

# Práctica 1

---

## Emulación y pruebas con Android

## Tabla de contenido

<b>1. Objetivos .....</b>	<b>3</b>
<b>2. Android Studio: Fundamentos .....</b>	<b>4</b>
2.1. Configuración inicial del entorno.....	4
2.2. Hola Mundo: creación de una app básica .....	6
2.3. Despliegue de la app en un emulador o dispositivo físico.....	8
2.4. Estudio de la app desarrollada .....	9
• Recursos utilizados por la app.....	9
• La actividad principal de la app .....	10
• El manifiesto de la app .....	13
2.5. Depuración de la ejecución de la app .....	14
• Obtención de logs .....	14
• Inserción de puntos de parada o <i>breakpoints</i> .....	15
<b>3. Estudio de varias apps.....</b>	<b>16</b>
3.1. App con componentes básicos (Ver ComponentesBasicos.zip) .....	17
• Reacción a eventos generados por el usuario .....	18
• Activación de otra actividad.....	19
3.2. App lista de la compra (Ver ComponentesAvanzados.zip) .....	20
3.3. Almacenamiento de información (Almacenamiento.zip).....	25
• Gestión automática de la configuración de la app: uso de PreferenceScreen .....	26
• Almacenamiento interno de datos .....	28
• Almacenamiento externo de información .....	31
3.4. Comunicaciones HTTPS (ComunicacionesHTTPS.zip) .....	35
• Peticiones http con Volley.....	36
• Comunicaciones http utilizando HttpURLConnection.....	37
- Peticiones http utilizando Handlers .....	37
- Comunicaciones http utilizando AsyncTasks .....	38
<b>4. Estudio de un sencillo malware Android (AlmacenamientoMalo.zip).....</b>	<b>39</b>
<b>5. Ejercicios a realizar .....</b>	<b>41</b>
<b>6. Evaluación de los ejercicios .....</b>	<b>42</b>

## 1. Objetivos

Si no están correctamente gestionados, los dispositivos móviles son actualmente un vector de ataque más para acceder a los datos privados del usuario del dispositivo o a los activos de la empresa en la que dicho usuario trabaja.

Para entender cómo un ciberataque puede localizar una vulnerabilidad en un dispositivo móvil y explotarla con fines ilícitos es necesario entender no sólo cuáles son las tecnologías utilizadas en este tipo de terminales, sino también cómo las aplicaciones hacen uso de dichas tecnologías. Por ello, resulta fundamental conocer bien el ciclo de desarrollo de una aplicación móvil (una app), lo que incluye tanto las herramientas como los lenguajes de programación utilizados para su implementación, depuración, emulación, y despliegue en un dispositivo real.

Las apps desarrolladas para la plataforma Android son actualmente las más numerosas en el mercado y las que sufren un mayor número de ataques de todo tipo según los informes que varias consultoras especializadas han emitido. Estas apps se programan en Java o Kotlin y, aunque pueden desarrollarse utilizando otros IDEs, el entorno “oficial” de desarrollo promovido por Google es *Android Studio*.

En esta práctica nos familiarizaremos con *Android Studio* y desarrollaremos una app Android en Java muy sencilla, la app *HolaMundo*. A través de esta app, en la Sección 2 de la práctica, veremos cómo compilar y lanzar a ejecución una aplicación móvil sirviéndonos de un emulador. También veremos cómo depurar dicha ejecución y cómo obtener trazas (*logs*) de su actividad. Finalmente, analizaremos el uso de la herramienta *adb* (*Android Debug Bridge*), que facilita la administración de un dispositivo o emulador desde el ordenador, permitiéndonos, entre otras cosas, instalar y borrar aplicaciones en el mismo u obtener volcados de memoria de gran interés a la hora de plantear un análisis forense.

En lo relativo a la programación de apps Android, y a pesar de lo complejo y extenso de la temática, la práctica ofrecerá una visión general de las principales funcionalidades que hay que conocer para entender cómo funcionan este tipo de aplicaciones. En la sección 3 analizaremos cómo las apps Android gestionan la interacción con los usuarios y la comunicación entre las distintas *Activities* que las conforman mediante *Intents*. La sección 4 nos llevará a estudiar cómo la plataforma gestiona los permisos y cómo las aplicaciones deben hacer uso de los mismos. Así mismo se planteará el código de un malware, que se servirá de un Servicio que se ejecutará en segundo plano y accederá de forma ilícita a los contactos del usuario para remitirlos a un servidor remoto. La práctica terminará en la sección 5 donde cada grupo de trabajo deberá dar respuesta a las preguntas planteadas con el objetivo de ver si los conceptos básicos requeridos han sido retenidos.

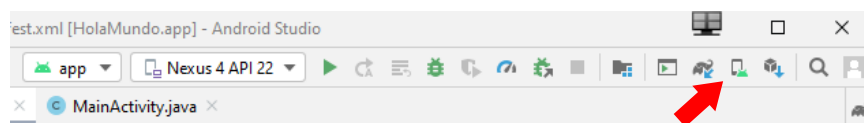
## 2. Android Studio: Fundamentos

Como ya se ha mencionado anteriormente, el entorno de desarrollo privilegiado para la programación de aplicaciones Android es Android Studio. Esta sección pretende introducir al alumno en su uso de manera rápida, pero eficaz.

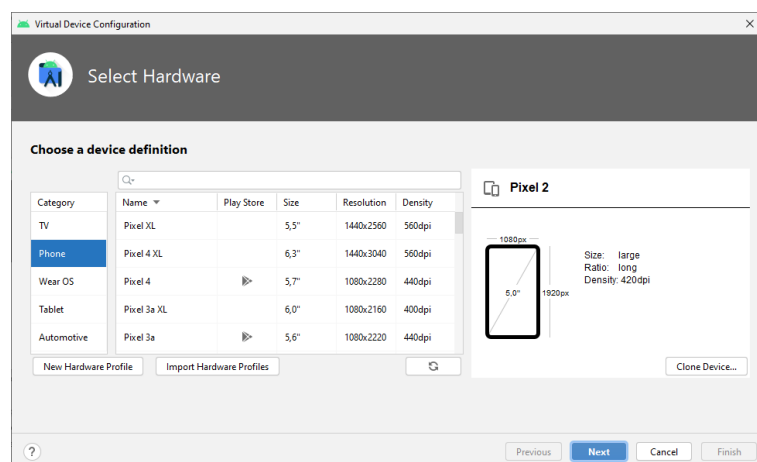
### 2.1. Configuración inicial del entorno

Lo primero que vamos a necesitar es instalar Android Studio. Para ello haremos uso de los tutoriales disponibles en <https://developer.android.com/studio/install> y que se detallan tanto para Windows, como para Mac, Linux y Chrome OS.

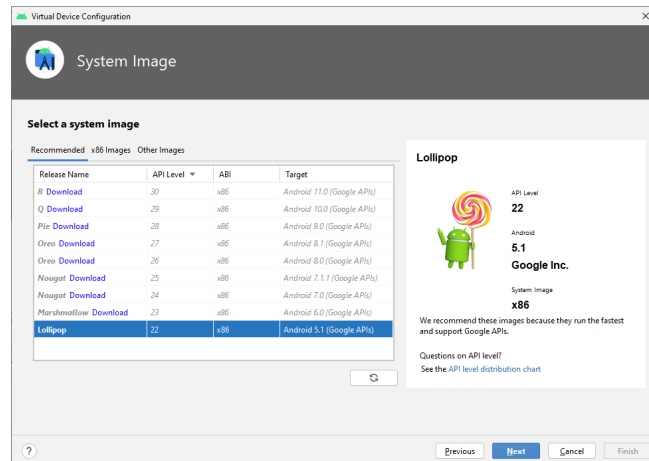
Una vez instalado el entorno de desarrollo, llega el momento de crear un emulador para trabajar. Para ello abriremos el AVDM (por *Android Virtual Device Manager*) haciendo click en la opción del entorno que señala la flecha roja en la siguiente figura:



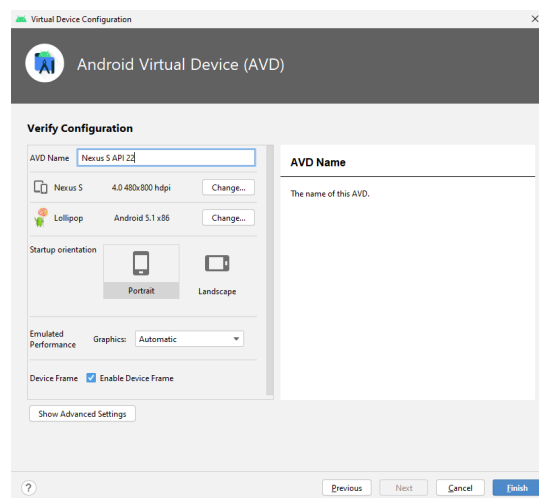
Solicitamos la creación de un dispositivo virtual (opción *Create Virtual Device*) haciendo click en el botón correspondiente. Esto nos llevará a la siguiente pantalla, en la que seleccionaremos el tipo de dispositivo que deseemos emular:



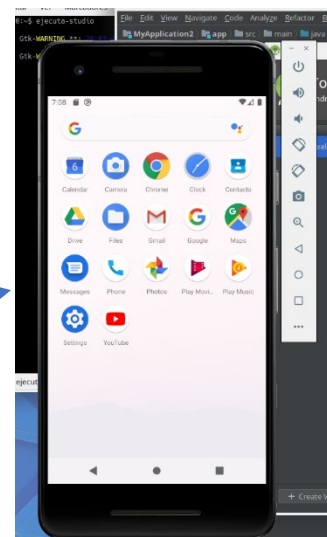
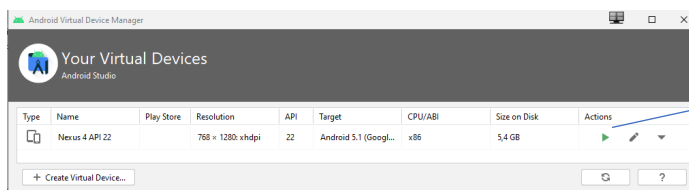
Seleccionaremos la versión de Android que deseemos utilizar en nuestras pruebas. En la pestaña *x86 Images* tendremos un listado completo de todos los sistemas disponibles, incluyendo versiones de cada versión de Android con y sin Google APIs. Elegid la que deseéis, por ejemplo, la versión Lollipop con Google APIs.



Finalmente daremos nombre al dispositivo y terminaremos:

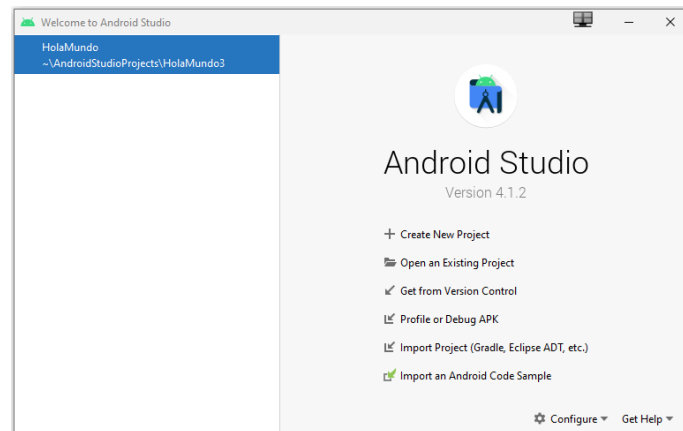


Si todo ha ido bien, deberíamos haber llegado ya a crear un dispositivo virtual. Desde el *Android Virtual Device Manager* simplemente tendremos que lanzarlo a ejecución, lo que pondrá en funcionamiento el emulador:

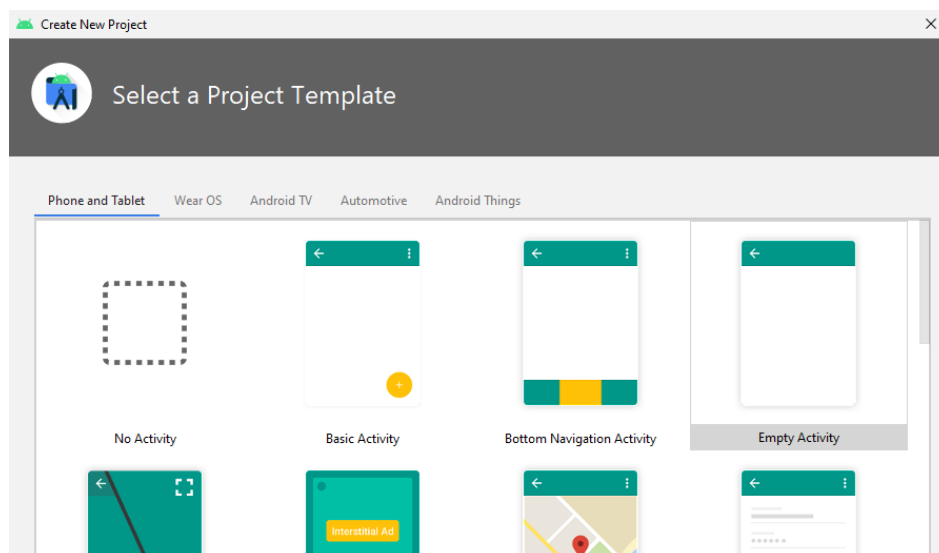


## 2.2. Hola Mundo: creación de una app básica

De las múltiples opciones que nos ofrece la pantalla de bienvenida del entorno de desarrollo seleccionaremos la opción “Start a new Android Studio Project”.

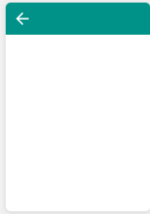


A continuación elegiremos como proyecto un proyecto “Phone and Tablet” y dentro del mismo una “Empty Activity”, tal y como se muestra en la siguiente imagen:



Si eligiéramos la actividad básica o cualquier otra la aplicación que se desplegaría sería un poco más compleja que la que vamos a considerar, ya que utilizaría *Frames* para poder ofrecer al usuario en una misma interfaz distintas ventanas de interacción, cada una con su propio comportamiento. Este aspecto de las aplicaciones móviles no se ha estudiado en clase, pero está bien saber que existe. A continuación, configura tu proyecto:

Configure your project



Empty Activity

Creates a new empty activity

Name

My Application

Package name

com.example.jcruizg.cdm.myapplication

Save location

/Users/jcruizg/Development/Android/Proyectos/MyApplication5

Language

Java

Minimum API level

API 21: Android 5.0 (Lollipop)

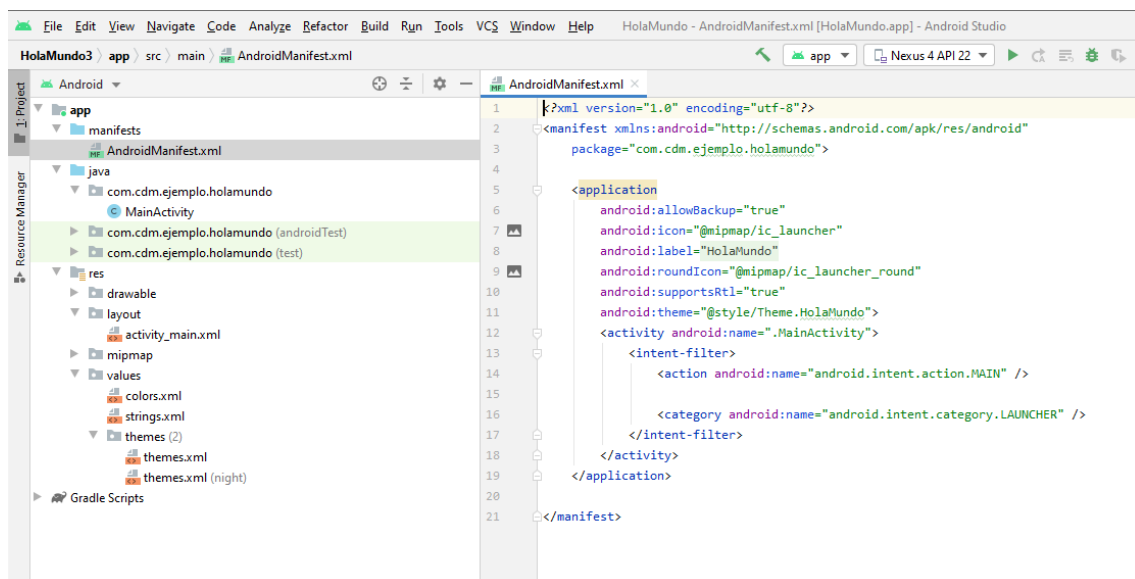
☒ Your app will run on approximately **85.0%** of devices.  
[Help me choose](#)

☐ This project will support instant apps

☐ Use AndroidX artifacts

Ten en cuenta que, en el ejemplo de configuración mostrada, se está eligiendo como lenguaje de programación Java y que el API de desarrollo seleccionada es Lollipop (Android 5.0). Esto último se hace así para que la app que vamos a desarrollar pueda ejecutarse en un amplio número de teléfonos y tablets (85% de los existentes actualmente). Podríamos haber elegido también el API 19, también conocida como KitKat (Android 4.4), y alcanzar así del 98% de los terminales Android en funcionamiento. Como ves, la elección del API mínima requerida para la ejecución de la aplicación es siempre responsabilidad de su programador.

Con la configuración definida, Android Studio genera automáticamente un proyecto que directamente puede ser ejecutado en un dispositivo móvil o emulador y que tiene, más o menos, el siguiente aspecto:



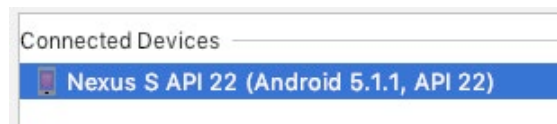
```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.cdm.ejemplo.holamundo">
4
5     <application
6         android:allowBackup="true"
7         android:icon="@mipmap/ic_launcher"
8         android:label="HolaMundo"
9         android:roundIcon="@mipmap/ic_launcher_round"
10        android:supportRtl="true"
11        android:theme="@style/Theme.HolaMundo">
12        <activity android:name=".MainActivity">
13            <intent-filter>
14                <action android:name="android.intent.action.MAIN" />
15
16                <category android:name="android.intent.category.LAUNCHER" />
17            </intent-filter>
18        </activity>
19    </application>
20
21 </manifest>
```

### 2.3. Despliegue de la app en un emulador o dispositivo físico

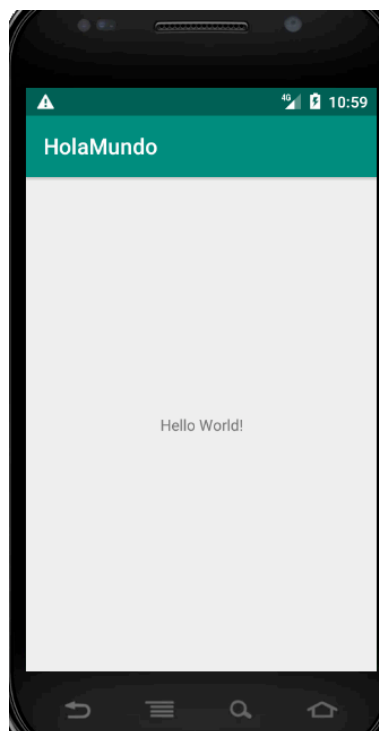
Como ya sabes, las aplicaciones de Android se pueden escribir en lenguaje de programación Java o Kotlin. Para la realización de esta práctica estamos considerando que lo están en Java. Las herramientas de Android SDK compilan tu código, junto con los archivos de recursos y datos, en un APK: un paquete de Android, que es un archivo de almacenamiento con el sufijo .apk. Un archivo de APK incluye todos los contenidos de una aplicación de Android y es el archivo que usan los dispositivos con tecnología Android para instalar la aplicación.



Si pulsamos sobre el botón de ejecutar la app (Ver imagen anterior) se nos pedirá sobre qué dispositivo deseamos desplegar nuestro proyecto. Si tenemos un dispositivo físico conectado a nuestra máquina éste aparecerá en el listado de dispositivos disponibles, pero si esto no es así, siempre podremos utilizar un emulador, como el que aparece en la siguiente imagen:



Para más información al respecto del despliegue de apps en dispositivos físicos podéis consultar la URL: <https://developer.android.com/studio/run/device>. Para crear un emulador y desplegar sobre el mismo una app existe una URL alternativa que podéis tomar como referencia: <https://developer.android.com/studio/run/emulator>. El resultado será algo como esto:





Recuerda que, tal y como hemos visto en clase, una vez instalada en el dispositivo, cada aplicación de Android se aloja en su propia zona de pruebas de seguridad, en su propio proceso. De esta manera, el sistema Android implementa el *principio de mínimo privilegio*. Es decir, de forma predeterminada, cada aplicación tiene acceso sólo a los componentes que necesita para llevar a cabo su trabajo y nada más. Esto crea un entorno muy seguro en el que una aplicación no puede acceder a partes del sistema para las que no tiene permiso.

Sin embargo, hay maneras en las que una aplicación puede compartir datos con otras aplicaciones y en las que una aplicación puede acceder a servicios del sistema. Una aplicación puede solicitar permiso para acceder a datos del dispositivo como los contactos de un usuario, los mensajes de texto, el dispositivo de almacenamiento (tarjeta SD), la cámara, Bluetooth y más. El usuario debe garantizar de manera explícita estos permisos. Más tarde en esta práctica veremos cómo trabajar con permisos del sistema.

Esto cubre los aspectos básicos sobre cómo una aplicación de Android existe en el sistema y cómo puede ser ejecutada en un dispositivo móvil físico o en un emulador. Pasemos ahora a estudiar los elementos que resultan fundamentales en el desarrollo de toda app Android.

## 2.4. Estudio de la app desarrollada

En primer lugar, decir que, aunque una app Android puede contener tanto Actividades, como Servicios, Receptores de Mensajes y Proveedores de contenido, la que vamos a estudiar en esta práctica sólo contiene Actividades ya que para abordarlo todo harían falta varias prácticas y no disponemos de tanto tiempo. Sin embargo, si deseas profundizar y saber algo más sobre estos componentes te sugiero visitar la siguiente URL: <https://developer.android.com/guide/components/fundamentals>.

En nuestro caso, la app *HolaMundo* que hemos generado consta de una única Actividad cuya interfaz (capa de presentación) se define en el fichero de layout *activity\_main.xml* y cuyo código (capa de negocio) se implementa en fichero *MainActivity.java*.

- Recursos utilizados por la app

Todo layout es un recurso, y por ello, el layout *activity\_main.xml* se almacena en el directorio *res/layout* del proyecto. Los strings, colores y estilos que utilice la aplicación, se almacenarán en el directorio *res/values*, y más concretamente en los ficheros *strings.xml*, *colors.xml* y *styles.xml* según el caso. Por otro lado, las imágenes se guardarán en el directorio *res/drawable*, y lo harán en distintas resoluciones para utilizar la más adecuada en función de las características concretas de la pantalla de cada dispositivo. Para más información acerca de la definición y uso de los distintos tipos de recursos que puede utilizar una app Android os sugiero visitar la URL <https://developer.android.com/guide/topics/resources/overview?hl=es-419>.

Centrándonos en nuestra app, el fichero de layout *activity\_main.xml* consta sólo de un *TextView* con el texto “Hello World!” y está asociado a la actividad de nombre *MainActivity* tal y como puede verse a continuación:

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      package="com.cdm.ejemplo.holamundo">
4
5      <application
6          android:allowBackup="true"
7          android:icon="@mipmap/ic_launcher"
8          android:label="@string/app_name"
9          android:roundIcon="@mipmap/ic_launcher_round"
10         android:supportsRtl="true"
11         android:theme="@style/Theme.HolaMundo">
12         <activity android:name=".MainActivity">
13             <intent-filter>
14                 <action android:name="android.intent.action.MAIN" />
15
16                 <category android:name="android.intent.category.LAUNCHER" />
17             </intent-filter>
18         </activity>
19     </application>
20
21 </manifest>

```

También podemos observar que en el fichero strings.xml que el entorno ha generado se define una cadena de caracteres que da el nombre a nuestra la app:

```

1  <resources>
2      <string name="app_name">HolaMundo</string>
3  </resources>
4

```

Si modificamos dicho nombre y re-ejecutamos la app observaremos el cambio.

Cabe señalar que la compilación del proyecto se traduce en la generación automática de una clase llamada *R* que permitirá a los programas acceder a los recursos almacenados por el desarrollador en el directorio *res/*. Por ejemplo, para acceder al layout definido en el fichero *activity\_main.xml* escribiremos *R.layout.activity\_main* en nuestro programa y para acceder al string *app\_name* que hemos visto utilizaremos el identificador *R.string.app\_name*.

- La actividad principal de la app

La clase *MainActivity.java* define el punto de entrada a nuestra aplicación móvil. Su código es el siguiente:

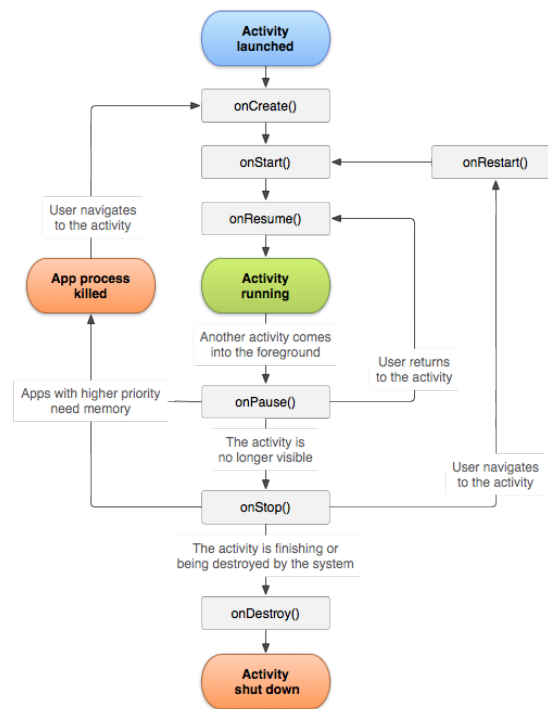
```

1  package com.example.jcruizg.cdm.holamundo;
2
3  import ...
4
5
6  public class MainActivity extends AppCompatActivity {
7
8      @Override
9      protected void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11         setContentView(R.layout.activity_main);
12     }
13 }
14

```

El ciclo de vida de una Actividad Android se gestiona, tal y como se menciona en <https://developer.android.com/guide/components/activities>, a través de una serie de call-backs, de los cuales *onCreate* es el primero en ser invocado (ver figura de al lado). Es por esto

que este método constituye el punto de entrada a la actividad en cuestión, en nuestro caso la principal de la app. Los métodos *onStart* y *onResume* son dos métodos que también puedes sobre-escribirse y que se invocan secuencialmente cuando una Activity se hace visible al usuario (*onStart*) y cuando está preparada para poder interactuar con el mismo (*onResume*). Contrariamente, cuando una Activity deja de ser interactiva se ejecuta el método *onPause*, cuando ya no está visible el método *onStop* y cuando finaliza su ejecución el método *onDestroy*. Todos se encadenan tal y como se muestra en la siguiente image:



Lo primordial en el método *onCreate* es asociar el layout que se desea utilizar con la actividad. Eso es lo que se hace con la llamada a *setContentView* pasándole el identificador del layout deseado, el layout *R.layout.activity\_main* en nuestro caso. Eso es lo que se ve en el código que se ha mostrado anteriormente.

Si deseáramos cambiar el mensaje que muestra la app, deberíamos proceder como sigue:

1. Generar un string con el mensaje a mostrar. En Android se recomienda siempre trabajar con recursos de tipo string en lugar de hacerlo con strings introducidos directamente en el código. Para definir nuestro string editamos el fichero *res/strings.xml* e introducimos un nuevo string con el mensaje a mostrar. En la siguiente figura tenéis un ejemplo de cómo hacerlo:

```

strings.xml x
Edit translations for all locales in the translations editor.
1 <resources>
2   <string name="app_name">HolaMundo</string>
3   <string name="mensaje">Hola Mundo</string>
4 </resources>
5
  
```

2. Reabrimos el layout de la actividad (fichero *activity\_main.xml*) y añadimos el mensaje que deseamos mostrar. Para ello tenemos dos alternativas:

2.1. Podemos cambiar el texto del atributo *android:text* del *TextView* por el identificador del string creado (*@string/mensaje*). Esta asignación es estática, con lo que el string asignado no se puede modificar a lo largo de la ejecución de la app. En este caso, la definición del *TextView* quedaría como sigue:

```
<TextView
    android:layout_width="3dp"
    android:layout_height="wrap_content"
    android:text="@string/mensaje"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

2.2. Alternativamente, podríamos asignar dinámicamente al *TextView* el texto que deseamos mostrar. Para ello necesitamos dotar de un identificador al *TextView* que será luego utilizado en el programa para poder modificar el texto mostrado. El identificador es un atributo más del *TextView* y podemos ponerle el nombre que queramos. Por ejemplo, vamos a utilizar el siguiente identificador *android:id="@+id/tvMensaje"*, con lo que la definición de la vista quedará así:

```
<TextView
    android:id="@+id/tvMensaje"
    android:layout_width="3dp"
    android:layout_height="wrap_content"
    android:text="@string/mensaje"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

A continuación, incluiremos en el método *onCreate* de la actividad la referencia al *TextView* y luego le asignaremos el texto a mostrar. Para lo primero utilizamos el método *findViewById* que nos permite obtener una referencia a una vista a partir de su identificador. Para lo segundo, simplemente tendremos que utilizar la referencia obtenida y solicitar a la misma el cambio del texto que muestra (llamada a *setText*). Obviamente, al método invocado no se proporcionaremos un string, sino el identificador al recurso string que hemos creado. El código de la actividad quedaría tal y como se muestra a continuación:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView s = findViewById(R.id.tvMensaje);
        s.setText(R.string.mensaje);
    }
}
```

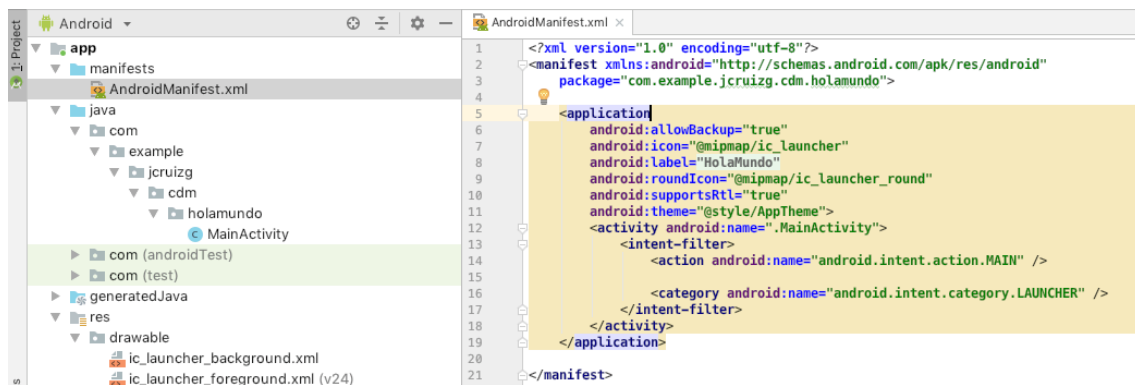
## • El manifiesto de la app

Todas las aplicaciones deben tener un archivo *AndroidManifest.xml* (con ese nombre exacto) en el directorio raíz de su proyecto. El archivo de manifiesto proporciona información esencial sobre la aplicación al sistema Android, información que el sistema debe tener para poder ejecutar el código de la app.

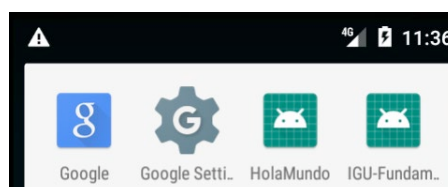
Tal y como se explica en <https://developer.android.com/guide/topics/manifest/manifest-intro?hl=es-419>, entre otras cosas, el archivo de manifiesto hace lo siguiente:

- Nombra el paquete de Java para la aplicación. El nombre del paquete sirve como un identificador único para la aplicación.
- Describe los componentes de la aplicación, como las actividades, los servicios, los receptores de mensajes y los proveedores de contenido que la integran. También nombra las clases que implementa cada uno de los componentes y publica sus capacidades, como los mensajes *Intent* a los que puede reaccionar. Estas declaraciones notifican al sistema Android los componentes y las condiciones para el lanzamiento.
- Declara los permisos que debe tener la aplicación para acceder a las partes protegidas de una API e interactuar con otras aplicaciones. También declara los permisos que otros deben tener para interactuar con los componentes de la aplicación.
- Declara el nivel mínimo de Android API que requiere la aplicación.

En el caso de nuestra app, el fichero de manifiesto tiene el siguiente aspecto:



En este fichero se indica que el paquete en el que se define nuestra app es el paquete *com.example.jcruizg.cdm.holamundo*, que la app tendrá como icono la imagen *ic\_launcher\_round* guardada como recurso en el directorio *res/mipmap*, que su actividad principal (la que responderá al intent *android.intent.action.MAIN*) es la actividad *MainActivity* y que a esta actividad se le debe asociar un icono que sirva de lanzadera en el panel de aplicaciones. Véase el icono que se genera para nuestra app *HolaMundo*:



## 2.5. Depuración de la ejecución de la app

Ahora que ya sabemos cuál es la estructura de nuestra app, y podemos leer su código, vamos a ver qué es lo que deberíamos hacer si deseamos saber lo que la app hace con fines de depuración.

- Obtención de logs

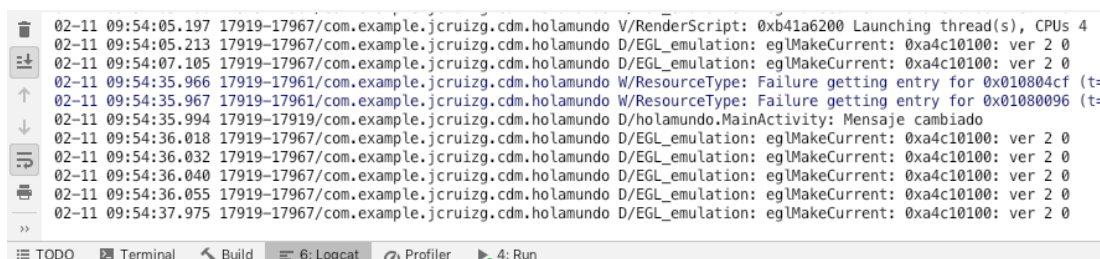
Normalmente introducimos en el código diversas llamadas a la función *printf* para obtener una traza del programa por consola. El problema es que en la noción de app no existe la noción de consola, con lo que la obtención de trazas debe hacerse de otra manera.

Para ello se ofrece la clase *Log* y el monitor *Logcat* que ofrece la utilidad *Android Monitor* que incluye el entorno de desarrollo. En la URL <https://developer.android.com/studio/debug/am-logcat?hl=es-419> se describe no sólo cómo funciona logcat, sino también la facilidades que nos ofrece a la hora de visualizar, filtrar y buscar logs (o mensajes de registro) en las trazas obtenidas.

A nivel de *programa*, todos los logs de Android tienen una etiqueta y una prioridad asociadas a ellos. La etiqueta de un mensaje de registro del sistema es una string breve que indica el componente del sistema a partir del cual se origina el mensaje (por ejemplo, *ActivityManager*). Esta etiqueta puede ser definida por el usuario utilizando cualquier string que te resulte útil; por ejemplo, el nombre de la clase actual (la etiqueta recomendada). Respetto a la prioridad, esta indica el tipo de mensaje que se está generando y puede ser V (detalle, es la prioridad más baja), D (depuración), I (información), W (advertencia), E (error), A (aserción).

Las imágenes que se incluyen a continuación muestran cómo introducimos en nuestra app un mensaje de depuración (*Log.d*) y cómo lo vemos a través del logcat cuando la ejecutamos. Cabe señalar que la clase *Log* se define en el paquete *android.util*, que debe ser importado.

Traza obtenida a través de *Logcat*. Observad que en una de las líneas se indica el cambio de mensaje del *TextView* que hemos realizado anteriormente:



```
02-11 09:54:05.197 17919-17967/com.example.jcruizg.cdm.holamundo V/RenderScript: 0xb41a6200 Launching thread(s), CPUs 4
02-11 09:54:05.213 17919-17967/com.example.jcruizg.cdm.holamundo D/EGL_emulation: eglMakeCurrent: 0xa4c10100: ver 2 0
02-11 09:54:07.105 17919-17967/com.example.jcruizg.cdm.holamundo D/EGL_emulation: eglMakeCurrent: 0xa4c10100: ver 2 0
02-11 09:54:35.966 17919-17961/com.example.jcruizg.cdm.holamundo W/ResourceType: Failure getting entry for 0x010804cf (t=
02-11 09:54:35.967 17919-17961/com.example.jcruizg.cdm.holamundo W/ResourceType: Failure getting entry for 0x01080096 (t=
02-11 09:54:35.994 17919-17919/com.example.jcruizg.cdm.holamundo D/holamundo.MainActivity: Mensaje cambiado
02-11 09:54:36.018 17919-17967/com.example.jcruizg.cdm.holamundo D/EGL_emulation: eglMakeCurrent: 0xa4c10100: ver 2 0
02-11 09:54:36.032 17919-17967/com.example.jcruizg.cdm.holamundo D/EGL_emulation: eglMakeCurrent: 0xa4c10100: ver 2 0
02-11 09:54:36.040 17919-17967/com.example.jcruizg.cdm.holamundo D/EGL_emulation: eglMakeCurrent: 0xa4c10100: ver 2 0
02-11 09:54:36.055 17919-17967/com.example.jcruizg.cdm.holamundo D/EGL_emulation: eglMakeCurrent: 0xa4c10100: ver 2 0
02-11 09:54:37.975 17919-17967/com.example.jcruizg.cdm.holamundo D/EGL_emulation: eglMakeCurrent: 0xa4c10100: ver 2 0
```

El código introducido en la clase *MainActivity* para obtener este log consiste en una simple llamada a *Log.d*, tal y como se muestra a continuación:



```

1 package com.example.jcruizg.cdm.holamundo;
2
3 import android.support.v7.app.AppCompatActivity;
4 import android.os.Bundle;
5 import android.widget.TextView;
6 import android.util.Log;
7
8 public class MainActivity extends AppCompatActivity {
9
10     @Override
11     protected void onCreate(Bundle savedInstanceState) {
12         super.onCreate(savedInstanceState);
13         setContentView(R.layout.activity_main);
14         TextView s = findViewById(R.id.tvMensaje);
15         s.setText(R.string.mensaje);
16         Log.d( tag: "holamundo.MainActivity", msg: "Mensaje cambiado");
17     }
18 }

```

### • Inserción de puntos de parada o *breakpoints*

Como en cualquier otro entorno de desarrollo, las apps desarrolladas en Android pueden ser depuradas utilizando puntos de parada (breakpoints) y estudiando el estado de la app en los mismos.

Para insertar un breakpoint en el código de la app, simplemente tendremos que hacer click con el ratón en el margen izquierdo de la línea de código donde deseamos posicionar dicho breakpoint. Tal y como se muestra en la siguiente imagen, aparecerá un punto rojo para indicarnos que allí se ha colocado un breakpoint.

```

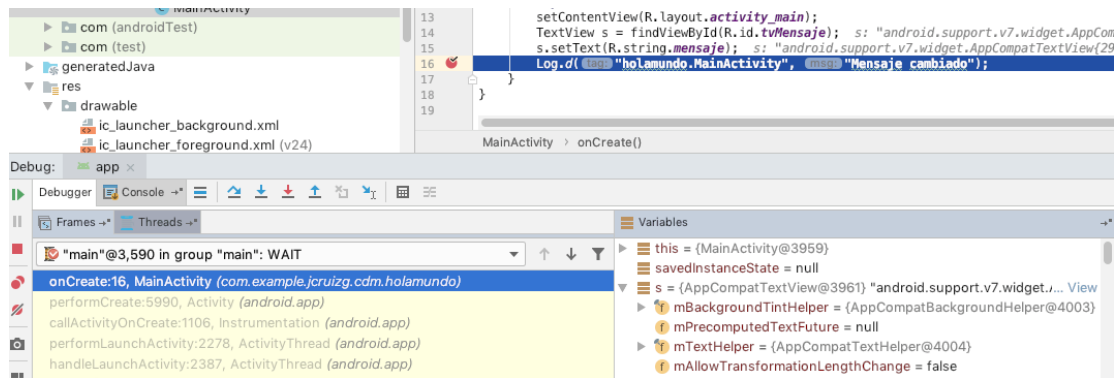
10
11
12
13
14
15
16
17
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    TextView s = findViewById(R.id.tvMensaje);
    s.setText(R.string.mensaje);
    Log.d( tag: "holamundo.MainActivity", msg: "Mensaje cambiado");
}

```

Con los breakpoints convenientemente introducidos en el código ejecutaremos la app en modo depuración. Esto puede hacerse utilizando la opción del menú Run > Debug 'app', el combo asociado a la misma (Ctrl+D) o simplemente haciendo click en el símbolo de depuración (un icono con forma de insecto) que ofrece la barra de herramientas del entorno. El símbolo en cuestión se muestra en la siguiente imagen:

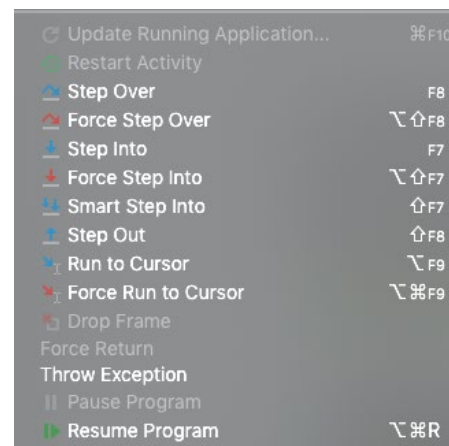


Cuando estemos depurando hay que tener en cuenta que la ejecución de la app será un poco lenta, aunque más rica en cuanto a la información que obtenemos de la app, tal y como se aprecia en la siguiente captura de pantalla, que se corresponde con la activación del breakpoint que hemos introducido en nuestra app:



Esto nos va a resultar especialmente interesante cuando analicemos una app, ya que podemos ver cómo evolucionan su estado en función de las entradas que recibe.

Al activar un breakpoint, podremos decidir si lo que deseamos es continuar desde el mismo con una ejecución paso a paso, definir otro breakpoint y continuar hasta el mismo, o simplemente continuar con la ejecución de la app hasta que el breakpoint que ya tenemos vuelva a activarse o la app simplemente termine su ejecución. Estas opciones y otras que el entorno ofrece (ver imagen al lado) están disponibles a través del menú *Run* del entorno y de los iconos que a tal fin aparecen cuando el código se ejecuta en modo depuración.



### 3. Estudio de varias apps

Aunque, no podríamos desarrollar nuestra propia app con lo que hemos visto hasta ahora, lo que sí que podríamos hacer es entender lo que una app hace y, lo más importante, si desensambláramos una app podríamos utilizar Android Studio para inspeccionar su código y los recursos que utiliza, depurarlo para comprenderlo mejor, e incluso ejecutarlo en un emulador o en un dispositivo físico.

Sin embargo, todavía hay muchas cosas que desconocemos de las apps Android y que son necesarias cuando abordamos el estudio de dichas apps desde la perspectiva de su seguridad. Cosas como ¿cómo interactúan las actividades con los usuarios de las apps? ¿cómo comunican las actividades entre sí? ¿cómo almacenan datos y qué permisos necesitan? ¿cómo gestionan las comunicaciones remotas? ¿cómo geolocalizan a sus usuarios? son cuestiones básicas para las que hay que tener una comprensión mínima si queremos poder descubrir vulnerabilidades en las aplicaciones o usos fraudulentos de sus recursos y/o permisos de ejecución.

Esta sección de la práctica se centra en abordar estos aspectos a través del estudio de 3 apps ya desarrolladas y cuyo código se suministra para su estudio.

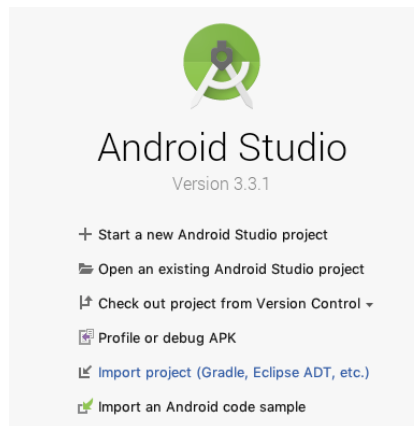


### 3.1. App con componentes básicos (Ver ComponentesBasicos.zip)

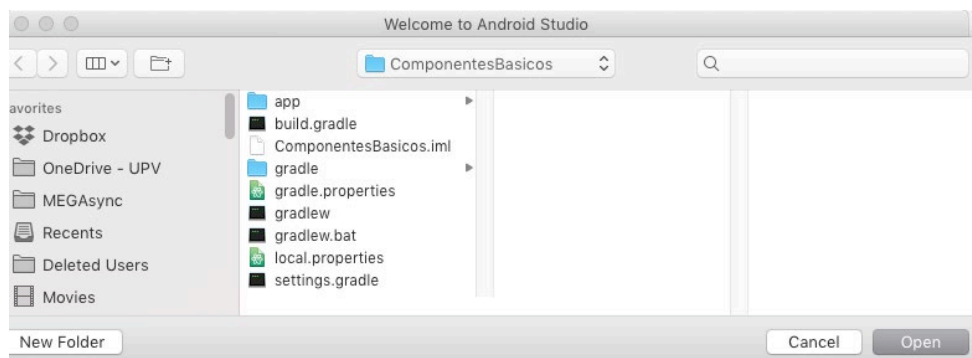
Ésta es una app muy sencilla que, además de mostrar cómo se utilizan los componentes más básicos para definir un IGU en Android, nos permitirá entender cómo podemos definir una app con más de una actividad, cómo podemos reaccionar a un evento generado por el usuario y cómo es posible activar una actividad desde otra.

La app se suministra en un zip, ComponentesBasicos.zip, que debe descomprimirse para poder ser utilizado. Luego lo importaremos en Android Studio siguiendo los siguientes dos pasos:

1. Usamos la opción “Import Project” de la pantalla de Bienvenida de Android Studio:



2. Seleccionar como origen de la importación el directorio que contiene el proyecto ComponentesBasicos y pulsar sobre Open.



Se recomienda ejecutar el proyecto para entender lo que hace la aplicación. Ésta posee sólo dos actividades. La primera tiene una interfaz que ofrece varios componentes. Su objetivo no es el de ofrecer una funcionalidad muy sofisticada, sino el de mostrar cómo pueden utilizarse distintas vistas Android (de tipo View), tales como `Checkbox`, `TextView`, `Button`, `RadioGroup`, `ImageView`, `EditText`, etc, para definir una IGU. La definición de las interfaces utilizadas puede consultarse en los ficheros de recursos `res/layout/igu_componentes_basicos.xml` y `res/layout/activity_about.xml`. Las dos actividades que gestionarán estas interfaces son las actividades definidas en los ficheros `ComponentesBasicosActivity.java` y `AboutActivity.java`.

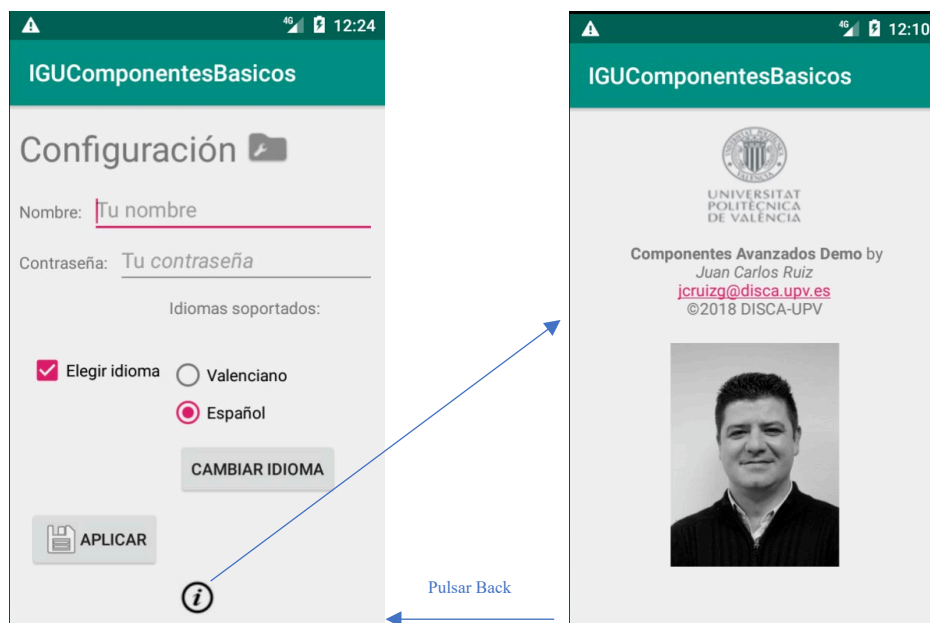
En cuanto al manifiesto de la app, refleja la definición de ambas actividades e informa al sistema de que la principal de ellas, es decir la que se activará al iniciar la app, es la actividad *ComponentesBasicosActivity*.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3    package="com.example.jcruizg.igucomponentesbasicos">
4
5    <application
6      android:allowBackup="true"
7      android:icon="@mipmap/ic_launcher"
8      android:label="IGUComponentesBasicos"
9      android:roundIcon="@mipmap/ic_launcher_round"
10     android:supportRtl="true"
11     android:theme="@style/AppTheme">
12     <activity android:name=".ComponentesBasicosActivity">
13       <intent-filter>
14         <action android:name="android.intent.action.MAIN" />
15
16         <category android:name="android.intent.category.LAUNCHER" />
17       </intent-filter>
18     </activity>
19     <activity android:name=".AboutActivity" />
20   </application>
21
22 </manifest>

```

En cuanto a las actividades, su apariencia es la siguiente:



- **Reacción a eventos generados por el usuario**

Además de lo ya mencionado, el código nos muestra cómo programar la reacción a eventos, mayoritariamente clicks, que puede producir el usuario al interactuar con la app. Para ello definiremos un manejador por cada evento a gestionar. Normalmente, estos manejadores serán métodos que asociaremos al listener de cada vista que pueda generar dicho evento. Por ejemplo, para reaccionar a la pulsación de un botón, hay que sobre-escribir el método *onClick* de la interfaz de escucha (listener) *View.OnClickListener*, instanciar un objeto de dicho tipo y asociárselo a la vista correspondiente (un botón en nuestro ejemplo) utilizando su método *setOnClickListener*. De esa forma, cuando el botón detecte un click, activará el manejador del evento *onClick* del listener que le hayamos suministrado.

Esto es lo que ocurre en el *ImageView* que contiene la vista definida en la interfaz *igu\_componentes\_basicos.xml* y que se implementa en el fichero

`ComponentesBasicosActivity.java`. En dicho fichero, apreciamos que en el método `onCreate` de la clase (actividad) `ComponentesBasicosActivity` aparece el siguiente código:

```

46
47
48
49
50
51
52
    findViewById(R.id.iv_About).setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            startActivity(new Intent(getApplicationContext(), AboutActivity.class));
        }
    });

```

En dicho código se obtiene con `findViewById` una referencia al `ImageView` de la interfaz y se asocia al dicha vista, utilizando el método `setOnClickListener`, un `View.OnClickListener` que sobrescribe el método `onClick`.

Otro ejemplo similar es el del botón Aplicar, que cuando es pulsado muestra un mensaje (el que genera la llamada a `Toast.makeText`) que informa al usuario sobre el estado actual de los distintos componentes de la inferfaz (nombre y contraseña introducidos e idioma elegido). La siguiente imagen muestra el punto del código en el que esto sucede:

```

((Button)findViewById(R.id.btn_AplicarConf)).setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        String idiomaElegido = "Idioma elegido: [DEFAULT:Español]";
        if (cbElegirIdioma.isChecked()) idiomaElegido = "Idioma elegido: "+idiomaActualmenteElegido;
        String nombreUsuario = "Nombre: " + ((EditText)findViewById(R.id.et_Nombre)).getText().toString();
        //
        // TENEMOS MUCHO CUIDADO A LA HORA DE MOSTRAR LA CONTRASEÑA PARA NO MOSTRAR LOS CARACTERES INTRODUCIDOS POR EL USUARIO, YA QUE ÉSTOS
        // SON SECRETOS. LO QUE HACEMOS ES REEMPLAZAR CADA UNO DE LOS CARACTERES POR EL CARACTER "?"
        //
        String contrasenaUsuario = "Contraseña: " +
            ((EditText)findViewById(R.id.et_Contrasena)).getText().toString().replaceAll(regex: ".", replacement: "?");
        Toast.makeText(getApplicationContext(), "text: nombreUsuario+"+"\n"+contrasenaUsuario+"n"+idiomaElegido, Toast.LENGTH_SHORT).show();
    }
});

```

Como podéis apreciar, el código de la aplicación está completamente comentado para facilitar su lectura e interpretación. Si surgieran dudas en cuanto a lo que se hace o cómo se hace no dudes en preguntar.

### • Activación de otra actividad

En uno de los ejemplos que acabamos de ver, el método `onClick`, instancia una nueva actividad, la actividad `AboutActivity`, y solicita al sistema su visualización. Esto se hace a través de la llamada al método `startActivity`. Este método requiere que se le suministre un `Intent`, que no es otra cosa que un mensaje que se enviará al sistema indicándole el contexto de ejecución de la app (obtenido a través de `getApplicationContext`) y la actividad a activar (indicada explícitamente a través de la clase `AboutActivity.class`). Es así como debemos proceder en general para activar una actividad desde otra actividad.

Cabe recordar que, tal y como se ha visto en clase, cuando se produce el cambio de una actividad A a otra B, la actividad A no se destruye si no hay falta de memoria en el dispositivo, sino que se apila en la pila de actividades del sistema. Cuando la actividad B finaliza su ejecución, algo que sucede cuando, por ejemplo, el usuario pulsa el botón Back del dispositivo, la actividad A volverá a activarse (se desapilará de la pila de actividades del sistema). Si durante la ejecución de B, el dispositivo recibiera una llamada entrante, B se apilaría también y la aplicación de llamada pasaría a ejecutarse. Cuando el usuario cuelgue, B volverá a ejecución y cuando, estando en B, pulse el botón Back, A pasará a ejecutarse. Es por esto que en la actividad `AboutActivity` no se programa nada para volver a la actividad `ComponentesBasicosActivity`, ya que el sistema gestionará automáticamente la activación de la segunda actividad cuando el usuario salga de la primera.

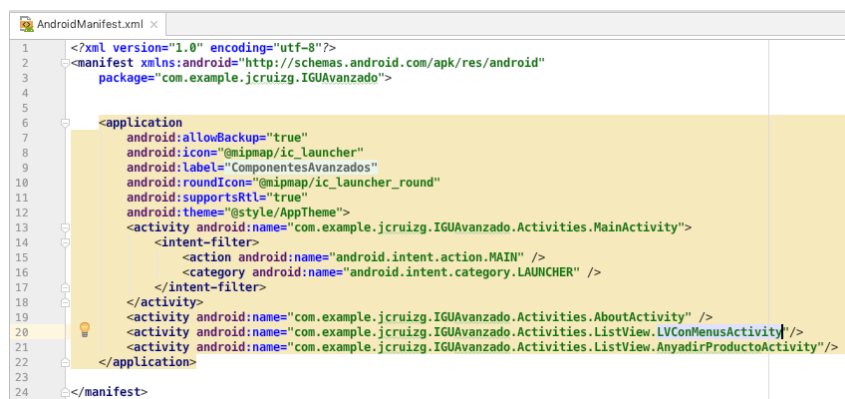
**Sugerencia:** Aunque esto pueda parecer un lío es muy sencillo. Podéis introducir en la aplicación *logs* o mensajes de tipo *Toast* y estudiar las trazas resultantes para entender cómo funciona y sobre todo, cómo se van encadenando los eventos que la app ejecuta.

### 3.2. App lista de la compra (Ver ComponentesAvanzados.zip)

Esta segunda aplicación trabaja con componente de la interfaz más avanzados y nos muestra:

- Cómo implementar un Dashboard (*MainActivity*);
- Una lista de la compra en la que se utiliza componentes IGU avanzados como *ListView*s, *Menus*, o *Dialogs*;

El manifiesto de la aplicación, que ya deberíamos ser capaces de interpretar, nos muestra que esta app integra 4 actividades distintas, tal y como se muestra a continuación:

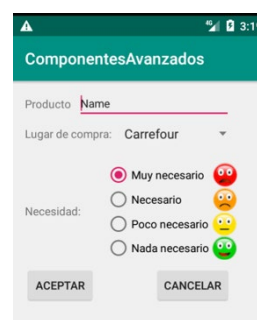


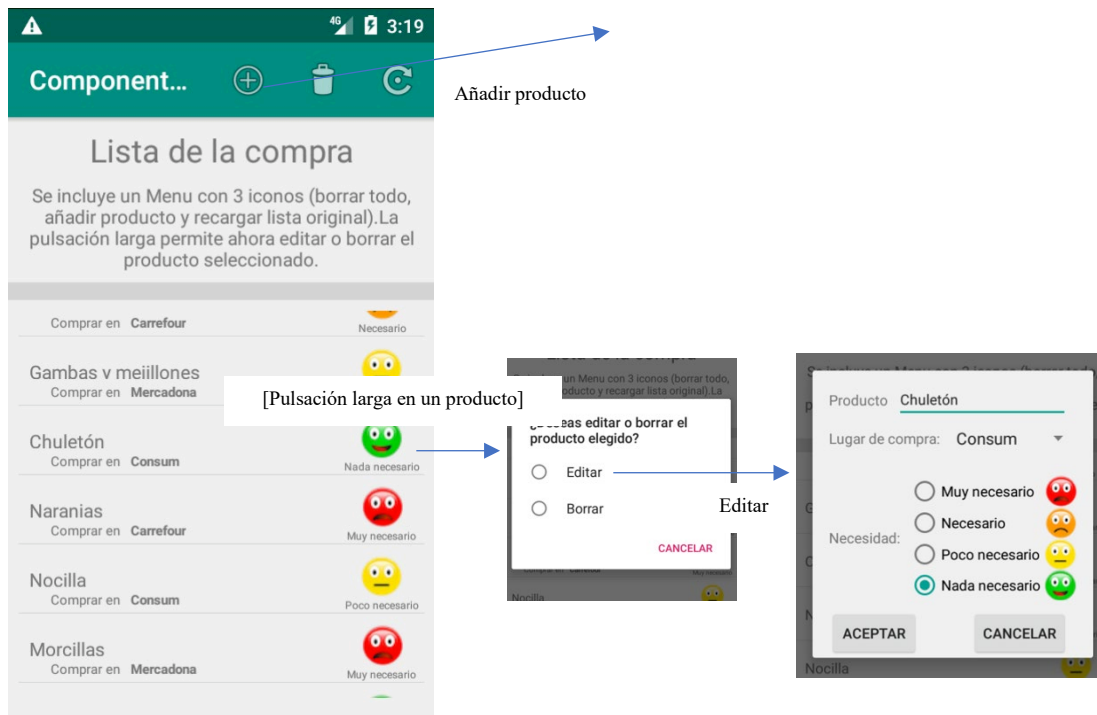
```

1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3    package="com.example.jcruizg.IGUAvanzado">
4
5
6
7    <application
8      android:allowBackup="true"
9      android:icon="@mipmap/ic_launcher"
10     android:label="ComponentesAvanzados"
11     android:roundIcon="@mipmap/ic_launcher_round"
12     android:supportRtl="true"
13     android:theme="@style/AppTheme">
14       <activity android:name="com.example.jcruizg.IGUAvanzado.Activities.MainActivity">
15         <intent-filter>
16           <action android:name="android.intent.action.MAIN" />
17           <category android:name="android.intent.category.LAUNCHER" />
18         </intent-filter>
19       </activity>
20       <activity android:name="com.example.jcruizg.IGUAvanzado.Activities.AboutActivity" />
21       <activity android:name="com.example.jcruizg.IGUAvanzado.Activities.ListView.LVConMenusActivity" />
22       <activity android:name="com.example.jcruizg.IGUAvanzado.Activities.ListView.AnyadirProductoActivity" />
23     </application>
24 </manifest>
  
```

El Dashboard (o tablero de mandos si lo tradujéramos al español) implementa la actividad principal y nos permite lanzar a ejecución las actividades *LVConMenusActivity* y *AboutActivity*. Es algo típico en muchas aplicaciones móviles que ofrecen una actividad central desde la que el usuario navega a todas las funcionalidades disponibles.

La última actividad (*AboutActivity*) ya la vimos en el ejemplo anterior. Por su parte, la actividad *LVConMenusActivity* implementa una lista de la compra. El menú ofrece 3 opciones: añadir un nuevo producto a la lista, borrar la lista y recargar una lista inicial. La lista de la compra se puede desplazar verticalmente. Si realizamos una pulsación corta sobre un elemento de la lista, no ocurrirá nada, pero si la pulsación es larga, entonces aparecerá una ventana emergente (un *Dialog* o cuadro de diálogo) que nos ofrecerá la posibilidad de editar o borrar el producto seleccionado. Si elegimos lo segundo, el producto desaparecerá de la lista, si deseamos editar el elemento, una segunda ventana emergente (en este caso un *Dialog personalizado*) aparecerá y nos permitirá aplicar las modificaciones que deseamos que se reflejarán en nuestra lista de la compra cuando las aceptemos.





El layout del menú se define como un recurso en el directorio `res/menu/menú_lista_compra.xml` y el código necesario para “inflar” el menú y añadirlo a la actividad, indicando cómo debe reaccionar ésta a la pulsación de cada una de las opciones propuestas, es el siguiente:

```

164 //
165 // GESTIÓN DE LAS OPCIONES DEL MENU
166 //
167 @Override
168 public boolean onCreateOptionsMenu(Menu menu) {
169     getMenuInflater().inflate(R.menu.menu_lista_compra, menu); //MOSTRAR EL MENU
170     return super.onCreateOptionsMenu(menu);
171 }
172
173 @Override
174 public boolean onOptionsItemSelected(MenuItem item) {
175     //
176     // REACCIONAR A LAS DISTINTAS OPCIONES QUE EL MENU OFRECE
177     //
178     switch (item.getItemId()){
179         case android.R.id.home: //SI SE PULSA HOME
180             return super.onOptionsItemSelected(item); //QUE EL SISTEMA LO GESTIONE
181         case R.id.menu_anyadir: //AÑADIR ELEMENTO
182             Intent i = new Intent( packageContext: this, AnyadirProductoActivity.class);
183             //
184             // LANZAMOS LA ACTIVIDAD QUE SERVIRÁ LA PETICIÓN INDICANDO QUE ESPERAMOS DE LA MISMA
185             // UN RESULTAD (EL NUEVO PRODUCTO A INTRODUCIR EN LA LISTA DE LA COMPRA EN NUESTRO CASO)
186             // EL MÉTODO DE CALLBACK EN ESTE CASO SERÁ onActivityResult (VER MÁS ABAJO)
187             //
188             startActivityForResult(i, ANYADIR_PRODUCTO_REQUEST);
189             break;
190         case R.id.menu_borrar:
191             borrarListaCompra(); //BORRAR TODA LA LISTA DE LA COMPRA
192             break;
193         case R.id.menu_restaurar:
194             recargaListaConCompraFake(); //GENERAR DE NUEVO LA LISTA DE LA COMPRA INICIAL
195             break;
196         default:
197             return super.onOptionsItemSelected(item); // POR SI ALGO SE NOS PASA QUE EL SISTEMA LO GESTIONE
198     }
199     return true;
200 }

```

Podéis encontrar más información al respecto del uso de menús en <https://developer.android.com/guide/topics/ui/menus?hl=es-419>.



En el código anteriormente mostrado vemos que cuando se solicita añadir un producto nuevo se lanza a ejecución la actividad *AnyadirProductoActivity.class*, pero en lugar de utilizar como hemos visto anteriormente el método *startActivity* utiliza el método *startActivityForResult*. La diferencia es que en este caso la actividad principal queda a la espera de un resultado, en nuestro caso el nuevo producto que se va a añadir. Si éste se comunica, entonces debe añadirse a la lista de la compra, y si no (el usuario cancela la acción), no debe hacerse nada. Es por esto que *startActivityForResult* necesita, además de un *Intent* que indique el contexto de la actividad y la clase a ejecutar (*AnyadirProductoActivity.class*), un código de petición, en nuestro caso *ANYADIR\_PRODUCTO\_REQUEST*. El método en el que se gestionará el call-back en caso de que éste se producto es el siguiente:

```

203 //
204 // MÉTODO QUE IMPLEMENTA EL CALLBACK RESULTANTE DE LA LLAMADA startActivityForResult(i,ANYADIR_PRODUCTO_REQUEST);
205 //
206 @Override
207 protected void onActivityResult(int requestCode, int resultCode, Intent data){
208     super.onActivityResult(requestCode, resultCode, data);
209     if (resultCode!=RESULT_CANCELED){ //SI EL USUARIO NO CANCELÓ LA ACCIÓN
210         switch(requestCode){
211             case ANYADIR_PRODUCTO_REQUEST: //Y LA PETICIÓN ES LA DE AÑADIR UN PRODUCTO
212                 Producto p = (Producto)data.getSerializableExtra( name: "Producto"); //EXTRAEMOS EL PRODUCTO DEL INTENT
213                 adapter.getCompra().anyadeProducto(p); // LO AÑADIMOS A LA LISTA
214                 adapter.notifyDataSetChanged(); //NOTIFICAMOS EL CAMBIO AL ADAPTER
215                 break;
216         }
217     }
218 }
219 }

```

Como vemos si la actividad que hemos ejecutado no termina con *RESULT\_CANCELED* y el código de petición es el que hemos emitido, entonces se extrae del mensaje (de hecho del *Intent data*) el producto y se le suministra al adaptador que gestiona la lista de la compra. Veremos lo que es ese adaptador un poco más tarde.

Centrándonos ahora en lo que hace la actividad que ha sido invocada para devolver el producto que defina el usuario, encontramos el siguiente código en la clase *AnyadirProductoActivity.java*.

```

66 Producto p = new Producto(nombre, lugar, necesidad); //CREAR EL PRODUCTO
67 Intent resultIntent = new Intent();
68 resultIntent.putExtra( name: "Producto",p); //INTRODUCIR EL PRODUCTO EN EL INTENT DE RESPUESTA
69 setResult(RESULT_OK, resultIntent); //INDICAR EN LA RESPUESTA QUE TODO HA IDO OK
70 finish(); //TERMINAR CON LA EJECUCIÓN DE LA ACTIVIDAD

```

Primero se instancia un *Intent* y se introduce el producto que se debe añadir utilizando el método *putExtra*. A continuación se indica que el resultado de la actividad será *RESULT\_OK* y que al mismo se asociará el intent que acabamos de generar. Finalmente la actividad terminará su ejecución de manera voluntaria (contrariamente a cuando lo hace porque el usuario pulsa el botón de *Back*). Esto activará el call-back que ya hemos visto en la actividad principal y el producto se añadirá a la Compra realizada. Cabe señalar que en el directorio *com/example/jcruizg/IGUAvanzado/Utilidad/Compras* encontraremos las clases *Compra.java* y *Producto.java*, que como su nombre indica definen lo que es en la práctica para la app una *Compra* y un *Producto*.

Como la lista de la compra es gestionada por la aplicación a través de una vista de tipo *ListView*, es posible asociarle al evento *OnItemLongClick* un manejador que, en nuestro caso, implementará un *Dialog* estándar de tipo *AlertDialog* para editar el producto seleccionado o bien eliminarlo. El código necesario para implementar esta funcionalidad es el siguiente:

```

54 //
55 // AHORA CUANDO EL USUARIO REALICE UNA PULSACIÓN LARGA SOBRE UN ITEM LE DAREMOS LA OPCIÓN DE EDITARLO O BORRARLO
56 // ESO LO HAREMOS UTILIZANDO UN AlertDialog
57 //
58 lv.setOnItemLongClickListener((adapterView, view, i, l) -> {
59
60     posicion = i;
61     AlertDialog.Builder builder = new AlertDialog.Builder(context: LVConMenusActivity.this);
62     builder.setTitle("¿Deseas editar o borrar el producto elegido?");
63     AlertDialog dialog;
64
65     String[] accion = {"Editar", "Borrar"}; //ACCIONES QUE OFRECEMOS AL USUARIO
66     //SÓLO PERMITIMOS QUE SE ELIJA UNA DE LAS DOS OPCIONES
67     builder.setSingleChoiceItems(accion, checkeditem: -1, new DialogInterface.OnClickListener() {
68         @Override
69         public void onClick(DialogInterface dialog, int which) { //EL -1 SIGNIFICA QUE NINGUNA ACCIÓN APARECERÁ COMO SELECCIONADA POR DEFECTO
70             switch(which){
71                 case 0:
72                     //Editar producto
73                     editarProducto(posicion);
74                     dialog.dismiss(); //TRAS TRATAR LA PULSACIÓN CERRAMOS EL DIALOG
75                     break;
76                 case 1:
77                     //Borrar producto
78                     borrarProducto(posicion);
79                     dialog.dismiss(); //TRAS TRATAR LA PULSACIÓN CERRAMOS EL DIALOG
80                     break;
81             }
82         }
83     });
84     builder.setNegativeButton(text: "Cancelar", listener: null); //EN CASO DE CANCELAR (ÚNICO BOTÓN MOSTRADO) NO HACER NADA (POR ESO LE PASAMOS UN NULL)
85     dialog = builder.create(); //CREAMOS EL DIALOG
86     dialog.show(); //MOSTRAMOS EL DIALOG
87     return true;
88 }
89 });
90

```

Si el usuario selecciona la edición del producto, entonces se le muestra un segundo *Dialog* que no es estándar, sino que se ha personalizado con un layout que se define como recurso en `res/layout/añadir_producto` y cuyo código se implementa en la clase *EditarProductoDialog* (almacenada en el directorio de código `com/jcruizg/IGUAvanzado/Dialogs`). Básicamente la clase implementa el método *onCreate* (que debe sobrescribir todo *Dialog*), y en el que inicializa la interfaz con la información del producto seleccionado, y el método *onClick*, que gestiona la validación de la información del producto, y en caso de aceptarse, se actualiza la vista que muestra el *ListView* que gestiona la lista de la compra.

Para más información acerca de los *Dialogs* utilizar la URL <https://developer.android.com/guide/topics/ui/dialogs> y en caso de desear personalizar uno, puede encontrarse más detalle en <https://developer.android.com/guide/topics/ui/dialogs>.

La gestión de la lista en sí es alrededor de lo que gravita toda la aplicación y tal vez, es la parte más compleja de comprender. A grandes rasgos, la idea es gestionar un conjunto de datos, pero separar, a través de un adaptador (o *Adapter* asumiendo la terminología Android), los datos de la interfaz. Por ello en la app encontramos que se define y utiliza un adaptador que, para que muestre un layout personalizado como el que tenemos, debe ser un adaptador creado por nosotros. El que la app utiliza está implementado en el fichero `com/example/jcruizg/IGUAvanzado/Adapters/MiCustomAdapterConViewHolder.java`. La instanciación del adaptador y su asignación a la lista se hacen en la actividad *LVConMenusActivity* y es relativamente sencilla, como puede apreciarse:

```

50 ListView lv = findViewById(R.id.lv_simpleLV);
51 adapter = new MiCustomAdapterConViewHolder(context: this, creaCompraFalsa()); // CREAMOS ARTIFICIALMENTE UNA LISTA DE LA COMPRA
52 lv.setAdapter(adapter);

```

Como vemos inicialmente se crea una compra falsa con el objetivo de que vosotros tengáis ya una lista de la compra de partida sin necesidad de introducir nada y podáis así jugar con las distintas funcionalidades que ofrece la aplicación. Esta lista false constituye el conjunto de datos inicial con el que va a trabajar el adaptador. El método *creaCompraFalsa()* lo hace es poblar la lista de la compra con los siguientes productos:

```

220 //
221 // MÉTODO DE CREACIÓN DE UNA LISTA FALSA, ES DECIR, UNA LISTA ARTIFICIALMENTE GENERADA PARA QUE TENGAMOS ALGO CON LO QUE
222 // PROBAR LA APLICACIÓN SIN NECESIDAD DE INTRODUCIR VARIOS PRODUCTOS MANUALMENTE
223 //
224 private Compra creaCompraFalsa(){
225     Compra compra = new Compra();
226     compra.anyadeProducto( nombre: "Judías", lugar: "Carrefour", R.drawable.necesario);
227     compra.anyadeProducto( nombre: "Arroz", lugar: "Mercadona", R.drawable.necesario);
228     compra.anyadeProducto( nombre: "Chorizos", lugar: "Consum", R.drawable.muy_necesario);
229     compra.anyadeProducto( nombre: "Leche", lugar: "Carrefour", R.drawable.poco_necesario);
230     compra.anyadeProducto( nombre: "Naranjas", lugar: "Carrefour", R.drawable.necesario);
231     compra.anyadeProducto( nombre: "Gambas y mejillones", lugar: "Mercadona", R.drawable.poco_necesario);
232     compra.anyadeProducto( nombre: "Chuletón", lugar: "Consum", R.drawable.nada_necesario);
233     compra.anyadeProducto( nombre: "Naranjas", lugar: "Carrefour", R.drawable.muy_necesario);
234     compra.anyadeProducto( nombre: "Nocilla", lugar: "Consum", R.drawable.poco_necesario);
235     compra.anyadeProducto( nombre: "Morcillas", lugar: "Mercadona", R.drawable.muy_necesario);
236     compra.anyadeProducto( nombre: "Cacao en polvo", lugar: "Consum", R.drawable.nada_necesario);
237     compra.anyadeProducto( nombre: "Agua embotellada", lugar: "Carrefour", R.drawable.poco_necesario);
238     return compra;
239 }
240

```

En cualquier momento podemos solicitar al adapter la compra que le hemos suministrado. Para ello nuestro adaptador ofrece los métodos necesarios para obtener la compra (*getCompra*) y modificarla (*setCompra*). El siguiente fragmento de código muestra cómo proceder para borrar completamente la lista de la compra o para recargar la lista con una compra dada (en nuestro caso la compra por defecto que devuelve el método *creaCompraFalsa*):

```

94 //
95 // BORRADO COMPLETO DE LA LISTA DE LA COMPRA
96 //
97 public void borrarListaCompra(){
98     adapter.getCompra().getListaDeProductos().clear();
99     adapter.notifyDataSetChanged();
100 }
101
102 //
103 // RECARGA DE LA LISTA DE LA COMPRA A SU VALOR ORIGINAL (LISTA DE LA COMPRA REALIZADA ARTIFICIALMENTE)
104 //
105 public void recargaListaConCompraFake() {
106     adapter.getCompra().getListaDeProductos().clear();
107     adapter.setCompra(creaCompraFalsa());
108     adapter.notifyDataSetChanged();
109 }

```

Señalar que el método *notifyDataSetChanged* es el método a través del cuál nuestro código (que define la capa de negocio de la app) comunica al adaptador un cambio en los datos que este gestiona, y por tanto, la necesidad de mostrar un nuevo conjunto de datos a través del *ListView*, es decir, actualizar la información que muestra la capa de presentación. Para cada elemento de la lista se activará el método *getView* que deberá implementar el adaptador y que básicamente proporciona al *ListView* el *View* asociado a cada elemento de la lista para que éste lo incluya en la lista que gestiona. En otras palabras, la lista de la compra está constituida por una serie de artículos que se deben de mostrar.

Cada vez que se actualiza la lista de la compra, se actualiza la información relativa a los artículos que esta contiene. Para ello, para cada artículo se genera (si no existe ya) la vista del artículo, y si ya existe se recupera. Luego se actualiza la información que contiene la vista con la información del artículo en cuestión. Finalmente, la vista asociada al artículo se devuelve. Esto sucede para cada artículo de la lista. El método *getView* de nuestro adaptador es el responsable de hacer todo lo indicado, tal y como sigue:



```

44 //
45 // SOBRESCRIBIMOS EL METODO GETVIEW PARA QUE, CONTRARIAMENTE A LO QUE HACE MICUSTOMADAPTER, AHORA SE UTILICE
46 // EL VIEWHOLDER QUE SE HA DEFINIDO
47 //
48 @Override
49 public View getView(int position, View convertView, ViewGroup parent) {
50     View view = convertView;
51     Contenedor contenedor = new Contenedor();
52     Producto producto = compra.getListaDeProductos().get(position);
53
54     //
55     // EL SECRETO ESTÁ EN INSTANCIAR UN VIEWHOLDER (LLAMADO CONTENEDOR EN NUESTRO EJEMPLO) Y GUARDARLO COMO TAG DEL VIEW QUE
56     // REPRESENTA A CADA ELEMENTO DE LA LISTA. ASÍ NO SERÁ NECESARIO HACER LAS LLAMADAS A FINDVIEWBYID CADA VEZ QUE EJECUTEMOS
57     // EL METODO GETVIEW (DE HECHO SÓLO SE HARÁN LA PRIMERA VEZ, CUANDO VIEW=NULL). ASÍ EL ACCESO A LOS VIEWS CONTENIDOS EN EL
58     // VIEWHOLDER (CONTENEDOR EN NUESTRO EJEMPLO) SERÁ DIRECTO, CON LO QUE GANAREMOS MUCHO TIEMPO Y GLOBALMENTE ACELERAREMOS
59     // MUCHO EL DIBUJADO DE LOS VIEWS EN LA LISTA. ESTA OPTIMIZACIÓN ES FUNDAMENTAL SI LA LISTA TIENE MUCHOS ELEMENTOS.
60     //
61
62     if (view == null) {
63         LayoutInflater inflater = (LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
64         view = inflater.inflate(R.layout.lista_compra_item, root, null); //LAYOUT DEFINIDO EN lista_compra_item.xml
65         contenedor.tvLugarDeCompra = (TextView) view.findViewById(R.id.tv_lugarCompra);
66         contenedor.ivNecesidad = (ImageView) view.findViewById(R.id.iv_necesidad);
67         contenedor.tvNecesidad = (TextView) view.findViewById(R.id.tv_necesidad);
68         contenedor.tvProducto = (TextView) view.findViewById(R.id.tv_producto);
69         view.setTag(contenedor);
70     }
71
72     contenedor = (Contenedor) view.getTag();
73     contenedor.tvProducto.setText(producto.getNombre());
74     contenedor.tvLugarDeCompra.setText(producto.getLugarDeCompra());
75     int necesidad = producto.getNecesidad();
76     contenedor.ivNecesidad.setImageResource(necesidad);
77     String msgNecesidad="Desconocida";
78     switch(necesidad){
79         case R.drawable.muy_necesario:
80             msgNecesidad="Muy necesario";
81             break;
82         case R.drawable.necesario:
83             msgNecesidad="Necesario";
84             break;
85         case R.drawable.poco_necesario:
86             msgNecesidad="Poco necesario";
87             break;
88         case R.drawable.nada_necesario:
89             msgNecesidad="Nada necesario";
90             break;
91         default:
92             break;
93     }
94     contenedor.tvNecesidad.setText(msgNecesidad);
95     return view;
96 }

```

El método aplica el patrón ViewHolder que recomienda Android para mejorar las prestaciones de las listas. El caso, es que dibujar una lista es muy costoso porque la llamada a *findViewById* lo es. Por ello, se almacena en un contenedor la referencias a los distintos *View* que hay en el layout de cada componente (un *ImageView* y tres *TextView* en nuestro caso) y en sucesivas llamadas para redibujar el contenido de la lista, se utilizan estas referencias directamente y sin necesidad de llamar al método *findViewById*.

Con todo esto ya deberíamos hacernos una idea de cómo funciona la app, y aunque no seamos capaces de desarrollar una app que gestione listas, deberíamos poder identificar el código que sirve para ello en una app si lo viéramos. Esto es muy importante sobre todo de cara a acciones de ingeniería inversa que pudiéramos necesitar hacer de cara a comprender como funciona una aplicación Android. Para profundizar en el uso de listas en Android se recomienda acceder al contenido de la siguiente URL: <https://developer.android.com/guide/topics/ui/layout/listview>.

### 3.3. Almacenamiento de información (Almacenamiento.zip)

Siguiendo con el ejemplo de la lista de la compra, la siguiente app implementa una variante de la app presentada en el apartado anterior, pero con capacidad para almacenar en memoria interna o externa (tarjeta SD) la lista de la compra confeccionada.

**Para esta app necesitaremos un dispositivo/emulador con Android 7 o superior para poder estudiar correctamente la gestión de permisos que se lleva a cabo.**

El almacenamiento interno es un almacenamiento considerado seguro, y por tanto, no requiere de la solicitud de permisos especiales, ni de la gestión de los mismos. Sin embargo, el uso de almacenamiento externo es considerado como peligroso, puesto que supone una vía de entrada y salida de datos con poco control. Esto implica que las apps deban solicitar una serie de permisos durante su instalación y que, a pesar de tener dichos permisos, deban respetar una serie de pautas en el acceso a dicha información.

La app que pasamos a estudiar nos va a mostrar no sólo dónde se deben definir y cómo se deben utilizar los permisos cuando accedemos a la información, sino que también nos enseñará a definir una actividad que almacene automáticamente la configuración de nuestra app.

El manifiesto de la app es el siguiente:



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

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="GestionFicheros"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportRtl="true"
        android:theme="@style/AppTheme">

        <activity android:name=".Activities.AboutActivity"
            android:parentActivityName=".Activities.MainActivity" />
        <activity android:name=".Activities.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name=".Activities.Ficheros.FicherosDemo"
            android:parentActivityName=".Activities.MainActivity" />
        <activity
            android:name=".Activities.Ficheros.AnyadirProductoActivity"
            android:parentActivityName=".Activities.Ficheros.FicherosDemo" />
        <activity
            android:name=".Activities.Ficheros.PantallaConfiguracionActivity"
            android:parentActivityName=".Activities.Ficheros.FicherosDemo"></activity>

    </application>
</manifest>
```

Vemos que la app declara en su manifiesto que pretende leer y escribir en almacenamiento externo. Cabe señalar que la necesidad de pedir permisos no afecta sólo al almacenamiento de información, sino como más tarde veremos a las comunicaciones http, o a otro tipo de operaciones que no vamos a ver, como el envío y recepción de mensajes SMS, llamadas telefónicas, etc. Para una más amplia información sobre los permisos y su gestión ir a <https://developer.android.com/guide/topics/security/permissions?hl=es-419>.

Siguiendo con nuestro ejemplo y con la información que refleja su manifiesto, vemos que la app consta de 5 actividades.

Los permisos que la app debe solicitar irán en función de donde se almacenen los datos. Esta decisión la debe poder configurar el usuario y a tal efecto la app bajo estudio ofrece una actividad de configuración. Veamos qué opciones ofrece dicha actividad antes de adentrarnos en cómo gestiona la app los permisos y almacena y recupera la información asociada a la lista de la compra.

- **Gestión automática de la configuración de la app: uso de PreferenceScreen**

De ellas, la actividad *PantallaConfiguracionActivity* es la encargada de mantener automáticamente el estado de la configuración de la app. Esta actividad es una

*PreferenceActivity* y simplemente gestiona la información que se define en su interfaz, que en lugar de ser un layout al uso, es una *PreferenceScreen* definida como un recurso en formato XML. Concretamente, ese fichero se encuentra en `res/xml/pantalla_configuracion.xml`.

```

1  <?xml version="1.0" encoding="utf-8"?>
2  <PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
3
4      <PreferenceCategory android:title="Estado de la app">
5          <SwitchPreference
6              android:defaultValue="true"
7              android:summaryOff="No restauraremos su lista de la compra, que estará vaci..."
8              android:summaryOn="Cuando ejecute la app, su lista de la compra será resta..."
9              android:title="Restaurar"
10             android:key="restaurar"
11          />
12      </PreferenceCategory>
13
14      <PreferenceCategory android:title="Almacenamiento">
15          <ListPreference
16              android:entries="@array/opciones_memoria"
17              android:entryValues="@array/opciones_memoria_valores"
18              android:summary="Pulse para especificar dónde se almacenarán, y desde dó..."
19              android:title="¿Memoria interna o externa?"
20              android:key="memoria"
21              android:defaultValue="0"
22          />
23          <ListPreference
24              android:defaultValue="0"
25              android:entries="@array/opciones_memoria_externa"
26              android:entryValues="@array/opciones_memoria_externa_valores"
27              android:key="memoria_ext"
28              android:summary="Pulse para escoger entre almacenamiento externo privado..."
29              android:title="Almacenamiento externo" />
30      </PreferenceCategory>
31  </PreferenceScreen>
32

```

Este tipo de ficheros siguen un formato muy estricto, que exige, por ejemplo que si la configuración de la app incluye una lista de preferencias (*ListPreference*), el array de entradas de la lista, y el de valores asociados, sean definidos en un fichero xml que se almacenará como recurso en `res/values` y que tendrá el nombre que nosotros deseemos darles, en nuestro caso `res/values/opviones_configuracion.xml`. El contenido de dicho fichero es el siguiente:

```

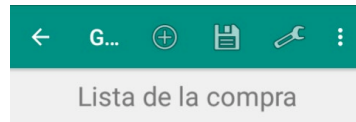
20  <string-array name="opciones_memoria">
21      <item>Memoria interna</item>
22      <item>Memoria Externa (SD)</item>
23  </string-array>
24
25  <string-array name="opciones_memoria_valores">
26      <item>0</item>
27      <item>1</item>
28  </string-array>
29
30  <string-array name="opciones_memoria_externa">
31      <item>Alm. externo público (/)</item>
32      <item>Alm. externo público (espec. [Music])</item>
33      <item>Alm. externo privado (/ de la app)</item>
34      <item>Alm. externo privado (espec. [Compras])</item>
35  </string-array>
36
37  <string-array name="opciones_memoria_externa_valores">
38      <item>0</item>
39      <item>1</item>
40      <item>2</item>
41      <item>3</item>
42  </string-array>

```

Como vemos la app ofrecerá la posibilidad de almacenar datos en memoria interna y externa, y en caso de hacerlo en memoria externa, lo podrá hacer en un espacio externo público (que no desaparecerá al desinstalar la app) o privado (relativo a la app y que desaparece cuando ésta se desinstala). Pues bien, las opciones de configuración que el usuario elija se almacenarán automática y localmente en el dispositivo sin que tengamos que hacer nada más ni solicitar permisos a nadie para ello. Cada vez que la app se ejecute dichas opciones se restaurarán y estarán disponibles para que la app actúe en consecuencia. Más información al respecto disponible en el siguiente enlace: <https://developer.android.com/guide/topics/ui/settings?hl=es-419>.

- Almacenamiento interno de datos

Almacenar datos internamente es algo considerado seguro. Esto significa que no deben solicitarse permisos especiales para hacerlo y, por tanto, que cualquier app puede almacenar información localmente. Nuestra app permite ahora almacenar la lista de la compra simplemente haciendo click en el icono del disco que aparece ahora en su menú.



De hecho, y aunque sólo 3 iconos del menú sean visibles, la app ofrece ahora 5 opciones que son las siguientes:

```

184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
...
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    cargaPreferencias();
    switch (item.getItemId()) {
        case android.R.id.home:
            return super.onOptionsItemSelected(item);
        case R.id.menu_anyadir:
            Intent i = new Intent( packageContext: this, AnyadirProductoActivity.class);
            startActivityForResult(i, ANYADIR_PRODUCTO_REQUEST);
            break;
        case R.id.menu_borrar:
            borrarListaCompra(); //VER BORRADO DE LISTA DE LA COMPRA
            break;
        case R.id.menu_restaurar:
            restauraCompra(); //VER OPERACIONES PARA GUARDAR Y RECUPERAR INFORMACIÓN
            break;
        case R.id.menu_guardar:
            guardaCompra(); //VER OPERACIONES PARA GUARDAR Y RECUPERAR INFORMACIÓN
            break;
        case R.id.menu_configurar:
            configuraApp(); // VER LANZAMIENTO A EJECUCIÓN DE LA ACTIVIDAD DE CONFIGURACIÓN DE LA APP
            break;
        default:
            return super.onOptionsItemSelected(item);
    }
    return true;
}

```

La información se guarda teniendo en cuenta cuales son las actuales opciones de almacenamiento definidas por el usuario en la configuración de la app:

```

242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
...
void guardaCompra(File fichero) {
    Compra c = adapter.getCompra();
    String compraAsString = Formateado.aXML(c);
    GestorFicheros.escribirDatos(fichero, compraAsString);

    String procedenciaDatos = null;
    switch (memoriaElegida) {
        case Interna:
            procedenciaDatos = "Memoria Interna";
            break;
        case SD:
            procedenciaDatos = "Memoria Externa (SD)";
            break;
    }
    Toast.makeText( context: this, text: "Compra guardada", Toast.LENGTH_LONG).show();
}

void guardaCompra(){
    File enDirectorio = null;
    switch (memoriaElegida) {
        case Interna:
            enDirectorio = getFilesDir();
            break;
        case SD:
            if (puedoGuardarEnSD()) enDirectorio = dameDirectorioExterno();
            break;
    }
    if (enDirectorio!=null) {
        File enFichero = new File( pathname: enDirectorio.getAbsolutePath() + File.separatorChar + "compra_backup.xml");
        guardaCompra(enFichero);
    }
}

```

Como puede observarse, al final todas las acciones de almacenamiento primero determinan el directorio en el que hay que almacenar la información y luego concatenan dicho directorio con el nombre del fichero (*compra\_backup.xml*) para obtener una dirección absoluta que es la utilizada para instanciar un objeto de tipo *File*. A continuación, la compra se formatea en XML (*Formateado.aXML(compra)*), y finalmente un gestor de ficheros (*GestorFicheros*) es el que escribe el string en formato XML que contiene la compra en el objeto de tipo *File*. Tanto la clase de *Formateado*

como la del *GestorFicheros* son clases de utilidad implementadas en el directorio *com/example/jcruizg/gestionficheros/Utilidad/Ficheros*.

A la hora de leer los datos trabajamos de manera muy similar:

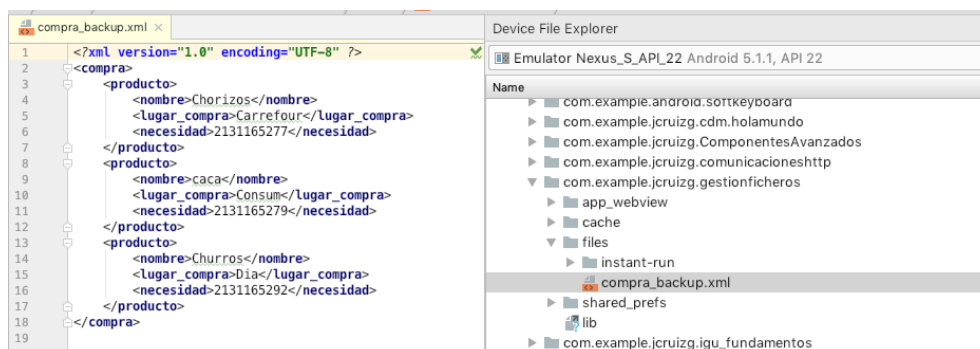
```

282     protected void restauraCompra() {
283         File desdeDirectorio = null;
284         switch (memoriaElegida) {
285             case Interna:
286                 desdeDirectorio = getFilesDir();
287                 break;
288             case SD:
289                 desdeDirectorio = dameDirectorioExterno();
290                 break;
291         }
292         File desdeFichero = new File( pathname: desdeDirectorio.getAbsolutePath() + File.separatorChar + "compra_backup.xml");
293         restauraCompra(desdeFichero);
294     }
295
296     void restauraCompra(File fichero) {
297         String contenidoFichero = GestorFicheros.leerDatos(fichero);
298         Compra c = Formateado.desdeXML(contenidoFichero);
299         adapter.getCompra().getListaDeProductos().clear();
300         adapter.setCompra(c);
301         adapter.notifyDataSetChanged();
302         String procedenciaDatos = null;
303         switch (memoriaElegida) {
304             case Interna:
305                 procedenciaDatos = "Memoria Interna";
306                 break;
307             case SD:
308                 procedenciaDatos = "Memoria Externa (SD)";
309                 break;
310         }
311         Toast.makeText( context: this,
312             text: "Compra restaurada desde " + procedenciaDatos,
313             Toast.LENGTH_LONG).show();
314     }
315
316 }

```

Lo único que ocurre al restaurar la compra es que, en este caso, se elimina y actualiza la compra que almacena el adapter, lo que obliga a notificar el cambio de datos para actualizar la lista de la compra que muestra el *ListView*.

Para finalizar con esta sección, señalaremos que *Android Studio* incorpora un explorador de archivos de dispositivo (*Device File Explorer*) que permite observar dónde se almacena la información y si se hace correctamente. En concreto, si el almacenamiento elegido es el interno, la información se almacena en el dispositivo en el directorio */data/data/paquete\_app/files*. En nuestro caso el directorio en cuestión tiene el siguiente nombre */data/data/com.example.jcruizg.gestionficheros/files* y allí encontramos el fichero *compra\_backup.xml*. El nombre del fichero, evidentemente, se lo hemos dado nosotros. Su contenido en un momento dado se muestra a título de ejemplo a continuación utilizando la utilidad de exploración de ficheros para dispositivos (*Device File Explorer*) que incorpora *Android Studio*:



## Gestión de información en formato XML

La escritura y lectura de un string en, o desde, un fichero es algo trivial que se hace tal y como se haría en Java. Ahora, la interpretación de los datos a leer o escribir ya no es tan

trivial, puesto que hay que hacerlo en XML. En la clase *Formateado* estos son los métodos utilizados para transformar una Compra en un String con formato XML:

```

28 @
29 public static String aXML(Producto p) {
30     String retStr = "\t<producto>\n";
31     retStr += "\t\t<nombre>" + p.getNombre() + "</nombre>\n";
32     retStr += "\t\t<lugar_compra>" + p.getLugarDeCompra() + "</lugar_compra>\n";
33     retStr += "\t\t<necesidad>" + p.getNecesidad() + "</necesidad>\n";
34     retStr += "\t</producto>\n";
35     return retStr;
36 }
37 // TRANSFORMA UNA COMPRA EN UN STRING CON FORMATO XML QUE LA REPRESENTA
38 //
39 @
40 public static String aXML(Compra c) {
41     String retStr = "<?xml version='1.0' encoding='UTF-8' ?>\n" +
42         "<compra>\n";
43     for (Producto p: c.getListaDeProductos()){
44         retStr += aXML(p);
45     }
46     retStr += "</compra>\n";
47     return retStr;
48 }

```

Para la lectura de dicha información, su interpretación y transformación en un objeto de tipo Compra, procedemos como sigue:

```

53 @
54 public static Compra desdeXML(String compraStr){
55     XmlPullParserFactory parserFactory;
56     try {
57         parserFactory = XmlPullParserFactory.newInstance();
58         XmlPullParser parser = parserFactory.newPullParser();
59         InputStream is = new ByteArrayInputStream(compraStr.getBytes(StandardCharsets.UTF_8));
60         parser.setFeature(XmlPullParser.FEATURE_PROCESS_NAMESPACES, false);
61         parser.setInput(is, null);
62         Log.d( tag: "[XMLParsing]:", msg: " Processing parsing");
63         return processParsing(parser);
64     } catch (XmlPullParserException e) {
65         Log.d( tag: "[XMLParsing:Exception]:", e.toString());
66         return null;
67     } catch (IOException e) {
68         Log.d( tag: "[XMLParsing:Exception]:", e.toString());
69         return null;
70     }
71 }
72
73 static final String nombre_TAG = "nombre";
74 static final String lugar_TAG = "lugar_compra";
75 static final String necesidad_TAG = "necesidad";
76 static final String producto_TAG = "producto";
77 static final String TAG_PARSING_XML = "[XMLParsing]";
78
79

```

Android ofrece para el parsing de archivos XML la clase *XMLPullParser* que deber ser instanciada a través de su factoría (*XMLPullParserFactory*). Tras ser instanciado el parser se asocia al flujo de datos (*InputStream*) asociado al string que representa la compra a analizar. Para su análisis hay que tener en cuenta los TAGs que hemos utilizado para formatear la compra, y con todo esto, el método *processParsing* analiza los datos como sigue:



```

84 private static Compra processParsing(XmlPullParser parser) throws IOException, XmlPullParserException{
85     Compra compra = new Compra();
86     Producto producto=null;
87
88     int eventType = parser.getEventType();
89
90     while (eventType != XmlPullParser.END_DOCUMENT) {
91         String name = null;
92         switch (eventType) {
93             case XmlPullParser.START_DOCUMENT:
94                 break;
95             case XmlPullParser.START_TAG:
96                 name = parser.getName();
97                 if (name.equalsIgnoreCase(producto_TAG)) { //NUEVO PRODUCTO
98                     Log.d(TAG_PARSING_XML, msg: "Creando nuevo producto");
99                     producto = new Producto();
100                 } else if (producto != null) {
101                     if (name.equalsIgnoreCase(nombre_TAG)) { //NOMBRE DEL PRODUCTO
102                         producto.setNombre(parser.nextText());
103                         Log.d(TAG_PARSING_XML, msg: "> Nombre: "+producto.getNombre());
104                     } else if (name.equalsIgnoreCase(lugar_TAG)) { // LUGAR DE COMPRA
105                         producto.setLugarDeCompra(parser.nextText());
106                         Log.d(TAG_PARSING_XML, msg: "> Lugar: "+producto.getLugarDeCompra());
107                     } else if (name.equalsIgnoreCase(necesidad_TAG)) { //NIVEL DE NECESIDAD DE LA COMPRA
108                         int necesidad = Integer.parseInt(parser.nextText());
109                         switch(necesidad){
110                             case R.drawable.muy_necesario:
111                             case R.drawable.nada_necesario:
112                             case R.drawable.necesario:
113                             case R.drawable.poco_necesario:
114                                 Log.d(TAG_PARSING_XML, msg: "> Necesidad conocida");
115                                 break;
116                             }
117                         producto.setNecesidad(necesidad);
118                     }
119                 }
120             }
121         break;
122     case XmlPullParser.END_TAG:
123         name = parser.getName();
124         if (name.equalsIgnoreCase(producto_TAG)) { //FIN DE PROCESAMIENTO DEL PRODUCTO
125             if (producto != null) {
126                 compra.anyadeProducto(producto);
127                 Log.d(TAG_PARSING_XML, msg: "Añadiendo nuevo producto"); //AÑADIENDO PRODUCTO A LA LISTA DE LA COMPRA
128             }
129         }
130         break;
131     }
132     eventType = parser.next();
133 }
134 }
135
136
137 return compra;
138 }

```

Básicamente nos encontramos en un bucle que, hasta el fin del documento, busca el nombre, el lugar de compra y la necesidad de cada producto, y luego cuando la definición del producto finaliza (*END\_TAG*), instancia un producto y lo añade a la compra que se está generando. Se ha añadido un log para poder depurar el código con *Logcat* y poder trazar el comportamiento de la aplicación. Cabe señalar que en muchas ocasiones estos logs no se borran y pueden darnos información de interés sobre cómo se ejecuta la app e incluso, a veces, nos ofrecen información muy sensible sobre la misma.

- Almacenamiento externo de información

La gestión del Almacenamiento externo es un poco más compleja, ya que a través dicho almacenamiento las aplicaciones pueden llegar a compartir información y, por tanto, hay que establecer mecanismos de control de acceso a la información almacenada.

Básicamente distinguimos tres casos: espacio de almacenamiento privado de la app (interno y externo), espacio de almacenamiento compartido para ficheros multimedia (videos, música, fotos, alarmas, etc.) y espacio compartido para cualquier otro tipo de fichero. Como muestra la tabla más abajo, los permisos necesarios en cada caso son distintos, así como la manera en la que el sistema gestionará los ficheros, ya que dependiendo de su tipo podrán ser abiertos, y por tanto consultados, por otras aplicaciones o no, así como ser eliminados en caso de desinstalar la app que los generó. Tal y como hemos dicho, la siguiente tabla sintetiza esta política de gestión de la información almacenada en un dispositivo Android.

	Contenido	Acceso	Permisos	¿Pueden ser abiertos por otras apps?	¿Se eliminan al desinstalar la app?
<b>Ficheros específicos de la app</b>	Aquellos que sólo la app debería usar	<b>Alm. Interno</b> - getFilesDir() - getCacheDir()  <b>Alm. Externo</b> - getExternalFilesDir() - getExternalCacheDir()	Ninguno Todos los ficheros se almacenan dentro del espacio de almacenamiento (interno o externo) de la app	No	Si
<b>Multimedia</b>	Ficheros compartidos (Videos, Música, Fotos)	MediaStore API	READ_EXTERNAL_STORAGE cuando se accede a los ficheros de otras apps desde Android 11 (API 30) o superior  READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE cuando se accede a los ficheros de otras apps desde Android 10 (API level 29)  Los permisos de lectura y escritura se requieren para todos los ficheros cuando se accede a ellos desde Android 9 (API 28) o inferior	Si, se debe de solicitar siempre el permiso READ_EXTERNAL_STORAGE	No
<b>Documentos y otros ficheros</b>	Otros ficheros que pueden compartirse	Storage Access Framework	Ninguno	Si, utilizando el explorador de archivos del sistema (el denominado system file picker)	No

En el caso de nuestra app podríamos tratar los ficheros en el espacio de almacenamiento externo privado de la app para no tener que gestionar los permisos, pero resultaría poco representativo. Lo que vamos a hacer es asumir que los ficheros se pueden almacenar tanto en el espacio interno (ya visto) como en el externo (SD) del dispositivo, y en este último, tanto en el espacio privado de la app como en el espacio público. Además, para simplificar la implementación nos aseguraremos de que la app funcione correctamente (si se le conceden los permisos que solicita claro) gracias a las librerías de compatibilidad. Estas librerías, normalmente, ya que depende de las implementaciones que realice el fabricante, nos aseguran que la app gestione correctamente su almacenamiento en dispositivos con Android 10 o superior. Por tanto, en el manifiesto de la app declaramos los permisos de lectura y escritura en almacenamiento externo que necesitamos:

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

A continuación, hay que ver si disponemos de los permisos necesarios cada vez que realizamos una operación de escritura o lectura. En el código de los métodos *guardaCompra* y *restauraCompra* que ya hemos visto hay unas comprobaciones que efectúan cuando la información debe leerse o escribirse en *SD*. Los métodos que realizan estas comprobaciones son *puedoGuardarEnSD* y *puedoLeerDeSD*.

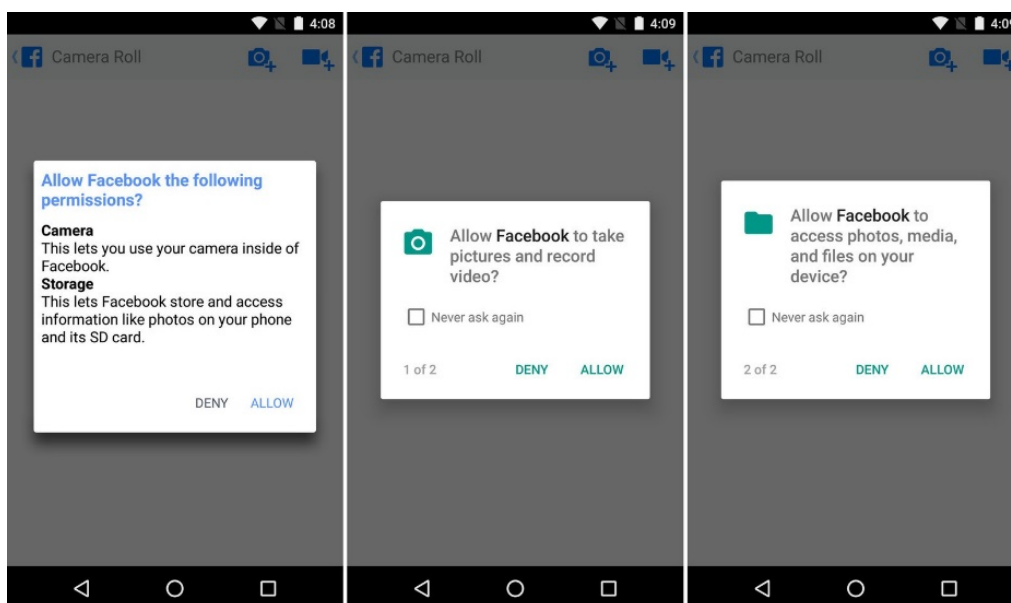


```

329 boolean puedoGuardarEnSD() {
330     if (Build.VERSION.SDK_INT >= 23) {
331
332         String state = Environment.getExternalStorageState();
333         //
334         // Miramos si la memoriaSD está presente y es accesible
335         //
336         if (!Environment.MEDIA_MOUNTED.equals(state)) return false;
337         if (Environment.MEDIA_MOUNTED_READ_ONLY.equals(state)) return false;
338
339         int checkWriteablePermission =
340             ContextCompat.checkSelfPermission( context: this, Manifest.permission.WRITE_EXTERNAL_STORAGE);
341         if (PackageManager.PERMISSION_GRANTED == checkWriteablePermission) {
342             return true;
343         } else {
344             ActivityCompat.requestPermissions(
345                 activity: this, new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE}, SOLICITAR_PERMISOS_ESCRITURA_EN_SD);
346             return false;
347         }
348     } else {
349         return true;
350     }
351 }
352
353 boolean puedoLeerDeSD() {
354     if (Build.VERSION.SDK_INT >= 23) {
355
356         String state = Environment.getExternalStorageState();
357         //
358         // Miramos si la memoriaSD está presente y es accesible
359         //
360         if (!Environment.MEDIA_MOUNTED.equals(state)) return false;
361
362         int checkWriteablePermission =
363             ContextCompat.checkSelfPermission( context: this, Manifest.permission.READ_EXTERNAL_STORAGE);
364         if (PackageManager.PERMISSION_GRANTED == checkWriteablePermission) {
365             return true;
366         } else {
367             ActivityCompat.requestPermissions(
368                 activity: this, new String[]{Manifest.permission.READ_EXTERNAL_STORAGE}, SOLICITAR_PERMISOS_LECTURA_EN_SD);
369             return false;
370         }
371     } else {
372         return true;
373     }
374 }

```

En ambos casos, comprobamos que la versión de la SDK sea superior o igual a 23, puesto que la gestión de permisos se impuso como obligatoria a partir de Android 6.0 (Marshmallow). Si es el caso, utilizamos la clase *Environment* para comprobar si el almacenamiento externo (la tarjeta SD) está montado y luego comprobamos (método *ContextCompat.checkSelfPermission*) que la app tiene los permisos que nos hacen falta (*READ\_EXTERNAL\_STORAGE* o *WRITE\_EXTERNAL\_STORAGE* según el caso). Si es así, se devuelve true y la operación puede realizarse. Si no es así, entonces se recomienda solicitar al usuario los permisos (llamada a *ActivityCompat.requestPermissions*). Esto conllevará que el usuario visualice un *Dialog* en el se le soliciten los permisos en cuestión.



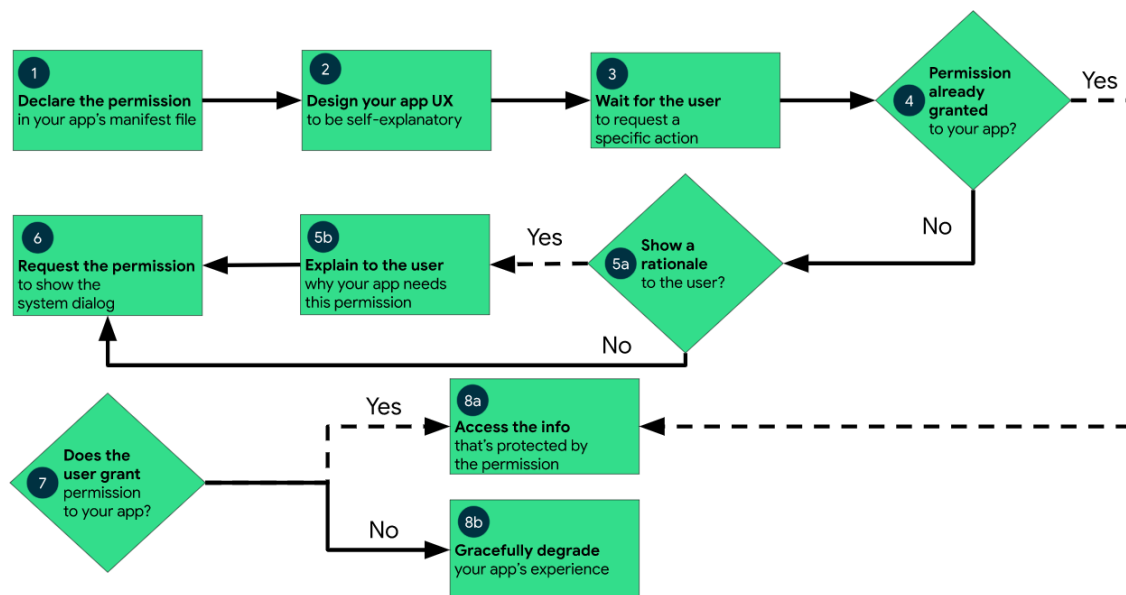
La respuesta del usuario se remite a la aplicación a través del siguiente método de callback que es activado por el sistema cuando el usuario contesta otorgando, o denegando, los permisos solicitados por la aplicación:

```

403 @Override
404 public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
405     super.onRequestPermissionsResult(requestCode, permissions, grantResults);
406
407     if (requestCode == SOLICITAR_PERMISOS_ESCRITURA_EN_SD) {
408         int grantResultsLength = grantResults.length;
409         if (grantResultsLength > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
410             Toast.makeText(getApplicationContext(), "Permisos concedidos para escribir en memoria externa.", Toast.LENGTH_SHORT).show();
411             guardaCompra();
412         }
413     }
414     if (requestCode == SOLICITAR_PERMISOS_LECTURA_EN_SD) {
415         int grantResultsLength = grantResults.length;
416         if (grantResultsLength > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
417             Toast.makeText(getApplicationContext(), "Permisos concedidos para leer desde la memoria externa.", Toast.LENGTH_SHORT).show();
418             restauraCompra();
419         }
420     }
421     else {
422         Toast.makeText(getApplicationContext(), "Imposible acceder a la SD por falta de permisos.", Toast.LENGTH_SHORT).show();
423     }
424 }

```

Si el usuario ha otorgado los permisos que se necesitan, entonces procedemos con la operación, si no, lo notificamos al usuario. En caso de intentar proceder sin los permisos adecuados, la app deberá asegurar un funcionamiento degradado correcto. Si los deseos del usuario no se respetan y se intenta acceder sin permiso al almacenamiento externo, la aplicación finalizará abruptamente su ejecución ya que el sistema matará al proceso que la ejecuta por una violación de privilegios. En el siguiente diagrama de flujo se refleja el protocolo a respetar para la gestión adecuada de permisos (algo que aplica a todos los permisos, no solo los de almacenamiento):



Al respecto de dónde se almacena la información cuando se guarda en memoria externa, la app que estamos estudiando ofrece distintas alternativas, que permiten almacenar la lista de la compra en distintos espacios de almacenamiento externo. En el caso 1 el directorio elegido es el directorio público (y por tanto accesible a todas las aplicaciones) donde el dispositivo almacena por defecto Música. Cabe señalar que esta elección se ha hecho a título demostrativo y que se podría haber elegido cualquier otro de los directorios públicos existentes (DIRECTORY\_AUDIOBOOKS, DIRECTORY\_MOVIES, DIRECTORY\_PICTURES, etc.). En el caso 2, la lista de la compra se almacena en el

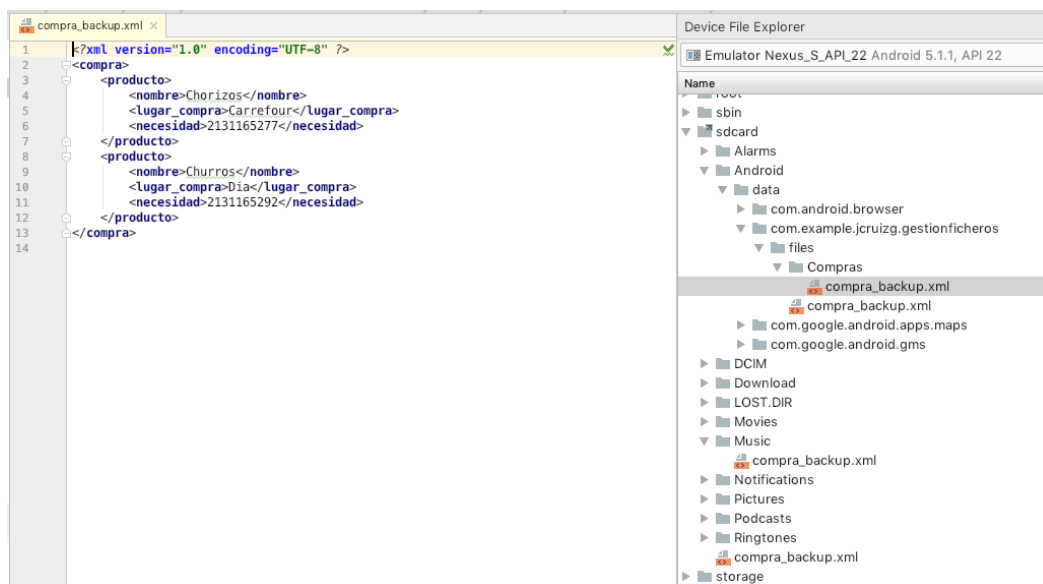
directorio raíz de almacenamiento externo, mientras que en el caso 3 se hace en directorio Compras, que si no existe se creará. Finalmente, el caso 4 nos muestra se muestran a continuación:

```

376 File dameDirectorioExterno() {
377     //
378     // Distintas opciones a la hora de almacenar la información en memoria externa.
379     //
380     //
381     switch (memoriaExtElegida){
382     case 1:
383         return Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_MUSIC);
384     case 2:
385         return this.getExternalFilesDir( type: null);
386     case 3:
387         return this.getExternalFilesDir( type: "Compras");
388     case 0:
389     default:
390         return Environment.getExternalStorageDirectory();
391     }
392 }
393

```

El explorador de archivos puede servirnos también para ver dónde y cómo se ha almacenado el fichero *compra\_backup.xml*. En este caso buscaremos el directorio /sdcard. La figura muestra el lugar donde se almacena el fichero para cada una de las 4 opciones que ofrece nuestra app:



Más información al respecto del almacenamiento en el siguiente enlace: <https://developer.android.com/guide/topics/data/data-storage?hl=es-419>.

### 3.4. Comunicaciones HTTPS (ComunicacionesHTTPS.zip)

La última de las apps que vamos a estudiar nos permite realizar peticiones http utilizando distintas alternativas que ofrece Android. En todas ellas, el objetivo es realizar una conversión entre divisas y para ello utilizaremos los servicios del banco central europeo, que nos proporciona un fichero XML en el que se reflejan los tipos de cambio entre divisas y que se actualiza diariamente. El fichero en cuestión está accesible a través de <https://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml?60634aa4076cc20682405e0785c50d27>.

La app suministrada realiza la conversión de divisas cargando el fichero XML tanto a través de la librería Volley (<https://developer.android.com/training/volley>) como de la clase

HttpsURLConnection (<https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection>). En ambos casos cabe señalar que en el manifiesto de la app deben solicitarse los permisos de conexión a internet, que no se consideran como peligrosos y que, por tanto, no conllevan una gestión particular por parte del usuario, es decir, se aplican automáticamente sin solicitar su conformidad.

```
5 <uses-permission android:name="android.permission.INTERNET" />
6
```

La interfaz de la app en ambos casos es la misma, puesto que lo único que cambia es cómo se obtienen las divisas y los valores de cambio asociadas a las mismas. Lo demás es idéntico.



Cabe señalar que las comunicaciones http no pueden ser gestionadas en primer plano, sino que tienen que ser servidas en hilos de ejecución distintos del hilo principal. La razón para ello es impedir que el hilo de comunicaciones bloquee al hilo de interacción con el usuario. Aunque la petición se efectúe a través de una clase dada, la clase *ProveedorDeDivisasPorHTTPS* en nuestro proyecto, la gestión de la petición puede llevarse a cabo utilizando la librería de Google Volley, utilizando tareas asíncronas (AsyncTasks) o simplemente utilizando manejadores de hilos (Handlers). A continuación, detallamos uno por uno estos métodos.

- **Peticiones http con Volley**

La librería Volley gestiona esto automáticamente, es decir sin que el programador tenga que preocuparse por el hilo de ejecución que servirá la petición. El siguiente fragmento de código muestra cómo la actividad *ConversorDivisas\_Volley* efectúa la petición http:

```

134 private void lanzaPeticiónWebParaObtencionDeDivisasporconVolley() {
135     requestQueue = Volley.newRequestQueue( context: this);
136     Response.Listener<String> responseListener = new Response.Listener<String>() {
137         @Override
138         public void onResponse(String response) {
139             //
140             // LO QUE DEBEMOS HACER SI TODO VA BIEN
141             //
142             actualizaAdapterData(response);
143             requestQueue.stop();
144         }
145     };
146     Response.ErrorListener errorListener =new Response.ErrorListener() {
147         @Override
148         public void onErrorResponse(VolleyError error) {
149             //
150             // LO QUE DEBEMOS HACER EN CASO DE PROBLEMAS CON LA PETICIÓN
151             //
152             requestQueue.stop();
153         }
154     };
155
156     new ProveedorDeDivisasPorHTTP().obtenerDivisas(requestQueue, responseListener, errorListener);
157 }
158

```

Cuando la petición es correcta, se activará el callback *onResponse* con lo que podremos actualizar la lista de divisas con los valores que se nos proporcionen. Si por el contrario hay un problema podremos tratarlo convenientemente en *onErrorResponse* si fuera necesario. La petición en sí (de tipo GET) la realiza el método *obtenerDivisas* de la clase *ProveedorDeDivisasPorHTTP*:

```

69 public void obtenerDivisas(RequestQueue requestQueue, Response.Listener<String> responseListener, Response.ErrorListener errorListener) {
70     try {
71         URL url = new URL( spec: "https://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml?60634aa4076cc20682405e0785c50d27");
72
73         // Initialize a new StringRequest
74         StringRequest stringRequest = new StringRequest(
75             Request.Method.GET,
76             url.toString(),
77             responseListener,
78             errorListener
79         );
80
81         // Add StringRequest to the RequestQueue
82         requestQueue.add(stringRequest);
83     } catch (IOException e) {
84         e.printStackTrace();
85     } finally {
86     }
87 }

```

El hecho de que Volley oculte toda gestión de hilo de ejecución hace que cuando se trata de interpretar el código de una app muchas veces resulte difícil saber si la app utiliza comunicaciones http o no simplemente por un análisis estático de código.

- Comunicaciones http utilizando *HttpsURLConnection*

Las cosas son algo más complejas cuando realizamos “a pelo” las peticiones http ya que la responsabilidad de crear y gestionar el ciclo de vida de los hilos que van a realizar la petición recae sobre los programadores.

Las peticiones http puede realizarse tanto a través de *Handlers* como a través de tareas asíncronas (o *AsyncTasks*). Ambos métodos están soportados por la clase *ConversorDivisas\_HTTP* que se os proporciona.

- Peticiones http utilizando *Handlers*

Para la gestión de la petición utilizando *Handlers* nuestra actividad provee el siguiente código:

```

134 protected void lanzaPeticiónWebParaObtencionDeDivisasporHandler() {
135
136     if (objSincronizacion == null) {
137         objSincronizacion = new Handler() {
138             @Override
139             public void handleMessage(Message msg) {
140                 //
141                 // Se activa cuando el proceso de petición de divisas por HTTP finaliza
142                 // con lo que se debe proceder a procesar la respuesta del servidor para
143                 // obtener las divisas y luego actualizar la IGU de la app
144                 //
145                 if (msg.what == CODIGO_PETICION) {
146                     Bundle bundle = msg.getData();
147                     if (bundle != null) {
148                         // Obtención del string en formato XML con las divisas
149                         HashDivisas d = (HashDivisas) bundle.getSerializable(CLAVE_RESPUESTA_PETICION);
150                         // Procesamiento del string y actualización del adapter
151                         actualizaAdapterData(d);
152                     }
153                 }
154             }
155         };
156     }
157
158     Thread hiloPeticiónHTTP = new Thread() {
159         @Override
160         public void run() {
161             HashDivisas d = new ProveedorDeDivisasPorHTTP().obtenerDivisas();
162
163             // Send message to main thread to update response text in TextView after read all.
164             Message message = new Message();
165
166             // Set message type.
167             message.what = CODIGO_PETICION;
168
169             // Create a bundle object.
170             Bundle bundle = new Bundle();
171             // Put response text in the bundle with the special key.
172             bundle.putSerializable(CLAVE_RESPUESTA_PETICION, d);
173             // Set bundle data in message.
174             message.setData(bundle);
175             // Send message to main thread Handler to process.
176             objSincronizacion.sendMessage(message);
177         }
178     };
179     hiloPeticiónHTTP.start();
180 }

```

Se genera un hilo secundario cuyo código (ver método *run*) realiza la petición y que al término de la misma genera un *Message* al que adjunta un código (*CODIGO\_PETICION*) y un *Bundle* que contiene las divisas recibidas a las que les asigna una clave de identificación (*CLAVE\_RESPUESTA\_PETICION*). El mensaje es entonces remitido al hilo principal de la app sirviéndose de un objeto de sincronización, el objeto *objSincronizacion* que se define en la misma clase de la siguiente manera:

```

38 public Handler objSincronizacion = null;

```

De hecho, el método que estamos analizando instancia este objeto de sincronización asociándole un *Handler* (de ahí el nombre del método) que provee un método *handleMessage* que será el activado en el hilo principal cuando el hilo secundario realiza la notificación *sendMessage*. Por tanto, el método *handleMessage* es el encargado de actualizar el conjunto de divisas que utiliza la app en el *Spinner* desplegable en el que aparecen sus nombres.



- Comunicaciones http utilizando AsyncTasks

Las tareas asíncronas son algo muy Android. Básicamente permiten definir una clase en la que los métodos *onPreExecute* y *onPostExecute* se van a ejecutar en primer plano y el



método *doInBackground* lo hará en segundo plano, sin necesidad de que el programador lo tenga que decir de manera explícita. Como su nombre indica, los métodos *onPreExecute* y *onPostExecute* se ejecutarán antes y después, respectivamente, del método *doInBackground*. Por su parte este último método será el que realice la petición http al ejecutarse en segundo plano. Además, el valor que retorne el método *doInBackground* será el que se suministre como argumento al método *onPostExecute*, con lo que la responsabilidad de actualizar la información de la interfaz, algo que como hemos dicho sólo puede hacer el hilo principal de la app, recae en este caso sobre este método.

```
182     protected void lanzaPeticiónWebParaObtencionDeDivisasporAsyncTask() {
183         new miAsyncTaskParaPeticiónHTTP().execute();
184     }
185
186     private class miAsyncTaskParaPeticiónHTTP extends AsyncTask<Void, Void, HashDivisas> {
187
188         @Override
189         protected HashDivisas doInBackground(Void... voids) {
190             return new ProveedorDeDivisasPorHTTP().obtenerDivisas();
191         }
192
193         @Override
194         protected void onPreExecute() { super.onPreExecute(); }
195
196         @Override
197         protected void onPostExecute(HashDivisas d) {
198             super.onPostExecute(d);
199             actualizaAdapterData(d);
200         }
201     }
202
203
204
205 }
```

## 4. Estudio de un sencillo malware Android (AlmacenamientoMalo.zip)

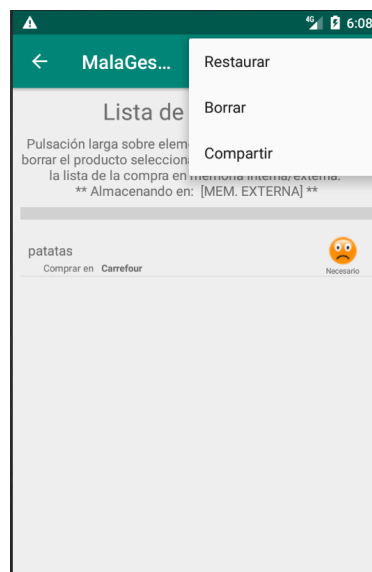
Como vemos la complejidad de las aplicaciones Android no es baja. A la complejidad de código “benigno” que pueden llegar a integrar se le suma el código “maligno” que también podrían contener. En esta sección del documento se plantea el estudio del código de un malware programado partiendo de una de las apps ya estudiada en el apartado anterior, la app de gestión de lista de la compra.

Lo que vamos a hacer no es explicar el código del malware. Ya hemos visto mucho código en el apartado 3 de este documento. Lo que vamos a hacer es intentar entender qué es lo que funciona mal en la aplicación simplemente instalándola en el emulador y estudiando su código. Tenemos la ventaja de que no vamos completamente a ciegas, puesto que disponemos del código limpio, sin malware, de la lista de la compra, lo que nos puede servir para abordar el estudio del código maligno que tenemos entre manos.

En primer lugar pongámonos en situación. Un cliente VIP de nuestra empresa se presenta un día diciendo “¡¡¡ Mis Contactos han desaparecido !!! ¿Podéis ayudarme a averiguar por qué?”. Este cliente nos cuenta que su dispositivo funcionaba correctamente hasta que se descargó. Tras cargarlo y encenderlo, el móvil comenzó a comportarse de manera extraña ya que, no sólo no encontraba ninguno de los contactos registrados en el mismo, sino que si introducía alguno, al poco tiempo (algo menos de un minuto) desaparecía. Sin embargo, no había instalado ninguna aplicación nueva desde que recargó el dispositivo,

con lo que no entendía por qué el dispositivo funcionaba correctamente hasta que el terminal se apagó, su batería se recargó y se volvió a encender.

Tras mucho investigar, hemos llegado a la conclusión de que el malware instalado en el dispositivo está en la app *AppMalvada* que un amigo de nuestro cliente le instaló recientemente. Tal y como refleja su manifiesto, esta app pide permisos de acceso a los contactos, con lo que sospechamos que tal vez los derechos adquiridos con estos permisos se utilicen para algo más que para compartir la lista de la compra con los contactos del usuario. Cabe señalar que la opción de Compartir es una nueva opción que la app ofrece y que no ofrecía en la versión libre de malware estudiada anteriormente. La opción de Compartir se incluye en el menú de opciones de la app, tal y como se muestra a continuación:



Ejecuta la app en un emulador (evita utilizar un dispositivo real) y estudia su código para intentar entender cómo hace para borrar los contactos del dispositivo. Lo que se busca no es entender el código de borrado de los contactos, sino el mecanismo utilizado para llegar a activar dicho código sin la intervención del usuario. Se plantean las siguientes dudas:

- ¿Quién se activa el código maligno? ¿Es el usuario al compartir su lista de la compra?
- ¿Cómo se explica que la aplicación se instale y todo funcione correctamente y que comiencen a desaparecer los contactos del usuario cuando el dispositivo se reenciende tras haberse apagado por falta de batería?
- ¿Cómo podemos resolver el problema? ¿Existe alguna solución que no requiera modificar el código Java de la aplicación, ni eliminar ninguna de las clases que la app integra? Buscamos encontrar una solución al problema que no sea excesivamente invasiva ni requiera de grandes destrezas en programación.
- ¿Es complejo el diseño de este malware o es sencillo y fácilmente replicable en otras apps? De poderse hacer, ¿qué sería necesario para reproducir en otra app el comportamiento observado en la app bajo estudio?



## 5. Ejercicios a realizar

El objetivo de los ejercicios propuestos no es el evaluar si en menos de 5 horas os habéis convertido en unos expertos desarrolladores en Android o no, porque eso es casi imposible. De lo que se trata es de que seáis capaces de desarrollar, partiendo de la app HolaMundo ya trabajada y de las apps suministradas para estudio, unas apps con algo más de funcionalidad, y que sepáis cómo implementar un fichero apk y programarlo en el dispositivo móvil.

Los **ejercicios básicos** a ejecutar son los siguientes:

**Ejercicio 1:** Modifica la app HolaMundo y reescribe todos los métodos relacionados con su ciclo de vida, es decir, los métodos onCreate, onStart, onResume, onPause, onStop y onDestroy. Añade en cada método un log de depuración y utiliza la herramienta Logcat para realizar un seguimiento de la ejecución de la app. Aporta en tu memoria logs mostrando distintas secuencias de activación de la app que respondan a distintas situaciones consideradas. Identifica al menos 2 situaciones distintas, descríbelas y proporciona los logs que muestras el funcionamiento descrito.

**Ejercicio 2:** Modifica la app HolaMundo y añádele a la misma una Actividad adicional en la que aparezcan los créditos de la app, que deben contener, al menos, el nombre y apellidos de los integrantes del grupo y sus emails. Para poder acceder a esta actividad añade un ImageView al que le asociéis una imagen (un icono de información por ejemplo) y que al pulsarlo os de acceso a la nueva actividad de créditos.

**Ejercicio 3:** Modifica la app resultante del ejercicio anterior y añádele al *TextView* que ya aparece mostrando el mensaje “Hola Mundo” un *EditText* en el que el usuario pueda introducir el texto que le apetezca. Añade también un botón que valide la introducción de datos y en respuesta a su pulsación genere un Toast con el texto que el usuario haya introducido en el nuevo *EditText* introducido.

Los **ejercicios avanzados** a ejecutar son los siguientes:

**Ejercicio 4:** Estudia el Malware proporcionado. Responde a las preguntas planteadas:

- ¿Quién se activa el código maligno? ¿Es el usuario al compartir su lista de la compra?
- ¿Cómo se explica que la aplicación se instale y todo funcione correctamente y que comiencen a desaparecer los contactos del usuario cuando el dispositivo se reenciende tras haberse apagado por falta de batería?
- ¿Cómo podemos resolver el problema? ¿Existe alguna solución que no requiera modificar el código Java de la aplicación, ni eliminar ninguna de las clases que la app integra? Buscamos encontrar una solución al problema que no sea excesivamente invasiva ni requiera de grandes destrezas en programación.
- **Ejercicio 5:** ¿Es complejo el diseño de este malware o es sencillo y fácilmente replicable en otras apps? De poderse hacer, ¿qué sería necesario para reproducir en otra app el comportamiento observado en la app bajo estudio? Aplica el malware a la aplicación de conversión de divisas. Incorpora a la memoria de la práctica una breve descripción del procedimiento seguido y proporciona en un fichero zip el proyecto resultante.

## 6. Evaluación de los ejercicios

Cada ejercicio valdrá 2 puntos. Alcanzarás un nivel **Junior** si resuelves los 3 primeros, un nivel **Máster** si resuelves el cuarto y un nivel **God** si resuelves el último. Intentar contestar a todas las preguntas planteadas. Tened en cuenta que en esto de la ciberseguridad muchas veces nos enfrentamos a problemas complejos y de idea feliz (necesidad de pensamiento lateral), por ello lo importante es intentarlo y aprender de los errores para poder tener más recursos la siguiente vez que uno afronta un reto similar.

Subid el informe en formato pdf con las respuestas y el fichero zip del ejercicio 5 al espacio compartido de poliformaT especificando claramente que son las respuestas a la práctica 1 de la asignatura. La práctica puede resolverse en grupos de hasta 2 alumnos. Si se hace en grupo, subid las respuestas sólo al espacio compartido de uno de los integrantes del grupo, pero dejad constancia en el espacio compartido del otro de que se ha hecho la práctica 1 y con quién se ha hecho.