Tema 5. Activities, Intents y permisos

5.1 Activities

La actividad (**Activity**) es el principal componente de una aplicación de Android y se encarga de gestionar gran parte de las interacciones con el usuario.

A nivel de lenguaje de programación, una actividad hereda de la clase **Activity** y se traduce visualmente, en una pantalla. Por tanto, en términos generales, podemos hablar de que una app tendrá tantas pantallas como actividades (activities).

Las actividades, tienen una secuencia de aparición según la lógica del código. Aparecerán unas u otras pantallas según el flujo del programa y las decisiones del usuario.

Las actividades funcionan dentro del dispositivo como procesos que pueden estar en diferentes estados o momentos de ejecución:

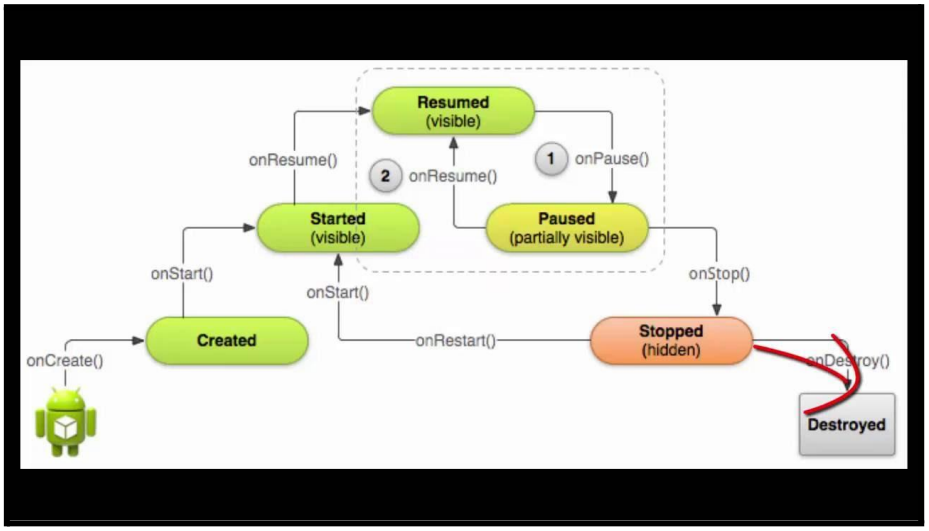
- **Foreground process.** La actividad que se está mostrando en pantalla y con la que el usuario puede interactuar.
- **Visible process.** Visible, pero no es posible interactuar con ella. Tiene un uso especial para el programador y el sistema operativo.
- **Service process.** Hace cosas en segundo plano y generalmente importantes.
- **Background process.** Actividad no visible.
- **Empty process.** Procesos vacíos sobre los que Android crea otro proceso nuevo.

5.1.2 Ciclo de vida de una aplicación

Teniendo en cuenta el estado de funcionamiento de una actividad, para Android, la actividad puede ser:

- **Inexistente**. No ocupa memoria y no se puede interactuar con ella.
- **Creada (Created).** Una actividad entra en este estado después de haber sido creada mediante el método *onCreate()*. En este punto, la actividad ha sido inicializada, pero aún no es visible al usuario.
- **Iniciada (Started)**. Después de *onCreate()*, la actividad pasa al estado "Iniciado" cuando se llama al método *onStart()*. En este estado, la actividad es visible, pero no está en primer plano.
- **En Primer Plano (Resumed)**: Una actividad entra en el estado "En primer plano" después de que *onStart()* llame a *onResume()*. En este estado, la actividad está en la parte superior de la pila de actividades y es completamente visible y activa para el usuario.
- **Pausada (Paused)**. Cuando otra actividad entra en primer plano o la actividad actual se minimiza, pasa al estado "Pausado". En este estado, la actividad sigue siendo visible pero no está en primer plano y no puede interactuar con el usuario.
- **Detenido (Stopped).** Cuando la actividad ya no es visible, por ejemplo, cuando el usuario navega a otra actividad o presiona el botón de inicio, entra en el estado "Detenido" después de que onPause() llame a onStop(). En este estado, la actividad no es visible y no puede interactuar con el usuario.
- **Destruido (Destroyed)**. Cuando la actividad se cierra o se elimina de la pila de actividades, entra en el estado "Destruido" después de que onStop() llame a onDestroy(). En este estado, la actividad ha sido liberada y eliminada.

Estos estados suelen ir en sucesión creciente cuando la actividad se pone en marcha, y en sentido decreciente cuando desaparece y se destruye. Este conjunto de estado es lo que se conoce por ciclo de vida.



Es necesario aprender a controlar el ciclo de vida de una aplicación y saber manejar cada uno de los métodos para hacer que la aplicación sea lo más robusta posible.

- **onCreate()**: es llamado “únicamente” cuando la actividad es invocada por primera vez. Al crear la actividad, se hace una reserva de memoria para los datos, se obtienen las referencias mediante findViewById() , se realizan las consultas de datos iniciales y se guardan las variables de entorno (p. ej. Bundle).

- **onStart()**: este método sucede al anterior. Se hace visible, pero aún no se puede interactuar con ella. Aquí se suelen ubicar instrucciones asociadas a la interfaz de usuario.

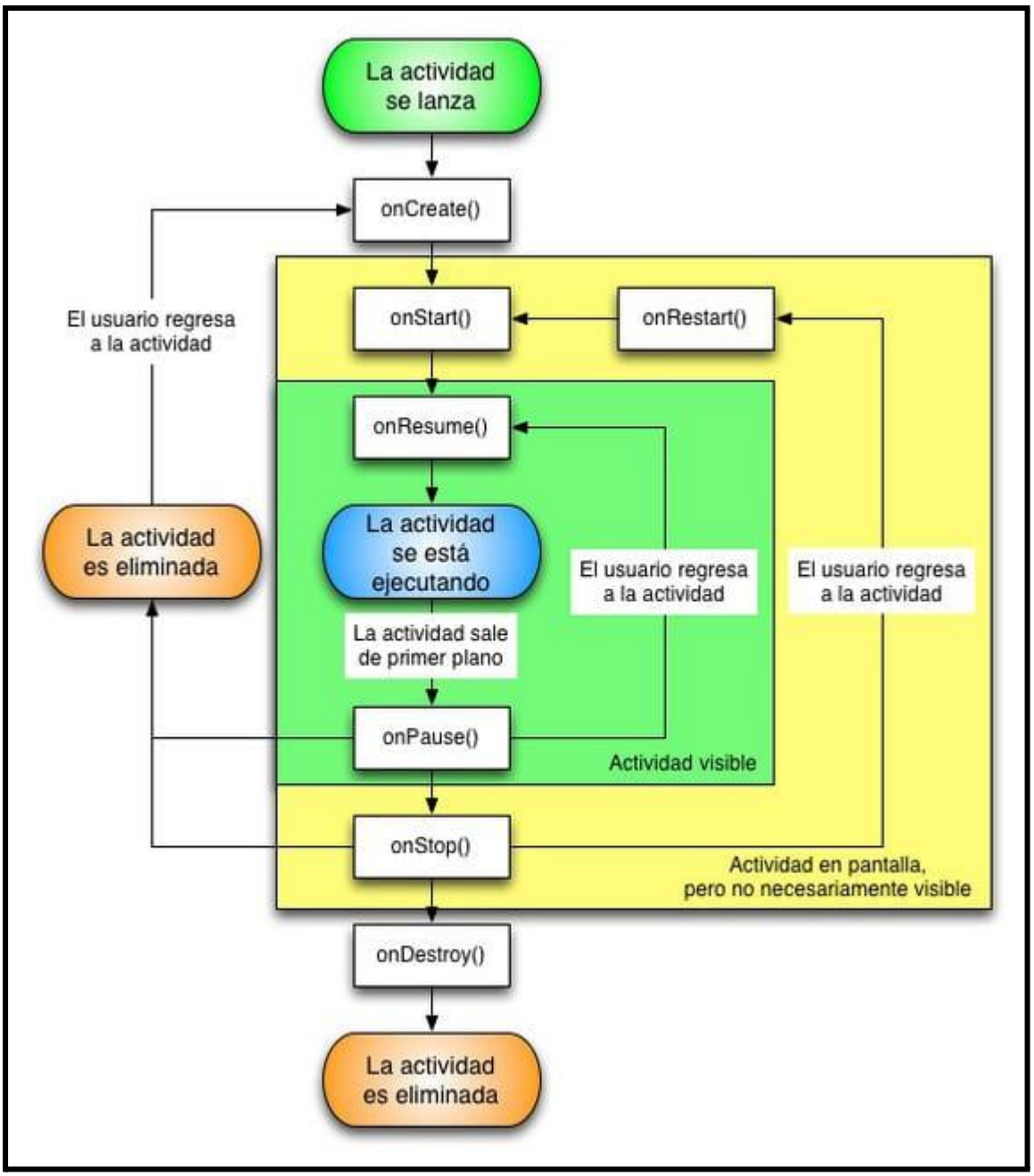
- **onResume()**: es llamado cuando la actividad puede interactuar con el usuario. Aquí se suelen activar sensores, cámara, ejecutar animaciones, actualizar información, etc.

- **onPause()**: ocurre cuando la actividad pasa a un segundo plano, bien porque se va a destruir o bien porque otra va a ocupar el primer plano. Aquí la actividad es parcialmente visible. Podría volver al primer plano con onResume() o ser parada con onStop().

- **onStop()**: este método es invocado cuando la actividad ya no es visible al usuario y otra actividad ha ocupado el primer plano. Si queremos reactivarla, se utiliza onRestart() y si queremos destruirla onDestroy().

- **onRestart()**: llamado cuando una actividad parada vuelve a primer plano. Siempre seguido de onStart().

- **onDestroy()**: es llamado cuando la actividad es definitivamente destruida y eliminada de memoria.


La actividad se lanza
onCreate()
El usuario regresa a la actividad
onStart()
onRestart()
onResume()
La actividad se está ejecutando
La actividad es eliminada
El usuario regresa a la actividad
El usuario regresa a la actividad
La actividad sale de primer plano
onPause()
Actividad visible
onStop()
Actividad en pantalla, pero no necesariamente visible
onDestroy()
La actividad es eliminada

5.1.3 Conservar el estado de una aplicación

Cuando se pausa o se detiene una actividad, se conserva el estado de la misma. Esto ocurre porque el objeto **Activity** aún se conserva en memoria, y toda la información de sus objetos y su estado actual permanece. Por tanto, cuando la actividad regrese a primer plano, esa información aún estará ahí.

Puede ocurrir que, estando parada, el sistema destruya una actividad para recuperar recursos de memoria; el objeto **Activity** también se destruye, y es necesario volver a crearlo para tenerlo en primer plano.

Si queremos que la actividad mantenga el mismo aspecto y la información previa, necesitamos de un método de callback llamado ***onSaveInstanceState()***.

Antes de la destrucción, el sistema llama a ***onSaveInstanceState()***, y almacena la información en un objeto ***Bundle***. Este tipo de objetos permiten el almacenamiento de datos usando métodos de tipo *putString()* o *putInt()*, e igualmente permiten recuperar la información usando métodos de *getString()* o *getInt()*.

Se podrían almacenar clases que hayamos creado nosotros mismos, si dicha clase implementa la interfaz ***Serializable***.

```
override fun onCreate(savedInstanceState: Bundle?)
{
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    Log.d( tag: "HolaMundo",  msg: "onCreate");

    /* Se recupera el valor almacenado al eliminarse el activity */

    val myValue: String? = savedInstanceState?.getString( key: "myKey",  defaultValue: "Valor no encontrado")

    if (myValue != null)
    {
        Toast.makeText( context: this, myValue, Toast.LENGTH_LONG).show()
    }
}
```

```
override fun onSaveInstanceState(outState: Bundle)
{
    //Hay que ponerlo antes de la llamada al constructor
    outState.putString("myKey", "Valor salvado")

    super.onSaveInstanceState(outState)


}
```

**Nota: gira la pantalla del emulador para comprobar que funciona correctamente.

5.2 Intents

¿Cómo podríamos hacer que un usuario pueda accionar determinados servicios (como llamar por teléfono o enviar mensajes) o bien navegar entre distintas pantallas o Activity?

En el desarrollo de cualquier aplicación es necesario emplear más de un objeto **Actividad**, por tanto, la comunicación entre ellas es importante.

De todo esto se encarga la clase ***Intent***. La clase ***Intent*** representa las acciones o intenciones que un usuario quiere realizar desde su dispositivo móvil. Las intenciones se pueden clasificar en dos tipos: *intents explícitos* e *intents implícitos*.

Por ejemplo, si al pulsar un botón debe lanzarse una nueva Activity o pantalla hablaremos de *intents explícitos* porque indicamos la clase que queremos lanzar, mientras que, si sólo conocemos lo que queremos realizar, pero no qué clase es la encargada de lanzarlo hablaremos de *intents implícitos* (cuando queremos lanzar una aplicación externa).

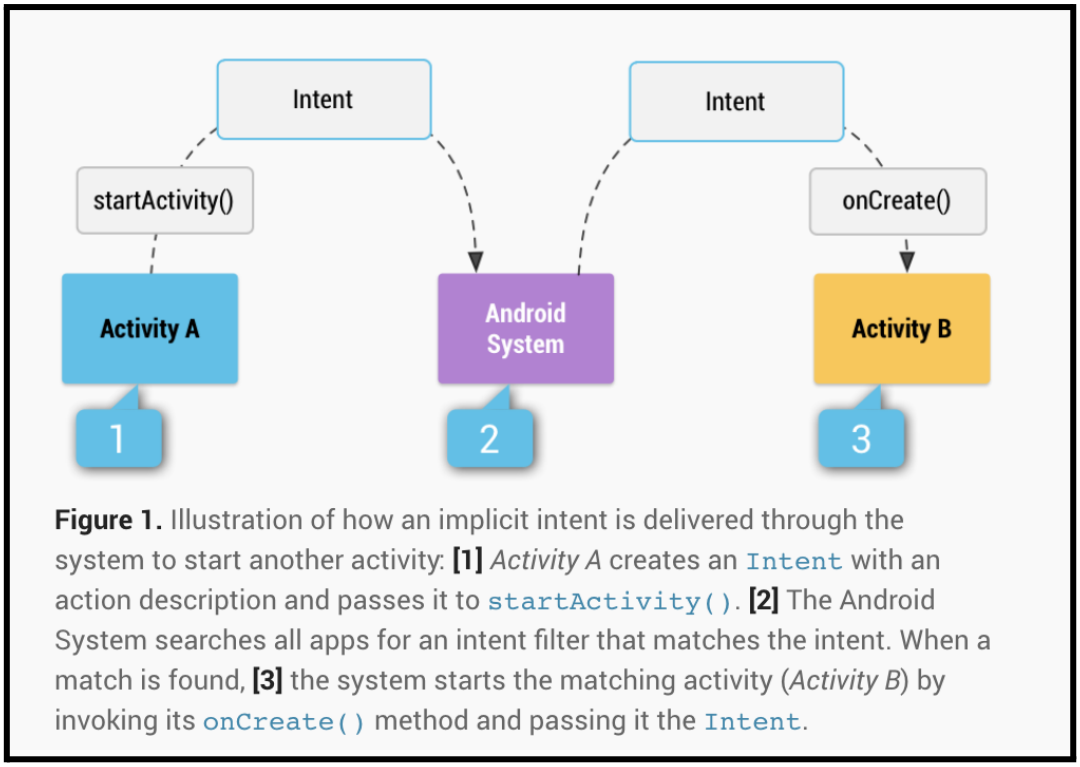


Figure 1. Illustration of how an implicit intent is delivered through the system to start another activity: **[1]** *Activity A* creates an `Intent` with an action description and passes it to `startActivity()`. **[2]** The Android System searches all apps for an intent filter that matches the intent. When a match is found, **[3]** the system starts the matching activity (*Activity B*) by invoking its `onCreate()` method and passing it the `Intent`.

5.2.1 Filtros de intent

Si observamos el fichero *AndroidManifest.xml* de un proyecto, podrás ver como se le indica al sistema operativo qué actividad es la que se debe lanzar a la hora de ejecutar la aplicación.

```
<activity
        android:name=".MainActivity"
        android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
    <meta-data
            android:name="android.app.lib_name"
            android:value=""/>
</activity>
```

Esto es gracias a *android.intent.action.MAIN* y a *android.intent.category.LAUNCHER* que son la acción y la categoría del intent que inicializa la app.

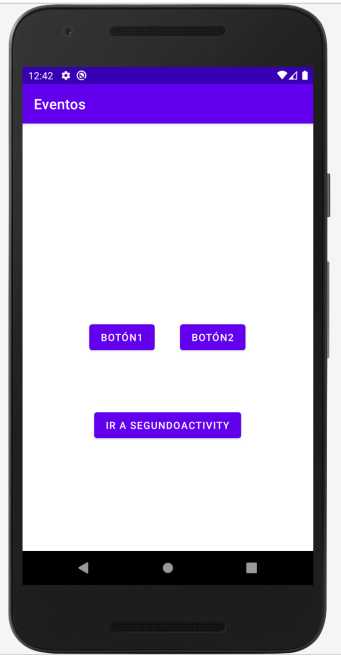
Si la aplicación tiene más de un activity, esta categoría sólo puede estar en uno de ellos.

5.2.2 Abrir una actividad nueva (intent explícito)

En el siguiente ejemplo de código, puedes ver el código de un **intent** explícito para pasar del **MainActivity** a **SegundoActivity**.

```
//Rescato el tercer botón
val btn3: Button = findViewById(R.id.button3)

//Al hacer click sobre el botón 3, se salta a SegundoActivity
btn3.setOnClickListener()
{ view ->
    val intent: Intent = Intent( packageContext: this, SegundoActivity::class.java)
    startActivity(intent)
    }

}
```

En el siguiente enlace tienes la documentación oficial de cómo iniciar otra actividad: https://developer.android.com/training/basics/firstapp/starting-activity?hl=es-419



5.2.2.1 Tasks y Flags

En Android, una ***"task" (tarea)*** se refiere a un grupo de ***actividades (activities)*** que se organizan juntas en una pila (stack) y están relacionadas entre sí de alguna manera. Las tareas son un concepto importante para la gestión de actividades y la navegación en aplicaciones Android. Aquí tienes algunos puntos clave sobre las tareas en Android:

- Tareas independientes: cada aplicación Android puede tener múltiples tareas en ejecución al mismo tiempo. Cada tarea puede contener una o más actividades, y estas actividades pueden pertenecer a la misma aplicación o incluso a diferentes aplicaciones.

- Relación de pila: las actividades en una tarea se organizan en una pila, con la actividad más reciente en la parte superior y las actividades anteriores debajo. Cuando una nueva actividad se inicia, generalmente se coloca en la parte superior de la pila.

- Pila de actividades: la pila de actividades es gestionada por el sistema Android y se utiliza para controlar la navegación entre las actividades. Cuando el usuario presiona el botón "Atrás" o cuando una actividad finaliza, la actividad superior en la pila se quita, y la siguiente actividad en la pila se convierte en la actividad activa.

- Tareas y actividades: las actividades en una tarea pueden pertenecer a la misma aplicación o a diferentes aplicaciones. Cada tarea se ejecuta en su propio contexto y puede tener su propia pila de actividades.

- Tareas y flags: los flags se utilizan para controlar cómo se administran las tareas y las actividades en la pila cuando se inician o interactúan con actividades.

Los ***flags*** son utilizados en Android para modificar el comportamiento de las actividades (activities) al iniciar o interactuar con ellas. Aquí se explica para qué se utilizan cada uno de ellos:

FLAG_ACTIVITY_NEW_TASK

Este flag se usa para iniciar una nueva tarea (task) en Android. Cuando se inicia una actividad con este flag, se crea una nueva tarea en lugar de agregarla a la tarea existente.

Esto puede ser útil cuando deseas iniciar una actividad como una tarea independiente, como una actividad de inicio o una actividad que debe estar aislada de otras tareas en curso.

FLAG_ACTIVITY_CLEAR_TOP

Este flag se usa en combinación con *FLAG_ACTIVITY_NEW_TASK* o simplemente al iniciar una actividad. Cuando se utiliza en conjunto con F*LAG_ACTIVITY_NEW_TASK*, se inicia una nueva tarea y, si la actividad ya se encuentra en la pila de actividades en esa tarea, se elimina cualquier actividad que esté encima de la actividad objetivo en la pila. Si se utiliza solo al iniciar una actividad, Android verifica si la actividad ya se encuentra en la pila y, si es así, la trae a la parte superior y elimina las actividades que estén encima de ella.

Esto es útil para volver a una actividad específica en una tarea existente y eliminar otras actividades intermedias.

FLAG_ACTIVITY_SINGLE_TOP

Este flag se utiliza para indicar que, si la actividad objetivo ya está en la parte superior de la pila de actividades, no se debe crear una nueva instancia de la actividad. En su lugar, se llamará al método onNewIntent() de la instancia existente de la actividad.

Esto es útil cuando deseas evitar la creación de múltiples instancias de la misma actividad si ya está en la parte superior de la pila.

Ejemplo

```
// En ActivityA, para iniciar ActivityB en una nueva tarea
val intent = Intent(this, ActivityB::class.java)
intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK
startActivity(intent)
```

Con este código, *FLAG_ACTIVITY_NEW_TASK* garantiza que **ActivityB** se inicie en una nueva tarea, incluso si ya existe una tarea en curso.

Esto puede ser útil si deseas que **ActivityB** se ejecute de manera independiente o aislada de las otras actividades en la aplicación.

5.2.2.2 Paso de información entre actividades

Cuando cambiamos de activity, es posible que cierta información disponible en el primero de ellos, sea necesaria en el segundo. Para poder pasar esa información se usa el sistema *clave-valor* y el método *putExtra()*.

En el primer activity incluiremos:

```
btn.setOnClickListener()
{
    it ->

    val intent: Intent = Intent( packageContext: this, ContactoActivity::class.java).apply { this: Intent

        putExtra( name: "PHONENUMBER",  value: "666111222")

    }
    startActivity(intent)
}
```

Y en el segundo:

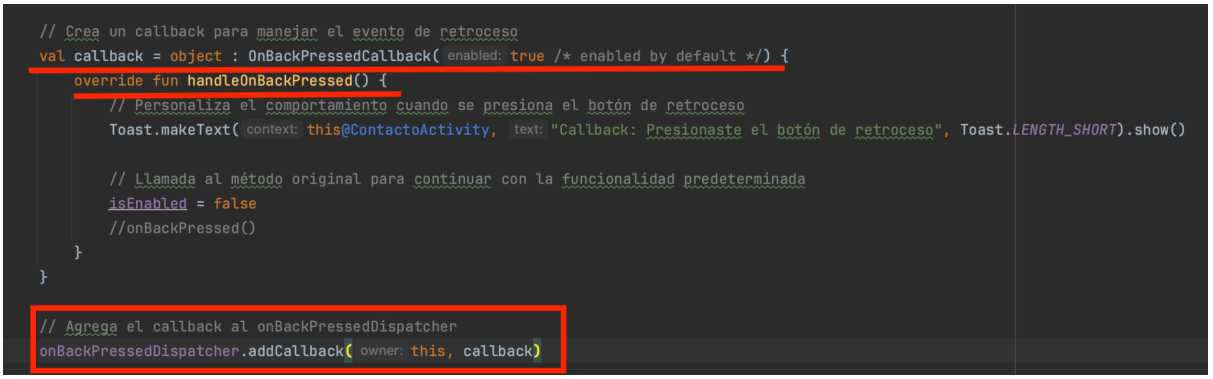
```
private fun makePhoneCall()
{
    val phoneNumber = intent.getStringExtra( name: "PHONENUMBER")

    val intent = Intent(Intent.ACTION_CALL, Uri.parse( uriString: "tel:$phoneNumber"))

    if (intent.resolveActivity(packageManager) != null)
        startActivity(intent)
    else
        Snackbar.make(binding.root,  text: "No se puede hacer la llamada porque " +
                "no hay ninguna app disponible", Snackbar.LENGTH_LONG).show()
}
```

5.2.2.3 Control del evento onBackPressed

onBackPressedDispatcher te permite interceptar y gestionar el evento de retroceso de la actividad.

En este ejemplo, se utiliza *OnBackPressedCallback* para crear un callback personalizado que maneja el evento de retroceso. El método *handleOnBackPressed()* se llama cuando se presiona el botón de retroceso. Puedes personalizar este método según tus necesidades.

Luego, el callback se agrega al *onBackPressedDispatcher*. La línea isEnabled = false se utiliza para desactivar el callback después de manejar el evento de retroceso, asegurando que no se ejecute en futuros eventos de retroceso.



```
// Crea un callback para manejar el evento de retroceso
val callback = object : OnBackPressedCallback( enabled: true /* enabled by default */) {
    override fun handleOnBackPressed() {
        // Personaliza el comportamiento cuando se presiona el botón de retroceso
        Toast.makeText( context: this@ContactoActivity,  text: "Callback: Presionaste el botón de retroceso", Toast.LENGTH_SHORT).show()

        // Llamada al método original para continuar con la funcionalidad predeterminada
        isEnabled = false
        //onBackPressed()
    }
}

// Agrega el callback al onBackPressedDispatcher
onBackPressedDispatcher.addCallback( owner: this, callback)
```

5.2.3 Lanzar una tarea (intent implícito)

Este caso suele darse cuando lo que hacemos es llamar a otra aplicación del dispositivo, como puede ser establecer una nueva alarma en el gestor de alarmas del dispositivo.

En el siguiente enlace tienes varios casos de ejemplo: https://developer.android.com/guide/components/intents-common?hl=es-419

Ejemplo

```
import android.content.Intent
import android.net.Uri
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // URL de la página web que deseas abrir
        val url = "https://www.ejemplo.com"

        // Crear un Intent para abrir el navegador web
        val intent = Intent(Intent.ACTION_VIEW, Uri.parse(url))

        // Comprobar si hay una aplicación que pueda manejar el Intent (naveg
        if (intent.resolveActivity(packageManager) != null) {
            startActivity(intent)
        } else {
            // Manejar el caso en el que no haya una aplicación para manejar
            // Puedes mostrar un mensaje de error o realizar otra acción
        }
    }
```

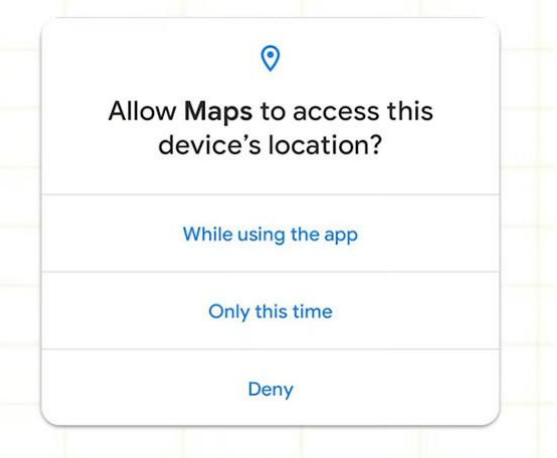
En este código lo más interesante es el método *resolveActivity* de la clase *Intent*, que sirve para comprobar previamente si es posible gestionar la petición.

5.3 Permisos

Cuando se quiere lanzar una tarea que no es propia de la app, se tendrá que tratar con otro tema, los **permisos**.

El tratamiento de los permisos en Android cambió a partir de la API 23 (Marshmallow). En el siguiente enlace puedes visitar todos los niveles de API Android.

Hasta entonces, los permisos se concedían durante la instalación de la app, y simplemente debían escribirse dentro del fichero *AndroidManifest*. Ahora, se deben conceder de manera explícita, en tiempo de ejecución.



Android categoriza los permisos en diferentes tipos, incluidos los del momento de la instalación, los permisos de tiempo de ejecución y los permisos especiales. Cada tipo de permiso indica el alcance de los datos restringidos a los que tu aplicación puede acceder, así como el alcance de las acciones restringidas que puede realizar, cuando el sistema le otorga ese permiso.

Permisos en el momento de la instalación

Los permisos en el momento de la instalación otorgan acceso limitado a los datos restringidos y permiten que realice acciones restringidas que casi no afectan al sistema u otras aplicaciones. Cuando declaras permisos en el momento de la instalación en la aplicación, el sistema le otorga automáticamente los permisos cuando el usuario la instala.

La tienda ***PlayStore*** muestra a los usuarios un aviso de permiso en el momento de la instalación cuando ve la página de detalles de la aplicación. Hay 2 tipos:

- **Permisos normales.** Permiten el acceso a los datos y las acciones que se extienden más allá de la zona de pruebas de tu aplicación. Sin embargo, los datos y las acciones presentan muy poco riesgo para la privacidad del usuario y el funcionamiento de otras aplicaciones. Por ejemplo: ***BLUETOOTH*** (permite que las aplicaciones se conecten a dispositivos bluetooth emparejados). Estos permisos tienen la categoría → Protection level: normal.

- **Permisos de firma.** Permiten el acceso a certificados digitales instalados para firmar digitalmente o identificar a una persona o entidad a través de su cuenta para descargar aplicaciones o acceder a servicios. Por ejemplo: ***BATTERY_STATS***

(permite que una aplicación recopile estadísticas de la batería.). Estos permisos tienen la categoría → Protection level: signature.

Permisos de tiempo de ejecución (o peligrosos)

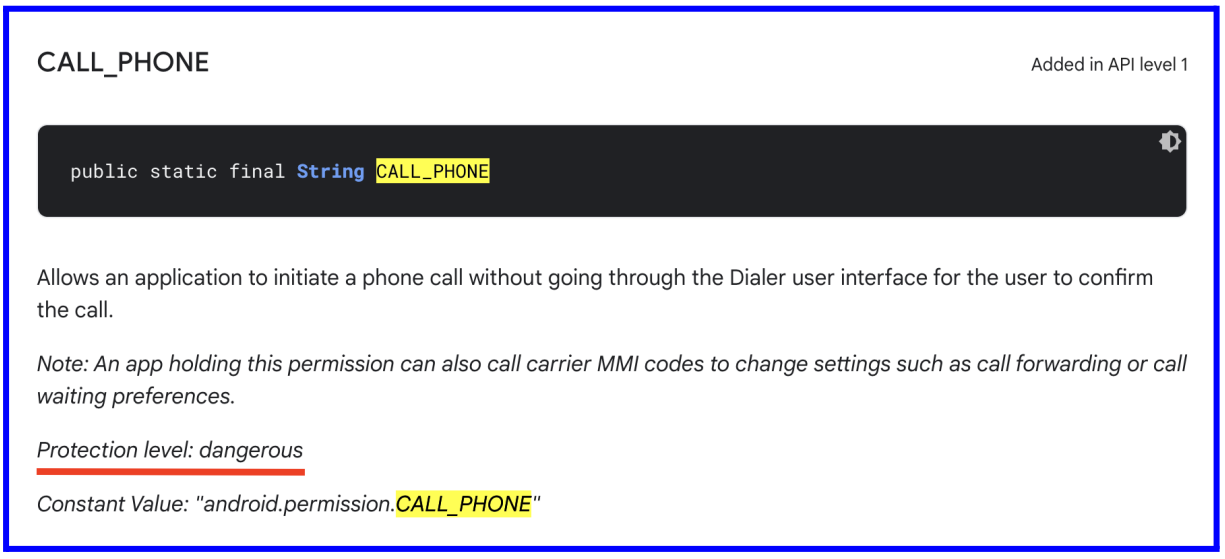
Estos permisos abarcan áreas en las cuales la aplicación requiere datos o recursos que incluyen información privada del usuario, o bien que podrían afectar los datos almacenados del usuario o el funcionamiento de otras aplicaciones. Cuando la aplicación solicita un permiso en tiempo de ejecución, el sistema presenta un mensaje de este permiso que debe ser aceptado por el usuario.

Muchos permisos en tiempo de ejecución acceden a los datos privados del usuario, un tipo especial de datos restringidos que incluye información que puede ser sensible. Algunos ejemplos de datos privados del usuario incluyen la ubicación y la información de contacto del usuario, la lista de los contactos del usuario, etc.

El micrófono, la cámara y la ubicación brindan acceso a información particularmente sensible. Por lo tanto, debes evitar acceder desde tu aplicación a estos dispositivos sin haber obtenido el permiso explícito para ello (permisos ***CAMERA*** para la cámara, ***RECORD_AUDIO*** para el micro y ***ACCESS_COARSE_LOCATION*** o ***ACCESS_FINE_LOCATION*** para la localización). Estos permisos tienen la categoría → Protection level: dangerous.

5.3.1 Tratamiento de permisos peligrosos

Vamos a ver un ejemplo, usando uno de los permisos de la categoría Protection level: dangerous. En este caso, vamos a probar con el permiso ***CALL_PHONE***, que nos permitiría realizar una llamada telefónica.



CALL_PHONE

Added in API level 1

```
public static final String CALL_PHONE
```

Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call.

Note: An app holding this permission can also call carrier MMI codes to change settings such as call forwarding or call waiting preferences.

Protection level: dangerous

Constant Value: "android.permission.CALL_PHONE"

En primer lugar, es necesario incluir en el fichero AndroidManifest, el permiso.

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

En segundo lugar, dentro del activity, escribiremos el siguiente código:

```
import ...

class MainActivity : AppCompatActivity() {
    // El código de solicitud para el permiso CALL_PHONE
    private val CALL_PHONE_PERMISSION_REQUEST = 123

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Verificar si ya tienes permiso para llamar
        if (ContextCompat.checkSelfPermission( context: this, android.Manifest.permission.CALL_PHONE) == PackageManager.PERMISSION_GRANTED) {
            makePhoneCall()
        } else {
            // Si no tienes permiso, solicítalo al usuario
            ActivityCompat.requestPermissions( activity: this, arrayOf(android.Manifest.permission.CALL_PHONE), CALL_PHONE_PERMISSION_REQUEST)
        }
    }

    // Este método se llama cuando el usuario responde a la solicitud de permiso
    override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<String>, grantResults: IntArray)
    {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults)

        if (requestCode == CALL_PHONE_PERMISSION_REQUEST) {
            if (grantResults.isNotEmpty() && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // Permiso otorgado, realiza la llamada
                makePhoneCall()
            } else {
                // Permiso denegado, puedes manejarlo de acuerdo a tus necesidades
            }
        }
    }
```

Y por último, tenemos la implementación del método ***makePhoneCall***().

```
    private fun makePhoneCall() {
        // Número de teléfono al que deseas llamar
        val phoneNumber = "1234567890"

        // Crear un Intent para realizar la llamada
        val intent = Intent(Intent.ACTION_CALL, Uri.parse( uriString: "tel:$phoneNumber"))

        // Comprobar si hay una aplicación que pueda manejar el Intent (aplicación de llamadas)
        if (intent.resolveActivity(packageManager) != null) {
            startActivity(intent)
        } else {
            // Manejar el caso en el que no haya una aplicación para manejar el Intent
            // Puedes mostrar un mensaje de error o realizar otra acción
        }
    }
}
```

Otra opción interesante sería no estar pidiendo permiso continuamente al usuario, si éste previamente ya rechazó otorgar dicho permiso. En ese caso, una posible solución sería esta:

```
binding.imageView.setOnClickListener{ it: View!

    if (ContextCompat.checkSelfPermission( context: this, android.Manifest.permission.CALL_PHONE) == PackageManager.PERMISSION_GRANTED)
        makePhoneCall()
    else
    {
        //El usuario ya ha rechazado previamente el permiso
        if (ActivityCompat.shouldShowRequestPermissionRationale( activity: this,
            android.Manifest.permission.CALL_PHONE))

            Snackbar.make(binding.root, text: "El permiso ha sido rechazado previamente." +
                    "Debes activarlo desde ajustes", Snackbar.LENGTH_LONG).show()

        else
            ActivityCompat.requestPermissions( activity: this, arrayOf(android.Manifest.permission.CALL_PHONE), CALL_PHONE_PERMISSION_REQUEST)
    }
```

5.3.2 Permiso "Solo mientras se usa la app"

La opción *"Solo mientras se usa la app"* se refiere a los permisos de **ubicación** en Android. Cuando una aplicación solicita permisos de ubicación, el usuario tiene la opción de otorgar esos permisos de tres formas:

- **Permitir siempre**. La aplicación puede acceder a la ubicación del dispositivo en cualquier momento, incluso cuando la aplicación está en segundo plano.

- **Permitir solo mientras se usa la app**. La aplicación solo puede acceder a la ubicación del dispositivo mientras está en primer plano, es decir, mientras el usuario está utilizando activamente la aplicación. Cuando la aplicación pasa a segundo plano, se suspende el acceso a la ubicación.

- **Denegar**. La aplicación no tiene acceso a la ubicación del dispositivo en absoluto.

La opción *"Solo mientras se usa la app"* es útil cuando una aplicación sólo necesita acceder a la ubicación del dispositivo mientras el usuario está interactuando con la aplicación, pero no necesita acceso continuo en segundo plano. Esto puede ser beneficioso para la privacidad del usuario, ya que limita el tiempo durante el cual la aplicación puede rastrear la ubicación.

Esta opción es comúnmente utilizada por aplicaciones que ofrecen características basadas en la ubicación, como mapas o servicios de entrega, donde la necesidad de acceder a la ubicación está directamente relacionada con la interacción activa del usuario con la aplicación.