Programación Multimedia y Dispositivos Móviles

UD 6. Intents, permisos y persistencia de la UI

Javier Carrasco

Curso 2024 / 2025

Intents, permisos y persistencia de la UI

6. Intents, permisos y persistencia de la UI

Los ***Intents*** son objetos que permitirán solicitar acciones a otros componentes de la aplicación (una *activity*, un servicio, un proveedor de contenido, etc), pero también permiten iniciar actividades en otras aplicaciones, como hacer una foto o ver un mapa.

Existen *Intents* de dos tipos:

- **Explícitos**, indicarán que deben lanzar exactamente, su uso típico es ejecutar diferentes componentes internos de una aplicación. Por ejemplo, una actividad (ventana nueva).
- **Implícitos**, se utilizan para lanzar tareas abstractas, del tipo *"quiero hacer una llamada"* o *"quiero hacer una foto"*. Estas peticiones se resuelven en tiempo de ejecución, por lo que el sistema buscará los componentes registrados para la tarea pedida, si encontrase varias, el sistema preguntará al usuario que componente prefiere.

La potencia de los *Intents* es muy grande, pero de momento se centrará la atención en los tres usos principales que se deben dominar.

- Para abrir una actividad:

Como ya sabes, una *activity* es una única pantalla de la aplicación. Se puede iniciar una nueva *activity* creando una nueva instancia, pasando un *Intent* mediante `startActivity()`. La *Intent* describirá que actividad se debe iniciar y los datos que hacen falta.

Si se necesita obtener un resultado al finalizar la actividad se utilizará el método `startActivityForResult()`. La actividad que lanza la llamada recibirá cuando finalice un objeto *Intent* separado en el *callback* de `onActivityResult()`.

- Para iniciar un servicio:

Un *Service* es un componente que realiza operaciones en segundo plano sin una interfaz de usuario. Se puede iniciar un servicio para realizar una única operación, como descargar un archivo, pasando un *Intent* a `startService()`. El *Intent* describe el servicio que se debe iniciar y contendrá los datos necesarios para la tarea.

Si el servicio está diseñado con una interfaz cliente-servidor, puedes establecer un enlace con el servicio de otro componente pasando un *Intent* a `bindService()`.

- Para entregar un mensaje:

Se utiliza para crear mensajes de aviso que cualquier aplicación puede recibir. También el sistema operativo puede enviar mensajes de eventos producidos en el sistema, como cuando el sistema arranca o el dispositivo comienza a cargarse. Se puede enviar un mensaje a otras *apps* pasando un *Intent* a `sendBroadcast()`, `sendOrderedBroadcast()` o `sendStickyBroadcast()`. Este tipo puede ser útil para la comunicación entre *apps*.

A continuación, se verá lo más básico de los *Intents*, implícitos y explícitos, para comenzar así a dominar las acciones básicas.

6.1. Filtros de intent

Como ya se ha comentado, cuando se lanza un *intent* implícito, es el propio sistema operativo el que busca la manera de poder realizarlo. Para que el sistema pueda encontrar la forma de hacerlo, deberá declararse en el fichero *manifest* de la aplicación esta posibilidad mediante el uso de la etiqueta `<intent-filter>`.

Si observas el fichero `AndroidManifest.xml` de un proyecto, podrás ver como se le indica al sistema operativo qué actividad es la que debe lanzar a la hora de ejecutar la aplicación. Esto es gracias a `android.intent.action.MAIN`, que sería la acción aceptada y a `android.intent.category.LAUNCHER` sería la categoría. Si la aplicación tiene más de una *activity*, esta categoría únicamente podrá estar en una de ellas.

```
<application
    ...
    <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>
    </activity>
</application>
```

Las acciones más utilizadas tienen definida su propia constante `ACTION` dentro de la clase `Intent`. Puedes consultar las constantes en el enlace al pie[1]. Una de las más comunes es `ACTION_VIEW`, que en el *manifest* se utilizaría como `android.intent.action.VIEW`. Esta acción se utiliza para poder hacer uso de los datos que se intercambian.

Las categorías[2] también disponen de sus propias constantes, en esta caso `CATEGORY`, dentro de la clase `Intent`.

En el siguiente código de ejemplo, se muestra la configuración de una actividad que puede enviar datos cuando estos sean de tipo texto mediante el uso de `ACTION_SEND`.

```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:mimeType="text/plain" />
    </intent-filter>
</activity>
```

Puedes encontrar más información sobre los filtros de *intent* en la documentación oficial[3] de Android.

---

1 *Standard Activity Actions* (https://developer.android.com/reference/kotlin/android/content/Intent#standard-activity-actions)
2 *Standard Categories* (https://developer.android.com/reference/kotlin/android/content/Intent#standard-categories)
3 *Intents* y filtros de *intents* (https://developer.android.com/guide/components/intents-filters?hl=es)

6.2. Abrir una actividad nueva (intent explícito)

Comienza por añadir un *EditText* y un *Button* a la `activity_main.xml`, quedando como puedes ver a continuación.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <com.google.android.material.textfield.TextInputLayout
        android:id="@+id/textInputLayout"
        style="@style/Widget.MaterialComponents.TextInputLayout.OutlinedBox"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:hint="@string/txt_editText"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <com.google.android.material.textfield.TextInputEditText
            android:id="@+id/ed_texto"
            android:layout_width="match_parent"
            android:layout_height="wrap_content" />
    </com.google.android.material.textfield.TextInputLayout>

    <Button
        android:id="@+id/button"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:layout_weight="1"
        android:text="@string/txt_button"
        app:layout_constraintEnd_toEndOf="@+id/textInputLayout"
        app:layout_constraintStart_toStartOf="@+id/textInputLayout"
        app:layout_constraintTop_toBottomOf="@+id/textInputLayout" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Fíjate en la propiedad `android:layout_marginStart`, que puede ser sustituida, o más bien complementada, por `android:layout_marignLeft` para añadir compatibilidad a la aplicación con APIs inferiores a la 17.

A continuación, se creará una segunda actividad vacía, la cual se abrirá al pulsar el botón creado en la actividad principal. Puedes utilizar la opción **File > New > Activity > Empty Activity**.

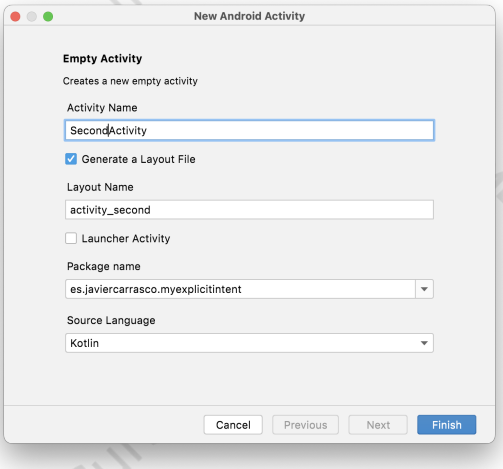


Figura 1

Esta acción añadirá automáticamente la siguiente información al fichero *Android Manifest* del proyecto.

```
<activity android:name=".SecondActivity" android:exported="false" />
```

Esta línea indica la existencia de una nueva *activity* en el proyecto. A continuación, se añadirá navegación, estableciendo la jerarquía de las *activities* utilizando la propiedad `android:parentActivityName`, esto le permitirá a la aplicación saber dónde volver cuando se pulse el botón *"atrás"*. Esta propiedad será la encargada de mostrar la flecha para volver en la barra de acción de la *app*.

```
<activity
    android:name=".SecondActivity"
    android:label="Second Activity"
    android:parentActivityName=".MainActivity"
    android:exported="false" />
```

La propiedad `android:label` permite cambiar el título de la *activity* mostrado en la barra de título de la aplicación. La propiedad `android:exported` permite indicar si esta *activity* puede ser accedida por otras aplicaciones por su nombre exacto con el valor a true, y todo lo contrario con su valor a false, en tal caso, sólo podrá ser accedida desde la propia aplicación.

De vuelta al *layout* de la segunda actividad, `activity_second.xml`, esta únicamente contendrá un *TextView* para mostrar el mensaje enviado desde la actividad principal como se verá más adelante.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SecondActivity">

    <TextView
        android:id="@+id/tv_msgReceived"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginLeft="16dp"
        android:layout_marginTop="32dp"
        android:layout_marginEnd="16dp"
        android:layout_marginRight="16dp"
        android:textSize="24sp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="Texto de muestra" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

La propiedad `tools:text` no requiere crear un valor en `string.xml`, esta se utiliza para ayudar en el diseño y ver como puede quedar, no aparecerá cuando se ejecute la aplicación. Ahora se modificará la lógica de la actividad que realiza la llamada, se comenzará con lo básico, llamar a otra *activity* sin pasarle valores.

```kotlin
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.button.setOnClickListener {
            // Se crea el objeto Intent.
            val myIntent = Intent(this, SecondActivity::class.java)
            // Se lanza la nueva activity con el Intent.
            startActivity(myIntent)
        }
    }
}
```

Se crea un objeto de tipo `Intent` y se le pasa el contexto y la clase `SecondActivity.kt` que se desea abrir, aunque ésta debe especificarse como de tipo Java. El contenido de la clase en cuestión será inicialmente el siguiente.

```
class SecondActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_second)
    }
}
```

6.2.1. Paso de información entre actividades

A continuación, se implementará el paso de información entre actividades, lo primero que se debe saber es que se utiliza el sistema **clave-valor**. Se modificará la llamada haciendo uso del método `putExtra()` que usa el sistema clave-valor.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    companion object {
        const val EXTRA_MESSAGE = "myMessage"
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.button.setOnClickListener {
            // Se crea el objeto Intent.
            val myIntent = Intent(this, SecondActivity::class.java).apply {
                // Se añade la información a pasar por clave-valor.
                putExtra(EXTRA_MESSAGE, binding.edTexto.text.toString())
            }
            // Se lanza la nueva activity con el Intent.
            startActivity(myIntent)
        }
    }
}
```

Se añade una constante con la clave para el valor que vaya a ser pasado, así se evitarán posibles errores a la hora de recoger los datos. Fíjate que para crear esta variable se debe utilizar un `companion object`[4].

A continuación se muestra el código necesario para recoger los datos en la segunda actividad.

```
class SecondActivity : AppCompatActivity() {
    private lateinit var binding: ActivitySecondBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
```

4 Companion Objects (https://kotlinlang.org/docs/object-declarations.html#companion-objects)

```
6           binding = ActivitySecondBinding.inflate(layoutInflater)
7           setContentView(binding.root)
8
9           // Se recuperan los datos y se asignan al TextView.
10          val message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE).apply { this: String?
11              binding.tvMsgReceived.text = this
12          }
13      }
14  }
```

Debes tener en cuenta que, el objeto `intent`, hace referencia al *intent* de la actividad que hace la llamada a la actividad que se quiere abrir. Haciendo uso del método `apply` puedes evitarte la declaración de la variable `message`. El resultado puede ser como el que se muestra en la siguiente figura.

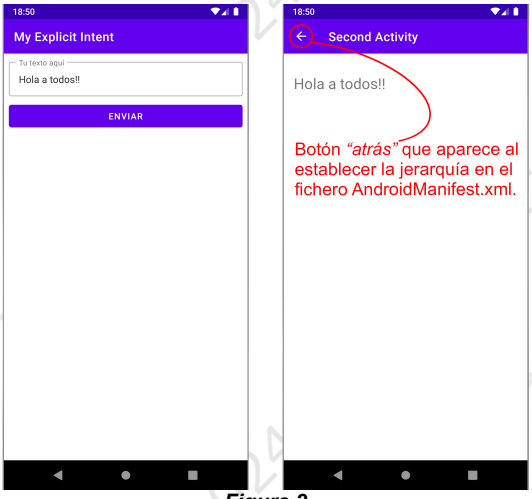


Figura 2

Puedes añadir un control de campos vacíos para evitar pasar a la segunda actividad sin texto en el cuadro, pasando únicamente, cuando el usuario haya escrito algo.

Figura 3

6.2.2. Devolución de un estado

Se ha visto como hacer un intercambio de información entre *activities* en una única dirección. Ahora se verá como hacer una comunicación bidireccional, ya que en ocasiones puede ser interesante recibir una respuesta de otra *activity*. Para el siguiente ejemplo se utilizará la llamada al método `startActivityForResult()`.

Para este ejemplo se creará un nuevo proyecto con dos actividades, la principal, será como la vista en el ejemplo anterior, a la cual se le añadirá un *TextView* para recibir aquello que devuelva la segunda actividad.

```
<TextView
    android:id="@+id/tv_Result"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    tools:text="Muestra el resultado" />
```

Figura 4

La segunda actividad (*activity_second.xml*) simulará una aceptación de condiciones, la cual permitirá devolver si se aceptan o no.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    ...
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SecondActivity">

    <TextView
        android:id="@+id/tv_nameReceived"
        android:layout_width="0dp"
        ...
        tools:text="Texto de muestra" />

    <Button
        android:id="@+id/bt_accept"
        ...
        android:text="@android:string/ok" />

    <Button
        android:id="@+id/bt_cancel"
        ...
        android:text="@android:string/cancel" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

El código que se omite corresponde a los ajustes de los componentes a la pantalla, estos pueden hacerse desde el editor sin problemas. Fíjate en las propiedades *text* de los botones, en este caso, se está utilizando el recurso *String* de Android, no es necesario crear las propiedades más comunes y además, estarán traducidas y se ajustarán al idioma del sistema operativo. El resultado buscado lo puedes ver en la siguiente figura.

Figura 5

A continuación se verá el código **Kotlin** necesario para el intercambio de información entre dos *activities*. En este primer caso, la actividad principal recibirá de la segunda actividad si la operación es correcta (*ok*) o no (*cancel*), no se obtendrá ningún dato más.

En primer lugar, en la clase `MainActivity.kt` se deberá añadir la funcionalidad para el botón *Enviar*.

```kotlin
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding
    private var REQUEST_CODE = 1234

    companion object {
        const val TAG_APP = "myExplicitIntent2"
        const val EXTRA_NAME = "userNAME"
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Se oculta el TextView que mostrará el resultado.
        binding.tvResult.visibility = View.INVISIBLE

        binding.button.setOnClickListener {
            if (!binding.edTexto.text.isNullOrEmpty())
                askConditions()
            else binding.textInputLayout.error = getString(R.string.txt_error)
        }
    }

    private fun askConditions() {
        Log.d(TAG_APP, "askConditions")
        // Se vuelve a ocultar el TV que mostrará el resultado.
        binding.tvResult.visibility = View.INVISIBLE
        // Se crea un objeto de tipo Intent.
        val myIntent = Intent(this, SecondActivity::class.java).apply {
            // Se añade la información a pasar por clave-valor.
            putExtra(EXTRA_NAME, binding.edTexto.text.toString().trim())
        }
```

```
        // Se lanza la activity (Deprecated).
        startActivityForResult(myIntent, REQUEST_CODE)
    }
}
```

En este caso, se utiliza la instrucción `startActivityForResult(myIntent, 1234)`, este método lanza una nueva *activity* y queda a la espera de recibir respuesta. Los parámetros son el *Intent* configurado con la *activity* que se quiere abrir, y un código de tipo entero que se asignará para identificar la respuesta cuando esta se produzca. No utilices siempre el mismo código para llamar a todas tus *activities*. A continuación se verá el código de `SecondActivity.kt`.

```
class SecondActivity : AppCompatActivity() {
    private lateinit var binding: ActivitySecondBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivitySecondBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Se recuperan los datos y se asignan al TextView.
        intent.getStringExtra(MainActivity.EXTRA_NAME).apply { this: String?
            binding.tvNameReceived.text = getString(
                R.string.msgAccept,
                this
            )

            binding.btAccept.setOnClickListener {
                setResult(Activity.RESULT_OK)
                Log.d(TAG_APP, "Valor devuelto OK")
                finish()
            }

            binding.btCancel.setOnClickListener {
                setResult(Activity.RESULT_CANCELED)
                Log.d(TAG_APP, "Valor devuelto CANCELED")
                finish()
            }
        }
    }
}
```

Para devolver el valor, aceptado o cancelado, verdadero o falso, se utiliza el método `setResult(valor)`, donde *valor* será `Activity.RESULT_OK` o `Activity.RESULT_OK`. Además, el método `finish()` de la clase *Activity* se utiliza para volver a la actividad que realizó la llamada, cerrando la actividad en la que se encuentra.

Observa la línea `binding.tvNameReceived.text = getString(R.string.msgAccept, this)`, concatena el valor definido en `strings.xml` con la variable con el valor pasado. Para hacer esto debes definir la variable `msgAccept` en `strings.xml` como se muestra a continuación.

```xml
<string name="msgAccept">Hola %s, ¿aceptas las condiciones?</string>
```

Ahora, vuelve a `MainActivity.kt`, deberás sobrescribir el método `onActivityResult()` para poder recibir la respuesta de la actividad llamada. Puedes añadirlo utilizando la opción **Ctrl+O** o en el menú **Code > Override Methods**. También puedes utilizar la opción de auto-completar según vas escribiendo.

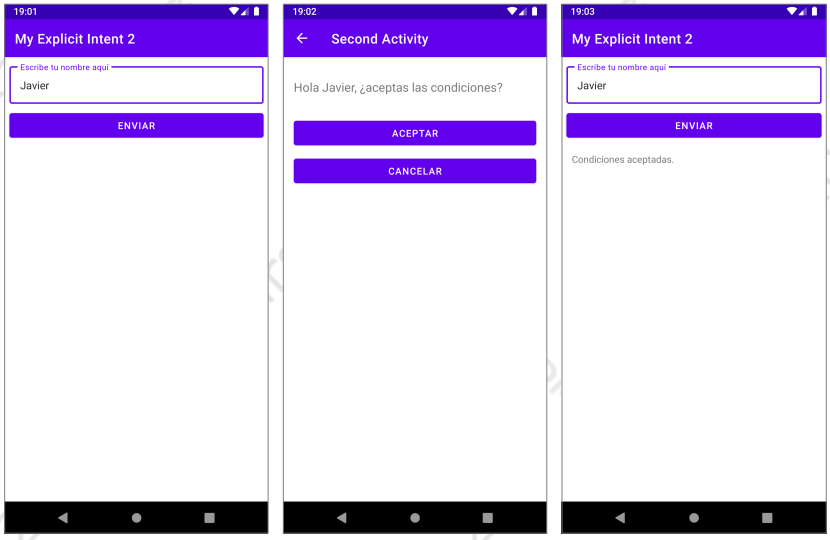
```kotlin
// Se obtiene el resultado de la Activity llamada.
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)

    if (requestCode == REQUEST_CODE) {
        if (resultCode == Activity.RESULT_OK)
            binding.tvResult.text = "Condiciones aceptadas."

        if (resultCode == Activity.RESULT_CANCELED)
            binding.tvResult.text = "Se canceló el contrato!"

        binding.tvResult.visibility = View.VISIBLE
    }
}
```

Al sobrecargar este método, se utiliza el parámetro `requestCode` que indicará el código de la *activity* que ha finalizado y así saber cual es y poder actuar en consecuencia. El parámetro `resultCode` indicará el resultado devuelto por la *activity* finalizada.



6.2.3. Intercambio de información por contrato

Si has seguido los pasos del ejemplo, te habrás dado cuenta que `startActivityForResult()` y `onActivityResult()` están *deprecated*. Esto se debe a que a partir de **AndroidX**, se cambia de estrategia para el intercambio de información entre actividades, y se utiliza para ello un sistema de contratos[5]. Este cambio se aplica para evitar el consumo excesivo de memoria, evitando así que el proceso destruya la actividad que ha lanzado el *intent*. Observa como quedaría el ejemplo visto con este nuevo sistema, solo afectará a la creación del *intent*, no a la devolución de datos.

Todos los cambios se verán reflejados en la clase principal. Desaparece la variable `REQUEST_CODE`, ya no es necesaria, así como el método `onActivityResult()`. En su lugar, se creará el siguiente objeto que se encargará de las acciones a realizar tras recibir la respuesta.

```
// Objeto para recoger la respuesta de la activity.
var resultadoActivity = registerForActivityResult(StartActivityForResult()) { result ->
        // Si fuese necesario recuperar información adicional
        // se obtendría mediante esta línea.
        val data: Intent? = result.data

        if (result.resultCode == Activity.RESULT_OK)
            binding.tvResult.text = "Condiciones aceptadas."

        if (result.resultCode == Activity.RESULT_CANCELED)
            binding.tvResult.text = "Se canceló el contrato!"

        binding.tvResult.visibility = View.VISIBLE
    }
```

En el método `askConditions()` cambia la llamada a la nueva *activity*, fíjate que ahora se utilizará el objeto `resultadoActivity` creado para lanzar la nueva vista.

```
// Se lanza la activity.
resultadoActivity.launch(myIntent)
```

Observa que ahora se lanzará el *intent* utilizando el método `launch()` del objeto creado, al cual se le deberá pasar el mismo *intent* que el visto anteriormente. El resultado de la aplicación será el mismo que en el ejemplo anterior.

6.2.4. Paso bidireccional de información entre actividades

Supón ahora que quieres devolver algo más de información, ya bien sea mediante la intervención del usuario o generada por la propia *activity* llamada. Ahora añade una *RatingBar* a la **activity_second.xml**.

```
<RatingBar
    android:id="@+id/ratingBar"
    android:layout_width="wrap_content"
```

5 Cómo obtener un resultado de una actividad (https://developer.android.com/training/basics/intents/result)

```
    android:layout_height="wrap_content"
    android:numStars="5"
    android:rating="10"
    android:stepSize="0.5" />
```

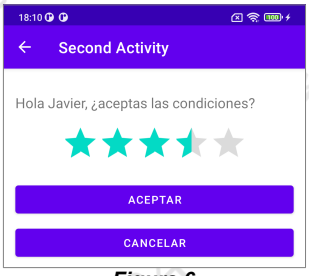


Figura 6

A continuación, modifica la clase `SecondActivity.kt` para que recoja el valor de la *RatingBar* y lo pase a la *activity* que ha realizado la llamada.

```
binding.btAccept.setOnClickListener {
    val intentResult: Intent = Intent().apply {
        // Se añade el valor del rating.
        putExtra("RATING", binding.ratingBar.rating)
    }

    setResult(Activity.RESULT_OK, intentResult)
    Log.d(TAG_APP, "Valor devuelto OK y valoración RatingBar")
    finish()
}
```

Únicamente, al pulsar el botón *Aceptar* se devolverá el resultado que el usuario haya establecido mediante la *RatingBar*. Se deberá crear un nuevo *Intent* al pulsar el botón, al que se le añadirá el resultado y el método `setResult()`, en este caso incluirá el *Intent* creado como parámetro. Ahora se deberá recoger desde la `MainActivity.kt` el resultado establecido.

```
// Objeto para recoger la respuesta de la activity.
var resultadoActivity = registerForActivityResult(StartActivityForResult()) { result ->
        // Se recupera la información adicional.
        val data: Intent? = result.data

        if (result.resultCode == Activity.RESULT_OK) {
            val valoracion = data?.getFloatExtra("RATING", 0.00F)
            binding.tvResult.text = getString(R.string.txt_Accepted, valoracion.toString())
        }
        if (result.resultCode == Activity.RESULT_CANCELED)
            binding.tvResult.text = getString(R.string.txt_noAccepted)

        binding.tvResult.visibility = View.VISIBLE
    }
```

En este caso se hace uso del objeto `data` que viene devuelto a través de `result`. Fíjate que se utiliza el operador `?`, esto es debido a que `data` puede ser nulo.

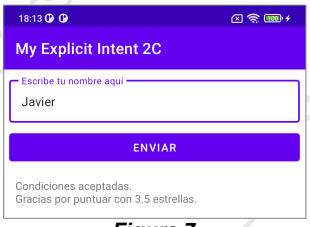


Figura 7

6.2.5. Paso de datos complejos entre actividades

En más de una ocasión te encontrarás con que debes pasar datos más complejos entre actividades, por ejemplo, una clase. Para ello, cuando se crea esa clase modelo, deberás decidir si quieres hacer que sea ***Parcelable***[6] o ***Serializable***[7]. A continuación, se verán las diferencias y como hacerlo.

Parcelable vs Serializable

Cuando comienzas a trabajar con Android, hasta que no chocas con el paso de referencias a objetos para pasarlos a otras actividades, o *Fragments*, no te das cuenta de que no se puede, y es así, no se puede, es necesario que los objetos se monten en un *Intent* o un *Bundle*.

Básicamente las diferencias entre ambos son las siguientes:

- ***Parcelable***: su uso está totalmente optimizado para Android, su implementación puede resultar algo más complicada, por suerte se utilizará Kotlin. Es mucho más rápido que Serializable.
- ***Serializable***: es más lenta, su implementación está basada en la interfaz estándar de Java, por lo que su implementación es más sencilla.

A continuación, se verá como implementar ambos sistemas en un proyecto Android, básicamente se pasará un objeto entre dos actividades, la primera actividad montará el objeto para pasarlo a la segunda que lo mostrará en pantalla.

Como se está trabajando con Kotlin, esto ayudará mucho en la implementación de ***Parcelable***, en primer lugar, en el *Build.Gradle* del módulo, deberás añadir el siguiente *plugin*.

```
plugins {
    ...
    id 'kotlin-parcelize'
}
```

6 Parcelable (https://developer.android.com/reference/android/os/Parcelable.html)
7 Serializable (https://developer.android.com/reference/java/io/Serializable.html)

Con esto, ya únicamente habrá que anotar la clase con la anotación `@Parcelize`, y todo los demás se hará de forma totalmente transparente. Ahora se creará una nueva *data class* para recoger los datos de alumnos.

```
import android.os.Parcelable
import kotlinx.parcelize.Parcelize

@Parcelize
data class Student(
    val idStudent: Int,
    val name: String,
    val surname: String,
    val age: Int
) : Parcelable
```

Cuidado con el *import*, si todavía están conviviendo, al añadir el *import* de la anotación es posible que te de dos opciones, no utilices *kotlinx.android.parcel.Parcelize*, está marcada como obsoleta.

Ahora que está "*parcelizada*" la clase, observa como se montará para pasarla a otra actividad.

```
val student = Student(1, "Javier", "Carrasco", 45)

binding.btnPasar.setOnClickListener {
    Intent(this, SecondActivity::class.java).apply {
        putExtra("STUDENT", student)
        startActivity(this)
    }
}
```

Para recoger el objeto en la segunda actividad, se hará de forma muy parecida a la vista anteriormente.

```
val student: Student?

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU)
    student = intent.getParcelableExtra("STUDENT", Student::class.java)
else student = intent.getParcelableExtra("STUDENT")

binding.tvResult.append("${student!!.idStudent}\n")
binding.tvResult.append("${student.name}\n")
binding.tvResult.append("${student.surname}\n")
binding.tvResult.append("${student.age}\n\n")
```

En este caso hay que hacer un control de versión, ya que el método *getParcelabeExtra* cambia a partir de la API 33. Con esto ya tienes pasado un objeto entre actividades, y también te serviría para pasarlo a un *fragment*.

Si lo que necesitas es pasar un conjunto de datos, se puede pasar un *array* de la siguiente forma, para añadir los datos al *intent*, deberás cambiar *putExtra* por *putParcelableArrayListExtra*.

```
val students: ArrayList<Student> = arrayListOf(
    Student(1, "Javier", "Carrasco", 45),
    Student(2, "Patricia", "Aracil", 44),
    Student(3, "Nicolás", "Royo", 43)
)

binding.btnPasar.setOnClickListener {
    Intent(this, SecondActivity::class.java).apply {
        putParcelableArrayListExtra("STUDENT", students)
        startActivity(this)
    }
}
```

La recepción también variará un poco, pero deberás seguir manteniendo el control de versiones.

```
val students: ArrayList<Student>?

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU)
    students = intent.getParcelableArrayListExtra("STUDENT", Student::class.java)
else students = intent.getParcelableArrayListExtra("STUDENT")!!

students!!.forEach {
    binding.tvResult.append("${it.idStudent}\n")
    binding.tvResult.append("${it.name}\n")
    binding.tvResult.append("${it.surname}\n")
    binding.tvResult.append("${it.age}\n\n")
}
```

A continuación, observa el mismo ejemplo, pero esta vez implementando ***Serializable***. Lo primero es que no hay que añadir ningún *plugin* al proyecto ni ninguna dependencia y, la clase *Student* será como se muestra.

```
import java.io.Serializable

data class Student(
    val idStudent: Int,
    val name: String,
    val surname: String,
    val age: Int
) : Serializable
```

La forma de pasar un único objeto a la segunda actividad no varía con respecto al ejemplo *Parcelable*. Sí variará ligeramente la recepción de la información en la segunda actividad.

```
val student: Student?

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU)
    student = intent.getSerializableExtra("STUDENT", Student::class.java)
else student = intent.getSerializableExtra("STUDENT") as Student

```

```
binding.tvResult.append("${student!!.idStudent}\n")
binding.tvResult.append("${student.name}\n")
binding.tvResult.append("${student.surname}\n")
binding.tvResult.append("${student.age}\n\n")
```

Como puedes ver, hay que cambiar el nombre del método utilizado y, además del control de versión, es necesario hacer un *cast* para APIs inferiores a la 33.

Ahora bien, como se ha visto en el ejemplo anterior, si necesitas pasar un conjunto de datos mediante *Serializable*, para añadir los datos al *intent* deberás modificar el código de la siguiente manera.

```
val students: ArrayList<Student> = arrayListOf(
    Student(1, "Javier", "Carrasco", 45),
    Student(2, "Patricia", "Aracil", 44),
    Student(3, "Nicolás", "Royo", 43)
)

binding.btnPasar.setOnClickListener {
    Intent(this, SecondActivity::class.java).apply {
        putExtra("STUDENT", students)
        startActivity(this)
    }
}
```

Como podrás ver, lo único que cambia es la fuente de datos, los datos se añaden al *intent* mediante un *putExtra*. En cuanto a la recepción, únicamente cambia el cast, que debe hacerse en ambos *getSerializableExtra*.

```
val students: ArrayList<Student>?

students = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU)
    intent.getSerializableExtra("STUDENT", Student::class.java) as ArrayList<Student>
else intent.getSerializableExtra("STUDENT") as ArrayList<Student>

students.forEach {
    binding.tvResult.append("${it.idStudent}\n")
    binding.tvResult.append("${it.name}\n")
    binding.tvResult.append("${it.surname}\n")
    binding.tvResult.append("${it.age}\n\n")
}
```

Esta opción mediante *Serializable* también funcionaría, para este ejemplo, apenas verás diferencias en cuanto a rendimiento, pero si los datos son mayores, podrás ver que el rendimiento baja con este último método.

Resumiendo, no está de más conocer ambos métodos, pero, si vas a trabajar con Android y, dada la facilidad de implementación que proporciona Kotlin mediante el *plugin kotlin-parcelize*, la mejor elección es utilizar ***Parcelable***.

6.2.6. Control de onBackPressed()

Ahora que ya sabes abrir otras actividades, te encontrarás con algunos inconvenientes al utilizar el botón atrás del sistema. Para poder controlar sus acciones, deberás sobrecargar el método `onBackPressed()` de la actividad. Algo simple, como desactivarlo, se podría hacer comentando la llamada al constructor, quedando como se muestra a continuación. Además, se muestra un *Toast* al pulsarlo, pero puedes añadir todo aquello que necesites que haga según sea el caso, como eliminar un registro, actualizar la base de datos, limpiar un formulario, etc.

```
override fun onBackPressed() {
    Toast.makeText(
        applicationContext,
        "onBackPressed() called!!",
        Toast.LENGTH_SHORT
    ).show()

    //super.onBackPressed()
}
```

6.2.7. Añadir flags a los intents

El uso de *flags* en la creación de los *intents* permite controlar como se manejará el objeto a lanzar. Para añadir un *flag* a un *intent* puede hacerse de la siguiente manera.

```
val myIntent = Intent(this, SecondActivity::class.java)
myIntent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP)
```

Según el tipo de *intent* que se está lanzando, se utilizará un tipo de *flag* u otro, en este caso, como se está hablando de lanzar actividades, se tres de los *flags* más utilizados y que pueden resultar de utilidad.

FLAG_ACTIVITY_NEW_TASK → Este *flag* convierte la *activity* que se lanza en la primera actividad de una nueva pila, creando un nuevo historial de navegación. Suele utilizarse en actividades que actúan como lanzadera o punto de entrada a la aplicación. No se recomienda para actividades que esperan un resultado.

FLAG_ACTIVITY_CLEAR_TOP → La actividad que se va a iniciar se lanzará sobre la misma tarea, de esta manera, todas las demás que haya por encima de ella en la pila se cerrarán, limpiando la pila. Esto puede evitar navegaciones incómodas.

FLAG_ACTIVITY_SINGLE_TOP → Con este *flag*, la actividad que se vaya a lanzar no se ejecutará si ya se encuentra en la parte de arriba de la pila del historial.

6.2.8. Cambiar el efecto de transición entre actividades

Por defecto, siempre que se abre una actividad se produce un cambio mediante un efecto de *slide*, si quieres cambiar esto, se puede comenzar por un efecto básico[8], un fundido entre actividades. Para hacerlo, deberás añadir en el *startActivity* un segundo parámetro como el siguiente.

```
startActivity(
    Intent(this, SecondActivity::class.java).addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP),
    ActivityOptions.makeSceneTransitionAnimation(this).toBundle()
)
```

Puedes personalizar los efectos según te interese, pero uno que puede resultar muy llamativo y darle un toque muy profesional a la aplicación es el efecto mediante elementos compartidos entre vistas.

La idea es la siguiente, se parte de dos actividades que tienen elementos en común, que serán los que animarán el cambio. Observa las dos siguientes *activities*, tienen el mismo elemento, exacto en ambas, cambiando la ubicación. Es importante que te fijes que la propiedad `transitionName`, que será la encargada de identificar los elementos que aplicarán el efecto.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout ... >

    <LinearLayout
        android:id="@+id/linear"
        ...
        android:transitionName="desliza"
        ...
        app:layout_constraintStart_toStartOf="parent">

        <ImageView
            android:id="@+id/imageView"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:scaleType="centerCrop"
            app:srcCompat="@drawable/person" />

        <TextView
            android:id="@+id/tvHola"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="16dp"
            android:text="@string/txt_label" />
    </LinearLayout>
</androidx.constraintlayout.widget.ConstraintLayout>
```

8 *Start an activity using an animation* (https://developer.android.com/develop/ui/views/animations/transitions/start-activity)

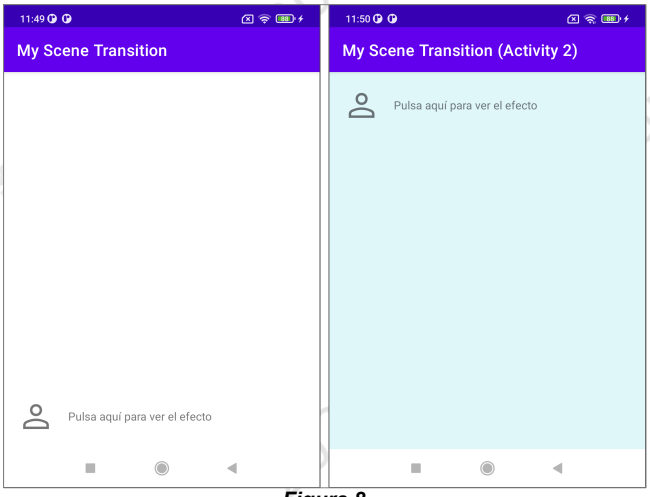


Figura 8

Ahora, en la *MainActivity*, en el método *onCreate*, se añadirá un *listener* sobre el LinearLayout, de manera que cuando se pulse sobre este, se abrirá la segunda actividad, observa como se crea la animación.

```
binding.linear.setOnClickListener {
    val intent = Intent(this, SecondActivity::class.java)
    val options = ActivityOptions.makeSceneTransitionAnimation(
        activity: this,
        binding.linear, sharedElementName: "desliza",
    )
    startActivity(intent, options.toBundle())
}
```

Al crearse el objeto *options*, se indica en primer lugar la actividad y a continuación, el elemento común y el nombre del elemento compartido, recuerda que debe existir en ambas *activities*.

Ahora, al cambiar de actividad, verás el efecto producido, el movimiento inverso se produce al pulsar “atrás” en el botón del sistema, pero si quieres finalizar desde código, la instrucción *finish* no te servirá, deberás utilizar en su lugar `finishAfterTransition()`.

Es posible que en un momento dado, quieras aplicar este efecto sobre elementos de un *RecyclerView*, y desde el *adapter* no sepas como acceder a la *activity* para poder crear las opciones de la transición. Una opción puede ser la siguiente, desde el *ViewHolder* puedes utilizar la siguiente instrucción para obtener de una vista la *activity* a partir del parámetro *view*.

```
val act = view as Activity
```

6.2.9. Organizar la creación de intents

Como has podido comprobar, cada vez que hay que llamar a otra *activity* hay que montar el código de forma muy similar, además de repetir el código una y otra vez, y si la actividad que hace las llamadas, hace muchas, puede ser ingobernable.

Lo que se puede hacer es trasladar la creación del *intent* a la propia actividad que quieres llamar, de forma que el código quede más legible y sea más sencillo realizar las llamadas. Retomando el ejemplo visto para el paso bidireccional de información, se harán las siguientes modificaciones.

En la clase `SecondActivity` se añadirá un *companion object* que se encargará de montar el *Intent*, quitando esta tarea de la actividad que hace la llamada.

```
companion object {
    const val EXTRA_NAME = "userNAME"

    fun navigateToSecondActivity(activity: AppCompatActivity, name: String): Intent {
        return Intent(activity, SecondActivity::class.java).apply {
            // Se añade la información a pasar por clave-valor.
            putExtra(EXTRA_NAME, name.trim())
            addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP)
        }
    }
}
```

En la clase `MainActivity`, el método que se había creado, *askConditions*, quedará ahora como se muestra.

```
private fun askConditions() {
    Log.d(TAG_APP, "askConditions")
    // Se vuelve a ocultar el TV que mostrará el resultado.
    binding.tvResult.visibility = View.INVISIBLE

    resultadoActivity.launch(
      SecondActivity.navigateToSecondActivity(this, binding.edTexto.text.toString())
    )
}
```

Para este caso, como se espera una respuesta, el método *navigateToSecondActivity*, debe devolver una *Intent* para poder recoger la respuesta, si no fuese así, se podría ahorrar la devolución del *Intent* y lanzar directamente la actividad, de esa forma bastaría con llamar al método.

```
fun navigate(activity: AppCompatActivity, itemId: Int = -1) {
    val intent = Intent(activity, SecondActivity::class.java).apply {
        putExtra(EXTRA_ID, itemId)
    }
    activity.startActivity(intent)
}
```

6.3. Gestión de 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, hasta entonces, se concedían durante la instalación. Ahora, debe concederse de manera explícita aquellos considerados **peligrosos**, ya no se pide permiso durante el proceso de instalación sino en tiempo de ejecución.

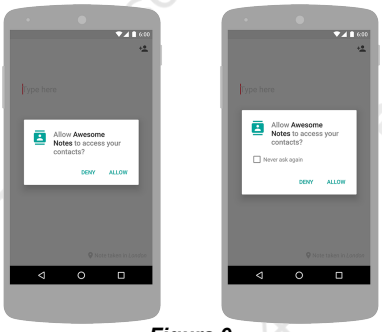


Figura 9

A raíz de este cambio se produce una clasificación de los permisos, básicamente se distinguirán tres tipos de permisos según sea su nivel de peligrosidad.

- **Permisos normales:** estos se utilizan cuando la aplicación necesita acceder a recursos o servicios fuera del ámbito de la *app*, donde no existe riesgo para la privacidad del usuario o para el funcionamiento de otras aplicaciones, por ejemplo, cambiar el uso horario.

  Si se declara en el *manifest* de la aplicación uno de estos permisos, el sistema otorgará automáticamente permiso para su uso durante la instalación. Además de no preguntarse al usuario por ellos, estos no podrán revocarlos.

- **Permisos de firma:** estos son concedidos durante la instalación de la *app*, pero sólo cuando la aplicación que intenta utilizar el permiso está firmada por el mismo certificado que la aplicación que define el permiso. Estos permisos no se suelen utilizar con aplicaciones de terceros, es decir, se utilizan entre aplicaciones del mismo desarrollador.

- **Permisos peligrosos:** estos permisos involucran áreas potencialmente peligrosas, como son la privacidad del usuario, la información almacenada por los usuarios o la interacción con otras aplicaciones. Si se declara la necesidad de uso de uno de estos permisos, se necesitará el consentimiento explícito del usuario, y se hará en tiempo de ejecución. Hasta que no se conceda el permiso, la *app* no podrá hacer uso de esa funcionalidad. Por ejemplo, acceder a los contactos.

Puedes encontrar todos los permisos que se pueden utilizar en la documentación de Google para Android[9]. El valor que necesites añadir al *manifest* lo encontrarás en *Constant Value*, y *Protection level* indica el tipo de permiso que es.

```
INTERNET

public static final String INTERNET

Allows applications to open network sockets.

Protection level: normal

Constant Value: "android.permission.INTERNET"
```

9 *Manifest permission* (https://developer.android.com/reference/android/Manifest.permission)

Los permisos normales no requieren de una solicitud al usuario para poder funcionar, es por eso que se comenzarán por los **permisos peligrosos**[10], concretamente, uno de los más habituales, el uso de la cámara de fotos del dispositivo. Según la documentación de *Google*, cuando uno se plantea la gestión de permisos debe plantearse el siguiente flujo de trabajo para una correcta gestión.

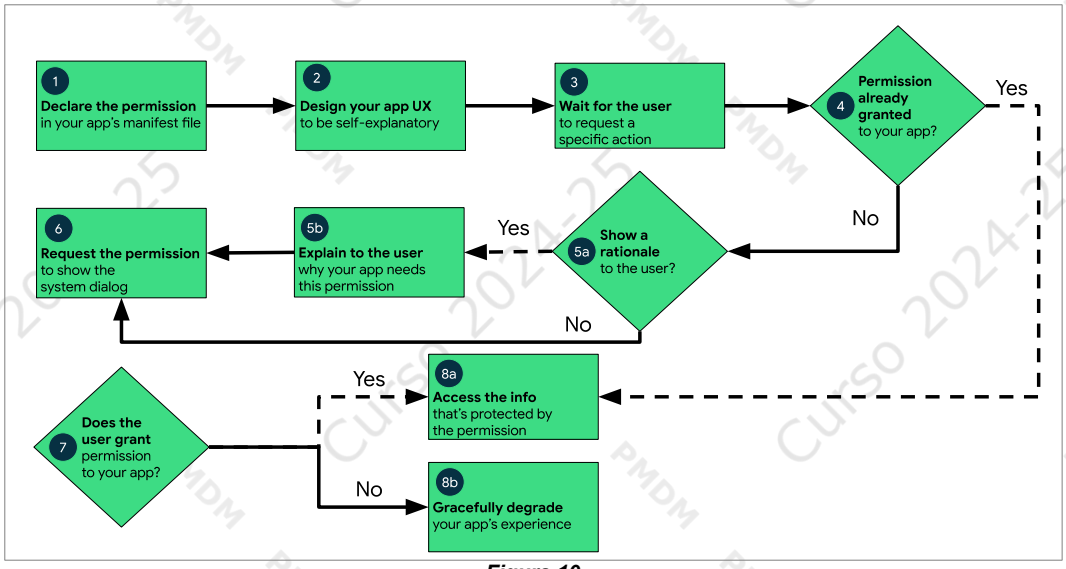


Figura 10

El primer paso para iniciar la gestión de permisos será añadir el permiso requerido en el fichero *Manifest* del proyecto. Además, la cámara puede añadir requisitos[11] adicionales como verás a continuación.

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

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

  <uses-feature android:name="android.hardware.camera" android:required="true" />
  <uses-feature android:name="android.hardware.camera.autofocus" android:required="true" />

  <application ...
```

La primera etiqueta `<uses-permission>` indicará que se necesitará permiso para acceder a la cámara, la siguiente `<uses-feature>`, indica que la aplicación requiere del *hardware* de la cámara y del *auto-focus* para poder funcionar.

Debes tener en cuenta que *Google Play* filtrará tu aplicación para comprobar que cumple con los estándares de seguridad establecidos.

10 Solicitud permisos de tiempo de ejecución (https://developer.android.com/training/permissions/requesting)
11 <uses-feature> (https://developer.android.com/guide/topics/manifest/uses-feature-element)

El siguiente paso será diseñar la UI, en este caso, se creará un *ImageView* para contener la imagen capturada (se verá más adelante), un botón para lanzar la cámara y una etiqueta que se utilizará para mostrar el estado del permiso.

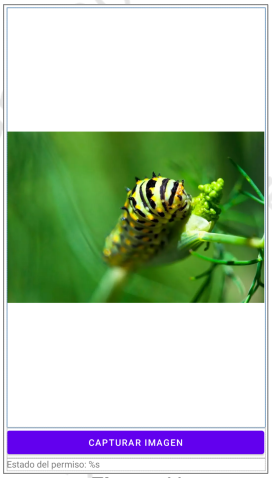


Figura 11

Una vez lanzada la aplicación, si accedes a la información de esta en el dispositivo, en la sección de permisos, podrás ver algo parecido a la siguiente imagen.

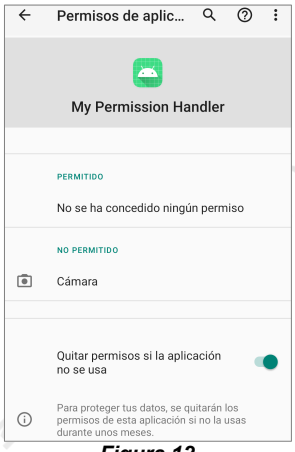


Figura 12

Antes de continuar, debes saber que existen dos formas de gestionar el código de solicitud del permiso, gestionar el código de manera manual o, permitir que el sistema lo haga por ti, ésta desde AndroidX. Se comenzará por esta última.

6.3.1. Código de solicitud administrado por el sistema

El primer paso será añadir la siguiente dependencia de *Activity*, de esta forma podrá automatizarse la gestión del código de solicitud por el sistema, añade en el build.gradle del módulo la siguiente biblioteca.

```
implementation 'androidx.activity:activity-ktx:1.6.1'
```

En la clase que hará uso del recurso que necesita el permiso, en este caso la *MainActivity*, se añadirá el siguiente *callback*, que se encargará de recoger la respuesta del usuario ante la pregunta.

```
val requestPermissionLauncher =
    registerForActivityResult(ActivityResultContracts.RequestPermission()) { isGranted ->
        if (isGranted) { // isGranted es de tipo Boolean.
            // Permission is granted. Acciones a realizar con el permiso concedido.
        } else {
            // Se daría una explicación al usuario sobre las consecuencias de denegar el
            // permiso. Acciones a realizar sin el permiso.
        }
    }
```

Para conseguir un código que pueda ser reutilizable, ya que la gestión de permisos es algo casi de uso diario en el desarrollo de aplicaciones móviles, se creará la clase `PermissionHandler` (**File > New > Kotlin Class/File**), aunque puedes elegir el nombre que quieras.

```
class PermissionHandler(val activity: Activity) {

    fun checkPermission(permission: String): Boolean {
        return when {
            ContextCompat.checkSelfPermission(
                activity, permission
            ) == PackageManager.PERMISSION_GRANTED -> {
                // Permiso ya concedido.
                true
            }
            // Este método devuelve TRUE si se ha denegado el permiso en algún momento.
            activity.shouldShowRequestPermissionRationale(permission) -> {
                // Se muestra una explicación sobre la necesidad de conceder el permiso.
                // Se muestra un Toast, pero lo ideal sería mediante un Dialog, de tal
                // forma que se pueda controlar que si acepta, se vuelva a pedir y si
                // cancela, continúe la aplicación sin bloquearla, pero sin acceso al
                // recurso.
                Toast.makeText(
                    activity,
                    activity.getString(R.string.showInContextUI),
                    Toast.LENGTH_LONG
                ).show()
                false
            }
            else -> {
                // You can directly ask for the permission.
                // Se recoge la respuesta del usuario mediante el callback creado.
                false
            }
        }
    }
}
```

Lo ideal es solicitar el permiso cuando realmente se vaya a necesitar, en este caso, cuando vaya a pulsarse el botón para abrir la cámara.

```
binding.button.setOnClickListener {
  if (PermissionHandler(this).checkPermission(Manifest.permission.CAMERA)) {
     binding.tvStatePermission.text = getString(R.string.txt_permissionState, "granted")
     val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
     startActivity(intent)
  } else {
     binding.tvStatePermission.text = getString(R.string.txt_permissionState, "no granted")
     requestPermissionLauncher.launch(Manifest.permission.CAMERA)
  }
}
```

6.3.2. Gestión manual del código de solicitud

Otra opción, si no utilizas la versión de AndroidX, es gestionar de manera manual el código de solicitud. En tal caso, no será necesario añadir la dependencia anterior.

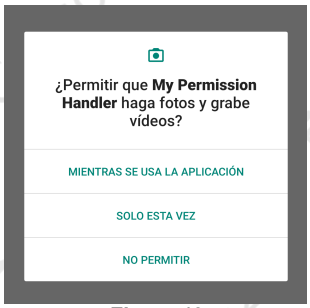


Figura 13

Sí que será necesario establecer un código de respuesta para la solicitud del permiso, por ejemplo, mediante una constante.

```
const val REQUEST_CODE_CAMERA = 12345

class MainActivity : AppCompatActivity() { ...
```

La clase ***PermissionHandler*** creada en el sistema anterior sirve para este caso, puedes importarla arrastrándola al proyecto. Los cambios se encuentran en la solicitud del permiso al pulsar el botón, ya no se utiliza el *callback*, ya que se debe indicar el código de respuesta en la solicitud al pulsar el botón.

```
binding.button.setOnClickListener {
  if (PermissionHandler(this).checkPermission(Manifest.permission.CAMERA)) {
     binding.tvStatePermission.text = getString(R.string.txt_permissionState, "granted")
     val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
     startActivity(intent)
  } else {
     binding.tvStatePermission.text = getString(R.string.txt_permissionState, "no granted")

     requestPermissions(arrayOf(Manifest.permission.CAMERA), REQUEST_CODE_CAMERA)
  }
}
```

Para este caso no se dispone del *callback* como ya se ha dicho, pero será necesaria una manera de poder recoger la respuesta del usuario, para eso, deberá sobrecargarse el método `onRequestPermissionsResult`, en el que deberá evaluarse según el código devuelto por la petición, el código creado con REQUEST_CODE_CAMERA.

```
override fun onRequestPermissionsResult(
    requestCode: Int, permissions: Array<out String>, grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)

    when (requestCode) {
        REQUEST_CODE_CAMERA -> {
            if (grantResults.isNotEmpty()
                && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // Permission is granted.
                binding.tvStatePermission.text =
                    getString(R.string.txt_permissionState, "granted")
            } else {
                // Explicación sobre las consecuencias de denegar el permiso.
                binding.tvStatePermission.text =
                    getString(R.string.txt_permissionState, "no granted")
```

```
            }
            return
        }
        else -> { // Nada que hacer.
            return
        }
    }
}
```

En ambos casos, como has podido ver, deberás evaluar la respuesta del usuario desde el punto en el que se pide el permiso.

A continuación, se verán algunos ejemplos de los *intents* implícitos de uso más común[12], para los casos en los que sea necesaria la solicitud de permisos, se utilizará la gestión del código de solicitud administrada por el sistema.

6.3.3. Ejemplos de intents implícitos

Se comenzará con *intents* cuyos permisos se consideran normales, por lo que se concederán durante el proceso de instalación de la aplicación. Partiendo de un proyecto nuevo con API mínima 25 y una *empty activity*, se añadirá un botón para lanzar cada *intent*.

Abrir una URL en el navegador

En primer lugar, se añadirá el permiso de uso de *Internet* en la aplicación, realmente no sería necesario, ya que la aplicación no accederá a Internet, sino que se hará uso de un navegador, pero así ya lo conoces.

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

    <application ...
```

Pero, desde Android 11 (API 30), si la aplicación tiene como objetivo la API 30 o superior, deben utilizarse las `queries` en el fichero `AndroidManifest.xml` del proyecto para poder interactuar con paquetes externos. Deberás añadir la siguiente etiqueta `queries` justo antes de la etiqueta `application`. Estas líneas indican que tipo de aplicación se desea buscar, en este caso, un navegador[13], y el tipo de datos que se van a utilizar.

```
<!-- Comprueba que existe un navegador en el sistema -->
<queries>
    <intent>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="https" />
    </intent>
</queries>
```

12 Intents comunes (https://developer.android.com/guide/components/intents-common)
13 *Browser* (https://developer.android.com/training/basics/intents/package-visibility-use-cases#check-browser-available)

También existe la posibilidad de añadir la siguiente línea, que indica que se quiere acceder a todos los paquetes disponibles, pero si subes la aplicación a *Google Play*, se aplicarán ciertas restricciones y no es muy recomendable.

```
<uses-permission android:name="android.permission.QUERY_ALL_PACKAGES"
    tools:ignore="QueryAllPackagesPermission" />
```

El código **Kotlin** que necesita la *app* para lanzar un *intent* implícito, en este caso abrir una página web, será como se muestra a continuación. En el método `onCreate()` de la clase `MainActivity.kt` se añade la funcionalidad del botón.

```
binding.btnWebPage.setOnClickListener { it:View!
    Intent(Intent.ACTION_VIEW, Uri.parse("https://www.javiercarrasco.es")).apply { this:Intent
        if (this.resolveActivity(packageManager) != null)
            startActivity(this)
        else Log.d("DEBUG", "Hay un problema para encontrar un navegador.")
    }
}
```

Al pulsar el botón se crea un *intent* que realizará la acción ACTION_VIEW[14], esta es la acción más común, se utiliza para que la información que se manda se muestre en la actividad a la que se está llamando. Por último se le indicará la URL utilizando la clase `Uri` [15] y su método `parse`.

Cuando el usuario pulse el botón, si el sistema detecta más de una forma de cumplir la petición preguntará como debe resolverla, permitiendo dejarla por defecto si se elige la opción *SIEMPRE*.

Fíjate en el uso de `resolveActivity()` antes de iniciar el *intent*, esto se hace para comprobar previamente que puede gestionarse la petición y evitar un posible fallo de la aplicación.

Además, debes tener en cuenta el protocolo que se utiliza, HTTPS, en este caso no se ha controlado, pero debería hacerse, ya que si no es un protocolo seguro no se abrirá y se producirá un error.

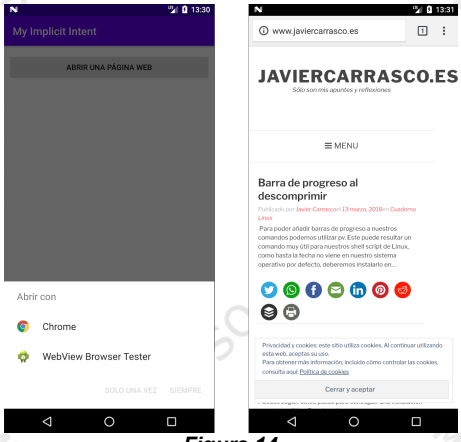


Figura 14

Marcar un número de teléfono

Este puede resultar útil, según el flujo de trabajo que estés buscando en tu aplicación. La idea es dejar preparado el número para llamar, a falta de pulsar el botón llamada, esto permite esquivar la necesidad de pedir permiso para llamar (que se verá más adelante).

Se añadirá un nuevo botón, etiquetado como “Marcar un número de teléfono”, el cual lanzará el siguiente *intent*. No es necesario añadir ningún permiso al *Manifest*.

14 *Action View* (https://developer.android.com/reference/kotlin/android/content/Intent.html#action_view)
15 Uri (https://developer.android.com/reference/android/net/Uri)

```
binding.btnDialPhone.setOnClickListener {
    val dial = "tel: "
    startActivity(Intent(Intent.ACTION_DIAL, Uri.parse(dial + "+34777111222")))
}
```

Para lanzar la acción ACTION_DIAL es necesario indicar el protocolo junto con el número de teléfono, para que el sistema detecte la aplicación a utilizar, el protocolo deberá ser **"tel: "**. Hecho esto, al pulsar el botón se abrirá la aplicación Teléfono, a la espera que el usuario pulse el botón para llamar.

Mandar un SMS

El siguiente código permite enviar un SMS mediante la aplicación del teléfono, por lo que no se necesita conceder permisos en la aplicación. Si quisieses que tu aplicación enviase el mensaje directamente sí sería necesario añadir el permiso SEND_SMS, considerado peligroso. Además, deberás añadir un *intent-filter* para esa tarea.

```
binding.btnSendSMS.setOnClickListener {
    Intent(Intent.ACTION_SENDTO).apply {
        data = Uri.parse("smsto:777666777")
        putExtra("sms_body", "Cuerpo del mensaje")

        startActivity(this)
    }
}
```

Fíjate como se indica el destinatario mediante el método `Uri` y como se añade texto al cuerpo del mensaje haciendo uso del método `putExtra()`. Según la versión del dispositivo, es posible que encuentres algún problema con *resolveActivity* para encontrar una aplicación para enviar SMS, por eso aquí se ha omitido.

Enviar un correo electrónico

Este ejemplo trata de compartir información con ese tipo de aplicaciones que así lo especifiquen mediante los ***intent-filter***, en ese caso se tratará de enviar un correo electrónico, y se verá como el correo electrónico aparece completado a falta de enviar mediante la aplicación, esto hace que no sea necesario indicar un permiso específico para ello en la aplicación.

```
binding.btnSendEmail.setOnClickListener {
    val TO = arrayOf("javier@javiercarrasco.es")
    val CC = arrayOf("")

    Intent(Intent.ACTION_SEND).apply {
        type = "text/html" // o también text/plain
        putExtra(Intent.EXTRA_EMAIL, TO)
        putExtra(Intent.EXTRA_CC, CC)
        putExtra(Intent.EXTRA_SUBJECT, "Envío de un email desde Kotlin")
        putExtra(Intent.EXTRA_TEXT, "Esta es mi prueba de envío de un correo.")

```

```
        if (this.resolveActivity(packageManager) != null)
            startActivity(Intent.createChooser(this, "Enviar correo..."))
        else Log.d("DEBUG", "Hay un problema para enviar el correo electrónico.")
    }
}
```

Los campos para `EXTRA_EMAIL` y `EXTRA_CC` deben crearse mediante el uso de *arrayOf*, esto es debido a que se puede especificar más de un correo. El campo *type* determinará el tipo de aplicación que se utilizará para enviar el correo, cuando el sistema pregunta, suele ofrecer más opciones si se trata texto plano.

Observa que se utiliza el método `createChooser` de la clase *Intent*, este muestra un diálogo con las diferentes aplicaciones que puedan utilizarse, o ninguna, para resolver el *intent*.

Abrir la aplicación de mapas

A continuación, puedes ver el código necesario para abrir la aplicación de mapas, generalmente *Google Maps*, con una localización. Al no necesitar establecer la ubicación, no es necesario solicitar el permiso, ya que éste sería peligroso.

```
binding.btnOpenMaps.setOnClickListener {
    startActivity(
        Intent(Intent.ACTION_VIEW, Uri.parse("geo:0,0?q=Alicante"))
    )
}
```

En este ejemplo puedes ver como se pide a la aplicación que localice la ciudad de Alicante en España. Si conoces las coordenadas, y además quieres añadir una etiqueta para la localización, puedes utilizar la siguiente *Uri*.

```
Uri.parse("geo:0,0?q=38.34,-0.49(Alicante)")
```

Añadir una alarma al despertador

Para este ejemplo es necesario establecer el permiso correspondiente para poder crear una alarma en el despertador, en el *manifest* deberás añadir la siguiente línea. Este permiso está catalogado como *normal*, por tanto no se necesita pedir permiso al usuario.

```
<uses-permission android:name="com.android.alarm.permission.SET_ALARM" />
```

El código que permite realizar esta acción será el siguiente que se muestra.

```
binding.btnAlarm.setOnClickListener {
    Intent(AlarmClock.ACTION_SET_ALARM).apply {
        putExtra(AlarmClock.EXTRA_MESSAGE, "Se acabó dormir")
        putExtra(AlarmClock.EXTRA_HOUR, 7)
        putExtra(AlarmClock.EXTRA_MINUTES, 45)

        if (intent.resolveActivity(packageManager) != null) {
            startActivity(this)
```

```
        }
    }
}
```

El método `resolveActivity` se utiliza para comprobar que haya al menos una aplicación capaz de resolver la petición, en tal caso no devolverá nulo. Sin esta comprobación la aplicación puede fallar y cerrarse.

Realizar una llamada de teléfono

Este caso ya hace uso de **permisos peligrosos**, por lo que se añadirá la gestión de permisos vista anteriormente. Para que la *app* pueda realizar la llamada deberás añadir el siguiente permiso al *manifest*.

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

Si consultas el manual de referencia de Android, podrás ver que el permiso CALL_PHONE[16] está clasificado como peligroso. El código para lanzar una llamada desde la aplicación será el siguiente.

```
binding.btnCallPhone.setOnClickListener {
    if (PermissionHandler(this).checkPermission(Manifest.permission.CALL_PHONE)) {
        Intent(Intent.ACTION_CALL, Uri.parse("tel:965555555")).apply {
            startActivity(this)
        }
    }else {
        requestPermissionCallPhone.launch(Manifest.permission.CALL_PHONE)
    }
}
```

Hacer uso de la cámara y recuperar el resultado

El siguiente ejemplo muestra cómo lanzar un *intent* hacia la cámara de fotos y obtener el resultado de la acción para mostrarlo en un *ImageView*, destacar que el resultado obtenido es un *thumbnail*[17], para conseguir la imagen completa consulta los anexos o la documentación de Google[18].

Para este ejemplo, se aplicará el permiso visto al inicio de este punto en el fichero *manifest*, además, se marcará como necesario para poder ejecutarse la acción y se añadirán los requisitos *hardware* que se utilizarán.

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

<uses-feature android:name="android.hardware.camera" android:required="true" />
<uses-feature android:name="android.hardware.camera.autofocus" android:required="true" />
```

Ahora, será necesario añadir el siguiente *callback* para poder recuperar el *thumbnail* que se obtendrá al hacer la foto.

16 *Call Phone* (https://developer.android.com/reference/android/Manifest.permission.html#CALL_PHONE)
17 Un *thumbnail* es una miniatura de una imagen, muy utilizado cuando la imagen es de gran tamaño.
18 Cómo tomar fotos (https://developer.android.com/training/camera/photobasics)

```
private var resultCaptura = registerForActivityResult(StartActivityForResult()) { result ->
    val data: Intent? = result.data

    if (result.resultCode == RESULT_OK) {
        val thumbnail: Bitmap? = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU)
            data?.getParcelableExtra("data", Bitmap::class.java)
        else data?.getParcelableExtra("data")

        binding.imageView.setImageBitmap(thumbnail)
    }
}
```

Observa que para obtener la información se comprueba la versión del sistema, esto es debido a que desde la API 33, ha cambiado la llamada a *getParcelableExtra*.

Por último, deberá comprobarse si está o no concedido el permiso y añadir el código para lanzar el *intent*. Se llamará al *callback* para recoger la imagen, recuerda que este sistema también se conoce como contrato.

```
binding.btnTakePicture.setOnClickListener {
    if (PermissionHandler(this).checkPermission(Manifest.permission.CAMERA)) {
        val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)

        if (intent.resolveActivity(packageManager) != null)
            resultCaptura.launch(intent)

    } else {
        requestPermissionCamera.launch(Manifest.permission.CAMERA)
    }
}
```

6.4. Persistencia de datos de la UI

Si no lo has visto todavía, debes saber que se puede perder información de la UI según cambien los estados de la aplicación, el simple hecho de girar la pantalla puede producir la pérdida de datos. Por suerte, no todos los elementos pierden información, pero para ilustrar este problema, crea la siguiente aplicación, la cual se llamará *My Litle ViewModel*.

Este nuevo proyecto constará de dos actividades, la segunda de momento estará vacía, pero la principal deberá tener el aspecto que muestra la figura. Básicamente la actividad contendrá un cuadro de texto, dos botones y una etiqueta que mostrará el texto escrito en el cuadro al pulsar el botón “Pulsa aquí”. El segundo botón abrirá la segunda actividad vacía.

Figura 15

Como mejoras añadidas al diseño, se ha modificado el tamaño de la barra de acción (*ActionBar*), para ello, puedes añadir las siguientes líneas al recurso de tema por defecto de la aplicación, no es necesario que esté en los dos ficheros de tema.

```
<!-- Status bar color. -->
<item name="android:statusBarColor">?attr/colorPrimaryVariant</item>
<item name="actionBarSize">100dp</item>
```

El código de la clase *MainActivity* de inicio será el que se muestra a continuación, suficiente para destacar el problema planteado.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        supportActionBar!!.setSubtitle(
            getString(R.string.txt_subtitle) + " - " + this.localClassName)
        Log.d("onCreate", "onCreate")

        binding.btnAccion.setOnClickListener {
            if (binding.tvSalida.text.isBlank())
                binding.tvSalida.text = binding.etTexto.text
            else binding.tvSalida.append("\n${binding.etTexto.text}")
        }

        binding.btnAccion2.setOnClickListener {
            startActivity(Intent(this, SecondActivity::class.java))
        }
    }

    override fun onDestroy() {
        Log.d("onDestroy", "onDestroy")
        super.onDestroy()
    }
}
```

Observa como se añade el título a la barra de estado de manera directa en código. Si ejecutas la aplicación, añades un texto al *EditText*, y pulsa el botón “Pulsa aquí”, verás que la etiqueta se rellenará, pero si ahora giras la pantalla, verás que la etiqueta pierde la información, algo que no ocurre con el texto del *EditText*. Lo mismo pasa al cambiar a la segunda actividad y volver con el botón “atrás” del sistema, pero no con la flecha de la barra de estado, que borra todo el contenido. Esto mismo ocurre cuando se cierra una actividad utilizando el método *finish()*.

6.4.1. savedInstanceState

A continuación, se resolverá este problema con una primera aproximación, haciendo uso del objeto `savedInstanceState` de tipo `Bundle` que tiene como parámetro el método *onCreate()*. Esto está considerado como un guardado ligero del estado de UI[19]. Para poder guardar el contenido de la etiqueta se deberá sobrecargar el siguiente método.

19 Guardar y restablecer estado de la IU (https://developer.android.com/guide/components/activities/activity-lifecycle#saras)

```kotlin
override fun onSaveInstanceState(outState: Bundle) {
    outState.run { putString("TEXTO", binding.tvSalida.text.toString()) }
    // Es necesario llamar al super para guardar los datos.
    super.onSaveInstanceState(outState)
}
```

Este método es llamado según se va cerrando la actividad, observa como los datos se guardan en el *Bundle* como clave-valor. En última instancia, se llamará al *super* para que se produzca el guardado.

Este sistema de guardado se utiliza cuando la cantidad de datos a salvar es trivial, esto es debido a que este proceso requiere de serialización en el sub-proceso principal y consume memoria del sistema. Ahora, para recuperar la información guardada, podrá utilizarse cualquiera de los dos métodos siguientes. El primer método consistirá en hacer la comprobación dentro del método *onCreate()* como se muestra a continuación.

```kotlin
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    if (savedInstanceState != null)
        with(savedInstanceState){ this: Bundle
            binding.tvSalida.text = this.getString("TEXTO")
        }
    ...
```

El segundo método sería sobrecargando el método `onRestoreInstanceState`, éste método se ejecutará tras el método *onStart()* y solo si se ha guardado un estado previamente, por lo que no es necesario comprobar el *Bundle*.

```kotlin
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    // Es necesaria la llamada al super para restaurar los datos.
    super.onRestoreInstanceState(savedInstanceState)

    savedInstanceState.run { this: Bundle
        binding.tvSalida.text = this.getString("TEXTO")
    }
}
```

En este ejemplo se ha utilizado la función de extensión `run`, de esta forma se devuelve el resultado de la función y se trabaja con el *Bundle* como *this*, algo que no podría hacerse con *with*. Comprueba que ahora, sí se guarda el estado de la UI al girar el dispositivo o, al volver de la segunda actividad mediante el botón “atrás” del sistema.

6.4.2. ViewModel

Ahora se añadirá un objeto `ViewModel` asociado a la actividad principal, estos objetos podrán sobrevivir más allá de las instancias específicas de las vistas. El alcance de un objeto *ViewModel* vendrá determinado por el `Lifecycle` que se pasará al `ViewModelProvider` al recibir el objeto.

Los objetos *ViewModel* permanecerán en memoria mientras el *Lifecycle* encargado de determinar su alcance no desaparezca definitivamente.

El *ViewModel* se solicitará en el método *onCreate()* de la actividad. Ten en cuenta que este método puede ser llamado varias veces, pero el *ViewModel* existirá desde la primera vez que sea solicitado.

Es muy importante que ningún *ViewModel* haga referencia directa a las vistas, *Lyfecycle* o referencia al contexto de una actividad.

Vuelve al proyecto y elimina todo lo relacionado con *savedInstanceState* para que vuelva al estado inicial (que se pierda la información en la etiqueta al girar la pantalla).

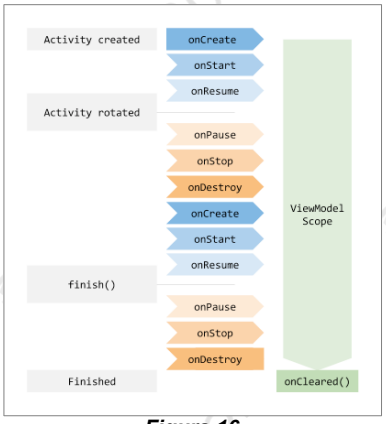


Figura 16

A continuación, crea una nueva clase que se llamará `MainViewModel` y que extenderá de la clase *ViewModel*. El nombre de la clase *ViewModel* creada permite identificar a que actividad pertenece, o se encuentra asociada.

```
class MainViewModel: ViewModel() {
    var textoEtiqueta: String? = null
}
```

En el *ViewModel* se declara la variable que se encargará de almacenar la información de la etiqueta de la UI, pero no sólo habrán propiedades, puedes añadir todo el código que consideres oportuno para la gestión de la información. El siguiente paso será asociar el *ViewModel* creado con la actividad que tendrá asociada. Añade las siguientes líneas en la clase *MainActivity*.

```
private lateinit var mainViewModel: MainViewModel

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    // Se establece la asociación del ViewModel.
    mainViewModel = ViewModelProvider(this).get(MainViewModel::class.java)

    // Se actualiza la UI.
    binding.tvSalida.text = mainViewModel.textoEtiqueta
    ...
```

Además, para mantener el *ViewModel* actualizado, deberá guardarse la información en la variable creada, esto se hará cada vez que se pulse el botón para mostrar el texto en la etiqueta.

```
// Se guardan los cambios en el ViewModel.
mainViewModel.textoEtiqueta = binding.tvSalida.text.toString()
```

Esta forma de vinculación sería la opción más básica, otra forma que puedes encontrar es añadiendo la siguiente dependencia al *Gradle*.

```
implementation 'androidx.activity:activity-ktx:1.6.1'
```

Ahora, en el método *onCreate()*, cambia la forma de hacer la asociación del *ViewModel* para que quede de la siguiente forma, deberás eliminar la línea de asociación del método *onCreate*, y cambia la propiedad *mainViewModel* de la siguiente forma.

```
private val mainViewModel: MainViewModel by viewModels()
```

La dependencia[20] añadida permite acceder a las API que admiten composiciones compiladas sobre *Activity*, en este caso a *viewModels()* mediante el operado *by*. El resto quedaría igual.

Cuando se utiliza *ViewModel*, suele utilizarse un sistema llamado ***backing*** (respaldo) para el manejo de las propiedades. De esta forma se evita que se acceda directamente a la propiedad, y se hace mediante una propiedad que haga de intermediaria.

```
private var _acumulador = 0
val acumulador: Int
    get() = _acumulador
```

En la clase que hereda de *ViewModel*, sí podrá trabajarse con la variable *_acumulador*, pero gracias a la sobrecarga del *getter*, la variable devuelta será la de respaldo.

```
fun incrementaDato(): Int {
    _acumulador++
    return acumulador
}
```

Uno de los objetivos principales es separar lo máximo posible la lógica de la parte visual. Cuando se trate el modelo MVVM se ampliará información.

6.5. Crear una pantalla de Splash

Puede resultar interesante que la aplicación que se esté diseñando tenga una pantalla de bienvenida, una "*splash screen*". Este tipo de pantallas suelen dar una buena sensación sobre la aplicación que se está cargando.

A continuación, se mostrará una forma fácil de crear una *splash screen* para las aplicaciones que se creen. Se partirá de un nuevo proyecto al que se llamará *"My Splash Screen"*. En primer lugar, se añadirá el siguiente estilo al recurso ***themes*** con las siguientes propiedades.

```
<!-- Modificaciones para el Splash Screen -->
<style name="SplashScreenTheme" parent="Theme.MaterialComponents.DayNight.NoActionBar">
    <item name="android:windowFullscreen">true</item>
    <item name="android:windowTranslucentNavigation">true</item>
</style>
```

20 activity-ktx (https://developer.android.com/jetpack/androidx/releases/activity)

De esta forma se activa la modo *full screen* y el efecto translúcido del *layout*. Seguidamente se creará una nueva *activity* vacía (***New > Activity > Empty Activity***), a la que se llamará `SplashScreenActivity`. En el *layout*, `activity_splash_screen.xml`, añade la imagen o texto que quieras que aparezca en el *splash*.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SplashScreenActivity">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:contentDescription="@string/app_name"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/logo_android" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

A continuación, en la clase `SplashScreenActivity.kt`, deberás añadir el siguiente código para controlar la carga y la posterior re-dirección a la actividad principal de la aplicación. La clase quedaría como se muestra.

```kotlin
class SplashScreenActivity : AppCompatActivity() {
    private lateinit var binding: ActivitySplashScreenBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivitySplashScreenBinding.inflate(layoutInflater)
        setContentView(binding.root)

        val intent = Intent(this, MainActivity::class.java)
        Handler(Looper.getMainLooper()).postDelayed({
            startActivity(intent)
            overridePendingTransition(android.R.anim.fade_in, android.R.anim.fade_out)
            this.finish()
        }, 1500)
    }
}
```

Como ves aquí, se hace uso de la clase `Handler`, esta es una clase *threading*, la cual permite pasar mensajes y procesar objetos *Runnables*, asociándolos a la cola de mensajes.

Junto con `Looper`, que se encargará de despachar la tarea encomendada, se programa su ejecución con *postDelayed*, en este caso lanzar la actividad creada tras 1500 mili-segundos.

Ya solo quedaría modificar el `AndroidManifest.xml`, cambiando el *intent-filter* de la `MainActivity` a la `SplashScreenActivity` y añadiendo las siguiente propiedades junto con el nuevo estilo. Observa como quedaría definida la actividad para el *splash* en el *manifest*.

```
<activity
    android:name=".SplashScreenActivity"
    android:exported="true"
    android:launchMode="singleTop"
    android:screenOrientation="portrait"
    android:theme="@style/SplashScreenTheme">

    <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>
```

Con esto, ya tendrías tu pantalla de *splash* en la aplicación, dando una sensación más profesional o seria.