

# Capítulo 4

## Intents, fichero de manifiesto y componentes

### Práctica 4.1: Una aplicación con dos ventanas

- Crea un proyecto que se llame *DosActivities* usando el asistente.
- Modifica la actividad predefinida quitándole la etiqueta "Hello world! " habitual y poniendo un botón con el texto "Púlsame". Asocia el evento de pulsación a un método `onPulsame(View v)`. De momento, deja el código vacío.
- Crea una actividad nueva. Para eso utiliza, por ejemplo, la opción del menú File - New - Activity. En la ventana de diálogo que aparece, despliega Android y selecciona Android Activity.
- Deja la nueva actividad en el proyecto actual (*DosActivities*), ponle como nombre *SecondaryActivity*, y como título (fichero manifiesto `android:label`) "Segunda ventana". Marca la casilla Launcher Activity.
- Modifica lo que necesites en el fichero `strings.xml` y en fichero `activity...xml` para que en esta segunda actividad se muestre el texto "Estas es la segunda ventana".
- Vuelve al código de `onPulsame()` de la primera actividad. Para invocar desde ahí a la segunda, necesitamos enviar a Android un intent ("intento"), en el que indiquemos qué queremos lanzar exactamente:

```
Intent i = new Intent(this, SecondaryActivity.class);
startActivity(i);
```

- Ejecuta la aplicación y pulsa el botón. Verás aparecer la segunda ventana, con el texto que hayas puesto.
- Pulsa "atrás". Verás que la ventana desaparece y se vuelve a la principal.

Un intent es una descripción abstracta de una operación que queremos que se realice, por lo que puede verse como un mensaje asíncrono. Es recogido por Android, que lo procesa y lanza el componente (activity) que se haya pedido.

En este ejemplo, estamos haciendo uso de lo que se conoce como modo de invocación explícito: hemos configurado con total exactitud la actividad que queremos que se lance, dando su clase Java.

Es importante ser consciente de que, al igual que ocurría al abrir un **AlertDialog**, el envío de un intent no detiene la ejecución de la actividad. Para comprobarlo, puedes añadir la generación de un toast tras el envío del intent y lo verás aparecer sobre la segunda ventana.

### Práctica 4.2: AndroidManifest.xml: Actividades

Tras realizar la práctica 4.1, si vas al lanzador de aplicaciones del dispositivo donde la hayas probado verás dos iconos referenciando a la aplicación, uno llamado Dos activities, y otro Segunda ventana. Si pulsas sobre el segundo icono, verás que se abre la segunda ventana.

El culpable de que aparezcan ambas es el fichero AndroidManifest.xml, que podrás encontrar en la raíz del proyecto (o del .apk). Contiene información esencial sobre la aplicación, que el sistema debe tener disponible antes de plantearse hacer uso de ella. El asistente de creación de nueva actividad lo ha modificado al activar la casilla de verificación Launcher Activity.

Si abres el fichero, verás que Eclipse organiza toda la información que contiene en cuatro pestañas diferentes (*Manifest*, *Application*, *Permission* e *Instrumentation*). También muestra una quinta pestaña con la vista del fichero en su versión en XML. No siempre hay una relación directa entre los elementos XML y las pestañas. Por ejemplo, aunque la pestaña Application sirve para definir todo lo referente al elemento XML **<application>** y sus subelementos, la pestaña Permissions sirve para definir elementos de tipo **<permissions>** y **<uses-permissions>** entre otros.

El número de opciones que podemos configurar en el fichero de manifiesto es muy grande, por lo que no analizaremos todas. Sí analizaremos sin embargo las que nos coloca el asistente en la versión predefinida.

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="libro.azul.dosactivities"
  android:versionCode="1"
  android:versionName="1.0" >

  <uses-sdk
    android:minSdkVersion="4"
    android:targetSdkVersion="10" />

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    ....
  </application>
</manifest>
```

- **<manifest>**: es el raíz del XML. Tiene tres atributos importantes:
  - ✓ **package**: contiene el nombre del paquete que sirve como identificador de la aplicación. No puede repetirse entre aplicaciones, pues se considerarían la misma.

- ✓ **versionCode**: número de versión. Debe ser siempre un número entero, e incrementarse de modo que una versión posterior tenga siempre un número mayor. Es un código interno no visible al usuario.
- ✓ **versionName**: nombre de la versión. Es el que verá el usuario, y puede contener cualquier texto.
- **<uses-sdk>**: mantiene la información sobre los niveles de API necesarios por la aplicación. En el atributo **minSdkVersion** se indica el mínimo nivel de API necesario para ejecutar la aplicación, y en **targetSdkVersion** la versión más alta contra la que se ha probado la aplicación.
- **<application>**: contiene información sobre la propia aplicación. Es el elemento que más ocupa. Tiene gran cantidad de atributos opcionales, aunque Eclipse sólo nos configura cuatro:
  - **allowBackup**: cierto para que una hipotética aplicación del sistema *de backup global* incluya también los datos de esta aplicación.
  - **icon**: enlace al recurso con el icono.
  - **label**: nombre de la aplicación para el lanzador.
  - **theme**: "tema" o estilo de los controles.

Observa que el valor de los atributos (salvo el primero) hace referencia a un recurso utilizando el esquema habitual **@<tipo>/<identificador>**.

Dentro del elemento **<application>** aparecen subelementos definiendo los componentes de la aplicación. De hecho, los atributos icon, label y theme que hemos visto sirven para darles valores por defecto, pero pueden ser sobrescritos en la definición de los componentes.

Por el momento, el único tipo de componente que conocemos son las actividades. En el fichero de manifiesto creado en la práctica 5.1 puedes comprobar que aparecen dos elementos de tipo **<activity>**, dado que hemos creado dos actividades:

```
<activity
  android:name="libro.azul.dosactivities.MainActivity"
  android:label="@string/app_name" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
<activity
  android:name="libro.azul.dosactivities.SecondaryActivity"
  android:label="@string/title_activity_secondary" >
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

El atributo más importante de los elementos XML que definen componentes es **android:name**, que indica el nombre de la clase que lo implementa. Cuando Android tiene que lanzar una actividad, busca el nombre de la clase en ese atributo, crea una nueva instancia suya y la ejecuta. En concepto, es similar a la referencia a la clase de un applet en una página HTML.

El nombre de la clase debe estar totalmente cualificado (debe incluir el paquete). Si se quiere omitir, se puede comenzar el nombre con un punto, y se asumirá el paquete definido en el atributo package del elemento <manifest>. Por tanto, Eclipse podría haber puesto android:name=".MainActivity".

Eclipse también nos ha establecido un nuevo atributo label para indicar el nombre de la actividad. Gracias a ello hemos podido diferenciar las dos ventanas en el lanzador de aplicaciones, al tener cada una su nombre.

Además, dentro de <activity> nos ha añadido un elemento adicional <intent-filter> o "filtro de intents. Este elemento sirve para indicar en qué circunstancias se lanzará esa actividad.

- Modifica el fichero de manifiesto y comenta (encerrando entre <!-- y --> la declaración completa de la segunda actividad.
- Ejecuta la aplicación de nuevo. Verás que falla al pulsar el botón para abrir la segunda ventana.
- Mira la salida del logcat. Verás que ha saltado una excepción del tipo ActivityNotFoundException. Sólo las actividades registradas en el fichero de manifiesto pueden ser ejecutadas, incluso aunque la clase exista realmente. Por tanto deben declararse todas.
- Observa que en el lanzador de aplicaciones, el icono a la segunda actividad ya no aparece.
- Vuelve al fichero de manifiesto, descomenta la declaración de la actividad .SecondaryActivity y comenta (o elimina) únicamente el elemento interno <intent-filter>.
- Ejecuta de nuevo la aplicación. Verás que ahora sí funciona.
- Ve al lanzador de aplicaciones del dispositivo, y comprueba que el icono de la segunda actividad sigue sin aparecer.

El significado de:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

es "esta actividad puede ser lanzada para realizar su tarea principal (MAIN) desde el lanzador de aplicaciones (LAUNCHER).

Al eliminar esta sección, quitamos esa posibilidad de inicio, y Android no incluye el icono en el lanzador.

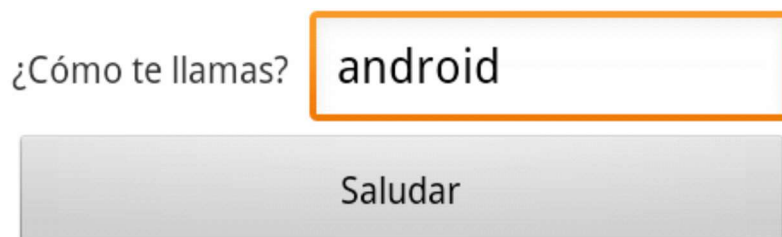
### Práctica 4.3: Enviando datos a la segunda ventana

Cuando configuramos un intent, podemos añadirle información adicional que podrá ser leída, en última instancia, por la actividad que Android termine lanzando. Esa información no es más que una lista de pares atributo-valor.

- Desde el lado del invocante, la clase Intent tiene varios métodos sobrecargados con el nombre putExtra(atributo, valor). El atributo siempre será una cadena, pero el valor puede tener una gran cantidad de tipos.
- Desde el punto de vista del invocado, la actividad tiene acceso al intent que ocasionó su invocación (o al menos a una copia) a través del método **getIntent()**. Una vez que lo tiene, puede usarse cualquiera de los métodos get\*Extra() (por ejemplo **getIntExtra()** o **getStringExtra()**) para conseguir los valores establecidos por el invocante. Todos esos métodos tienen el atributo buscado (en la lista de pares) como primer parámetro y, algunos, tienen un segundo parámetro con el valor por defecto que deseamos obtener si no hay ningún atributo con ese nombre en los datos enviados por el invocante.

Con esta información:

- Haz una copia del proyecto de la práctica anterior y ponle el nombre DosActivitiesPutExtra.
- Modifica la actividad principal para que pregunte al usuario su nombre.
- En el evento del botón, obtén el nombre del cuadro de texto. Si está vacío, muestra un toast indicando el problema. En otro caso, envía a la segunda actividad el nombre a través del atributo extra de clave nombre.



```
Intent i = new Intent(this, SecondaryActivity.class);  
i.putExtra("nombre", elNombre); // Sacado del EditText  
startActivity(i);
```

- En la segunda ventana, haz que se se muestre "Hola, <nombre>, desde la segunda ventana". Para eso, añade en el método onCreate() el acceso al atributo pasado como parámetro.

```
String nombre = getIntent().getStringExtra("nombre");  
String saludo = getResources().getString(R.string.<idStr>);
```

```
saludo = String.format(saludo, nombre);  
TextView etiqueta = (TextView) findViewById(R.id.<idLabel>);  
etiqueta.setText(saludo);
```

## Práctica 4.4: Recogiendo valores de la segunda ventana

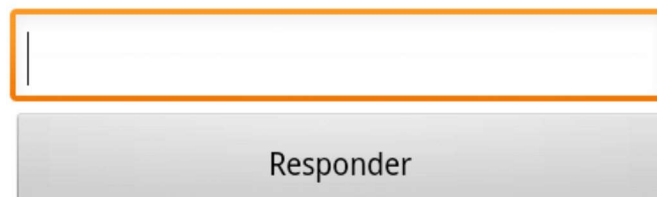
Al igual que desde la actividad invocante podemos enviar datos a la nueva, también podemos enviar datos en el sentido inverso. Para eso:

- Al "ejecutar" el intent, usaremos el método `startActivityForResult()`.
- La segunda ventana creará un nuevo intent que enviará de vuelta a la actividad invocante usando el método `setResult()`.
- La primera actividad recogerá de manera asíncrona el resultado en su método `onActivityResult()`.

La invocación a `startActivityForResult()` tampoco es bloqueante. La ejecución continúa; el resultado lo recibiremos por una llamada asíncrona al método `onActivityResult()`. Dado que una actividad podría querer abrir varias actividades en diferentes momentos, y siempre se recibirá el resultado a través del método `startActivityForResult()`, éste recibe un entero (que normalmente se denomina `requestCode`) que será enviado de vuelta por Android en la invocación a `onActivityResult()` correspondiente. De ese modo, la actividad puede utilizar códigos numéricos diferentes en la apertura de diferentes actividades y así conocer qué respuesta debe procesar en cada momento.

- Haz una copia del proyecto `DosActivitiesPutExtra` y dale el nombre `DosActivitiesForResult`.
- Modifica la segunda actividad para que incluya un nuevo cuadro de texto y un botón, donde el usuario podrá introducir un saludo de vuelta. Por ejemplo, si el usuario introduce "Android" como nombre, deberá ver algo similar a:

Hola, Android, desde la segunda ventana

The image shows a user interface element consisting of a white rectangular text input field with a thin orange border. Below the input field is a gray rectangular button with the word "Responder" centered on it in a dark gray font.

- Modifica `MainActivity.java` para que haga uso del mencionado método `startActivityForResult()`. Para eso, define una constante nueva para ser usada como "marca" para diferenciar la actividad que envía un valor de vuelta:

```
private static final int SECONDARY_ACTIVITY_TAG = 1;
```

y sustituye la invocación original con:

```
startActivityForResult(i, SECONDARY_ACTIVITY_TAG);
```

- Añade a SecondaryActivity.java el código para el evento de la pulsación del botón, donde se enviará de vuelta a la actividad invocante el nombre recibido. Además, se cerrará la actividad con el método finish().

```
String respuesta;  
EditText et = (EditText) findViewById(R.id.etRespuesta);
```

```
respuesta = et.getText().toString();
```

```
Intent datos = new Intent();  
datos.putExtra("respuesta", respuesta);  
setResult(RESULT_OK, datos);  
finish();
```

Fíjate que la respuesta está compuesta de dos elementos:

- ✓ Un código de respuesta (en este caso RESULT\_OK). Podemos también devolver RESULT\_CANCELED o, en general, cualquier valor numérico mayor o igual que la constante RESULT\_FIRST\_USER.
  - ✓ Un intent nuevo con datos extra.
- De nuevo en MainActivity.java, crea el nuevo método al que llamará onActivityResult(). En él, mostraremos en un toast el saludo que ha escrito el usuario en la segunda ventana, recogiénolo del intent que recibimos como parámetro. Ten en cuenta que si el usuario cierra (con el botón "volver") la actividad secundaria, Android nos avisará de todos modos, indicando RESULT\_CANCELED como resultado:

```
@Override  
protected void onActivityResult(int requestCode,  
                                int resultCode,  
                                Intent data) {  
    String respuesta;  
    if ((resultCode == RESULT_CANCELED) ||  
        (data.getStringExtra("respuesta").equals("")))  
        respuesta = getResources().getString(  
            R.string.antipatico);  
    else  
        respuesta = data.getStringExtra("respuesta");  
    Context contexto = getApplicationContext();  
    Toast.makeText(contexto, respuesta,  
        Toast.LENGTH_LONG).show();  
} // onActivityResult
```

- Ejecuta el programa y pruébalo. Comprueba que al volver desde la segunda ventana, en la primera aparece el toast con el nombre. Si en la segunda ventana no escribes nada o pulsas "Volver", la primera ventana te llama antipático.

### Práctica 4.5: Llamando a una actividad de otra aplicación

Ya se ha mencionado que cuando se envía un intent, es Android quién lo recoge, lo analiza y se encarga de lanzar la actividad correspondiente. En principio esto parece demasiado trabajo para lanzar una ventana de la misma aplicación.

El objetivo, sin embargo, es conseguir que sea igual lanzar una actividad implementada en nuestra aplicación, que una ventana de otra.

En esta práctica vamos a hacer una nueva aplicación que será una copia de la anterior, pero que tendrá únicamente la primera actividad, e invocará a la segunda de la original.

- Haz una copia del proyecto DosActivitiesForResult y dale como nombre InvocacionRemota.
- Elimina la segunda actividad. Para eso:
  - ✓ Borra el fichero SecondaryActivity.java
  - ✓ Borra el fichero activity"secondary.xml
  - ✓ En el fichero de manifiesto, elimina el elemento <activity> que declaraba la actividad SecondaryActivity.
- Renombra el paquete de la aplicación, para que sea diferente al original y podamos instalar ambas en el mismo dispositivo. Para eso:
  - ✓ En el package explorer, selecciona el paquete original cuyo nombre era libro.azul.dosactivities, pulsa con el botón derecho y elige Refactor - Rename...
  - ✓ Pon como nombre libro.azul.invocacionremota.
  - ✓ En el fichero de manifiesto, modifica el atributo package del elemento <manifest> y pon el nuevo nombre.
- Modifica el nombre de la aplicación que verá el usuario. Para eso, en el fichero strings.xml modifica la cadena cuyo identificador es app\_name y pon "InvocacionRemota". Elimina, si quieres, las cadenas que se usaban en la actividad secundaria y que ahora no referencia nadie.
- Dado que hemos copiado la actividad principal, para diferenciarla de la original modifica el layout de la nueva que hemos creado aquí y ponle android:background="#80ff80".
- Eclipse mostrará un error en MainActivity.java, en la línea de código fuente donde se configura el objeto de la clase Intent. Originalmente lo configurábamos pasándole directamente la clase de la actividad a invocar. Ahora esa clase no la tenemos disponible, por lo que tenemos que indicar la actividad a partir de su nombre. Sustituye el código original:

```
Intent i = new Intent(this, SecondaryActivity.class);
```



por:

```
Intent i = new Intent();
i.setClassName("libro.azul.dosactivities",
    "libro.azul.dosactivities.SecondaryActivity");
```

y deja el resto igual (establecimiento del nombre e invocación a la actividad).

- Ejecuta la práctica 4.4 para estar seguro de que el dispositivo donde lo pruebes tiene la última versión. Cierra el programa.
- Ejecuta ahora el proyecto nuevo de invocación remota, introduce un nombre y pulsa el botón. Al hacerlo, obtendrás un error. Mira en el logcat y verás que se trata de un problema de seguridad. La aplicación libro.azul.dosactivities no permite la invocación externa a su actividad secundaria. Para permitirlo, debe decirlo explícitamente en su fichero de manifiesto.
- Modifica el fichero de manifiesto de la práctica anterior (de nombre libro.azul.dosactivities). En la declaración de la actividad secundaria, añade un intent-filter que dé permiso para la ejecución externa:

```
<activity
    android:name=".SecondaryActivity"
    android:label="@string/title_activity_secondary" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
    </intent-filter>
</activity>
```

Fíjate que no hemos añadido la categoría LAUNCHER, por lo que la segunda actividad seguirá sin aparecer en el lanzador de aplicaciones.

- Vuelve a ejecutar la práctica original para que se envíe al dispositivo el cambio en su manifiesto.
- Ejecuta ahora la nueva, y verás que la invocación remota sí funciona.

Ten en cuenta que la actividad principal está en una aplicación, y la segunda está en otra. Android protege las aplicaciones ejecutándolas en procesos diferentes. Vamos a comprobarlo:

- Detén las dos aplicaciones. Para eso, si estás ejecutándolas en un AVD, puedes usar la consola:

```
$ ./adb shell
# ps
# kill <pid de dosactivities>
# kill <pid de invocacionremota>
```

Si estás usando un dispositivo físico, ve a la configuración de las aplicaciones, selecciona cada una de ellas y pulsa, si está habilitado, el botón Forzar detención.

- Usando el adb shell, asegúrate de que ninguna está lanzada (esto podrás hacerlo también sobre un dispositivo físico, aunque no seas root). Deja abierta la conexión con el shell.
- Lanza Invocación remota. Usa ps en el shell para ver la aparición del proceso. Asegúrate de que el proceso de la práctica anterior Dos Activities no aparece.
- Introduce un nombre y pulsa el botón.
- Vuelve al shell, ejecuta ps y nota la aparición del nuevo proceso para esa actividad.

## Práctica 4.6: Invocaciones implícitas con Intents

En los dos ejemplos anteriores hemos invocado una segunda actividad usando el llamado método explícito, en el que se proporciona la clase exacta que deberá procesar nuestra solicitud.

Sin embargo, los intents se han diseñado para usarlos de una manera mucho más potente llamada uso implícito, en el que no se proporciona qué actividad debe procesar la solicitud. El objetivo es que los intents creen un "lenguaje" en el sistema para que una aplicación pueda decir "quiero llamar a casa", y el sistema sepa qué actividad es capaz de ejecutar ese deseo. El nombre intent (intención) encaja con esa idea, en el que las aplicaciones notifican su deseo de que algo ocurra.

Con esta idea en mente, los intents poseen, además del nombre de la clase que hemos usado y los datos extra (setClassName() y putExtra()) varios campos más. Ten en cuenta que en las invocaciones implícitas nunca se establece el nombre de la clase, y el sistema hará uso del resto de campos para deducir qué actividad es capaz de atender la solicitud. En concreto:

- **Acción:** es una cadena que indica la acción que se desea realizar. Se establece con setAction(), o incluso pasándola en el constructor.
- **Datos:** una URI sobre la que ejecutar la acción. Se establece con el método setData(Uri u). Si la URI se posee a través de una cadena, será necesario convertirla, por lo que la construcción más habitual será setData(Uri.parse("...")).
- **Tipo de los datos:** si a partir de la Uri no es posible identificar el tipo de datos, podemos indicárselo explícitamente al intent con setType(). El parámetro es una cadena, indicando el tipo MIME (por ejemplo image/\*). Podemos establecer la URI y el tipo simultáneamente con setDataAndType(Uri, String). En ocasiones, queremos especificar un tipo de datos pero no una URI.
- **Categorías:** un intent puede especificar varias categorías para la solicitud. Cada categoría es una cadena.

De todos estos campos, el más importante es la acción. Android especifica una gran cantidad de acciones, declaradas como constantes dentro de la clase Intent. Podemos considerar a las acciones como el nombre de un método. Es la acción la que determina cómo se utilizará el resto de la información contenida en el intent.

Como se ha dicho, dentro de la clase Intent se definen bastantes constantes con cadenas de acciones predefinidas que comprende el sistema. En esta práctica vamos a probar algunas de ellas, enviando diferentes solicitudes al sistema Android utilizando intents para ver algunas de las posibilidades que nos ofrece. Por comodidad, reutilizaremos el mismo proyecto, y nos limitaremos a ir cambiando la configuración del intent que generaremos.

- Crea un nuevo proyecto en Eclipse y llámalo IntentImplicito. Elige las opciones habituales.
- Modifica la ventana para que tenga un único botón que ejecute el método `onBotonEjecutar()`. Por ejemplo, suponiendo que mantienes el `RelativeLayout` inicial:

```
<Button
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_centerInParent="true"
    android:text="@string/ejecutarIntent"
    android:onClick="onBotonEjecutar" />
```

donde la cadena `@string/ejecutarIntent` se habrá definido como "Ejecutar intent implícito".

- En el código Java, crea siguiente método:

```
public void onBotonEjecutar(View v) {
    Intent i = new Intent();
    i.setAction(Intent.ACTION_POWER_USAGE_SUMMARY);
    startActivity(i);
}
```

También podríamos haber usado la inicialización directamente en el constructor:

```
public void onBotonEjecutar(View v) {
    Intent i = new Intent(Intent.ACTION_POWER_USAGE_SUMMARY);
    startActivity(i);
}
```

- Ejecuta el programa. En este caso, es preferible que lo ejecutes en un dispositivo físico.

Verás que la acción solicitada sirve para lanzar la actividad que muestra el resumen de uso de la batería. Nos muestra una estimación del tiempo que durará la batería, y los porcentajes de uso de batería de las diferentes aplicaciones disponibles<sup>1</sup>. Dado que los

---

<sup>1</sup> El resultado podría variar dependiendo del teléfono concreto.

AVD no simulan la batería con tanta exactitud, en este caso el resultado en el emulador será mucho más pobre.

La acción seleccionada no necesita información adicional, por lo que no hemos necesitado establecer más campos.

Hagamos uso ahora de una acción que necesita una URI.

- Comenta, si quieres conservarlo, el código anterior donde configurábamos el intent.
- Configura el intent para que la acción sea ACTION\_VIEW, y en la URI pon una dirección Web:

```
public void onBotonEjecutar(View v) {  
    Intent i = new Intent();  
    /*  
    // Lanzamos la actividad que muestra el resumen  
    // de uso y estado de la batería.  
    i.setAction(Intent.ACTION_POWER_USAGE_SUMMARY);  
    /**/  
    // Lanzamos un navegador Web.  
    i.setAction(Intent.ACTION_VIEW);  
    i.setData(Uri.parse("http://www.programa-me.com"));  
    startActivity(i);  
}
```

- Ejecuta la aplicación. Si no tienes tarifa de datos en el móvil, utiliza mejor un AVD.

Al hacer uso de la acción ACTION\_VIEW, el sistema interpreta que queremos ver la URI que le pasamos como datos, por lo que nos abre un navegador Web y nos la muestra.

Es interesante observar que la acción ACTION\_VIEW está "sobrecargada", y dependiendo de la URI que pongamos se abrirá una aplicación u otra.

- Comenta la configuración del intent de la prueba anterior.
- Establece la siguiente configuración:

```
// Abrir la edición de un SMS  
i.setAction(Intent.ACTION_VIEW);  
i.setData(Uri.parse("sms:5554433"));
```

- Ejecuta el programa. Verás aparecer la aplicación para escribir mensajes cortos, dado que en la URI hemos puesto sms: en lugar de http://.

Esto nos demuestra que en algunas ocasiones Android hace uso únicamente de la acción para decidir qué aplicación lanzar (como ocurría en el ejemplo de las estadísticas de uso de la batería), pero a veces necesita analizar también la URI.

Precisamente eso es lo que ocurre aquí; como la acción ACTION\_VIEW es muy genérica, Android necesita analizar la URI y al ver que hace referencia a mensajes cortos lanza la aplicación adecuada.

En este caso, podríamos haber lanzado el editor de mensajes SMS de otro modo. En lugar de establecer los datos, podríamos haber establecido el tipo MIME a "mensaje de texto", y haber proporcionado el número de teléfono a través de los datos extra:

```
i.setAction(Intent.ACTION_VIEW);  
i.setType("vnd.android-dir/mms-sms");  
i.putExtra("address", "5554433");
```

Si ejecutas la aplicación, el resultado será exactamente el mismo. Las aplicaciones receptoras de intents especifican "el protocolo" que comprenden, y que deberá ser utilizado por los usuarios que quieran invocarlas. No obstante, esta prueba nos demuestra que Android puede utilizar también el tipo (en lugar de la URI) para seleccionar la actividad que se hará cargo de atender un intent.

Aunque en el ejemplo de los SMS tengamos dos alternativas, normalmente habrá solo una. Además, los datos extra (que ponemos con putExtra()) no pueden ser utilizados por Android para seleccionar la actividad receptora del intent. Esa es la razón de que se consideren "datos extra": Android los ignora durante la toma de la decisión.

En este sentido se debe tener en cuenta que las aplicaciones pueden crear sus propias acciones para que otras las utilicen<sup>2</sup>. Incluso, pueden reutilizar las acciones existentes, y diferenciarse de las demás a través de la URI o del tipo MIME. Por ejemplo, la aplicación Skype informa a Android de que es capaz de procesar intents con la acción VIEW ACTION, pero con URIs del tipo skype:<usuario>.

La conclusión de esto es que la información más importante para una acción irá siempre en la URI, y es el uso que se recomienda. Los datos extra deben utilizarse para información adicional que no afecte a la elección de la actividad destino.

- Modifica la aplicación para añadir al intent información extra:

```
i.putExtra("sms_body",  
    "Hola! Estoy aprendiendo Android :-");
```

- Lanza la aplicación, pulsa el botón, y verás que se abre la ventana de edición de SMS, con un mensaje parcialmente escrito. La aplicación de los SMS mira si se ha establecido en los datos extra algo asociado al atributo sms\_body y lo establece como texto inicial.

---

<sup>2</sup> Para eso, necesitarán conocer la cadena de la acción. Fíjate que nosotros aquí hemos estado usando las constantes definidas en la clase Intent, que serán cadenas.

- Si das hacia atrás, para cerrar la escritura del mensaje, la actividad funciona como normalmente y dejará el mensaje en los borradores.

Esta opción funciona tanto en la configuración del intent a través de la URI como a través del tipo MIME.

La invocación a otras actividades también podemos utilizarla para recibir datos. Aunque no vamos a entrar en ello, sí podemos probar que, al menos, la actividad se lanza. Además, en este ejemplo, lo importante no es la URI (que no hay) sino únicamente el tipo de datos:

- Comenta el código de configuración del intent que hiciste en la prueba anterior.
- Establece el siguiente:

```
i.setAction(Intent.ACTION_GET_CONTENT);
i.setType("image/*");
```

- Lanza la aplicación y pulsa el boton. Verás aparecer la galería multimedia, mostrando las imágenes y fotos que tengas en el dispositivo. En este caso es preferible utilizar un móvil físico, porque los AVDs no traen ninguna.
- Selecciona una foto. Observa que no se muestra. En su lugar, se vuelve a nuestra actividad. Si hubiéramos usado `startActivityResult()`, habríamos recibido una URI de vuelta con la localización de la imagen seleccionada.
- Modifica el tipo MIME pedido, y pon `audio/*`. Relanza la aplicación y nota que ahora se solicita un fichero de audio.

Para llamar por teléfono no usamos `ACTION_VIEW` como en el caso de los mensajes cortos:

- Comenta la configuración del intent de la última prueba.
- Configúralo para usar la acción `ACTION_DIAL`, y establece en la URI el número de teléfono al que quieres llamar:

```
i.setAction(Intent.ACTION_DIAL);
i.setData(Uri.parse("tel:5554433"));
```

- Ejecuta la aplicación y pulsa el botón.
- Observa que se abre la ventana de marcación, con el teléfono indicado en la URI ya escrito.
- Si lo ejecutas sobre un teléfono físico y pones un número de teléfono que esté en la agenda, la aplicación buscará el nombre y te lo mostrará.

La prueba anterior no realizaba la llamada. Tan sólo la "configuraba" para que el usuario la aceptara. Podemos iniciar la llamada usando una acción diferente.

- Copia la configuración del intent de la última prueba, y comenta una de las dos para que puedas conservarla.
- Modifica la acción para usar ACTION\_CALL en lugar de ACTION\_DIAL.
- Lanza la aplicación y pulsa el botón.
- La aplicación falla.
- Mira el contenido del logcat. Observa que se debe a un problema de seguridad y permisos.

## Práctica 4.7: Permisos

Como medida de protección, las aplicaciones que necesitan realizar tareas que pueden perjudicar al sistema (por ejemplo, impedir que entre en modo de suspensión) o al usuario (por ejemplo, realizar llamadas o detectar la posición a través del GPS) deben declararlo en su fichero de manifiesto. Cuando se instala una aplicación nueva, el usuario debe conceder explícitamente todos esos permisos. De otro modo, la aplicación no se instalará.

Una vez que la aplicación se ha instalado, el usuario no vuelve a ser preguntado por el uso de esos permisos, lo que evita la aparición de mensajes continuos que el usuario dejaría pronto de leer y aceptaría por costumbre.

Si una aplicación intenta hacer uso de una característica de Android delicada para la que no ha pedido permiso explícito en su fichero de manifiesto, Android la detendrá generando una excepción de seguridad, tal y como hemos visto.

- En la práctica anterior, abre el fichero de manifiesto.
- Ve a la pestaña Permissions, y pulsa Add...
- Selecciona Uses permission, que se utiliza para solicitar permiso de uso de una característica protegida.
- Despliega los posibles permisos disponibles. Selecciona el que tiene por nombre android.permission.CALL\_PHONE.
- En la vista XML, comprueba que ha aparecido un nuevo elemento:

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

- Vuelve a ejecutar la aplicación. Verás que ahora sí se realiza la llamada.
- Modifica el teléfono y pon un número que conozcas. Ejecútalo en tu teléfono y verás que la llamada se realiza, y aparece el nombre correspondiente de la agenda.

## Práctica 4.8: Exportación e instalación

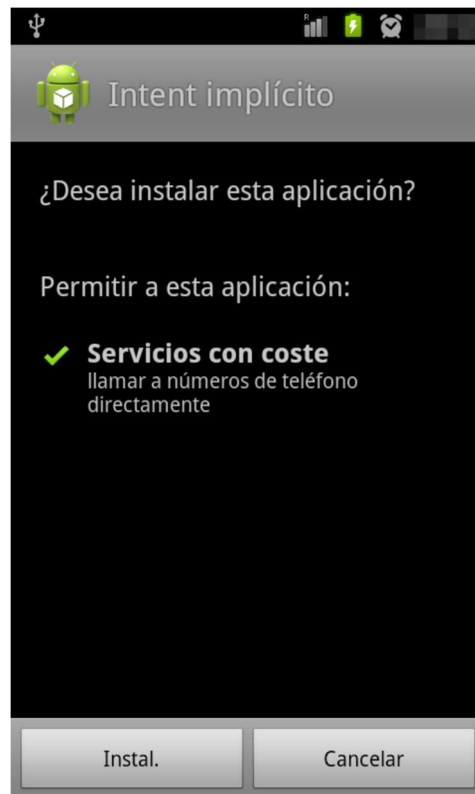
En la práctica 4.7 creamos una aplicación que requería un permiso determinado, y sin embargo cuando la ejecutamos no nos preguntó si queríamos darle ese permiso. Por tanto, ¿dónde está la protección de la que alardea Android?

Si no nos preguntó fue porque lanzamos la aplicación con la depuración por USB habilitada. Durante el desarrollo, el dispositivo no nos molesta con las preguntas de los permisos, pues asume que sabe lo que estamos haciendo.

En esta práctica vamos a crear un .apk que podríamos distribuir (o publicar en Google Play) y lo instalaremos, para ver la solicitud de concesión de permisos.

- En Eclipse, selecciona la opción del menú File - Export.
- En el cuadro de diálogo, escoge Android - Export Android Application.
- Selecciona el proyecto que quieres exportar (el de la práctica 4.7).
- Crea el certificado digital que usarás para firmar el proyecto. El certificado te identifica como desarrollador, y debe ser el mismo para todas las versiones futuras de la aplicación. De otro modo se considerarían aplicaciones diferentes, dificultando la actualización.
- Elige la localización y nombre del fichero .apk que vas a generar.
- Copia el .apk al móvil. Para eso:
  - ✓ Deshabilita la depuración por USB.
  - ✓ Conecta el móvil al ordenador.
  - ✓ Copia el .apk a algún lugar de la memoria interna.
- Ya puedes desconectar el móvil del ordenador.
- Ve a la configuración de las aplicaciones y marca "Fuentes desconocidas ". De otro modo, el teléfono se negará a instalar un .apk que no venga de Google Play u otros markets reconocidos por él.
- Ve al gestor de aplicaciones y asegúrate de que la versión que se instaló para depurar desde Eclipse no está ejecutándose. Puedes incluso desinstalarla.
- Abre el explorador de archivos del móvil, y selecciona el .apk que copiaste antes, para iniciar la instalación.
- El móvil te muestra los permisos que el .apk declara que necesita. Verás que solicita un servicio con coste ("llamar a números de teléfono directamente"). Acepta la instalación.





- Ejecuta el programa y comprueba que funciona correctamente.
- Para evitar problemas en el futuro:
  - ✓ Desinstala la aplicación.
  - ✓ Borra el .apk.
  - ✓ Desactiva la opción "Fuentes desconocidas".

### Práctica 4.9: Receptores de mensajes del sistema

Hasta ahora hemos utilizado únicamente actividades. En Android, las aplicaciones están creadas de componentes, y las actividades son únicamente uno de los tipos de componentes posibles.

Es importante darse cuenta de que desde el punto de vista del programador, el sistema no ejecuta procesos, sino componentes. En nuestra aplicación no hay programa principal (igual que no lo hay en los applets). En su lugar, en el fichero de manifiesto debemos declarar los componentes que proporcionamos, y bajo qué circunstancias deben ser lanzados.

Los componentes se ejecutan siempre a través de intents. Incluso el lanzador de aplicaciones del teléfono utiliza un intent para lanzar los programas (¿recuerdas la acción MAIN y la categoría LAUNCHER?). El sistema controla el ciclo de vida de los componentes y de los procesos, eliminando los menos importantes cuando surgen necesidades adicionales de memoria.

En esta práctica vamos a ver un nuevo tipo de componente, llamado broadcast receivers o receptores de mensajes del sistema. Estos componentes se lanzan

automáticamente cuando se produce algún evento del sistema en el que están interesados. Para eso, en el fichero de manifiesto de la aplicación deben indicar de qué sucesos quieren ser notificados, y sólo cuando ocurran Android instanciará el componente y le dará la oportunidad de ejecutarse.

Estos componentes no poseen interfaz de usuario. Algunas veces introducen avisos en la barra de notificaciones. También pueden, naturalmente, ocasionar la apertura de una actividad enviando un intent, para que el usuario pueda actuar en consecuencia.

- Crea un proyecto nuevo en Eclipse llamado BroadcastReceiver. Como nombre del paquete pon libro.azul.broadcastreceiver. En este caso no crees una actividad.
- El proyecto creado estará vacío. Crea una clase nueva con el menú File - New - Class.
- En el cuadro de diálogo que se muestra, escribe como nombre del paquete libro.azul.broadcastreceiver (el mismo que pusiste en el nombre del paquete de la aplicación). Como nombre de la clase escribe DeteccionEnchufado, y como superclase escribe (o busca) android.content.BroadcastReceiver, el equivalente a la clase android.app.Activity de la que heredan siempre nuestras actividades. Asegúrate de que la casilla de verificación Inherited abstract methods está marcada para que Android incluya sus prototipos automáticamente.
- El código fuente que nos crea Eclipse tiene definido un método llamado onReceive(), sin código, que es abstracto en la clase padre. Ese método será llamado automáticamente por Android cuando ocurra un suceso de nuestro interés. Observa que recibe como segundo parámetro un intent con la información que ha ocasionado la invocación. Esto demuestra que los mensajes del sistema también se envían a través de intents. En este caso, la acción del intent llevará la información sobre el suceso ocurrido.
- Vamos a hacer un receptor de mensajes del sistema que escriba en el registro (logcat) un aviso cada vez que se conecta o desconecta el cable de alimentación. Para eso, escribe el siguiente código en el método onReceive():

```
if (i.getAction().equals(
    Intent.ACTION_POWER_CONNECTED))
    android.util.Log.i(TAG, "Cargador conectado");
else if (i.getAction().equals(
    Intent.ACTION_POWER_DISCONNECTED))
    android.util.Log.i(TAG, "Cargador desconectado");
```

Añade también el atributo estático TAG:

```
private static final String TAG = "BroadcastReceiver";
```

- Antes de poder ejecutar la aplicación, necesitamos declarar el nuevo componente en el fichero de manifiesto. Eclipse no lo habrá hecho por nosotros como hizo con la actividad, dado que no hemos usado un asistente del

ADT, sino uno genérico de Eclipse. Abre el fichero de manifiesto, y dentro del elemento <application> añade la definición del nuevo componente:

```
<receiver android:name=".DeteccionEnchufado">
</receiver>
```

El nombre de la clase comienza con punto, por lo que se da por supuesto que pertenece al mismo paquete que la aplicación.

- Nos falta definir en qué sucesos estamos interesados. Para eso tenemos que hacer uso de un elemento <intent-filter>, que establece filtros a los intents que deberían llegarnos. Dentro del elemento <receiver> que acabas de crear, añade:

```
<intent-filter>
    <action android:name=
        "android.intent.action.ACTION_POWER_CONNECTED"/>
</intent-filter>
<intent-filter>
    <action android:name=
        "android.intent.action.ACTION_POWER_DISCONNECTED"/>
</intent-filter>
```

Esto indica que nos interesan los mensajes relacionados con la conexión y desconexión del cargador. Fíjate que en el código fuente utilizamos la constante Intent.ACTION\_POWER\_DISCONNECTED. En el fichero de manifiesto debemos usar su valor (que siempre es una cadena).

- La aplicación la ejecutaremos sobre un AVD para no tener que conectar y desconectar el móvil físico a una toma de corriente para probar nuestro programa. Lanza, si no lo habías hecho ya, un AVD.
- Para estar seguros del estado de partida del AVD, usa el interfaz en consola y "desconecta" el cable de alimentación. Para eso, conéctate por telnet con el puerto par en el que está escuchando el AVD (recuerda que puedes saber cual es mirándolo en el título de la ventana del AVD):

```
$ telnet localhost 55xx
...
OK
power ac off
OK
```

El comando power ac off "desconecta" el cable de alimentación.

No cierras esta conexión; la vamos a utilizar más adelante.

- Ejecuta la aplicación sobre el AVD. Verás que no ocurre nada. Al fin y al cabo, nuestra aplicación no declara ninguna actividad principal.

- En una ventana nueva, abre una sesión con el shell y lista los procesos.

```
$ ./adb shell
# ps
```

Asegúrate de que no está lanzada nuestra aplicación.

- En el mismo shell, muestra el registro:

```
# logcat
```

Verás aparecer mucha información de sucesos, y se quedará bloqueado para mostrar a continuación cualquier otro mensaje que llegue.

- Vuelve a la ventana donde tenías la conexión de telnet abierta. Simula ahora la conexión del cable de alimentación:

```
power ac on
OK
```

- Ve a la ventana del shell donde se está mostrando la salida de logcat. Observa la aparición de nuestro mensaje:

```
I/BroadcastReceiver( PID): Cargador conectado
```

- Repite el proceso, desconectando la alimentación (con power ac off). Comprueba que aparece el mensaje de log correspondiente.
- Detén la salida de logcat pulsando Ctrl-C. Ejecuta ps para comprobar la aparición de nuestro proceso (cuyo PID coincide con el que aparecía en el logcat).
- Puedes eliminarlo con kill, y volver a simular la conexión y desconexión del cable de alimentación. Android volverá a ejecutarlo para notificarle el cambio, y el proceso reaparecerá (con otro PID).

Esta práctica ejemplifica que los receptores de mensajes son lanzados directamente por el sistema y no tienen que mantener un "demonio" en ejecución. Al registrar en el fichero de manifiesto los mensajes que nos interesan, Android lanzará el componente cuando ocurran.