

Desarrollo de aplicaciones e introducción a juegos con Android Studio y AndEngine.

Tema 5: Persistencia de datos en Android



Jose Manuel Fornés Rumbao (© Copyright)

Departamento de Ingeniería Telemática
Universidad de Sevilla

jfornes@us.es

Índice

1. Introducción
2. Estructura interna de la aplicación
3. Preferencias y estado de la UI
4. Sistema de ficheros
 1. Ficheros dentro de la aplicación: Privados
 2. Ficheros dentro de la aplicación: Públicos
 3. Raw Files
 4. Ficheros XML personalizados
5. Bases de datos. SQLite
6. Content Provider
7. Objeto Application

Introducción

- En aplicaciones típicas de escritorio, el sistema operativo ofrece el sistema de ficheros para compartir datos entre aplicaciones
- En Android, los ficheros son privados por aplicación
- Para compartir información se usan los ContentProvider
- Los mecanismos de persistencia son:
 - Preferencias y estado de UI
 - Ficheros locales
 - Bases de datos



Estructura interna de la aplicación

- Cada aplicación que se instala en el dispositivo crea una carpeta con el nombre de su paquete principal dentro de la ruta /data/data de la memoria interna del dispositivo. Dentro de esa carpeta privada (que no puede ser accedida por ningún otro proceso salvo que se disponga de un terminal de desarrollo o se “rootee” el dispositivo) se aloja una estructura en la que, entre otros, se encuentran las siguientes carpetas:
 - **/shared_prefs**: contiene un conjunto de ficheros XML que se corresponden a los nombres de las preferencias compartidas. Almacena un mapa con pares clave-valor.
 - **/files**: contiene los archivos del sistema, que pueden ser binarios o de texto.
 - **/databases**: contiene las bases de datos SQLite asociadas a la aplicación.
 - **/cache**: contiene los datos temporales de la aplicación

Preferencias y estado de UI (I)

- Para guardar el estado se utiliza `SaveInstanceStateHandler`, que se ha diseñado específicamente para ello
 - Antes de que una actividad termine, se llama a `onSaveInstanceState()` para guardar el estado de la UI y que ésta pueda recuperar en el futuro

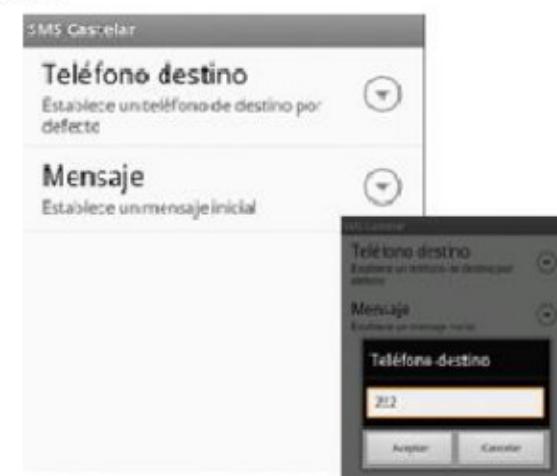
```
@Override  
public void onSaveInstanceState(Bundle state) {  
    TextView tv = (TextView) findViewById(R.id.view);  
    String textViewValue = tv.getText().toString();  
    state.putString(TEXTVIEW_STATE_KEY, textViewValue);  
    super.onSaveInstanceState(state);  
}
```

- La recuperación del estado se realiza mediante `onRestoreInstanceState`

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
    TextView tv = (TextView) findViewById(R.id.view);  
    String text = "";  
    if (savedInstanceState != null && savedInstanceState.containsKey(TEXTVIEW_STATE_KEY)) {  
        text = savedInstanceState.getString(TEXTVIEW_STATE_KEY);  
        myTextView.setText(text);  
    }  
}
```

Preferencias y estado de UI (II)

- Para guardar las preferencias
 - Los datos se almacenan en pares key/value en un fichero XML que puede gestionarse desde el código
 - Usado típicamente para guardar las preferencias de la aplicación (fuentes, colores..)
 - Permite guardar datos primitivos (Boolean, float...)
 - Esta información persiste aunque la aplicación muera.
 - Las Actividades tienen los métodos `onSaveInstanceState()` o `onStop()` para guardar los datos antes de acabar
 - Se invoca el método `getSharedPreferences (File)` si tenemos varios ficheros o `getPreferences()` para un sólo fichero



Preferencias y estado de UI (III)

- Creando y salvando las preferencias:

```
SharedPreferences pref = null;  
protected void savePreferences() {  
    int mode = Activity.MODE_PRIVATE;  
    pref = getSharedPreferences("mypre", mode);  
    SharedPreferences.Editor editor = preferences.edit();  
    editor.putBoolean("isSilentMode", true);  
    editor.commit();  
}
```

- Se puede acceder a las preferencias a través del Contexto
- Una actividad hereda del contexto

Preferencias y estado de UI (IV)

- Recuperando preferencias

```
SharedPreferences pref;
protected void restorePreferences() {
    int mode = Activity.MODE_PRIVATE;
    pref = getSharedPreferences(MYPREFS, mode);
    nameValue = pref.getString("name", "");
    phoneValue = pref.getString("phone", "");
}
```

- Existen 4 modos de acceso:
 - Context.MODE_PRIVATE: sólo la aplicación puede acceder al archivo de preferencias.
 - Context.MODE_WORLD_READABLE: otras aplicaciones pueden leer el archivo de preferencias, pero no modificarlas.
 - Context.MODE_WRITEABLE: otras aplicaciones pueden leer y escribir el archivo de preferencias.
 - Context.MODE_MULTI_PROCESS: varios procesos pueden acceder simultáneamente al fichero de preferencias.

Sistema de ficheros

- Existen varias formas de acceder al sistema de ficheros. Depende de la ubicación y formato de los mismos.
 - Ficheros dentro de la aplicación
 - Raw files que son incluidos como recursos
 - Ficheros XML personalizados

Ficheros dentro de la aplicación

- Se gestionan como los ficheros en JAVA mediante I/O, se
- deben crear inputs y outputs streams
- Sólo se soportan archivos creados en la misma carpeta que la aplicación
- Las aplicaciones van a la ruta: /data/app (gratuitas) y /data/app-private (de pago), las nuevas versiones soportan almacenamiento en la SDCard.
- Se usan los métodos: `openFileInput` y `openFileOutput`:
 - El nombre del archivo no puede contener separadores de path
 - El archivo creado es por defecto privado a la aplicación
 - Hay que recordar llamar al `close` al finalizar

```
FileOutputStream fos;
FileInputStream fis;
String FILE_NAME = "temp.tmp";
//Crea un fichero de salida privado a la aplicación.
fos = openFileOutput(FILE_NAME, Context.MODE_PRIVATE);
//Crea un fichero de entrada FileInputStream
fis = openFileInput(FILE_NAME);
```

Ficheros dentro de la aplicación: Privados

- Para acceder a los archivos alojados en la tarjeta interna, usaremos los métodos `openFileInput()` y `openFileOutput()`. La escritura sería de la siguiente forma:

```
// Declaramos una constante con el nombre del fichero
private static final String CONFIG_FILENAME = "config.txt";
[...]

try {
    // Abrimos un fichero privado asociado con el paquete del contexto de la aplicacion
    // en modo de escritura.
    // El fichero estará en este caso en          /data/data/com.dam.practicas/files/config.txt
    FileOutputStream streamFichero = openFileOutput(CONFIG_FILENAME, Activity.MODE_PRIVATE);
    // Usamos un OutputStreamWriter para transformar una cadena de texto en un array de bytes
    // y almacenarlos en el archivo.
    OutputStreamWriter writer = new OutputStreamWriter(streamFichero);
    // Escribimos la cadena "Hola, mundo." y un retorno de carro.
    writer.write("Hola, mundo." + System.getProperty("line.separator"));
    // Forzamos la escritura y cerramos el OutputStreamWriter
    writer.flush();
    writer.close();
}
catch(IOException e) {
    Log.e(this.getClass().getName(), "escribirConfiguracion()", e);
}
```

Ficheros dentro de la aplicación: Privados

- La lectura se realizaría de forma similar:

```
try {  
    // Abrimos un fichero privado asociado con el paquete del contexto de la aplicación  
    // en modo de solo lectura  
    FileInputStream inputStream = openFileInput(CONFIG_FILENAME);  
    // Construye un objeto que transformara el fichero en un flujo de caracteres, es decir,  
    // convertira el array de bytes en una cadena de texto.  
    InputStreamReader inputStreamReader = new InputStreamReader(inputStream);  
    // En lugar de utilizar directamente el InputStreamReader, creamos a partir de el  
    // un BufferedReader, que encapsula el InputStreamReader y proporciona ya un buffer  
    // para los datos de entrada. De este modo nos despreocupamos de mantener el buffer.  
    BufferedReader bufferedReader = new BufferedReader(inputStreamReader);  
    // Leemos linea a linea  
    String configLine = bufferedReader.readLine();  
    while(configLine != null)  
    {  
        procesarLinea(configLine); // Usamos la linea de texto para lo que queramos  
        configLine = bufferedReader.readLine(); // Leemos la siguiente linea  
    }  
} catch(IOException e) {  
    Log.e(this.getClass().getName(), "leer()", e);  
}
```

Ficheros dentro de la aplicación: Públicos

- Para leer y escribir en un fichero almacenado en la tarjeta SD necesitaremos añadir los permisos adecuados en el manifiesto, que serán los siguientes:

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

- La escritura se realiza de forma similar:

```
// Obtenemos la ruta de la tarjeta SD
File sdCardPath = Environment.getExternalStorageDirectory();
// Creamos el fichero pasándole la ruta y el nombre del fichero
// En este caso, será /mnt/sdcard/sdcard/config.txt
File ficheroExt = new File(sdCardPath.getAbsolutePath(), CONFIG_FILENAME);
// Abrimos el flujo de salida del fichero para escribir en él.
// El booleano que se pasa como segundo parámetro indicará si queremos añadir nuevo texto
// (true) o no (false).
FileOutputStream streamFicheroExt = new FileOutputStream(ficheroExt, false);
// Obtenemos un OutputStreamWriter para transformar el texto en un array de bytes
OutputStreamWriter writerExt = new OutputStreamWriter(streamFicheroExt);
// Escribimos la cadena "Hola, mundo." y un retorno de carro.
writerExt.write("Hola, mundo." + System.getProperty("line.separator"));
// Forzamos la escritura y cerramos el OutputStreamWriter
writerExt.flush();
writerExt.close();
```

Ficheros dentro de la aplicación: Públicos

- Además de escribir en el almacenamiento interno y externo, es posible hacer uso de otras dos rutas predefinidas:
 - la caché de la aplicación, pensada para almacenar datos temporales y que se corresponderían primero con el directorio /cache, que se accedería de la siguiente forma:

```
// Se corresponderá con el directorio  
/data/data/com.dam.practica/cache/config.txt567822742.tmp  
File ficheroCache = File.createTempFile(CONFIG_FILENAME, null, getCacheDir());
```

- El segundo será el directorio público de la aplicación asociado a un tipo determinado de dato. Así, si quisiésemos almacenar fotos, podríamos hacer que éstas se almacenaran en el directorio de la carpeta pública destinada a ello por defecto.

```
// Se corresponderá con el directorio /mnt/sdcard/DCIM  
File ficheroExtRel = new  
File(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DCIM),  
CONFIG_FILENAME);
```

Ficheros dentro de la aplicación: Públicos

- Los valores para estos directorios son los siguientes:
 - Environment.DIRECTORY_PICTURES: Directorio público de imágenes
 - Environment.DIRECTORY_DCIM: Directorio público de fotografías
 - Environment.DIRECTORY_ALARMS: Directorio público de alarmas
 - Environment.DIRECTORY_DOWNLOADS: Directorio público de descargas
 - Environment.DIRECTORY_MOVIES: Directorio público de películas
 - Environment.DIRECTORY_MUSIC: Directorio público de música
 - Environment.DIRECTORY_NOTIFICATIONS: Directorio público de sonidos de notificación
 - Environment.DIRECTORY_PODCASTS: Directorio público de podcasts
 - Environment.DIRECTORY_RINGTONES: Directorio público de tonos de llamada

Raw files

- Son archivos incluidos como recursos y guardados en /res/raw y se pueden leer:
Resources.openRawResource(R.raw.nombre_del_fichero)
- El fichero se identifica sólo con su nombre, sin su extensión
- El acceso es de sólo lectura
- Estos archivos **no son compilados**. Pueden ser binarios.

```
//Ejemplo de lectura de un fichero binario
Resources resources = this.getResources();
InputStream is = null;
try {
    is = resources.openRawResource(R.raw.people);
    byte[] reader = new byte[is.available()];
    while (is.read(reader) != -1) {}
    this.readOutput.setText(new String(reader));
} catch (IOException e) {
    Log.e("ReadRawResourceFile", e.getMessage(), e);
}
```

Ficheros XML personalizados

- Son archivos incluidos como recursos y guardados en /res/xml
- No se usa un stream para acceder a ellos
- Estos archivos **son compilados** por la plataforma (eficiencia)
- Se pueden leer con un XMLPullParser

```
//Supongamos un fichero llamado test.xml y guardado en /res/xml/test.xml
private String getEventsFromAnXMLFile(Activity activity) throws XmlPullParserException, IOException
{
    StringBuffer sb = new StringBuffer();
    Resources res = activity.getResources();
    XmlResourceParser xpp = res.getXml(R.xml.test);
    xpp.next();
    int eventType = xpp.getEventType();
    while (eventType != XmlPullParser.END_DOCUMENT){
        if(eventType == XmlPullParser.START_DOCUMENT){
            sb.append("*****Start document");
        }
        else if(eventType == XmlPullParser.START_TAG){
            sb.append("\nStart tag "+xpp.getName());
        }
        else if(eventType == XmlPullParser.END_TAG){
            sb.append("\nEnd tag "+xpp.getName());
        }
        else if(eventType == XmlPullParser.TEXT){
            sb.append("\nText "+xpp.getText());
        }
        eventType = xpp.next();
    }//eof-while
    sb.append("\n*****End document");
    return sb.toString();
}//eof-function
```

Bases de datos

- Android hace uso de la biblioteca SQLite para aquellas aplicaciones que necesiten una base de datos. Cada propia aplicación corre su propia base de datos, evitando problemas de concurrencia, corrupción de datos, etc.
- SQLite:
 - Open source
 - Proporciona capacidades de una base de datos relacional
 - La usan el Iphone, el ipod, etc...
 - Ligera. No requiere excesivos recursos
 - Para más información: <http://www.sqlite.org>
- Las bases de datos, al igual que los ficheros internos y las preferencias compartidas, se almacenan en el directorio interno de la aplicación, concretamente en el subdirectorio /data/data/nombre_paquete/databases.
- Permite gestionar actualizaciones de las bases de datos (realizar ALTER TABLE cuando se necesite)
- Por defecto las bases de datos son privadas a la aplicación. Para compartir datos entre aplicaciones se usarán los ContentProviders.

Bases de datos: Cursor

- Android devuelve el resultado de las consultas (querys) a la base de datos en un objeto Cursor. Un Cursor no es más que un puntero a un conjunto de datos.
- Incluye métodos para moverse por los datos:
 - moveToFirst: mueve el cursor a la primera fila de los registros
 - moveToNext: mueve el cursor a la siguiente fila de los registros
 - moveToPrevious: mueve el cursor a la fila anterior de los registros
 - getCount: devuelve el número de registros de la query
 - getColumnIndexOrThrow: devuelve un índice para la columna dada
 - getColumnName: dado un índice, devuelve el nombre de esa columna
 - getColumnNames: devuelve un array con los nombres de las columnas
 - moveToPosition: mueve el cursor al registro que hay en esa posición
 - getPosition: devuelve la posición actual del cursor

Bases de datos: Cursor

- El programador debe llamar el método `close` del cursor para liberar los recursos tomados y llamar a `deactivate` y `requery` cuando sea necesario.
- Sin embargo, Android ofrece el método `managedQuery` (...) el cual gestiona el ciclo de vida del cursor por nosotros asociándolo al ciclo de vida de la actividad.
- Llamar a este método es la manera más sencilla y óptima para trabajar con cursores desde una actividad
- Métodos importantes de un cursor:
 - Close Cierra el cursor liberando completamente los recursos tomados
 - Deactivate Desactiva el cursor haciendo que fallen todas las llamadas al cursor hasta que se llame a `requery`. (Libera recursos)
 - Requery Lanza de nuevo la consulta que creó el cursor, refrescando su contenido.

Bases de datos: SQLite (I)

- Creación de la base de datos: se crea una nueva clase que herede de la clase `SQLiteOpenHelper`, que implementará por defecto dos nuevos métodos: `onCreate()` y `onUpdate()`.
 - El primero de estos métodos se ejecutará una sola vez si la base de datos no existe, y se utilizará para crear la estructura de la base de datos (tablas) y una carga de datos inicial (si procede).
 - El segundo método se ejecutará si la versión de la base de datos aumenta desde el último acceso (es decir, si el desarrollador modifica la base de datos, debe modificar el valor de la versión para que `SQLiteOpenHelper` ejecute el método `onUpdate()`, realizando operaciones como por ejemplo añadir un nuevo campo a una tabla).
- SQLite proporciona un conjunto muy limitado de tipos de dato: enteros, cadenas, reales y blobs (además del valor `NONE`). Por lo tanto, el resto de datos, incluyendo fechas, tendrán que ser “emulados” de alguna forma. La fecha la podemos codificar como una cadena de texto con el formato “YYYY-MM-DD HH:MM:SS.SSS” (“1970-01-01 00:00:00”), como un entero (número de segundos desde 1970-01-01 00:00:00) o como un número real (días desde 4714-11-24 A.C.).

Bases de datos: SQLite

- Ejemplo: Una BD con una tabla “configuracion” con tres campos: clave [text], valor [text] y fechaModif [datetime].

```
public class SQLConfigManager extends SQLiteOpenHelper {

    private static final String SQL_DROP_CONFIG = "drop table if exists configuracion";
    private static final String SQL_CREATE_CONFIG = "create table configuracion(" +
                                                    "clave text primary key," +
                                                    "valor text," +
                                                    "fechaModif datetime)";

    // Constructor por defecto
    public SQLConfigManager(Context context, String nombre, CursorFactory factory, int version)
    {
        super(context, nombre, factory, version);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        // Al instalar la base de datos, creamos la tabla "configuracion"
        db.execSQL(SQL_CREATE_CONFIG);
        Log.d(TAG, "onCreate()");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        // En caso de actualizar la base de datos, eliminamos la tabla y la volvemos a crear.
        db.execSQL(SQL_DROP_CONFIG);
        db.execSQL(SQL_CREATE_CONFIG);
    }
}
```

Bases de datos: SQLite (II)

- Lanzando una consulta

```
String[] resultColumns = new String[] {  
    KEY_ID, NAME, DESCRIPTION };  
Cursor cursor = db.query(DATABASE_TABLE, resultColumns, null, null, null,  
    null);
```

- En este ejemplo tenemos:
 - DATABASE_TABLE: Nombre de la base de datos
 - resultColumns: es un array con los nombres de las columnas a obtener:
 - KEY_ID: será el nombre de la primera columna a obtener
 - NAME: será el nombre de la segunda columna a obtener
 - DESCRIPTION: será el nombre de la tercera columna a obtener
 - Null: Filtro WHERE, del tipo p.e: "clave=? ". Si null entonces todos
 - Null: es un array con los nombres de los parámetros que sustituyen al «?» Anterior
 - Null: Cláusula GROUP BY
 - Null: Cláusula HAVING
 - Null: Cláusula ORDER BY

Bases de datos: SQLite (III)

- Por ejemplo, el SQL:

```
select nombre, apellidos from empleados  
    where estadoCivil = 'S' and edad > 38  
        order by apellidos
```

- Se realiza así:

```
String[] filtroSelect = {"nombre", "apellidos"};  
String[] filtroColumnas = {"S", "38"};  
  
cSelect = db.query("empleados", filtroColumnas, "estadoCivil=? and edad >?",  
    filtroSelect, null, null, "apellidos");
```

- También es posible ejecutar el SELECT directamente:

```
String SQL_SELECT_VALOR = "select valor from configuracion " +  
                            " where clave=?";  
cSelect = db.rawQuery(SQL_SELECT_VALOR, filtroSelect);
```

Bases de datos: SQLite (IV)

- Recorriendo el resultado de la consulta

```
public Contactos recuperarCONTACTO(int id) {
    SQLiteDatabase db = getReadableDatabase();
    String[] valores_recuperar = {"_id", "nombre", "telefono",
"email"};
    Cursor c = db.query("contactos", valores_recuperar, "_id=" + id,
        null, null, null, null);
    if(c != null) {
        c.moveToFirst();
    }
    Contactos contactos = new Contactos(c.getInt(0), c.getString(1),
        c.getInt(2), c.getString(3));
    db.close();
    c.close();
    return contactos;
}
```

- Seleccionando el primer valor

```
if(cSelect.getCount() == 1)
{
    // Recuperamos el campo "valor" correspondiente al elemento
    "clave".
    cSelect.moveToFirst();
    resultado = cSelect.getString(0);
}
```

Bases de datos: SQLite (V)

- Insertando un nuevo registro

```
ContentValues newValues = new ContentValues();
newValues.put(NAME, "android");
newValues.put(DESCRIPTION, ...);
db.insert(DATABASE_TABLE, null, newValues);
//null es por defecto para indicar que no queremos un registro nulo
```

- Actualizando filas:

```
ContentValues updatedValues = new ContentValues();
updatedValues.put(NAME, "peter");
String where = KEY_ID + "=" + 1;
db.update(DATABASE_TABLE, updatedValues, where, null);
```

- Borrando filas

```
db.delete(DATABASE_TABLE, KEY_ID + "=" + rowId, null);
```

Content Provider (I)

- Proveen de contenidos a las aplicaciones y permite compartir datos entre ellas
- Desacopla la aplicación de la capa de datos
- Modelo URI simple para operaciones CRUD
- Muchas aplicaciones nativas publican sus proveedores de contenidos
 - ContactsProvider
 - MediaProvider
- También se pueden crear content provider nuevos

Content Provider (II)

- Un content provider implementa la interfaz
 - `query(Uri, String[], String, String[], String)`
 - `insert(Uri, ContentValues)`
 - `update(Uri, ContentValues, String, String[])`
 - `delete(Uri, String, String[])`
 - `getType(Uri)`
- El acceso a un content provider es gestionado por un content resolver
- El content provider analizará el resto de la URI mediante un UriMatcher

Content Provider (III)

- Un content provider expone una URI pública, la cual identifica un conjunto de datos, “una tabla”. La URIs de los proveedores comienzan por “**content://**”
- Normalmente un content provider expone dos URIs.:
 - `content://contacts/people` información de todas las personas
 - `content://contacts/people/1` información persona con `id = 1`



Content Provider (IV)

- Consultando un content provider (I)

```
String[] projection = new String[] {
    People._ID,
    People._COUNT,
    People.NAME,
    People.NUMBER
};
Uri contacts = People.CONTENT_URI;
Cursor cursor = managedQuery(contacts,
    projection, // Which columns to return
    null, // Which rows to return (all rows)
    null, // Selection arguments (none)
    People.NAME + " ASC"); // order asc by name
```

- Consultando un content provider (II)

```
Uri myPerson = ContentUris.withAppendedId(People.CONTENT_URI, 23);
Cursor cursor = managedQuery(myPerson, null, null, null, null);
```

Content Provider (V)

- Insertando datos en un content provider

```
ContentValues values = new ContentValues();
values.put(People.NAME, "Abraham Lincoln");
values.put(People.STARRED, 1);
Uri uri = getContentResolver().insert(People.CONTENT_URI, values);
```

Content Provider (VI)

- Ejemplo simple. Listado de contactos

```
<ScrollView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/contactos"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</ScrollView>

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView texto = (TextView) findViewById(R.id.contactos);
        Cursor c = managedQuery(ContactsContract.Contacts.CONTENT_URI,
        new String[]
            {ContactsContract.Contacts.DISPLAY_NAME}, ContactsContract.Contacts.IN_VISIBLE_GROUP,
        null, ContactsContract.Contacts.DISPLAY_NAME);
        while (c.moveToNext()) {
            String contactos = c.getString(c.getColumnIndex(ContactsContract.Data.DISPLAY_NAME));
            texto.append(contactos);
            texto.append("\n");
        }
    }
}
```

Objeto Application (I)

- La clase Application, es la clase base de la aplicación y puede ser usada para mantener un estado global. Se puede realizar una implementación propia de esta clase simplemente dándole el nombre en el fichero AndroidManifest.xml. Si tiene el nombre puesto, provocará que se cree una instancia de esa clase cuando se cree la aplicación

```
<application
    android:icon="@drawable/icon"
    android:label="@string/app_name"
    android:name="MiClaseAplicacion">

public class MiClaseAplicacion extends Application {

    @Override
    public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
    }
    @Override
    public void onCreate() {
        super.onCreate();
    }
    @Override
    public void onLowMemory() {
        super.onLowMemory();
    }
    @Override
    public void onTerminate() {
        super.onTerminate();
    }
}
```

Objeto Application (II)

- Los métodos básicos que tiene esta clase son:
 - onConfigurationChanged(). Lo llama el Sistema cuando cambia la configuración del dispositivo mientras que su componente se está ejecutando.
 - onCreate() Se llama cuando se inicia la aplicación, antes de que ningún otro objeto de la aplicación se cree.
 - onLowMemory(). Se llama cuando el Sistema completo está funcionando con poca memoria y quisiera ejecutar procesos para ajustarse.
 - onTerminate(). Se usa en entornos de procesos emulados. Nunca se llamaría en dispositivos Android en producción, donde los procesos se quitan simplemente matando el proceso.
- Esta clase está presente en todo momento y puede ser usada por cualquier actividad de la aplicación, de manera que lo que se guarde en ella puede ser accesible desde todas las actividades.
- Esto permite almacenar datos en ella y que sirvan de paso entre actividades en lugar de otros métodos de persistencia o de envío de objetos intents entre actividades

Objeto Application (III)

- Otro uso posible es usarlo como si fuera un patrón Singleton, es decir, un mecanismo para asegurar que sólo hay una instancia de la aplicación:

```
public class MiClaseAplicacion extends Application {  
    private static MiClaseAplicacion singleton;  
  
    public MiClaseAplicacion getInstance(){  
        return singleton;  
    }  
    @Override  
    public void onCreate(){  
        super.onCreate();  
        singleton = this;  
    }  
}
```

- Así cuando se ejecuta el método `getInstance()` nos devuelve la instancia de la aplicación que se creó y se guardó en la variable `singleton` en el método `onCreate()`

Objeto Application (IV)

- Para usarlo en cualquier Activity, simplemente declaramos una variable de esta clase, le damos el valor del objeto creado y ya podemos acceder a su contenido:

```
public class MiArmario extends Application {
    private String b = null;
    public MiArmario () {
        super();
    }
    public String getB(){
        return b;
    }
    public void setB(String newS){
        b = newS;
    }
}

public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        MiArmario app = (MiArmario) getApplication();
        if(app.getB()==null)
            app.setB("Primera Actividad Creada");
    }
}

public class SecondActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        MiArmario app = (MiArmario) getApplication();
        if(app.getB()!=null)
            Toast.makeText(getApplicationContext() , app.getB() , Toast.LENGTH_LONG).show();
    }
}
```

Objeto Application (V)

- Hay que tener cuidado con el uso de esta clase ya que los datos que se almacenen en ella no siempre pueden estar disponibles:
 - El objeto Application creado, no permanece en memoria para siempre. Si el usuario abandona la aplicación mediante el botón de Home, quedando ésta en segundo plano, puede ocurrir que si Android necesita memoria más adelante, de forma silenciosa mate la aplicación para conseguirlo ya que la prioridad de esta aplicación es baja en ese estado
 - Si el usuario reabre la aplicación, **Android crea una nueva instancia, distinta a la anterior y vacía de todo contenido.**
 - Si una Activity intenta obtener datos de ella puede que ya no estén y según como se accedan, puede provocar la caída de la aplicación.
 - Para evitar la caída hay que tener en consideración esta posibilidad