el sistema inicializa todos los controles con sus valores, pero puede ser que algunos de estos valores los necesitemos para implementar otras partes de nuestra aplicación. Si no, ¿de qué serviría que hayamos elegido un nivel "Profesional" en el control `ListPreference` de la pantalla de preferencias, si después cuando iniciemos el juego no se actualizara el nivel?

# Ejercicio Resuelto

Siguiendo el ejemplo de nuestra pantalla de preferencias en cualquiera de las versiones, debes crear una opción en el menú principal de la actividad llamado “**Ver Preferencias**” que nos muestre los datos almacenados en cada uno de los controles de preferencia.

Mostrar retroalimentación

1.- Importa uno de los proyectos de ejemplo de preferencias del apartado “**Mostrar la pantalla de preferencias**”.

2.- Crea el fichero XML activity\_main.xml en el directorio /res/menu/ y añade las opciones del menú.

3.- En el fichero MainActivity.java implementa el método onOptionsItemSelected(MenuItem item) y cuando se haya seleccionado la opción action\_show\_settings se ejecutará un método que muestra los valores almacenados:

```
public boolean onOptionsItemSelected(MenuItem item) {  
    switch (item.getItemId()) {  
        .....  
        case R.id.action_show_settings:  
            showPreferences();  
            return true;  
        case R.id.action_settings:  
            showSettingsFragment();  
            return true;  
        default:  
            return super.onOptionsItemSelected(item);  
    }  
}  
  
private void showPreferences() {  
    SharedPreferences sharedpreferences =  
        PreferenceManager.getDefaultSharedPreferences(this);  
    String user = sharedpreferences.getString("preference_user_key","No declarado");  
    Boolean saveuser=sharedpreferences.getBoolean("preference_saveuser_key",true);  
    String nameavatar =  
        sharedpreferences.getString("preference_name_avatar_key","No declarado");  
    int age = sharedpreferences.getInt("preference_age_key",18);  
    String level = sharedpreferences.getString("preference_level","1");  
    Set<String> n = new HashSet<>(); n.add("sonido");  
    Set<String> options=  
        sharedpreferences.getStringSet("preference_options_key",n);  
    String message = new String("Nombre de usuario: "+user+"\n");  
    message.append("Salvar usuario: "+saveuser+"\n");  
    message.append("Nombre avatar: "+nameavatar+"\n");  
    message.append("Edad: "+age+"\n");  
    message.append("Nivel: "+level+"\n");  
    message.append("Opciones: "+options.toString());  
    Toast.makeText(this,message,Toast.LENGTH_SHORT).show();  
}
```

4.- Sólo queda desplegar la aplicación en un emulador y ver los resultados al pulsar sobre la opción del menú que ya existía para abrir la pantalla de preferencias y por supuesto probar la opción que muestre los valores de preferencia almacenados.

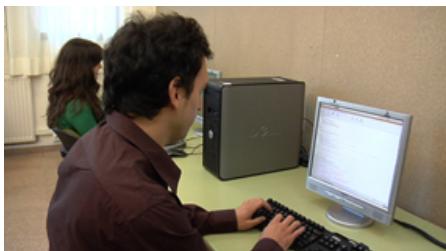
## Para saber más

En este apartado se ha explicado la utilidad de la clase SharedPreferences, unida a la clase Preferences que muestran los valores clave-valor mediante controles.

Esta clase también tiene otra utilidad que es guardar valores en un fichero de preferencias, aunque **NO tenga interfaz gráfica** de preferencias. Este es el supuesto de querer guardar cierta información de la aplicación que siempre esté accesible.



### Caso práctico



Trabajando con las preferencias **Juan** ha comprobado cómo se pueden almacenar ficheros XML de forma casi transparente para el desarrollador para después recuperar esa información. Ahora bien, ¿se puede guardar otro tipo de ficheros? ¿Dónde se pueden almacenar? ¿El proceso es tan sencillo como los ficheros de preferencias?

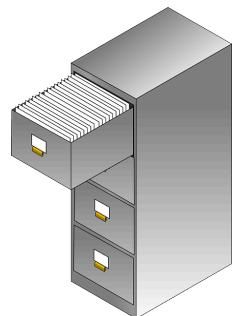
**Juan** recuerda que el núcleo de Android está basado en Linux y además se utiliza Java como lenguaje de programación, por tanto intuye que no será muy diferente a lo que estudió en el módulo de **Programación** en el ciclo formativo **DAM** sobre la gestión de ficheros.

La gestión de ficheros en Android está basada en el paquete `java.io`. Si ya has trabajado con Java no te resultará nada difícil manipular ficheros en Android, pero si no es el caso, no te preocupes, con un poco más de esfuerzo llegarás a comprender los métodos necesarios para trabajar con ficheros tanto de lectura como de escritura.

Debido a las particularidades de los dispositivos móviles en cuanto al almacenamiento se refiere, debemos tener en cuenta el lugar dónde se ubicarán los ficheros. Estos dispositivos cuentan con una capacidad de almacenamiento limitada, por tanto a la hora de guardar los ficheros diferenciaremos tres lugares:

1. En la **memoria interna** del dispositivo. Diferenciamos entre la propia memoria interna y la carpeta de recursos de la aplicación.
2. En la **memoria externa** del dispositivo (tarjeta de memoria), si existe.

Veamos cómo se gestionan los ficheros dependiendo del lugar donde se almacenan.

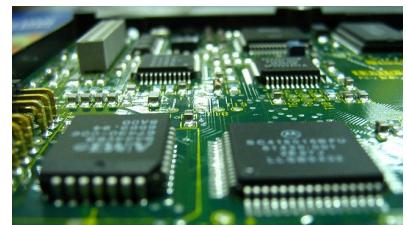


Ciker-Free-Vector-Images  
(Licencia Pixabay)

## 2.1.- Almacenamiento en memoria interna

Cuando trabajamos con ficheros en la memoria interna del dispositivo debemos tener en cuenta que ésta es muy limitada y no debemos sobrecargarla. Cuando se instala una aplicación en el dispositivo móvil, el sistema reserva un espacio para que se almacenen los ficheros de datos necesarios. Este espacio no hay que confundirlo con la estructura de carpetas del proyecto de la aplicación. La ruta de la memoria interna de una aplicación es la siguiente:

/data/data/paquete.aplicación/files/



CristianIS (Licencia Pixabay)

Algunos de los métodos que utilizaremos para manipular los ficheros son:

- openFileOutput(): abrir un fichero en modo escritura.
- openFileInput(): abrir un fichero en modo lectura.
- getFilesDir(): recuperar la ruta absoluta del directorio donde se encuentran los ficheros internos.
- getDir(): crear o abrir un directorio interno de la aplicación.
- deleteFile(): eliminar un fichero.
- fileList(): devuelve un array con todos los ficheros internos de la aplicación.
- openRawResource(): leer los ficheros de recursos almacenados en la carpeta /res/raw/.

Aunque los ficheros que se almacenan en la memoria interna son de carácter privado, el modo de acceso a estos ficheros puede variar utilizando las siguientes constantes asociadas al contexto:

- MODE\_PRIVATE: otras aplicaciones no pueden acceder al fichero.
- MODE\_WORLD\_READABLE: otras aplicaciones pueden leer el fichero.
- MODE\_WORLD\_WRITABLE: otras aplicaciones pueden escribir sobre el fichero.

En la versión API 17 se ha declarado como obsoleto el uso de los dos últimos modos debido al riesgo de que una aplicación pueda manipular ficheros internos de otra. Por tanto, lo más aconsejable es utilizar el modo privado. Veamos el código necesario para guardar una cadena de texto en un fichero de la memoria interna:

```
//Abrimos el fichero en modo privado para escribir.  
OutputStreamWriter fout= new OutputStreamWriter(  
    openFileOutput("fichero.txt", Context.MODE_PRIVATE));  
//Escribimos en el fichero  
fout.write("Esto es un texto de prueba");  
//Cerramos el fichero  
fout.close();
```

Y si queremos leer el fichero de texto anterior podríamos escribir algo así:

```
//Abrimos el fichero para leer.  
BufferedReader fin =new BufferedReader(new InputStreamReader(  
    openFileInput("fichero.txt")));  
// Leemos el fichero.  
String texto = fin.readLine();  
//Cerramos el fichero  
fin.close();
```

¿Cómo podemos consultar la ubicación del fichero almacenado? Android Studio incorpora un explorador de archivos que permite visualizar el sistema de ficheros del dispositivo. Puedes abrirlo desde Tools - Android - Android Device Monitor. En la ventana que se abre podemos ver la pestaña File Explorer.

### Para saber más

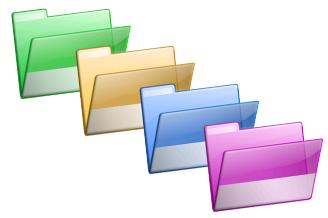
Si necesitas recordar los conceptos sobre la gestión de ficheros en Java puedes consultar el siguiente enlace donde se ven algunos ejemplos.

[Lectura y escritura de ficheros en Java](#)

## 2.1.1.- Almacenamiento en los recursos de la aplicación

Creamos en esta carpeta un fichero que deseamos consultar desde la propia aplicación y para leerlo podemos hacerlo así:

```
//Abrimos el fichero de recursos de la carpeta raw en modo lectura.  
InputStream fraw = getResources().openRawResource(R.raw.fich_raw);  
//Leemos el fichero.  
BufferedReader brin = new BufferedReader(new InputStreamReader(fraw));  
//Guardamos el contenido en la variable texto.  
String texto = brin.readLine();  
//Cerramos el fichero.  
fraw.close();
```



[OpenClipart-Vectors \(Licencia Pixabay\)](#)

En el ejemplo anterior vemos la utilización del método `openRawResource()` que abre el fichero de recurso cuyo identificador le pasamos como parámetro. El resto del código es similar a leer un fichero de texto. Obviamente en la carpeta de recursos `/res/raw/` podremos tener almacenados cualquier tipo de fichero binario (imagen, vídeo, etc.) que también podremos leer tal y como se hace en Java.

### Recomendación

En el siguiente enlace puedes descargar un proyecto de ejemplo donde se puede ver el funcionamiento de los ficheros en la memoria interna. Consta de un objeto `EditText` donde se puede introducir el texto a guardar y tres objetos `Button` (uno para escribir o guardar el fichero, otro para leerlo y por último un botón para leer un fichero de texto almacenado en la carpeta de recursos `/res/raw/` de la propia aplicación). Así mismo podrás estudiar su código para aprender más sobre la gestión de ficheros.

[Aplicación de ejemplo sobre los ficheros internos en Android](#) (zip - 13.63 MB)

### Autoevaluación

Si el nombre del paquete de mi aplicación es `com.example.ejemploficheros`, la ruta donde se almacenan los ficheros internos de la aplicación será:

`/data/data/com.example.ejemploficheros/files/`.

- Verdadero  Falso

**Verdadero**

La afirmación es verdadera, ese es el espacio que Android reserva para los datos internos de tu aplicación.

Los ficheros incluidos en la carpeta `/res/raw/` se pueden leer y escribir utilizando los permisos apropiados.

- Verdadero  Falso

**Falso**

La afirmación es falsa porque en esta carpeta sólo tendremos acceso de lectura, no de escritura.

## 2.2.- Almacenamiento en memoria externa

El almacenamiento de ficheros en memoria externa suele hacerse en las tarjetas de memoria SD extraíbles. Estas tarjetas suelen tener mayores capacidades y los ficheros que almacenamos en ellas pueden corresponder a archivos privados de la aplicación que se instala en la tarjeta o bien ficheros públicos compartidos por varias aplicaciones. En cualquier caso, para trabajar con los ficheros almacenados en este tipo de memoria, debemos tener en cuenta varios aspectos:

- **Seguridad y permiso del usuario:** los ficheros pueden ser públicos y por tanto accesibles a varias aplicaciones. Es necesario solicitar el permiso correspondiente al usuario para tener acceso a la tarjeta de memoria. En el archivo de manifiesto será necesario incluir lo siguiente:

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

```
    android:minSdkVersion="8"
    android:targetSdkVersion="17" />

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

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

- **Estado de la tarjeta de memoria:** la tarjeta SD puede no estar presente en el dispositivo o aunque esté insertada no estar debidamente montada. Para ello será necesario antes de comenzar a tratar los ficheros, recuperar su estado de la siguiente forma:

```
String estado = Environment.getExternalStorageState()
```

- Android devolverá una constante dependiendo del estado en el que se encuentre la tarjeta SD. Aquí tienes algunos ejemplos:
  - MEDIA\_MOUNTED: la tarjeta está montada y disponible para lectura y escritura.
  - MEDIA\_MOUNTED\_READ\_ONLY: la tarjeta está montada en modo lectura.
  - MEDIA\_UNMOUNTED: la tarjeta está insertada pero no se encuentra montada.
  - MEDIA\_REMOVED: la tarjeta no está insertada.
- **Acceso al directorio de la tarjeta de memoria:** utilizamos uno de los siguientes métodos:
  - getExternalFilesDir() devuelve la ruta de una carpeta determinada de nuestra aplicación. Podemos pasar como parámetro null y nos recuperará la carpeta raíz de la aplicación en el directorio:

```
/mnt/sdcard/Android/data/paquete.aplicación/files/
```

- getExternalStoragePublicDirectory() devuelve la ruta de una carpeta pública.

En Android podemos indicar alguno de los subdirectorios definidos por alguna de las siguientes constantes para clasificar los contenidos, como por ejemplo se hace en la galería multimedia.

### Subcarpetas multimedia

Constante	¿Qué almacena?	¿Dónde se almacena?
DIRECTORY_PICTURES	Ficheros de imágenes, fotos	Pictures/
DIRECTORY_MOVIES	Ficheros de películas.	Movies/
DIRECTORY_MUSIC	Ficheros de música.	Music/
DIRECTORY_RINGTONES	Ficheros de tonos.	Ringtones/

- **Configurar la memoria externa en el emulador:** para probar el correcto funcionamiento del almacenamiento de ficheros en memoria externa debemos configurar en un emulador (AVD) la opción SD Card Size con un tamaño de memoria adecuado. Para visualizar los ficheros utilizaremos la misma opción que en los ficheros internos (File Explorer).



## 2.2.1.- Escribiendo en memoria externa

Para escribir un fichero en una tarjeta de memoria externa tendremos en cuenta los aspectos anteriormente descritos. Así, después de especificar el permiso correspondiente en el fichero de manifiesto, podríamos utilizar el siguiente código para escribir un fichero de texto dentro del espacio reservado a nuestra aplicación en la tarjeta SD:

```
String estado = Environment.getExternalStorageState();
if (estado.equals(Environment.MEDIA_MOUNTED)){
    File file = new File(getExternalFilesDir(null), "fichero.txt");
    OutputStreamWriter fout =
        new OutputStreamWriter(
            new FileOutputStream(file));
    fout.write("Texto de prueba almacenado en tarjeta SD.");
    fout.close();
}
```



Ciker-Free-Vector-Images (Licencia Pixabay)

Recordemos que primero debemos recoger el estado en el que se encuentra la tarjeta, y si ésta se encuentra debidamente montada en el dispositivo, entonces podremos acceder a ella para indicar que en la ruta raíz del espacio reservado para la aplicación dentro de la tarjeta de memoria crearemos un fichero llamado "fichero.txt". Escribimos el texto deseado y cerramos el fichero.

Si deseamos escribir un fichero en uno de los subdirectorios que la tarjeta SD tiene establecido podríamos realizar la siguiente modificación al código anterior:

```
File file = new File(
    Environment.getExternalStoragePublicDirectory(
        Environment.DIRECTORY_PICTURES), "fotografia.jpg");
```

Así obtenemos acceso al directorio público /Pictures/ para almacenar posteriormente el fichero que le pasamos. No debemos olvidar que estos directorios pueden ser accesibles desde cualquier aplicación, por tanto es conveniente saber qué tipo de fichero se guarda en cada zona.

## 2.2.2.- Leyendo de la memoria externa

Para leer un fichero que está almacenado en la memoria externa del dispositivo móvil tendremos en cuenta los mismo aspectos que se ha visto para la escritura.

Tanto si queremos leer un fichero que está en el directorio de la aplicación como si éste es público en alguno de los subdirectorios de Android, sólo cambiará el método para recuperar la ruta del fichero.

Como ejemplo, veamos cómo podemos leer un fichero de texto situado en la zona reservada para la aplicación:

```
String estado = Environment.getExternalStorageState();
if (estado.equals(Environment.MEDIA_MOUNTED)){
    File file = new File(getExternalFilesDir(null), "fichero.txt");
    BufferedReader br = new BufferedReader(
        new InputStreamReader(
            new FileInputStream(file)));
    String texto = br.readLine()
    br.close();
}
```

 Ilustración de una lupa

[TheUjulala \(Licencia Pixabay\)](#)

### Recomendación

En el siguiente enlace puedes descargar un proyecto de ejemplo donde se puede ver el funcionamiento de los ficheros en la memoria externa. Consta de un objeto EditText donde se puede introducir el texto a guardar y tres objetos Button (dos botones para escribir y leer el fichero dentro de la tarjeta de memoria pero en el espacio reservado para la aplicación y por último un botón para escribir el fichero en la subcarpeta pública DOWNLOAD). Así mismo podrás estudiar su código para aprender más sobre la gestión de ficheros.

[Aplicación de ejemplo sobre los ficheros externos en Android.](#) (zip - 13.62 MB)

1	2013-09-23 15:51	download	-rwxrwxr-x
2	2013-09-23 15:51	dcim	-rwxrwxr-x
3	2013-09-23 15:51	Download	-rwxrwxr-x
4	2013-09-23 16:01	fichero.txt	----rwxrwx
5	2013-09-23 15:58	DCIM	d---rwxrwx
6	2013-07-02 15:58	Downloads	d---rwxrwx
7	2013-09-23 15:58	Alarms	d---rwxrwx
8	2013-07-02 15:58	Android	d---rwxrwx
9	2013-09-23 15:58	data	d---rwxrwx
10	2013-09-23 15:58	com.example.ejemploficherosexternos	d---rwxrwx
11	2013-09-23 15:58	files	d---rwxrwx
12	2013-09-23 15:58	fichero.txt	----rwxrwx
13	2013-09-23 15:58	LOST.DR	d---rwxrwx
14	2013-07-02 15:58	Movies	d---rwxrwx
15	2013-07-02 15:58	Music	d---rwxrwx
16	2013-07-02 15:58	Notifications	d---rwxrwx
17	2013-07-02 15:58	Pictures	d---rwxrwx
18	2013-07-02 15:58	Podcasts	d---rwxrwx
19	2013-07-02 15:58	RingTones	d---rwxrwx

Captura tomada del IDE Eclipse cuya licencia es Eclipse Public License

### Autoevaluación

Es necesario solicitar permiso al usuario en el archivo de manifiesto para gestionar ficheros en:

- La memoria interna.
- La memoria externa.
- La carpeta de recursos /res/raw/
- Todas las respuestas son correctas.

¡Casi! Repasa mejor los contenidos.

¡Exacto!

No es correcto.

Sólo es necesario cuando queremos gestionar ficheros de la memoria externa (tarjeta SD).

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

**Para obtener la ruta de una carpeta determinada de nuestra aplicación situada en la memoria externa utilizaremos el siguiente método:**

- `getExternalStorageState()`
- `getExternalFilesDirectory()`
- `getExternalFilesDir()`
- `getExternalStoragePublicDirectory()`

No es así.

¡Casi, casi! Fíjate bien en el método.

¡Muy bien! En efecto, ese es el método.

No es correcto, repasa los contenidos.

## Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

### 3.- Base de Datos

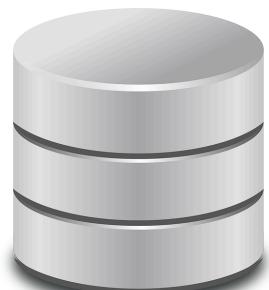
#### Caso práctico



**María** es experta en base de datos, y desde que cursó el módulo “Base de Datos” en el ciclo de grado superior de DAM ha seguido estudiando todo tipo de sistemas gestores de datos. Cuando comenzaron su andadura en Android, estudiando la arquitectura de la plataforma se percató de que se utilizaba un motor de base de datos específico llamado SQLite, el cual desconocía. Ahora ha llegado el momento de aprender un poco más sobre Android: ¿cómo gestiona Android una base de datos? ¿Se utilizará también el lenguaje SQL?

Las bases de datos han aportado enormes ventajas al tratamiento de la información con respecto al sistema de ficheros tradicional, pero sus características particulares pueden hacernos pensar que no pueden utilizarse en dispositivos con ciertas limitaciones como los móviles. ¿Existe algún motor de base de datos adaptado a este tipo de dispositivos? Android incorpora la librería SQLite que tiene las siguientes características:

- Necesita un tamaño reducido.
- Se basa en el modelo relacional.
- Utiliza el lenguaje SQL.
- Soporta transacciones.
- Requiere pocos recursos del sistema.
- Es de código abierto.



[OpenClipart-Vectors](#) ([Licencia Pixabay](#))

Estas características convierten a SQLite en la herramienta más apropiada para gestionar los datos en los dispositivos móviles. Vamos a ver cómo podemos crear una base de datos sencilla con alguna tabla y gestionar los datos en ella desde una aplicación Android.

#### Para saber más

Si deseas conocer un poco más SQLite te invitamos a que visites su página oficial.

[Página oficial de SQLite.](#)

# 3.1.- Estructura de SQLite

¿Cómo se integra SQLite con Android?

En la API de Android se encuentran las clases relacionadas con la gestión de BD. Aquí nos centraremos únicamente en las tres clases más utilizadas que nos permita desarrollar un ejemplo sencillo utilizando base de datos:



[Richard Hipp/Mike Toews](#) (Dominio público)

- **SQLiteOpenHelper:** gestiona la creación y actualización de las bases de datos. Entre los métodos más utilizados de esta clase destacan:
  - `onCreate()`: se lanza para crear la base de datos por primera vez.
  - `onUpgrade()`: se lanza para actualizar la estructura de la base de datos.
  - `getReadableDatabase()`: abrir una base de datos en modo lectura.
  - `getWritableDatabase()`: abrir una base de datos en modo lectura y escritura
  - `getDatabaseName()`: recupera el nombre de la base de datos
  - `close()`: cerrar la base de datos.
- **SQLiteDatabase:** incluye los métodos para gestionar una base de datos. Entre los métodos más utilizados de esta clase destacan:
  - `insert()`: insertar un nuevo registro.
  - `update()`: modificar un registro.
  - `delete()`: eliminar un registro.
  - `execSQL()`: ejecuta una sentencia SQL que no devuelve ningún registro.
  - `rawQuery()`: lanza una sentencia de consulta utilizando un comando completo SQL y se recupera el resultado en un cursor.
  - `query()`: lanza una sentencia de consulta mediante varios parámetros y se recupera el resultado en un cursor.
  - `isOpen()`: comprobar si está abierta la base de datos.
- **SQLiteDatabase:** gestiona el cursor para recorrer las filas recuperadas mediante una sentencia SQL. Entre los métodos más utilizados de esta clase destacan:
  - `moveToFirst()`: posiciona el cursor en el primer registro.
  - `moveToNext()`: posiciona el cursor en el siguiente registro.
  - `getString()`: recupera en un String el valor del campo.
  - `getInt()`: recupera en un entero el valor del campo
  - `getFloat()`: recupera en un Float el valor del campo.
  - `getLong()`: recupera en un Long el valor del campo.
  - `getColumnCount()`: recupera el número de columnas.

## Para saber más

En la documentación oficial de Android puedes encontrar la API con todas las clases que incorpora SQLite y que se incluyen en el paquete `android.database.sqlite`.

[SQLite en Android](#)

## Autoevaluación

La clase de SQLite que contiene los métodos para insertar, actualizar, eliminar y consultar los registros de una base de datos es:

- `SQLiteDatabase`.
- `SQLiteOpenHelper`.
- `SQLiteCursor`.
- `SQLiteExecSQL`.

No es la opción correcta. Repasa los contenidos.

¡Muy bien!

Ésta no es la clase.

Fallaste, fíjate bien.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## 3.2.- Creación de una base de datos

¿Cómo empezamos a trabajar con SQLite?

Si nos fijamos en las tres clases anteriores, la primera clase que tenemos que tener en cuenta es SQLiteOpenHelper. En ella encontramos entre otros, los métodos de crear por primera vez una base de datos o actualizar la estructura de la misma. Por tanto vamos a crear una clase que extiende de ella de la siguiente forma:

```
public class MiBaseDatosHelper extends SQLiteOpenHelper
```



[OpenClipart-Vectors \(Licencia Pixabay\)](#)

Esta clase contendrá un constructor que no necesitamos sobrescribir y al que se le pasa un objeto Context, el nombre de la base de datos, un objeto CursorFactory y la versión de la base de datos.

```
public MiBaseDatosHelper(Context contexto, String nombre,<br />                                CursorFactory factory, int version) {  
    super(contexto, nombre, factory, version);  
}
```

El motivo de extender la clase SQLiteOpenHelper nos obliga a sobrescribir los métodos onCreate() y onUpgrade(). El método onCreate() se lanza la primera vez que se ejecute la aplicación para crear la base de datos. Le pasamos un objeto de tipo SQLiteDatabase para ejecutar la sentencia SQL que nos crea la tabla. Recuerda que el método execSQL() para ejecutar sentencias SQL está en la clase SQLiteDatabase.

```
@Override  
public void onCreate(SQLiteDatabase db) {  
    db.execSQL("CREATE TABLE miTabla (campo1 INTEGER, campo2 TEXT)");  
}
```

Como se ha comentado antes, también será obligatorio sobrescribir el método onUpgrade(). Este método se utiliza para actualizar la base de datos. Supongamos que deseamos modificar la estructura de la tabla e incluir un campo nuevo: este método es el que realiza esa actualización de forma automática. Para ello le pasamos como parámetros además de un objeto SQLiteDatabase, el número de versión actual y el número de la nueva versión de la BD. La implementación de este método dependerá de los cambios que se hayan producido en la propia estructura de la BD, pero de momento para simplificar el ejemplo de este método lo único que hacemos es borrar la tabla y crearla de nuevo, aunque en un caso real habría que migrar todos los datos a la nueva estructura de la tabla que vayamos a crear entre algunas de las acciones a realizar.

```
@Override  
public void onUpgrade(SQLiteDatabase db, int versionAnterior, int versionNueva) {  
    //Se elimina la versión anterior de la tabla  
    db.execSQL("DROP TABLE IF EXISTS miTabla");  
    //Se crea la nueva versión de la tabla  
    db.execSQL("CREATE TABLE miTabla (campo1 INTEGER, campo2 TEXT)");  
}
```

Con todo esto, ya tendríamos implementada la clase auxiliar que nos permitirá crear, actualizar y conectarnos a una base de datos en Android.

### Autoevaluación

Los métodos que son obligatorios sobrescribir al extender la clase SQLiteOpenHelper son:

- onCreate()

[insert\(\)](#)

[onUpgrade\(\)](#)

[update\(\)](#)

[Mostrar retroalimentación](#)

## Solución

1. Correcto
2. Incorrecto
3. Correcto
4. Incorrecto

### 3.3.- Operaciones sobre una base de datos

Y ahora, ¿cuándo y cómo utilizamos esta clase auxiliar en nuestra aplicación Android?

La utilizaremos cada vez que queramos lanzar una actividad donde se vaya a realizar alguna operación sobre la base de datos.

Estas operaciones se pueden clasificar en dos grupos dependiendo de si se recupera o no algún resultado tras su ejecución. Así pues, tendremos:

- Inserción, actualización y eliminación de registros que no devuelven ningún registro, sólo se modifican.
- La selección, que retorna un conjunto de registros en un cursor.

Como ya hemos visto, las diferentes pantallas de Android se lanzan desde sus respectivas clases codificadas en Java. Aquellas pantallas que realicen algún tipo de operación sobre la base de datos necesitan apoyarse en la clase auxiliar. Pero dejando a un lado de momento el diseño que tendrán las pantallas, nos centramos en la implementación de sus respectivas clases.

Para esta implementación se pueden seguir los siguientes pasos:

- 1.- Crear un objeto de la clase SQLiteOpenHelper para poder abrir la base de datos. A este objeto le pasamos el contexto (que será la propia actividad), el nombre de la base de datos, un objeto CursorFactory que normalmente no es necesario y lo dejamos a null y el número de versión de la BD. Cuando se crea el objeto puede ocurrir alguna de las siguientes situaciones:
  - 1.1.- Si la base de datos existe y su versión coincide con la que le pasamos como parámetro, entonces se crea la conexión.
  - 1.2.- Si la base de datos existe pero su versión es anterior a la que le pasamos como parámetro, entonces se lanzará de forma automática el método onUpgrade() para actualizar la nueva BD y realizar así la conexión.
  - 1.3.- Si la base de datos no existe se lanzará de forma automática el método onCreate() y después se realizará la conexión.
- 2.- Abrir la base de datos en modo lectura y/o escritura según la operación que deseamos realizar.
- 3.- Comprobar que la base de datos está abierta y utilizar los métodos apropiados para realizar cualquier tipo de operación sobre la misma.
- 4.- Para finalizar no debemos olvidar cerrar siempre la base de datos.

El código podría ser muy similar al que mostramos a continuación:

```
//Creamos un objeto de la clase auxiliar.  
MiBaseDatosHelper bdhelp =  
    new MiBaseDatosHelper(this, "nombreBD", null, 1);  
  
//Abrimos en modo lectura y escritura la base de datos 'MiBD'  
SQLiteDatabase bd = bdhelp.getWritableDatabase();  
  
//Si la base de datos está abierta.  
if(bd.isOpen())  
{  
    //Aquí hay que añadir el código necesario para realizar  
    //cualquier tipo de operación sobre la base de datos.  
    //Veremos algunos métodos en los apartados siguientes.  
  
    //Cerramos la base de datos  
    bd.close();  
}
```



[Ciker-Free-Vector-Images \(Licencia Pixabay\)](#)

## 3.3.1.- Inserción, actualización y eliminación de registros

Revisando la clase `SQLiteDatabase` nos damos cuenta de que existen varios métodos para ejecutar sentencias SQL. Entonces, ¿cuáles son los más apropiados para cada tipo de operación? ¿Se pueden emplear indistintamente aunque la operación no devuelva ningún resultado?

Android nos proporciona una serie de métodos para que trabajemos sobre la base de datos dependiendo del tipo de operación que se realiza sobre la misma. En este apartado vamos a trabajar con los métodos necesarios para realizar operaciones de manipulación de datos, es decir, la inserción, la actualización y la eliminación de registros que recordamos no devuelven ningún registro.

Empezaremos viendo una forma muy sencilla.: el método `execSQL()`, que recibe como parámetro una sentencia SQL completa y la ejecuta. Aquí tienes algunos ejemplos:

```
bd.execSQL("INSERT INTO miTabla (campo1, campo2) VALUES (1, 'unTexto')");  
bd.execSQL("UPDATE miTabla SET campo2 = 'otroTexto' WHERE campo1 = 1 ");  
bd.execSQL("DELETE FROM miTabla WHERE campo1 = 1 ");
```



Everaldo Coelho and  
YellowIcon (GNU Lesser GPL)

Otra opción que se incluye en Android es utilizar métodos en los que se separan las tablas, los valores y las condiciones en parámetros. Éstos métodos son: `insert()`, `update()` y `delete()`.

El método `insert()` recibe tres parámetros en el siguiente orden:

- **Nombre de la tabla.**
- **Condiciones:** normalmente en la operación de inserción este valor será null porque no se precisan.
- **Valores del registro:** utilizaremos un objeto `ContentValues` para almacenarlos en forma de clave-valor.

Aquí tienes un ejemplo de inserción:

```
//Creamos el objeto ContentValues con los valores del registro.  
ContentValues miRegistro = new ContentValues();  
miRegistro.put("campo1", 1);  
miRegistro.put("campo2", "unTexto");  
//Insertamos el registro.  
bd.insert("miTabla", null, miRegistro);
```

El método `update()` recibe cuatro parámetros en el siguiente orden:

- **Nombre de la tabla.**
- **Valores:** el objeto `ContentValues` como en el caso anterior.
- **Condición:** la cláusula WHERE de la sentencia que podemos dejar a null si no los necesitamos.
- **Argumentos de la condición:** que podemos dejar a null si no los necesitamos.

Aquí tienes un ejemplo de actualización:

```
//Valores a actualizar.  
ContentValues miRegistro = new ContentValues();  
valores.put("campo2", "otroTexto");  
//Actualizar el registro  
bd.update("miTabla", miRegistro, "campo1=1", null);
```

El método `delete()` recibe tres parámetros en el siguiente orden:

- **Nombre de la tabla.**
- **Condición:** la cláusula WHERE de la sentencia que podemos dejar a null si no los necesitamos.
- **Argumentos de la condición:** que podemos dejar a null si no los necesitamos.

Aquí tienes un ejemplo de eliminación:

```
//Eliminamos el registro.  
bd.delete("miTabla", "campo1=1", null);
```

En los métodos `delete()` y `update()` hemos visto el parámetro de argumentos de la condición. Este parámetro lo podemos utilizar para simplificar la cadena de texto que pasamos como condición. Para ello, utilizamos un array con los valores de cada argumento en el mismo orden que aparecerán en la cláusula WHERE y en la condición introducimos el carácter (?) por cada valor que deseamos reemplazar. Veamos un ejemplo:

```
String[] argumentos = new String[]{"valor1","valor2"};  
bd.update("miTabla", miRegistro, "campo2=? OR campo2=?", argumentos);
```

También se puede utilizar este parámetro en el método `execSQL()`:

```
String[] argumentos = new String[]{"valor1","valor2"};  
bd.execSQL("DELETE FROM miTabla WHERE campo2=? OR campo2=?", argumentos);
```

## Para saber más

Es recomendable que consultes cada uno de los métodos que se han visto en el apartado en la documentación oficial de Android.

[Documentación oficial de la clase SQLiteDatabase](#)

## Autoevaluación

Marca las sentencias en las que se produciría un error:

`bd.insert("miTabla", null, miRegistro);`

`bd.delete("miTabla", null, null);`

`bd.execSQL("DELETE FROM miTabla WHERE campo1 = ? ", null);`

`bd.insert("miTabla", miRegistro);`

[Mostrar retroalimentación](#)

## Solución

1. Incorrecto
2. Incorrecto

- 3. Correcto
- 4. Correcto

## 3.3.2.- Recuperación o consulta de registros

Ya conocemos las distintas formas de manipular los registros con SQLite; ahora vamos a aprender a recuperar los registros y mostrarlos.

Si nos fijamos en la clase SQLiteDatabase encontramos dos métodos que nos pueden servir: rawQuery() y query().

¿En qué situación utilizamos cada uno?

Al igual que pasa en las operaciones de manipulación de registros vamos a tener dos formas de ejecutar una sentencia de consulta:

- rawQuery(): ejecuta directamente la sentencia SQL de consulta que pasamos como parámetro.
- query(): utiliza varios parámetros para construir la sentencia a ejecutar.

En cualquier caso, la ejecución de una sentencia de consulta nos devolverá un conjunto de registros en un objeto de tipo Cursor que hereda de la clase SQLiteDatabase. Este cursor nos permite procesar de forma individual cada fila del resultado de la consulta. Inicialmente este cursor apunta a la posición previa de la primera fila con lo cual tenemos que mover el cursor al primer registro mediante el método booleano moveToNext(). A continuación iremos iterando por cada fila hasta llegar al final del resultado donde el método moveToNext() devolverá el valor false indicando que no hay más filas que procesar.

Veamos la primera opción de implementación, bastante sencilla, utilizando el método rawQuery(). Fíjate cómo también se podría pasar como segundo parámetro los argumentos de la condición tal y como vimos en el apartado anterior. En este ejemplo se pone a null porque no se necesita.

```
Cursor cur = bd.rawQuery("SELECT campo1, campo2 FROM miTabla", null);
```

Para utilizar el método query() debemos tener en cuenta los parámetros que se pasan:

- **Nombre de la tabla.**
- **Array con los nombres de campos.**
- **Condición:** la cláusula WHERE de la sentencia.
- **Argumentos de la condición:** que podemos dejar a null si no los necesitamos o bien pasarlo como parámetros en un array.
- **Cláusula GROUP BY** que podemos dejar a null si no los necesitamos.
- **Cláusula HAVING** que podemos dejar a null si no los necesitamos.
- **Cláusula ORDER BY** que podemos dejar a null si no los necesitamos.

Aquí se muestra un ejemplo de recuperación de registros utilizando este método:

```
String[] misCampos = new String[] {"codigo", "nombre"};
Cursor cur = bd.query("miTabla", misCampos, "campo1=1", null, null, null, null);
```

Una vez recuperados los registros en el objeto Cursor, utilizamos los métodos moveToFirst() y moveToNext() que devuelven true en caso de haber realizado correctamente el movimiento correspondiente y los métodos get\*(índice\_columna) para obtener el valor de cada columna a través del índice y según su tipo de dato como muestra el siguiente ejemplo:

```
if (cur.moveToFirst()) {
    do {
        int var1 = cur.getInt(0);
        String var2 = cur.getString(1);
    } while(cur.moveToNext());
}
```

Con los valores de los campos en guardados en las distintas variables ya podremos mostrarlos por pantalla de la forma que más nos interese.



Everaldo Coelho and  
YellowIcon (GNU Lesser GPL)

### Para saber más

Además de los métodos utilizados para recorrer el cursor y obtener los valores de sus campos, puedes consultar otros métodos que te pueden ser útiles en la documentación oficial de Android.

## Autoevaluación

Para ejecutar una sentencia **SQL** de consulta podemos utilizar los métodos `query()`, `rawQuery()` y `execSQL()`.

- Verdadero  Falso

**Falso**

La afirmación es falsa porque `execSQL()` se utiliza exclusivamente para sentencias de manipulación de registros (inserción, actualización y eliminación) que no devuelven ningún registro.

### Caso práctico



ADA

MARÍA

JUAN

Ada ha hablado con del departamento de I+D+I y ven conveniente conocer la nueva librería que Google ha lanzado para trabajar con base de datos: **Room**.

Ada ha pensado en **María** y **Juan** que ya conocen la biblioteca androidx.\*. Efectivamente **Juan** se da cuenta que **Room** es una nueva herramienta que pertenece a los componentes de la arquitectura de **Jetpack**, pero, ¿en qué consiste **Jetpack**? ¿Cómo funciona **Room**? ¿Qué ventajas tiene usar **Room** en vez de **SQLite**?

**Juan y María** se dan cuenta que es hora de profundizar en esta nueva colección de bibliotecas que Google ha lanzado y están ilusionados, ya que están innovando y ¡se

están adaptando a los cambios!

**Android Jetpack** es un conjunto de librerías y herramientas cuyo objetivo es agilizar el desarrollo de aplicaciones mejores. Las librerías de **Jetpack** utilizan el nombre de paquete androidx.\* y junto a las herramientas se distribuyen en cuatro categorías:

- ✓ Interfaz de usuario.
- ✓ Base.
- ✓ Arquitectura.
- ✓ Comportamiento.

Pero si antes ya existían los fragmentos, las notificaciones, los permisos, ¿qué hace nuevo a **Jetpack**?

Pues **Android Jetpack** ha organizado todos los componentes y los ha separado en estas cuatro categorías para entender mejor el mundo Android además de unificar todas las librerías (v4, v7, v13, v8, ...) en un mismo paquete androidx.\*



Imagen de Android Jetpack

Robot de Android de Google. Uso conforme a Licencia ([CC-BY](#))

### Librerías equivalentes entre Support Library 28.0.0 y las Bibliotecas de Android X

Support Library 28.0.0 (última versión de la biblioteca de compatibilidad.)	Bibliotecas de androidx
android.support.constraint.ConstraintLayout	androidx.constraintlayout.widget.ConstraintLayout
android.app.Fragment	androidx.fragment.app.Fragment
android.support.v4.app.ActivityCompat	androidx.core.app.ActivityCompat

Además **Jetpack** optimiza las clases ya creadas, en este módulo ya hemos visto sus ventajas en las preferencias de usuario que pertenecen a la categoría "**Comportamiento**" como puedes ver en la [documentación oficial sobre Jetpack](#)

Ya hemos avanzado bastante en el conocimiento de Android y vemos que una aplicación puede tener varios puntos de entrada ya que puede tener actividades, fragmentos, servicios, proveedores de contenido (**content providers**) y receptores de emisión (**broadcast receivers**). Llegamos a la conclusión que la arquitectura de una aplicación móvil es compleja y coordinar todos los componentes puede ser complicado. El objetivo de las nuevas clases y herramientas que forman parte de la categoría **Arquitectura de Componentes** de **Jetpack** es simplificar la comunicación de todos los componentes de una aplicación centrándonos en el ciclo de vida de cada componente y crear aplicaciones robustas, testeables y fáciles de mantener.

**Room** forma parte de esta arquitectura y convierte fácilmente nuestros objetos Java en datos de una tabla de **SQLite** de una forma robusta. Realizar las operaciones **CRUD** en una base de datos va a ser más sencillo que directamente hacerlo en **SQLite**. ¡Véamnos cómo es posible!

Lo primero que tenemos que hacer es preparar el proyecto para que pueda usar la librería **Room**.

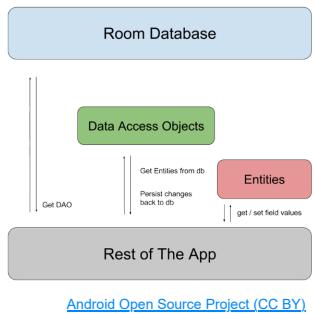
Para este apartado vamos a utilizar el proyecto GamePreferences realizado en el apartado 1.3.- Mostrar la pantalla de preferencias y añadir las dependencias de **Room** en el fichero build.gradle del módulo app:

```
implementation "androidx.room:room-runtime:2.2.3"
annotationProcessor "androidx.room:room-compiler:2.2.3"
```

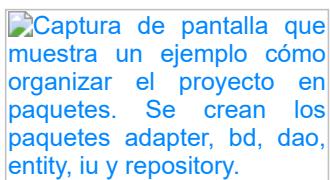
## 4.1.- Componentes de la librería

Los componentes de la librería **Room**, como puede verse en el esquema de la derecha, que engloba todos los componentes, son:

- **Entidades**: son las clases de Java que comúnmente se llaman POJO. Son clases simples, que representan un objeto en el mundo real. Normalmente estará compuesto por sus atributos privados y sus correspondientes **getters** y **setters** públicos. En nuestro ejemplo tenemos una clase POJO y es la clase Ranking.
- **Objetos de acceso a datos o DAO**: estos objetos incluyen métodos que ofrecen un acceso abstracto a la base de datos. Es abstracto, porque serán interfaces o clases abstractas donde se definirán todas las operaciones de lectura y escritura necesarias en la aplicación.
- **Base de datos**: habrá que declarar la base de datos de nuestra aplicación y definir el objeto que nos permita acceder.



Ya tenemos clara la estructura de componentes de nuestro proyecto pero, ¿cómo Room trabaja con SQLite? Pues es a través de las **Anotaciones** que la librería Room genera código SQLite internamente. Cuando una anotación precede a la definición de un componente, Room implementa el código para interactuar con la base de datos. Por ejemplo para indicar que la clase Ranking es una entidad se añadirá la anotación `@Entity` en la definición de clase:



`@Entity`

```
public class Ranking {  
    private int position;  
    private String avatar;  
    .....}
```

Captura del IDE Android Studio. Apache 2.0

El número de clases que tendrá nuestro proyecto crecerá considerablemente, con lo cual es deseable organizarlas en base a su función. Planteamos estructurar el proyecto en los siguientes package que mostramos en la imagen de la derecha. Desde el package gamepreferences:

- 1.- Seleccionamos las siguientes opciones **Refactor-> Rename** y le damos otro nombre.
- 2.- Añadimos los diferentes package mediante **File -> New -> Package**.

Esta solución es orientativa, puedes darles otro nombre o bien organizarlas de otra forma siempre y cuando las clases que contenga el paquete estén relacionadas.

## Autoevaluación

En Room, las clases abstractas o interfaces que definen los métodos de lectura y escritura para acceder a la base de datos son:

- Entidades.
- DAO
- Clases POJO
- Clases Helper.

Incorrecto

¡Correcto! Veamos cómo se implementa en Room estas clases abstractas o interfaces.

Incorrecto

Incorrecto

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## Para saber más

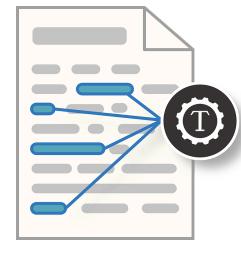
Puedes consultar el taller que Google ofrece para conocer esta librería bien en [Java](#) o en [Kotlin](#).

## 4.2.- Crear Entidades

Una entidad es una tabla de base de datos y una clase POJO representa una fila de esa tabla. ¿Cómo se consigue esta traducción?

De la siguiente forma:

- 1.- La clase de entidad o POJO está marcada con la anotación `@Entity`. Por defecto el nombre de la tabla coincide con la clase, se puede cambiar usando la propiedad `tableName` en `@Entity`.
- 2.- Cada atributo de la clase es una columna de la tabla. Por defecto el nombre del atributo es el nombre de la columna pero se puede modificar mediante la anotación `@ColumnInfo(name = "other_name")`.



**En SQLite, los nombres de tabla no distinguen entre mayúsculas y minúsculas.**

Además se pueden utilizar las siguientes anotaciones:

- `@Ignore`: con esta anotación se indica a Room que ignore un campo en particular.
- `@PrimaryKey (autoGenerate = true)`: con esta anotación indicamos a Room que una variable (normalmente ID) es un valor que se autogenera al crear un objeto de esta clase y que no podrá repetirse. Es un valor único y se puede utilizar para localizar un objeto concreto.
- `@NonNull`: esta anotación indica que un campo no puede ser nulo.

Room tiene ciertas **restricciones** que si no se cumplen dan error al generar la tabla en la base de datos y son las siguientes:

- 1.- Hay que tener en cuenta que Room debe ser capaz de crear un objeto de esta clase, con lo cual nos obliga a tener un constructor sin parámetros o bien un constructor con todos los campos no marcados con `@Ignore`.
- 2.- Si los atributos de la clase los marcamos como `private`, tenemos que definir sus correspondientes `setter` y `getter`

Finalmente el código de nuestra clase Ranking quedaría de la siguiente forma:

```
@Entity(primaryKeys = {"date", "avatar"})
public class Ranking implements Parcelable {
    public static final String TAG = "Ranking";
    @NonNull
    private String avatar;
    @NonNull
    private String date;
    private int points;
    public Ranking(String avatar, String date, int points) {
        this.avatar = avatar;
        this.date = date;
        this.points = points;
    }
    <br /> //region: Implementación métodos GET y SET
    public String getAvatar() {
        return avatar;
    }
    public void setAvatar(String avatar) {
        this.avatar = avatar;
    }
    public String getDate() {
        return date;
    }
    public void setDate(String date) {
        this.date = date;
    }
    public int getPoints() {
        return points;
    }
    public void setPoints(int points) {
        this.points = points;
    }
    //endregion
}
```

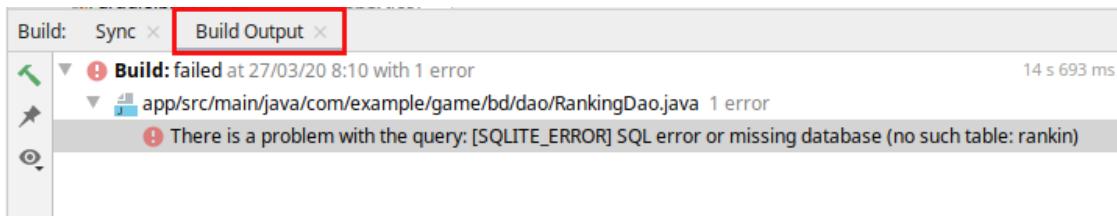
## Para saber más

En el caso que necesites implementar índices o quieras saber cómo implementar otras opciones puedes consultar la documentación oficial en el siguiente enlace:

[Cómo definir datos mediante entidades.](#)

## 4.3.- Crear DAO

Un DAO en Room es una clase abstracta o una interfaz que define un conjunto de métodos que ofrecen acceso a la base de datos utilizando consultas en SQL. Estos métodos los implementa Room en tiempo de compilación y realiza ciertas comprobaciones. Por ejemplo, en los métodos @Query Room verifica si hay un problema con la consulta, si fuera así muestra un error de compilación en el panel **Build** como muestra la siguiente imagen:



Captura del IDE Android Studio. [Apache 2.0](#)

En Room se pueden realizar las siguientes consultas:

- **@Query:** se hacen consultas directamente a la base de datos. A veces se necesita filtrar una serie de datos en base a un parámetro que se pasa al método, en este caso Room hace coincidir el parámetro de vinculación :avatar con el parámetro del método avatar. Cualquier error que hubiera se muestra en tiempo de compilación.

```
@Query("SELECT * FROM ranking WHERE avatar=:avatar")
Ranking findByShortName(String avatar);
```

- **@Insert:** inserta todos los parámetros en la base de datos con una sola transacción.

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
long insert(Ranking ranking);
```

Si sólo se inserta un único objeto este método puede devolver un valor long que muestra el id de la fila o rowId insertada en la base de datos. Si el parámetro es un Array o una colección, debería ser long[] o List<Long>.

- **@Update:** modifica un conjunto de entidades proporcionadas que se pasan por parámetros. Normalmente suele ser un objeto que se actualiza en la base de datos teniendo en cuenta la clave primaria o PrimaryKey:

```
@Update
void update(Ranking ranking);
```

Este método puede devolver un valor int que indica la cantidad de filas actualizadas en la base de datos.

- **@Delete:** elimina un conjunto de registros de la base de datos en base a los parámetros del método de igual forma que el método **@Update**.

```
@Delete
void delete(Ranking ranking);
```

### Para saber más

Puedes consultar la documentación oficial donde se muestran otras soluciones que se pueden dar en las operaciones a realizar con la base de datos:

[Cómo acceder a datos mediante DAO](#)

### Autoevaluación

El método que devuelve el valor rowId de una fila es:

- Delete
- Insert
- Update
- Query

No es correcto.

¡Correcto! SQLite utiliza internamente una columna llamada rowid que es autoincrementable de forma que cuando se inserta una nueva fila en la tabla su valor no se repite ya que se incrementa en uno.

No es la opción correcta.

No has acertado.

## Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto
4. Incorrecto

## 4.4.- Crear Base de Datos

Para crear la base de datos partimos del código del apartado [7. Add a Room Database](#) del taller sobre la arquitectura de componentes de **Jetpack** pero que adaptaremos y modificaremos al ser publicada bajo la licencia de [Apache 2.0](#). Para crear la base de datos realizaremos los siguientes pasos:

- 1.- En primer lugar tenemos que crear una clase para la base de datos, esta clase debe ser abstracta y heredar de RoomDatabase.
- 2.- Encima de la declaración de la clase se debe añadir la anotación @Database y mediante el parámetro `<span class="pln">entities</span>` enumerar tablas y especificar a continuación la versión de la base de datos.
- 3.- Ya dentro de la clase, se debe definir por cada **DAO** un método abstracto sin parámetros que devuelva la clase que está anotada con `@Dao`.



[IO-Images \(Licencia Pixabay\)](#)

```
<strong>@Database(entities = {Ranking.class},version = 1, exportSchema = false)</strong>
public <strong>abstract</strong> class GameDatabase extends <strong>RoomDatabase</strong> {<br /><br />
    public abstract <strong>RankingDao</strong> getRankingDao();<br /><br />
    private static volatile GameDatabase INSTANCE;
    private static String DATABASE_NAME="game";
    private static final int NUMBER_OF_THREADS = 4;
    public static final ExecutorService databaseWriteExecutor =
        Executors.newFixedThreadPool(NUMBER_OF_THREADS);
```

4.- A continuación se crea un objeto [singleton](#) de la base de datos llamado INSTANCE que será el que utilicen el resto de clases de la aplicación y accederán al objeto mediante un método estático de la clase.

5.- Room impide el acceso a la base de datos desde el hilo principal para solucionarlo Google utiliza la clase ExecutorService cuya función es ejecutar operaciones de la base de datos de forma asíncrona en un subproceso en segundo plano. Utilizar esta clase evitará tener que crear clases AsyncTask cada vez que se quiera realizar una operación con la base de datos.

```
public static void <strong>createDatabase</strong>(final Context context) {
    if (INSTANCE == null) {
        synchronized (GameDatabase.class) {
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                    GameDatabase.class, DATABASE_NAME)
                    .build();
            }
        }
    }
}
public static GameDatabase <strong>getDatabase</strong>(){
    return INSTANCE;
}
```

6.- Si ves la documentación oficial de Google en el método `getDatabase()` se crea la base de datos si no existiera y se devuelve la instancia. Esto conlleva un problema, ya que desde cualquier parte de la aplicación que quiera acceder a nuestra base de datos hay que pasarle un objeto Context. Para solucionarlo, nosotros hemos creado dos métodos `onCreateDatabase()` y mantenemos `getDatabase()`.

Para poder crear el objeto de la base de datos definimos una clase personalizada Application y en el método `onCreate()` llamamos al método de creación de la base de datos:

```
public class GameApplication extends Application {
    public static final String CHANNEL_ID = "1";
    @Override
    public void onCreate() {
        super.onCreate();
        <strong>    GameDatabase.onCreateDatabase(this);</strong>
    }
}
```

Es importante definir en el fichero manifiesto que nuestra clase `Application`, corresponde con el objeto que automáticamente crea

cuando se ejecuta la aplicación. Esto se hace añadiendo el atributo android:name al nodo <application>.

```
<application  
<strong>android:name=".iu.GameApplication"</strong>  
.....
```

## Autoevaluación

**Si se crea una clase que hereda de Application hay que indicar que corresponde al nodo <application> del fichero manifiesto.**

- Verdadero  Falso

**Verdadero**

Si no se declara el atributo android:name dentro del nodo <application> no se vinculará la clase creada con la clase Application que por defecto crea el sistema.

## 4.5.- Clase Repository: accediendo a Room desde la app



PIRO4D ([Licencia Pixabay](#))

Ya tenemos todos los componentes de Room definidos, y nos queda acceder a la base de datos desde la aplicación y lo haremos mediante el **patrón Repository**.

¿De qué se encarga este patrón?

En primer lugar se llama patrón a una solución que resuelve problemas comunes en el desarrollo de software y que después de comprobar su efectividad se convierte en una solución estándar. Un patrón ahorra tiempo a los programadores ya que tienen la seguridad que su solución es válida. El patrón **Repository** añade una capa dentro de nuestra aplicación cuya misión es mover la información entre los objetos de nuestra aplicación (clases **POJO**) y el sistema de almacenamiento que puede ser una base de datos **SQLite**, un sistema de archivos o bien estar en un servidor remoto, etc.

En nuestro ejemplo esta clase será la responsable de interactuar con la base de datos de Room en nombre de la vista y deberá proporcionar métodos que utilicen el **DAO** para insertar, eliminar y consultar registros del ranking. Estas operaciones de base de datos deberán realizarse en subprocesos separados del subproceso principal usando la clase **AsyncTask**.

```
public void insert(final Ranking ranking) {  
    new AsyncTask<Void, Void, Void>() {  
        @Override  
        protected Void doInBackground(Void... voids) {  
            rankingDao.insert(ranking);  
            return null;  
        }  
        .execute();  
    }<br />
```

¿Cómo accede toda la aplicación a esta clase repositorio?

Pues mediante una única instancia de la clase Repository de forma que el resto de la aplicación puede consultar la información del ranking de nuestra aplicación sólo a través de esta clase. Esto se consigue aplicando el **patrón Singleton** que consiste en:

- 1.- Declarar el constructor de la clase como privado de forma que sólo la propia clase puede crear una instancia de ella misma.
- 2.- Declarar un método de obtención de esta instancia comprobando que si es la primera vez que se accede a ella se crea el objeto y se devuelve. Este método suele llamarse `getInstance()` como se muestra en el siguiente código:

```
private RankingRepository() {  
    rankingDao = GameDatabase.getDatabase().getRankingDao();  
}  
public static RankingRepository getInstance() {  
    if (instance == null)  
        instance = new RankingRepository();  
    return instance;  
}
```

Ya hemos dicho que cualquier componente (normalmente actividades y fragmentos) pueden llamar a cualquier método de la clase repositorio y hemos dicho que estos métodos son asíncronos, es decir, que la ejecución de la aplicación sigue su curso. Entonces, nos planteamos la siguiente pregunta:

¿Cómo avisa el repositorio a las actividades o fragmentos que ya tiene los datos o qué se ha eliminado un elemento?

Declarando una interfaz en la clase repositorio de forma que toda clase que quiera una respuesta del repositorio debe implementar la siguiente interfaz:

```
public interface RankingRepositoryCallback {  
    void onSuccess(List<Ranking> list);  
    void onFailure();  
}
```

Veamos el ejemplo a la hora de eliminar un ranking. Este método recibe por parámetro la clase que implementa la interfaz `RankingRepositoryCallback` de forma que cuando se elimina un ranking se llama al método `onSuccess()` de esta clase que ejecutará las acciones pertinentes, en nuestro ejemplo actualizará el listado de ranking:

```
public void delete(final Ranking ranking, final RankingRepositoryCallback callback) {  
    if (ranking != null) {  
        new AsyncTask<Void, Void, Void>() {  
            @Override  
            protected Void doInBackground(Void... voids) {  
                rankingDao.delete(ranking);  
                callback.onSuccess();  
                return null;  
            }  
            .execute();  
        }  
    }  
}
```

## Ejercicio Resuelto

En el siguiente enlace te puedes descargar el proyecto entero que hemos explicado en este apartado. Presta atención a cómo las clases `ListRankingFragment` y `AddRankingFragment` se comunican con la clase repositorio `RankingRepository` y viceversa.

[Descarga el proyecto GameRoom](#) (zip - 2.07 MB).

## 5.- Proveedores de Contenidos

### Caso práctico

**Ada** ha comprobado cómo están gestionando los datos **Juan** y **María** y ahora les hace el siguiente planteamiento:

¿Podríamos crear alguna aplicación en Android que necesitara datos de otra aplicación ya existente? Por ejemplo, ¿podríamos acceder a la lista de contactos de nuestro dispositivo móvil desde nuestra aplicación?



**Juan** y **María** suponen que sí es posible y no están nada equivocados, de hecho, **Ada** les ha dado una pista para que investiguen sobre el tema: los **proveedores de contenidos** o **ContentProvider**.

Ya hemos visto que los datos se pueden almacenar en un dispositivo móvil a través de ficheros o mediante una base de datos. Ahora bien, ¿son estos datos accesibles únicamente desde la aplicación donde se han generado? No tendría mucho sentido prohibir el acceso a determinados datos que puedan necesitar otras aplicaciones. Evitaríamos así utilizar más espacio de memoria y hablando de dispositivos móviles, ya sabemos que esta limitación es muy importante.

Android dispone de un mecanismo que se encarga de compartir los datos entre varias aplicaciones que se llama Content Provider o proveedores de contenidos. La forma de acceder a esos contenidos se hace mediante el uso de URI.



Everaldo Coelho and YellowIcon (GNU Lesser GPL)

**Una URI** es una cadena de caracteres que identifica de forma única un recurso (ej: una llamada de teléfono, un **SMS**, un correo electrónico, etc.)

Una URI está formada por varias partes. La primera es el prefijo que nos indica el tipo de recurso al que se desea acceder. Aquí tienes tan sólo algunos ejemplos de las distintas URI que ofrece Android:

- http:// para acceder a una página web.
- smsto: para envío de mensajes.
- mailto: para envío de correo.
- content:// para acceder a los proveedores de contenido que veremos en este apartado con más profundidad.

Sobre los datos de los proveedores de contenidos podemos realizar cualquier operación de inserción, actualización, borrado y por supuesto consulta. Además como ya sabemos utilizar el motor de base de datos SQLite la forma de trabajar será muy parecida utilizando incluso sentencias SQL y utilizando cursor para recorrer los resultados obtenidos.

También hay que tener en cuenta que debemos solicitar permiso en el fichero de manifiesto para realizar cualquier tipo de operación (lectura y/o escritura). Así por ejemplo si deseamos acceder al proveedor de contenidos Browser para consultar la tabla del historial de navegación, tendremos que incluir el siguiente código dependiendo del permiso que necesitemos:

```
<uses-permission android:name="com.android.browser.permission.READ_HISTORY_BOOKMARKS"/>
<uses-permission android:name="com.android.browser.permission.WRITE_HISTORY_BOOKMARKS"/>
```

En este apartado nos centraremos en detalle en cómo gestionar datos de otros proveedores de contenidos ya que normalmente cuando creamos una aplicación nos puede ser muy útil utilizar los datos de otras aplicaciones como el registro de llamadas, la lista de contactos, páginas visitadas, etc.

También veremos cómo podemos crear un proveedor de contenidos para que otras aplicaciones puedan utilizar los propios datos que genere nuestra aplicación.

# 5.1.- Utilizando proveedores de contenidos

¿Cómo accedemos a los datos que ya existen en otras aplicaciones?



Como ya hemos visto, cada URI indica el recurso al que deseamos acceder, además del prefijo que identifica el tipo de recurso y que en nuestro caso será `content://` debemos especificar también el proveedor o aplicación a la que deseamos acceder y por último la tabla a la que vamos hacer referencia. Así por ejemplo, si queremos utilizar la tabla donde se almacena la lista de favoritos del navegador tendremos que utilizar la URI correspondiente que tendrá el formato `content://browser/bookmarks` aunque es más cómodo utilizar la constante preestablecida `Browser.BOOKMARKS_URI` que nos proporciona Android.

Everaldo Coelho and  
YellowIcon (GNU Lesser GPL)

## Debes conocer

¿Y qué tipos de proveedores existen? ¿Cómo están identificados?

En el siguiente enlace puedes encontrar toda la información referente a los Content Provider de Android.

[Proveedores de contenido en Android.](#)

Vamos a realizar un ejemplo sencillo para recuperar desde nuestra nueva aplicación la lista de favoritos. Después de solicitar los permisos adecuados en el fichero de manifiesto para acceder al proveedor de contenidos (sólo hace falta el de lectura porque únicamente queremos recuperar los datos), creamos una instancia de la URI que necesitamos para acceder al recurso Browser y la tabla correspondiente utilizando la constante que Android nos proporciona (`BOOKMARKS_URI`). Después creamos una referencia a un objeto `ContentResolver` mediante el método `getContentResolver()` lo cual nos permitirá realizar operaciones sobre el proveedor de contenidos. La tabla a la que vamos a acceder será `BookmarkColumns` que contiene una serie de columnas que debemos indicar en la consulta, para ello crearemos un array indicando las dos columnas que deseamos obtener. Recuerda que cada proveedor dispone de diferentes tablas y columnas que puedes consultar en la documentación oficial de Android.

```
Uri miUri = Browser.BOOKMARKS_URI;  
  
ContentResolver contRes = getContentResolver();  
  
String[] columnas = new String[] {  
    Browser.BookmarkColumns.URL,  
    Browser.BookmarkColumns.VISITS };  
  
Cursor cur = contRes.query(miUri,  
    columnas,  
    null,  
    null,  
    null);
```

A partir de aquí sólo nos queda recorrer el cursor para recoger los valores de las dos columnas que hemos pasado. Esto lo hacemos indicando el índice que ocupa cada columna en la tabla de la siguiente forma:

```
int columnaURL = cur.getColumnIndex(Browser.BookmarkColumns.URL);  
int columnaVisits = cur.getColumnIndex(Browser.BookmarkColumns.VISITS);
```

Con el índice ya podemos acceder al campo de cada registro y recuperar su valor.

```
cur.getString(columnaURL);  
cur.getString(columnaVisits);
```

## Recomendación

En el siguiente enlace puedes descargar un proyecto de ejemplo donde se puede ver cómo se accede a los datos de la lista de favoritos del navegador desde una aplicación que creamos nosotros pulsando sobre un botón. Estudia su código y verás que no es muy complicado. Prueba a hacer tú lo mismo accediendo por ejemplo al listado de llamadas.

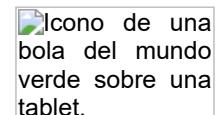
[Aplicación de ejemplo sobre la utilización de un Proveedor de Contenidos](#) (zip - 13.61 MB).

## 5.2.- Creando un proveedor de contenidos

La posibilidad de compartir datos entre las distintas aplicaciones de Android nos ofrece una gran ventaja. Ya sabemos cómo podemos ver datos que han sido almacenados por otras aplicaciones o recursos de Android, pero, ¿podemos compartir los datos de una aplicación que hemos hecho nosotros?

Por supuesto que sí. Veamos cómo se haría.

Los pasos a seguir para que nuestra aplicación pueda compartir sus propios datos con otras aplicaciones serían los siguientes:



Everaldo Coelho and  
YellowIcon (GNU Lesser GPL)

- 1.- Identificar el tipo de información que vamos a compartir y si están almacenados en ficheros o en bases de datos.
- 2.- Crear una clase que extienda de la clase ContentProvider donde se debe implementar los métodos abstractos onCreate(), query(), insert(), delete(), update(), getType() con los cuales otras aplicaciones podrán acceder a nuestros datos y realizar cualquier operación sobre ellos y definir las URI que sean necesarias para su acceso.
- 3.- Por último registrar el ContentProvider en el fichero de manifiesto.

En nuestra aplicación de ejemplo podemos crear un ContentProvider sobre la tabla Alumnos (que contiene los campos \_ID, nombre, email). Una primera consideración a tener en cuenta para crear un proveedor de contenidos es que los registros almacenados se identifiquen de forma única por un campo que funcione como clave primaria. En nuestro caso tenemos el campo \_ID que además se autoincrementa conforme se insertan nuevos registros.

### Debes conocer

En el siguiente enlace puedes encontrar la información necesaria para la creación y posterior uso de un ContentProvider sobre una tabla de Base de Datos que podrías utilizar para tu aplicación **Alumnos**.

[Creación y uso de un ContentProvider](#)

# Condiciones y términos de uso de los materiales

Materiales desarrollados inicialmente por el Ministerio de Educación, Cultura y Deporte y actualizados por el profesorado de la Junta de Andalucía bajo licencia Creative Commons BY-NC-SA.



Antes de cualquier uso leer detenidamente el siguiente [Aviso legal](#)

## Historial de actualizaciones

Versión: 03.00.03

Fecha de actualización: 14/11/23

Actualización de materiales y correcciones menores.

Versión: 03.00.00

Fecha de actualización: 13/06/20

Autoría: Lourdes Rodríguez Morón

Ubicación: 3 Bases de datos

Mejora (tipo 3): Cambiar todo el apartado. Seguir la recomendación de Android y usar Room como abstracción para el acceso a la base de datos.

Ubicación: apartado 3.3.2.

Mejora (tipo 2): No se explican correctamente los cursores que devuelven las consultas a la base de datos.

Ubicación: apartado 1.2

Mejora (tipo 2): La explicación de este apartado se basa en dos ejemplos que no en la versión actual no compilan correctamente.

Ubicación: Nodos del mapa

Mejora (Mapa conceptual): Revisión y actualización en base a los nuevos contenidos

Ubicación: Tabla de contenidos

Mejora (Orientaciones del alumnado): Revisión y actualización en base a los nuevos contenidos

Versión: 02.00.00

Fecha de actualización: 25/04/14

Autoría: Víctor Gil Rodríguez

Nuevos contenidos, mapa conceptual y orientaciones actualizados con Android.

Versión: 01.00.00

Fecha de actualización: 25/04/14

Versión inicial de los materiales.

