Apuntes

Descargar estos apuntes

Tema 3 . Componentes de una aplicación Android

Índice

Introducción

En este tema estudiaremos los componentes de una aplicación Android y sus respectivas funciones, además de la comunicación entre activitis.
Para cualquier aplicación Android, en general, será necesario construir una ventana en la que se irán añadiendo componentes, cada componente es un punto de entrada por el que el sistema o un usuario interactúa con tu aplicación. Algunos componentes dependen de otros.
Lo primero que haremos será comprender que es el contexto de una aplicación y como permite relacionar los componentes.
Luego veremos el concepto de **Actividades, Intents, Servicios, BroadCast Receivers y Content Providers**.

Contexto de una aplicación

El contexto de una aplicación es una interfaz entre la aplicación y el sistema operativo, la cual describe la información que representa tu aplicación dentro del ambiente del sistema operativo. También permite acceder a los recursos de la aplicación y coordinar el funcionamiento de los bloques de la aplicación.
El contexto representa toda la meta-información sobre las relaciones que tiene la aplicación con otras aplicaciones o el sistema y podemos implementarlo a través de la **clase abstracta Context**.
En Android nos encontramos con los siguientes tipos de contextos:

- **Aplicación**. Este contexto engloba a todos los demás, y cubre todo el ciclo de vida de la aplicación desde que la arrancamos hasta que muere. Por lo que cada aplicación tiene ***un único contexto de aplicación***. El Context de la aplicación vive hasta que se termina por completo la aplicación (hasta que muere, no hasta que se pausa)
  Se puede acceder desde una ***Activity*** o un ***Service*** con `getApplication()` o desde cualquiera que herede de Context con `getApplicationContext()`.
- **Activity o Service**. Como hemos dicho, un Context vive tanto como el elemento al que pertenece, por lo que depende del ciclo de vida. Así, un Context de una Activity vivirá el mismo tiempo que vive esta la activity y siempre será vivirá menos que el de la Aplicación.

Componentes de una aplicación

Activity

Las Activitys son los componentes visibles de la aplicación, tienen una interface de usuario que permite interactuar con ellas.
La mayoría de las aplicaciones permiten la ejecución de varias acciones a través de la existencia de una o más pantallas. Por ejemplo: *una aplicación de mensajes de texto podrá tener la lista de contactos que se muestra en una ventana, mediante el despliegue de una segunda ventana el usuario puede escribir el mensaje al contacto elegido, y en otra tercera puede repasar su historial de mensajes enviados o recibidos*.
Cada una de estas ventanas puede estar representada a través de un componente Activity, de forma que navegar de una ventana a otra implica lanzar una actividad o dormir otra.

Services

Un servicio es una entidad que **ejecuta instrucciones en segundo plano** sin que el usuario lo note en la interfaz. Son muy utilizados para realizar acciones de larga duración mientras las actividades muestran otro tipo de información. Por ejemplo guardar la información en la base de datos, escuchar música mientras se ejecuta la aplicación, administrar conexiones de red, etc.
Los servicios también tienen un ciclo de vida como las actividades, pero este es más corto. Solo se comprende de los estados de Creación, Ejecución y Destrucción.

View

Las vistas (view) son los componentes básicos con los que se construye la interfaz gráfica de la aplicación, análogo por ejemplo a los controles de Java o .NET. De inicio, Android pone a nuestra disposición una gran cantidad de controles básicos, como cuadros de texto, botones, listas desplegables o imágenes, aunque también existe la posibilidad de extender la funcionalidad de estos controles básicos o crear nuestros propios controles personalizados.

Fragment

Los fragmentos (fragment) se pueden entender como secciones o partes (habitualmente reutilizables) de la interfaz de usuario de una aplicación. De esta forma, una actividad podría contener varios fragmentos para formar la interfaz completa de la aplicación, y adicionalmente estos fragmentos se podrían reutilizar en distintas actividades o partes de la aplicación.

Widget

Los widgets son elementos visuales, normalmente interactivos, que pueden mostrarse en la pantalla principal del dispositivo Android y recibir actualizaciones periódicas. Permiten mostrar información de la aplicación al usuario directamente sobre la pantalla principal.

Content provider

Con el componente **Content Provider**, cualquier aplicación en Android puede almacenar datos en un fichero, en una base de datos SQLite o en cualquier otro formato que considere.
Además, estos **datos pueden ser compartidos entre distintas aplicaciones**. Una clase que implemente el componente Content Provider contendrá una serie de métodos que permite almacenar, recuperar, actualizar y compartir los datos de una aplicación.
Por defecto, el API de Android trae consigo Content Providers predefinidos para intercambiar información de audio, vídeo, imágenes, e información personal. Pero si en algún momento deseas intercambiar información con una estructura personalizada debes crear tu propia subclase heredada de la clase ContentProvider.

Broadcast receiver

Finalmente, el último componente clave son los **Broadcast Receives**, o receptores de emisión, que **reaccionan a intents específicos pudiendo a su vez ejecutar una acción** o iniciando una activity o pueden devolver otro intent al sistema para que siga la ejecución. Realmente este tipo de componentes son capaces de procesar los mensajes del sistema operativo.

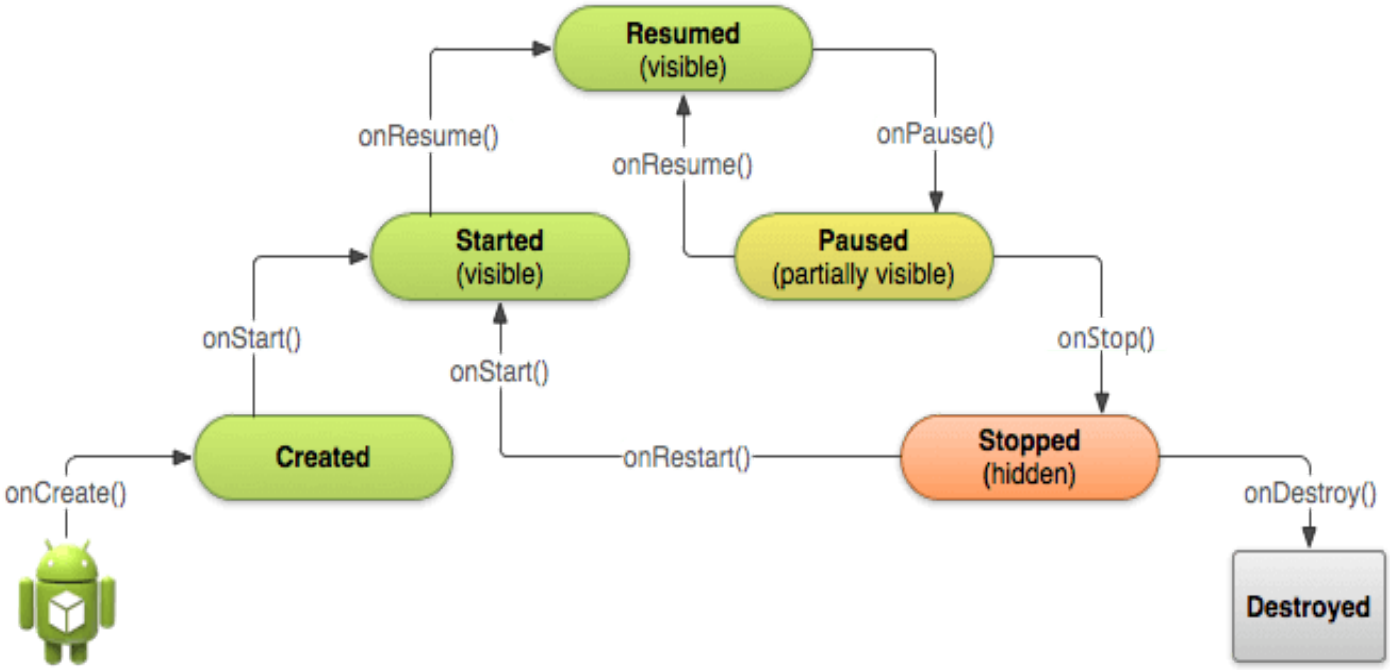
Intens

Un Intent **es un objeto de mensajería** que puedes usar para solicitar una acción de otro componente de una app. Los intents facilitan la comunicación entre componentes de varias formas, desde la comunicación entre actividades, entre fragments, servicios, etc. En un punto posterior de este tema se explicarán más extensamente.

Ciclo de vida de una Activity

Como hemos comentado en el punto anterior. La activity es uno de los componentes esenciales de una aplicación Android, además que es el componente que tiene asociada la interfaz de usuario. Una misma aplicación puede tener una o varias actividades y Android permite controlar por completo el ciclo de vida de cada una de estas.
Los estados por los cuales puede pasar una Activity son los siguientes: **Creación, Ejecución, Reanudación, Pausa, Parada y Destrucción**.

A la relación entre ellos se le llama **Ciclo de vida de una Actividad**.



El ciclo de vida de una Activity nos describe los estados y las transiciones entre estados que una determinada Activity, como hemos comentado son:

- Creación: una actividad se ha creado cuando su estructura se encuentra en memoria, pero esta no es visible aun. Cuando el usuario presiona sobre el icono de la aplicación en su dispositivo, el método `onCreate()` es ejecutado inmediatamente para cargar el layout de la actividad principal en memoria.
- Ejecución-Reanudación: después de haber sido cargada la actividad se ejecutan en secuencia el método `onStart() y onResume()`. Aunque `onStart()` hace visible la actividad, es `onResume()` quien le transfiere el foco para que interactúe con el usuario.
- Pausa: una actividad está en pausa cuando se encuentra en la pantalla parcialmente visible. Un ejemplo: *cuando se abren diálogos que toman el foco superponiéndose a la actividad*. El método llamado para la transición hacia la pausa es `onPause()`.
- Detención: una actividad está detenida cuando no es visible en la pantalla, pero aún se encuentra en memoria y en cualquier momento puede ser reanudada. Cuando una aplicación es enviada a segundo plano se ejecuta el método `onStop()`. Al reanudar la actividad, se pasa por el método `onRestart()` hasta llegar a el estado de ejecución y luego al de reanudación.
- Destrucción: cuando la actividad ya no existe en memoria se encuentra en estado de destrucción. Antes de pasar a destruir la aplicación se ejecuta el método `onDestroy()`. Es común que la mayoría de actividades no implementen este método, a menos que deban destruir procesos como servicios en segundo plano.

Aunque tu aplicación puede tener varias actividades en su estructura, se debe definir una actividad principal. Para hacerlo se debe especificar en tu archivo Android Manifest un componente

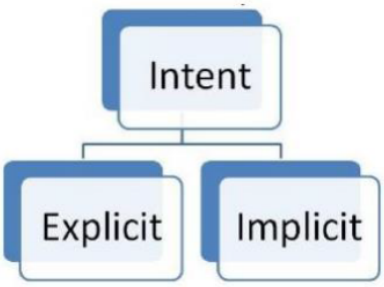
`<actitivy>`, con un componente hijo `<intent-filter>`. Dentro de este componente declararas dos nuevos componentes, `<action>` y `<category>`. Al primero le asignaras el elemento enumerado `MAIN` y al segundo el elemento enumerado `LAUNCHER`.

Crea un proyecto nuevo y copia el código de CicloVidaActivity en la clase principal. Prueba la salida que muestra Logcat en Debug y filtrado por "SALIDA". Fíjate por que métodos pasa la App según los distintos cambios de estado.

Comunicación de Activitys a través de Intents

Uno de los componentes que hemos comentado que forman parte de una aplicación android son los intents. Como dijimos, son los que permiten la comunicación con el resto de componentes, también se pueden definir como la voluntad de realizar una acción o una tarea determinada, estas tareas pueden ser una llamada de teléfono o visualizar una página web entre otras.

Android soporta dos tipos de Intentos **explícitos e implícitos**.



Son explícitos cuando ejecutamos un componente en específico. Son implícitos cuando se entrega una referencia genérica sobre alguna condición, y aquellas entidades que cumplan ese requisito serán presentadas como candidatas para la tarea.

Intent explícito

Se especifica exactamente el componente a lanzar. Se suele utilizar cuando se quiere ejecutar los componentes internos de una aplicación.
Un ejemplo de **Intent explicito** se da cuando invocamos una actividad con un tipo de clase especifico. Por ejemplo, *supón que dentro de nuestra aplicación tenemos dos actividades llamadas A y B. Si decidimos iniciar la actividad B desde la actividad A usaríamos un Intent explicito debido a que "B" es un tipo definido*.

En el inicio de una nueva actividad se pueden dar varias circunstancias:

- Inicio de una actividad sin argumentos ni devolución de resultados:

```
var intento= Intent(this, Activity2::class.java)
//var intento=Intent(applicationContext, Activity2::class.java)
startActivity(intento)
```

- Inicio de una actividad con argumentos y sin devolución de resultados. Podemos pasar información a las actividades añadiendo datos al objeto intent con el método putExtra:

```
var intento= Intent(applicationContext, Activity2::class.java)
intento.putExtra("DATO", "Este es el valor que se manda desde Main Activity")
startActivity(intento)
```

  En la clase invocada obtendremos los datos recibidos de la siguiente manera:

```
var intento=intent
findViewById<TextView>(R.id.textoActivity2).setText(intento.getStringExtra("DATO"))
```

- Inicio de una actividad con argumentos y devolución de resultados. Cuando se inicia una actividad para obtener un resultado, es posible (y casi seguro en el caso de las operaciones que consumen mucha memoria, como el uso de la cámara) que se destruya tu proceso y tu actividad debido a la poca memoria. Por lo que se debe separar la devolución de llamada del resto de código, para ello la librería de Android nos proporciona distintos componentes:
  - `registerForActivityResult()` . **Línea6** Es el elemento principal y que se usa para registrar la devolución de resultados. Tiene varias sobrecargar, en un principio usaremos la que toma un objeto `ActivityResultContract` . Devuelve un elemento `ActivityResultLauncher` que se usará para iniciar la otra actividad.
  - `ActivityResultContract` . **Línea6** y parámetro del elemento anterior. Un contrato que especifica que una actividad se puede llamar con una entrada de tipo I y producir una salida de tipo O. La API proporciona contratos predeterminados para acciones de intent básicas, como tomar una foto, solicitar permisos, etc. También se pueden crear contratos personalizados.
  - `ActivityResultCallback` . **Línea12** se usa este objeto, creado en la línea 6. Es una devolución de llamada con seguridad de tipos que se llamará cuando el resultado de una actividad esté disponible. Es una interfaz de método único con un método onActivityResult() que toma un objeto del tipo de salida definido en `ActivityResultContract`
  - `ActivityResultLauncher` . **Línea12** manda el intent que queremos usar. Es un lanzador para una llamada preparada previamente para iniciar el proceso de ejecución de un `ActivityResultContract` .

```
class MainActivity:AppCompatActivity(){
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val openPostActivity =
6           registerForActivityResult(ActivityResultContracts.StartActivityForResult()) {
                result -> if (result.resultCode == Activity.RESULT_OK) {
                    Toast.makeText(applicationContext,result.data?.
                    getStringExtra("DATORETURN"),Toast.LENGTH_SHORT).show()
10              }
            }
12      findViewById<TextView>(R.id.textoActivity1).setOnClickListener(View.OnClickLister
            openPostActivity.launch(
                Intent(applicationContext, Activity2::class.java).apply
                {putExtra("DATO", "Este es el valor que se manda desde Main Activity")})}
16  }
}
```

📌 De **Línea 6 a Línea 10**, se registra la actividad para esperar resultados de otras activitys o fragments. Se debe registrar la actividad para posibles resultados en el método *onCreate()*.Si el resultado es OK realizamos el código necesario, en este caso se ha extraído la información del intent recibido `result.data` . De **Línea 12 a 16** se lanza el intent de la actividad que se quiere iniciar, pero con el objeto `ActivityResultCallback` creado en el bloque anterior.

En la clase invocada devolveremos los datos de la siguiente manera:

```
    textView.setOnClickListener {
            val intentoResultado=Intent()
            intentoResultado.putExtra("DATORETURN","Este es el dato")
4           setResult(Activity.RESULT_OK,intentoResultado)
5           finish()}
```

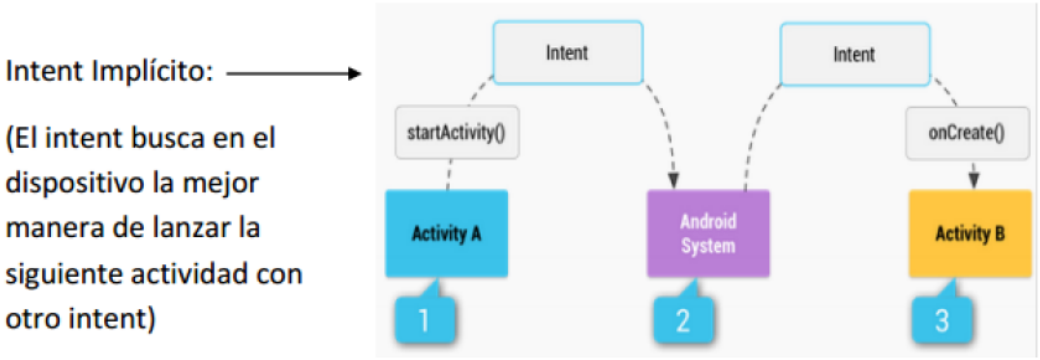
📌 en la **Línea 4** preparamos el resultado de la actividad indicando que ha sido correcto y añadiendo un intent con información si lo creemos necesario. **Línea 5** cerramos la activity.

Ejercicio ResueIto2Activitys

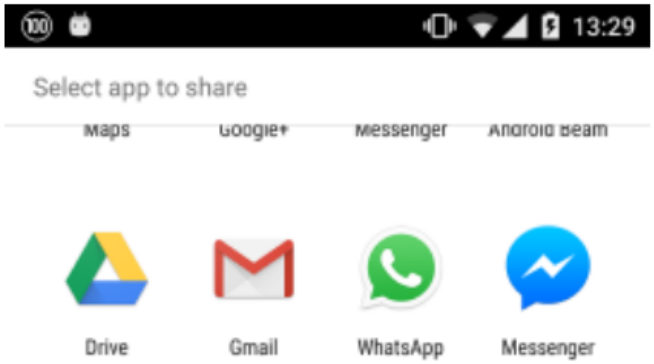
Ejercicio Propuesto3ActivitysRetornoDatos

Intent Implícito

Los Intent implícitos son aquellos que le preguntan al sistema qué servicio o componente es el más adecuado para realizar la petición, es decir, no específica un componente en especial, y solo se especifica la funcionalidad requerida a través de una acción (ver, hacer búsqueda, hacer foto, marcar número…) y un dato (URL, número a marcar…)En este caso el sistema determina el componente para su resolución, desencadenando un "intent" sin conocer exactamente cuál es la aplicación o componente que lo recibirá, si no puede establecer exactamente el componente a utilizar porque hay más de uno, muestra una ventana con las opciones a seleccionar.



Un ejemplo de **Intent implícito**: *sería cuando deseas compartir un sitio web en alguna red social. Para ello Android nos ofrece un dialogo listando las app sociales instaladas en el dispositivo en ese momento, para que elijamos la de nuestro interés. No especificamos que aplicación queríamos, simplemente se mostraron todas las aplicaciones que podrían responder a ese tipo de acción*.



Básicamente los pasos para lanzar una actividad con un **intent implícito** serían los siguientes:

- Declaramos el intent con la acción apropiada (ACTION_VIEW, ACTION_WEB_SEARCH, etc).
- Adjuntamos información adicional necesaria dependiendo de la acción del intent.
- Lanzamos el intent dejando que el sistema encuentre la actividad adecuada).

Para iniciar la actividad utilizaremos igualmente startActivity. Algunos ejemplos de Intents Comunes:

- Para mostrar una web dentro de un navegador:

```
Intent intento = new Intent(Intent.ACTION_VIEW, Uri.parse("http://www.google.es"));
startActivity(intento);
```

  Requiere los permisos: `<uses-permission android:name=" android.permission.INTERNET "/>`

- Para utilizar la búsqueda de google;

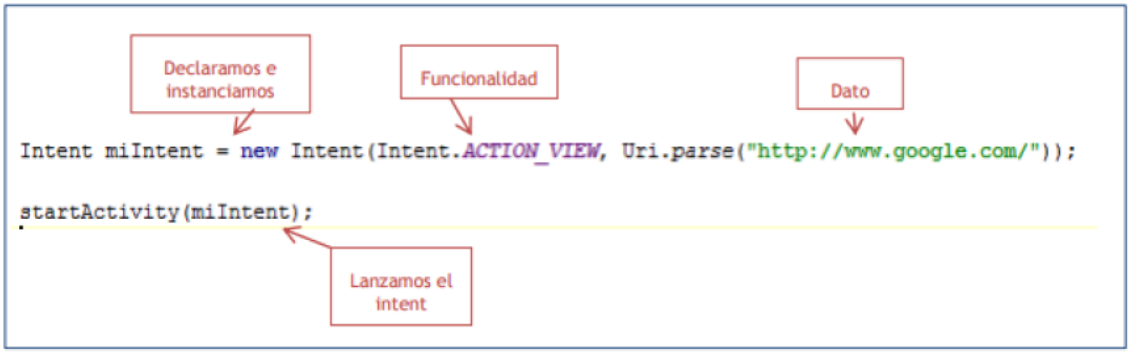
```
Intent intento = new Intent(Intent.ACTION_WEB_SEARCH);
intento.putExtra(SearchManager.QUERY, "I.E.S. Doctor Balmis");
startActivity(intento);
```

  Requiere los permisos `<uses-permission android:name="android.permission.INTERNET"/>`

- Para mostrar los contactos.

```
Intent intento = new Intent(Intent.ACTION_VIEW, Uri.parse("content://contacts/people/"));
startActivity(intento);
```

  Requiere los permisos: `<uses-permission android:name="android.permission.READ_CONTACTS"/>`



Es posible que en ocasiones no se pueda acceder a una aplicación debido a que no está instalada en el dispositivo, a restricciones de perfil o a la configuración de un administrador. En ese caso, no se produce la llamada y se bloquea la app. Para verificar que una actividad recibirá la intent, es muy importante usar antes `resolveActivity()` en el objeto Intent. Si el resultado no es nulo, hay al menos una aplicación que puede administrar la intent y es seguro llamar a `startActivity()`.

```
Intent intento = new Intent(Intent.ACTION_VIEW, Uri.parse("content://contacts/people/"));

if (intento.resolveActivity(packageManager) != null) startActivity(intento);
```

Ejercicio ResueltoIntentImplicito

Ejercicio PropuestoIntentImplicito

Requerir permisos al usuario

Android está apostando cada vez más fuerte por la seguridad, por lo que en las versiones más recientes del sistema operativo no basta con añadir la etiquete `<uses-permission ...>` en el Manifiesto. Si se quiere acceder a recursos ajenos a la aplicación en funcionamiento, se deberá realizar una solicitud de permisos de forma explicita al usuario.
Esto es debido a que cada app se ejecuta en una zona de pruebas con acceso limitado, si necesita usar recursos o información ajenos a su propia zona de pruebas deberás seguir los siguientes pasos:

1. En el archivo de manifiesto de la app, declara los permisos que necesites.
2. Esperar a que el usuario invoque la tarea o acción de la app que requiere acceso a datos privados específicos. En ese momento, la app puede solicitar el permiso de tiempo de ejecución necesario para acceder a esos datos. Para ello:
    1. Verifica si el usuario ya otorgó el permiso de tiempo de ejecución que requiere la app. De ser así, esta ya puede acceder a los recursos necesarios.
    2. En caso contrario, solicitar el permiso en tiempo de ejecución, en ese caso el sistema mostrará un dialogo solicitando el permiso y si es otorgado se pasará al acceso de los recursos. En este paso se puede optar por mostrar un dialogo explicando porque motivo son necesarios los recursos.
3. Si el usuario rechaza la solicitud, la app debería proporcionar una opción elegante al usuario.

Podemos crear un código parecido al siguiente, en el que se pide permiso para acceder a los contactos y a la memoría externa del dispositivo:

```
0   val RESPUESTA_PERMISOS = 111
    @RequiresApi(Build.VERSION_CODES.Q)
3   fun solicitarPermisos()
    {
5       if (checkSelfPermission(READ_EXTERNAL_STORAGE) == PackageManager.PERMISSION_DENIED
            || checkSelfPermission(READ_CONTACTS) == PackageManager.PERMISSION_DENIED)
        {
8           if (shouldShowRequestPermissionRationale(READ_EXTERNAL_STORAGE))
                //Si se decide explicar los motivos de los permisos con  similar a un dialogo
10          if (shouldShowRequestPermissionRationale(READ_CONTACTS))
                //Si se decide explicar los motivos de los permisos con  similar a un dialogo
            //Con requestPermissions se pide al usuario que permita los permisos,
            //en este caso dos
14          requestPermissions(arrayOf(READ_EXTERNAL_STORAGE, _CONTACTS), RESPUESTA_PERMISOS)
        }
16      else lanzarFuncionalidadQueRequierePermisos()
17  }
```

```
0   override fun onRequestPermissionsResult(
        requestCode: Int,//código de identificación del resultado
        permissions: Array<out String>,//array con los nombres de los permisos
        grantResults: IntArray//array de 0 y -1 (permitido, no permitido) en orden)
    {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults)
7       when (requestCode) {
            RESPUESTA_PERMISOS -> {
                if (grantResults[0] == PackageManager.PERMISSION_GRANTED)
                    Toast.makeText(
                        applicationContext,
                        permissions[0] + " Permiso concedido",
                        Toast.LENGTH_SHORT
                    ).show()
                if (grantResults[1] == PackageManager.PERMISSION_GRANTED)
                    Toast.makeText(
                        applicationContext,
                        permissions[1] + " Permiso concedido",
                        Toast.LENGTH_SHORT
                    ).show()
            }
        }
22  }
```

📌 **Línea 0** creamos una constante identificativa de la petición de permisos. Desde la **Línea 3 hasta la 17** creamos el método encargado de lanzar los diálogos con la petición de los dos permisos (en este caso). **Línea 5** se comprueba que todavía no se han concedido, y en ese caso se lanza la petición **Línea 16**. Las **líneas 8 y 10**, servirían para añadir alguna forma de explicación al usuario. Si el permiso ha sido concedido con anterioridad se pasará a lanzar la funcionalidad que lo usa, **Línea 16**.

El hilo que controla la respuesta del usuario, se recoge mediante el método sobreescrito **`onRequestPermissionsResult`** , a este método le llega la información necesaria para poder saber a que petición de permisos se refiere (usando la constante creada anteriormente **Línea 7**), el array con los nombres de los permisos y el resultado de si han sido concedido o no. Con esta información pasaremos a lanzar la funcionalidad que los requiere o a dar otras opciones al usuario.

Requerir permisos al usuario

Android está apostando cada vez más fuerte por la seguridad, por lo que en las versiones más recientes del sistema operativo no basta con añadir la etiquete `<uses-permission ...>` en el Manifiesto. Si se quiere acceder a recursos ajenos a la aplicación en funcionamiento, se deberá realizar una solicitud de permisos de forma explicita al usuario.
Esto es debido a que cada app se ejecuta en una zona de pruebas con acceso limitado, si necesita usar recursos o información ajenos a su propia zona de pruebas deberás seguir los siguientes pasos:

1. En el archivo de manifiesto de la app, declara los permisos que necesites.
2. Esperar a que el usuario invoque la tarea o acción de la app que requiere acceso a datos privados específicos. En ese momento, la app puede solicitar el permiso de tiempo de ejecución necesario para acceder a esos datos. Para ello:
    1. Verifica si el usuario ya otorgó el permiso de tiempo de ejecución que requiere la app. De ser así, esta ya puede acceder a los recursos necesarios.
    2. En caso contrario, solicitar el permiso en tiempo de ejecución, en ese caso el sistema mostrará un dialogo solicitando el permiso y si es otorgado se pasará al acceso de los recursos. En este paso se puede optar por mostrar un dialogo explicando porque motivo son necesarios los recursos.
3. Si el usuario rechaza la solicitud, la app debería proporcionar una opción elegante al usuario.

Podemos crear un código parecido al siguiente, en el que se pide permiso para acceder a los contactos y a la memoria externa del dispositivo:

```
0   val RESPUESTA_PERMISOS = 111
    @RequiresApi(Build.VERSION_CODES.Q)
3   fun solicitarPermisos()
    {
5       if (checkSelfPermission(READ_EXTERNAL_STORAGE) == PackageManager.PERMISSION_DENIED
            || checkSelfPermission(READ_CONTACTS) == PackageManager.PERMISSION_DENIED)
        {
8           if (shouldShowRequestPermissionRationale(READ_EXTERNAL_STORAGE))
                //Si se decide explicar los motivos de los permisos con  similar a un dialogo
10          if (shouldShowRequestPermissionRationale(READ_CONTACTS))
                //Si se decide explicar los motivos de los permisos con  similar a un dialogo
            //Con requestPermissions se pide al usuario que permita los permisos,
            //en este caso dos
14          requestPermissions(arrayOf(READ_EXTERNAL_STORAGE, _CONTACTS), RESPUESTA_PERMISOS)
        }
16      else lanzarFuncionalidadQueRequierePermisos()
17  }
```

```
0   override fun onRequestPermissionsResult(
        requestCode: Int,//código de identificación del resultado
        permissions: Array<out String>,//array con los nombres de los permisos
        grantResults: IntArray//array de 0 y -1 (permitido, no permitido) en orden)
    {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults)
7       when (requestCode) {
            RESPUESTA_PERMISOS -> {
                if (grantResults[0] == PackageManager.PERMISSION_GRANTED)
                    Toast.makeText(
                        applicationContext,
                        permissions[0] + " Permiso concedido",
                        Toast.LENGTH_SHORT
                    ).show()
                if (grantResults[1] == PackageManager.PERMISSION_GRANTED)
                    Toast.makeText(
                        applicationContext,
                        permissions[1] + " Permiso concedido",
                        Toast.LENGTH_SHORT
                    ).show()
            }
        }
22  }
```

📌 **Línea 0** creamos una constante identificativa de la petición de permisos. Desde la **Línea 3 hasta la 17** creamos el método encargado de lanzar los diálogos con la petición de los dos permisos (en este caso). **Línea 5** se comprueba que todavía no se han concedido, y en ese caso se lanza la petición **Línea 16**. Las **líneas 8 y 10**, servirían para añadir alguna forma de explicación al usuario. Si el permiso ha sido concedido con anterioridad se pasará a lanzar la funcionalidad que lo usa, **Línea 16**.

El hilo que controla la respuesta del usuario, se recoge mediante el método sobreescrito **`onRequestPermissionsResult`** , a este método le llega la información necesaria para poder saber a que petición de permisos se refiere (usando la constante creada anteriormente **Línea 7**), el array con los nombres de los permisos y el resultado de si han sido concedido o no. Con esta información pasaremos a lanzar la funcionalidad que los requiere o a dar otras opciones al usuario.

Intent-filter

Las actividades y servicios pueden declarar el tipo de acciones que pueden llevar a cabo y los tipos de datos que pueden gestionar. De esta forma se informa al sistema del tipo de intenciones implícitas que puede atender dicho componente. Los intent-filter se definen en el archivo de manifiesto de la aplicación dentro de la activity correspondiente.

Como se ha explicado en el punto anterior, cuando se crea un intent implícito el sistema Android busca el componente apropiado que pueda resolver la petición dada en el intent.

La búsqueda la realiza analizando los **intent-filter** definidos en los archivos de manifiesto de las aplicaciones instaladas en el dispositivo. Si existe coincidencia lanza dicha aplicación. Si la coincidencia es múltiple (varias aplicaciones pueden dar respuesta a esa petición), el sistema muestra un cuadro de diálogo con el fin de que el usuario decida. A un Intent podemos asociarle una acción, unos datos y una categoría.

Las actividades pueden declarar el tipo de acciones que pueden llevar a cabo y los tipos de datos que pueden gestionar.

- Las acciones son cadenas de texto estándar que describen lo que la actividad puede hacer. Por ejemplo `android.intent.action.VIEW`, es una acción que indica que la actividad puede mostrar datos al usuario.
- La misma actividad puede declarar el tipo de datos del que se ocupa, por ejemplo `vnd.android.cursor.dir/person, indica` que la actividad manipula los datos de la agenda*.
- También pueden declarar una categoría, que básicamente indica si la actividad va a ser lanzada desde el lanzador de aplicaciones `android.intent.category.DEFAULT`, desde el menú de otra aplicación o directamente desde otra actividad.
  Un posible ejemplo del AndroidManifest.xml que cumpla las anteriores condiciones:

```
<intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="vnd.android.cursor.dir/person"/>
</intent-filter>
```

Una activity puede tener varios intent-filter definidos. Igualmente cada uno de esos intent-filter puede tener varias acciones definidas.

Manifest ejemplo de varios intent-filter:

```
<activity  class = ".NotesList"  android:label = "@string/title_notes_list" >
    <intent-filter>
        <action  android:name = "android.intent.action.MAIN"  />
        <category  android:name = "android.intent.category.LAUNCHER"  />
    </intent-filter>
    <intent-filter>
        <action  android:name = "android.intent.action.VIEW"  />
        <action  android:name = "android.intent.action.EDIT"  />
        <action  android:name = "android.intent.action.PICK"  />
        <category  android:name = "android.intent.category.DEFAULT"  />
        <data  android:mimeType = "vnd.android.cursor.dir/ vnd.google.note "  />
    </intent-filter>
    <intent-filter>
        <action  android:name = "android.intent.action.GET_CONTENT"  />
        <category  android:name = "android.intent.category.DEFAULT"  />
        <data  android:mimeType = "vnd.android.cursor.item/ vnd.google.note "  />
    </intent-filter>
</activity>
```

> 📌El primer intent-filter define que esta activity es punto de entrada para la aplicación. La categoría especifica que debe aparecer en el lanzador de aplicaciones. El segundo, permite a la activity visualizar y editar el directorio de datos (a través de las actions VIEW y EDIT), o recuperar una entrada particular y devolverla (a través de la action PICK). En data se especifica el tipo de datos que se manejan. Como está definido con la Categoría DEFAULT, estamos indicando que atenderá a los intents implícitos que soliciten otras aplicaciones.

Trabajando con Parcelables

Cuando estamos desarrollando en Android y necesitamos enviar datos a otra Activity lo podemos realizar mediante un **Intent y Bundle** pero qué sucede si deseamos enviar un objeto para economizar una llamada a la API. Android nos brinda las opciones de **Serializable** (propia de Java) y **Parcelable**, sin embargo esta última es hasta 10 veces más rápida por lo que nos vamos a enfocar en ella.
Parcelable es una interfaz y podemos implementarla manual o mediante un plugin que, en las últimas versiones de Android Studio, viene incluido por defecto.
Claramente tendremos que crear la clase POJO, como se hace siempre, pero está deberá derivar de la `Interfaz Parcelable`. Como es de suponer esto nos obligará a implementar una serie de métodos y una variable CREATOR que hereda de Creator y que tendrá todos sus miembros estáticos gracias el modificador `companion`.
Veamos un ejemplo con dos activitys, una en la que se pueden introducir los datos de un contacto y que al pulsar un botón envíe los datos existentes en la pantalla a una segunda actividad que mostrará los datos de forma sencilla.

Clase **Contacto** que hereda de la `Interfaz Parcelable`:

```
class Contacto() : Parcelable {
    var nombre: String?
    var telefono: String?
    var email: String?
    var foto: Bitmap?
    //Constructor generado automáticamente
    constructor(parcel: Parcel) : this() {
        nombre = parcel.readString()
        telefono = parcel.readString()
        email = parcel.readString()
        foto = parcel.readParcelable(Bitmap::class.java.classLoader)
    }
    init{
         this.nombre=""
         this.telefono=""
         this.email=""
         this.foto=null
     }
    constructor(nombre: String?, telefono: String?,
                email: String?, foto: Bitmap?):this(){
        this.nombre = nombre
        this.telefono = telefono
        this.email = email
        this.foto = foto
    }
    //funciones generadas automáticamente
    override fun describeContents(): Int {
        return 0
    }
    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeString(nombre)
        parcel.writeString(telefono)
        parcel.writeString(email)
        parcel.writeParcelable(foto, flags)
    }
    //Clase creada automáticamente. Un companion object es la forma que tienen Kotlin
    //de definir miembros staticos dentro de una clase, todos los elementos
    //de este object serán static
    companion object CREATOR : Creator<Contacto> {
        override fun createFromParcel(parcel: Parcel): Contacto {
            return Contacto(parcel)
        }
        override fun newArray(size: Int): Array<Contacto?> {
            return arrayOfNulls(size)
        }
    }
}
```

Clase **MainActivity** que inicia la actividad segunda mandando el objeto parcelado.

```
class MainActivity : AppCompatActivity()
{
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val nombre = findViewById<TextView>(R.id.campo_nombre)
        val tlf = findViewById<TextView>(R.id.campo_tlf)
        val correo = findViewById<TextView>(R.id.campo_correo)
        val img = findViewById<ImageButton>(R.id.placeImage)
        val imagen = BitmapFactory.decodeResource(resources, R.drawable.noimagen)
        val button: Button = findViewById<Button>(R.id.boton)
        button.setOnClickListener {
                val c = Contacto(nombre.text.toString(), tlf.text.toString(),
                                 correo.text.toString(), imagen)
15              val intent = Intent(applicationContext, Activity2::class.java)
16              intent.putExtra("CONTACTO", c)
                startActivity(intent)}
    }}
```

📌 **Línea 15** se crea el intent para la actividad 2 . **Línea 16** como se puede ver, aunque el elemento que añadimos al intent es un objeto de tipo Contacto, se puede hacer con `putExtra` debido a que el contacto hereda de Parcelable.

En la **Activity2** recuperaremos el objeto pasado, con la **línea 6**.

```
class Activity2: AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity2);
        //recuperar datos y visualizar
6       val c = this.intent.getParcelableExtra<Contacto>("CONTACTO");
        val nombre = findViewById<TextView>(R.id.txt_nombre);
        val tlf = findViewById<TextView>(R.id.txt_tlf);
        val correo = findViewById<TextView>(R.id.txt_correo);
        val img = findViewById<ImageButton>(R.id.placeImage);
        if (c != null) {
            nombre.setText(c.nombre)
            tlf.setText(c.telefono)
            correo.setText(c.email)
            img.setImageBitmap(c.foto)}}
}
```

✋ **Importante:** No olvides declarar en el Manifest las dos actividades.

Ejercicio PropuestoParcelabe