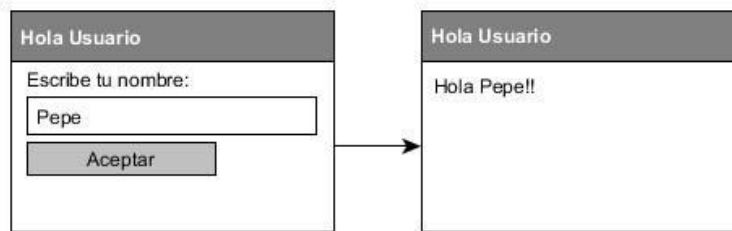


## Actividad 3.8 Navegación entre ventanas

La aplicación consta de dos pantallas, por un lado la pantalla principal que solicitará un nombre al usuario y una segunda pantalla en la que se mostrará un mensaje personalizado para el usuario. Así de sencillo e inútil, pero aprenderemos muchos conceptos básicos, que para empezar no está mal.

Por dibujarlo para entender mejor lo que queremos conseguir, sería algo tan sencillo como lo siguiente:



Android Studio ha creado por nosotros la estructura de carpetas del proyecto y todos los ficheros necesarios de un Hola Mundo básico, es decir, una sola pantalla donde se muestra únicamente un mensaje fijo.

Lo primero que vamos a hacer es diseñar nuestra pantalla principal modificando la que Android Studio nos ha creado por defecto. Aunque ya lo hemos comentado de pasada, recordemos dónde y cómo se define cada pantalla de la aplicación. En Android, el diseño y la lógica de una pantalla están separados en dos ficheros distintos. Por un lado, en el fichero `/src/main/res/layout/activity_main.xml` tendremos el diseño puramente visual de la pantalla definido como **fichero XML** y por otro lado, en el fichero `/src/main/java/com.example.actividad_3_1/MainActivity.kt`, encontraremos el código kotlin que determina la lógica de la pantalla.

Vamos a modificar en primer lugar el aspecto de la ventana principal de la aplicación añadiendo los controles (views) que vemos en el esquema mostrado al principio del apartado. Para ello, vamos a sustituir el contenido del fichero `activity_main.xml` por el siguiente:

```
8 <LinearLayout android:id="@+id/lytContenedor"  
9     android:layout_width="match_parent"  
10    android:layout_height="match_parent"  
11    android:orientation="vertical">  
12  
13    <TextView android:id="@+id/lblNombreModulo"  
14        android:layout_width="wrap_content"  
15        android:layout_height="wrap_content"  
16        android:text="@string/modulo"  
17        android:textColor="#FF0000" />  
18  
19    <TextView android:id="@+id/lblNombre"  
20        android:layout_width="wrap_content"  
21        android:layout_height="wrap_content"  
22        android:text="@string/nombre" />  
23  
24    <EditText android:id="@+id/txtNombre"  
25        android:layout_width="match_parent"  
26        android:layout_height="wrap_content"  
27        android:inputType="text" />  
28  
29    <Button android:id="@+id/btnAceptar"  
30        android:layout_width="wrap_content"  
31        android:layout_height="wrap_content"  
32        android:text="@string/aceptar" />  
33  
34    <TextView android:id="@+id/lblNombreAlumno"  
35        android:layout_width="wrap_content"  
36        android:layout_height="wrap_content"  
37        android:text="@string/alumno"  
38        android:textColor="#0000FF" />  
39  
40 </LinearLayout>
```

Aparecerán algunos errores marcados en rojo, en los valores de los atributos `android:text`. Es normal, lo arreglaremos pronto.

En este XML se definen los elementos visuales que componen la interfaz de nuestra pantalla principal y se especifican todas sus propiedades.

Lo primero que nos encontramos es un elemento **LinearLayout**. Los layout son elementos no visibles que determinan cómo se van a distribuir en el espacio los controles que incluyamos en su interior. En este caso, un **LinearLayout** distribuirá los controles simplemente uno tras otro y en la orientación que indique su propiedad **android:orientation**, que en este caso será "vertical".

Dentro del layout hemos incluido **5 controles**: **3 etiquetas (TextView)**, un **cuadro de texto (EditText)**, y un **botón (Button)**. En todos ellos hemos establecido las siguientes propiedades:

- **android:id**. ID del control, con el que podremos identificarlo más tarde en nuestro código. Vemos que el identificador lo escribimos precedido de "@+id/". Esto tendrá como efecto que al compilarse el proyecto se genere automáticamente una nueva constante en la clase R para dicho control. Así,

por ejemplo, como al cuadro de texto le hemos asignado el ID TxtNombre, podremos más tarde acceder al él desde nuestro código haciendo referencia a la constante R.id.txtNombre.

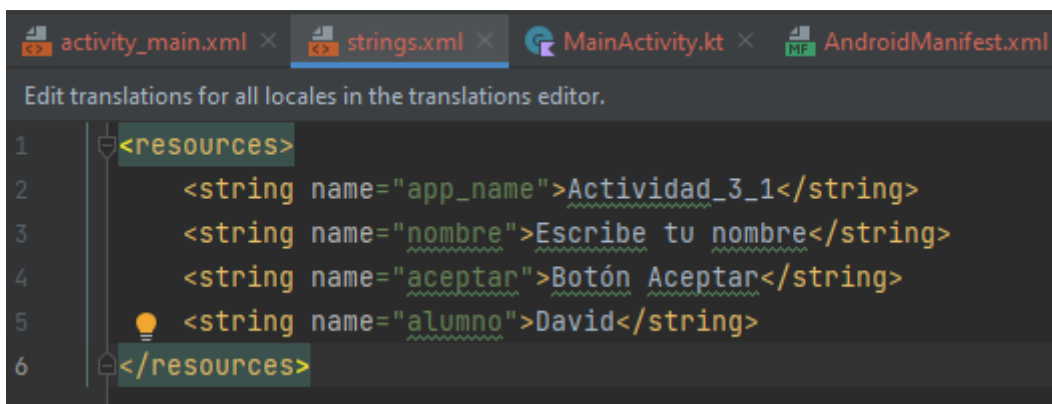
- **android:layout\_height** y **android:layout\_width**. Dimensiones del control con respecto al layout que lo contiene (height=alto, width=ancho). Esta propiedad tomará normalmente los valores "wrap\_content" para indicar que las dimensiones del control se ajustarán al contenido del mismo, o bien "match\_parent" para indicar que el ancho o el alto del control se ajustará al alto o ancho del layout contenedor respectivamente.

Además de estas propiedades comunes a casi todos los controles que utilizaremos, en el cuadro de texto hemos establecido también la propiedad **android:inputType**, que indica qué tipo de contenido va a albergar el control, en este caso será texto normal (valor "text"), aunque podría haber sido una contraseña (valor "textPassword"), un teléfono ("phone"), una fecha ("date"), ....

Por último, en la etiqueta y el botón hemos establecido la propiedad **android:text**, que indica el **texto que aparece en el control**. Y aquí nos vamos a detener un poco, ya que tenemos dos alternativas a la hora de hacer esto. En Android, el texto de un control se puede especificar directamente como valor de la propiedad **android:text**, o bien utilizar alguna de las cadenas de texto definidas en los recursos del proyecto, en cuyo caso indicaremos como valor de la propiedad **android:text** su identificador precedido del prefijo "@string/". Dicho de otra forma, la primera alternativa ha sido indicar directamente el texto como valor de la propiedad en la etiqueta de esta forma:

```
<TextView android:id="@+id/lblNombreModulo"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="DEIN"  
    android:textColor="#FF0000" />
```

Y la segunda alternativa **utilizada**, consistente en definir primero una nueva cadena de texto en el fichero de recursos */src/main/res/values/strings.xml*, por ejemplo con identificador "nombre" y valor "Escribe tu nombre:":



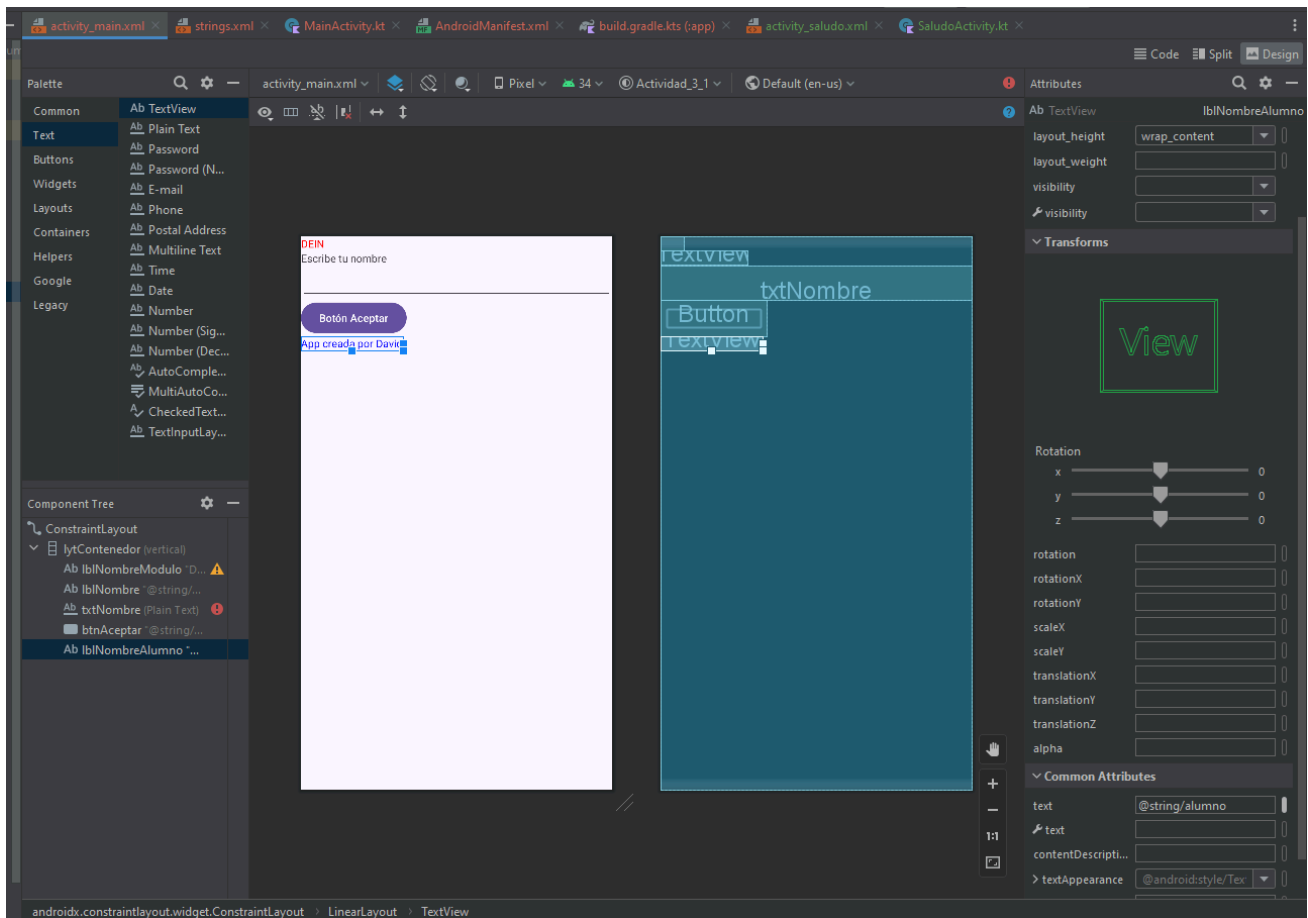
```
1 <resources>  
2     <string name="app_name">Actividad_3_1</string>  
3     <string name="nombre">Escribe tu nombre</string>  
4     <string name="aceptar">Botón Aceptar</string>  
5     <string name="alumno">David</string>  
6 </resources>
```

y posteriormente indicar el identificador de la cadena como valor de la propiedad **android:text**, siempre precedido del prefijo "@string/", de la siguiente forma:

```
<TextView android:id="@+id/lblNombreAlumno"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/alumno"  
    android:textColor="#0000FF" />
```

Esta segunda alternativa nos permite tener perfectamente **localizadas y agrupadas** todas las cadenas de texto utilizadas en la aplicación, lo que nos podría facilitar por ejemplo la traducción de la aplicación a otro idioma. Haremos esto para las dos cadenas de texto utilizadas en el layout, "nombre" y "aceptar". Una vez incluidas ambas cadenas de texto en el fichero *strings.xml* **deberían desaparecer los dos errores** marcados en **rojo** que nos aparecieron antes en la ventana *activity\_main.xml*.

Nos quedará así:



Con esto ya tenemos definida la presentación visual de nuestra ventana principal de la aplicación, veamos ahora la lógica de la misma. Como ya hemos comentado, **la lógica de la aplicación se definirá en ficheros kotlin independientes**. Para la pantalla principal ya tenemos creado un fichero por defecto llamado *MainActivity.kt*. Empecemos por comentar su código por defecto:

```
activity_main.xml x strings.xml x MainActivity.kt x
package com.example.actividad_3_1

import ...

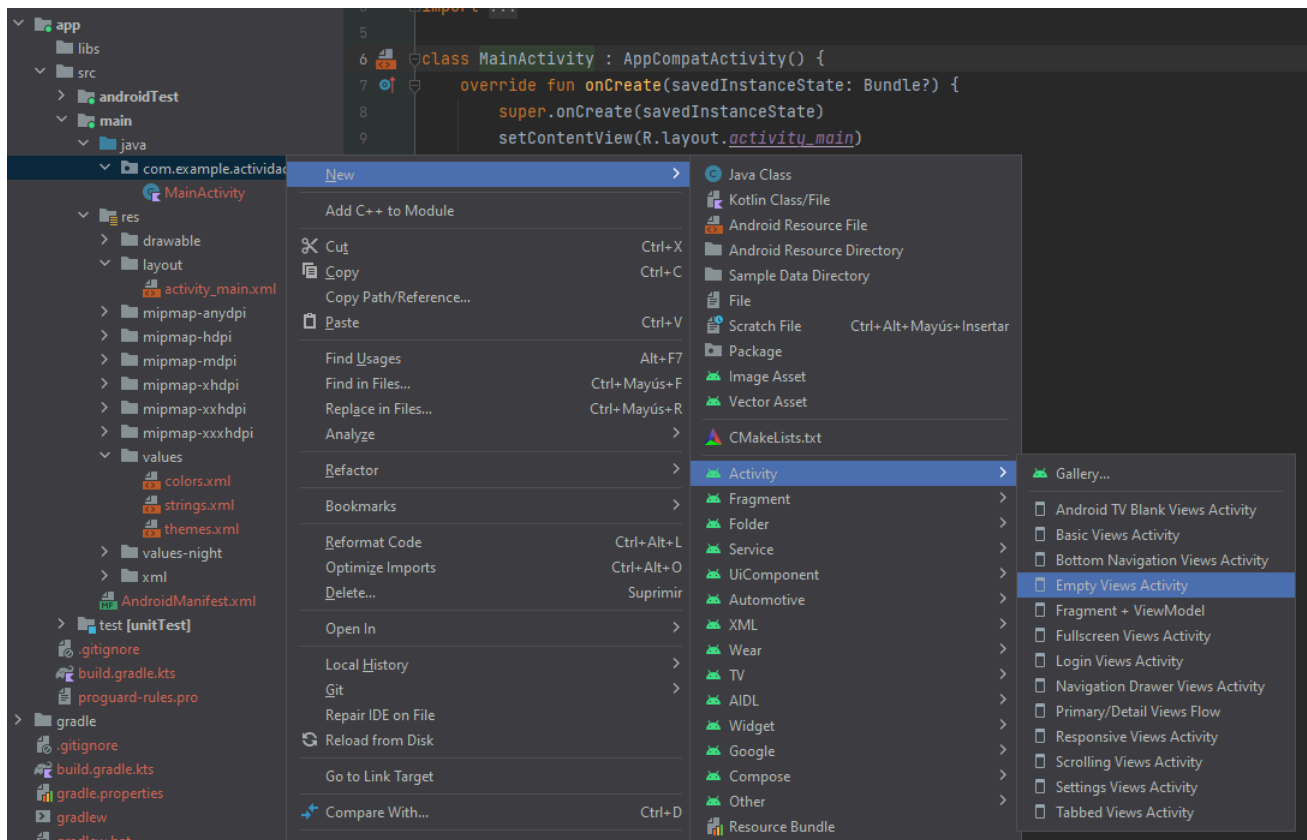
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Como ya vimos en un apartado anterior, las diferentes pantallas de una aplicación Android se definen mediante objetos de tipo **Activity**. Por tanto, lo primero que encontramos en nuestro fichero kotlin es la definición de una nueva **clase MainActivity** que extiende en este caso de un tipo especial de **Activity**

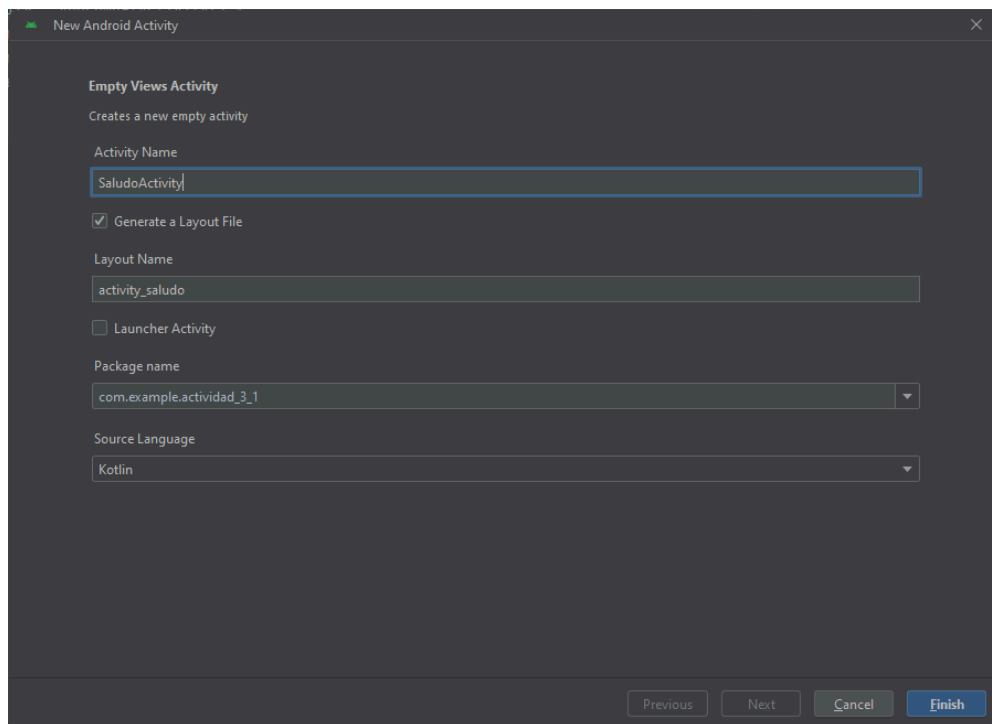
llamado **AppCompatActivity**, que soporta entre otras cosas la utilización de la **Action Bar** en nuestras aplicaciones (la action bar es la barra de título y menú superior que se utiliza en la mayoría de aplicaciones Android). El único método que modificaremos por ahora de esta clase será el **método onCreate()**, llamado cuando se crea por primera vez la actividad. En este método lo único que encontramos en principio, además de la llamada a su implementación en la clase padre, es la llamada al método **setContentView(R.layout.activity\_main)**. Con esta llamada estaremos indicando a Android que debe establecer como interfaz gráfica de esta actividad la definida en el recurso **R.layout.activity\_main**, que no es más que la que hemos especificado en el fichero **/src/main/res/layout/activity\_main.xml**. Una vez más vemos la utilidad de las diferentes constantes de recursos creadas automáticamente en la clase **R** al compilar el proyecto.

Antes de modificar el código de nuestra actividad principal, vamos a crear una nueva actividad para la segunda pantalla de la aplicación análoga a esta primera, a la que llamaremos **SaludoActivity**.

Para ello, pulsaremos el botón derecho sobre la carpeta **/src/main/java/com.example.actividad\_3\_1/** y seleccionaremos la opción de menú **New / Activity / Empty ViewsActivity**:



En el cuadro de diálogo que aparece indicaremos el nombre de la actividad, en nuestro caso **SaludoActivity**, el nombre de su layout XML asociado (Android Studio creará al mismo tiempo tanto el layout XML como la clase kotlin), que llamaremos **activity\_saludo**, y el nombre del paquete de la actividad, donde podemos dejar el valor por defecto.



Pulsaremos Finish y Android Studio creará los nuevos ficheros **SaludoActivity.kt** y **activity\_saludo.xml** en sus carpetas correspondientes.

De igual forma que hicimos con la actividad principal, definiremos en primer lugar la interfaz de la segunda pantalla, abriendo el fichero **activity\_saludo.xml**, y añadiendo esta vez tan sólo un LinearLayout como contenedor y una etiqueta (TextView) para mostrar el mensaje personalizado al usuario.

Para esta segunda pantalla el código que incluiríamos sería el siguiente:

```
9      <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
10          android:id="@+id/lyyContenedorSaludo"
11          android:layout_width="match_parent"
12          android:layout_height="match_parent"
13          android:orientation="vertical" >
14
15          <TextView android:id="@+id/txtSaludo"
16              android:layout_width="wrap_content"
17              android:layout_height="wrap_content"
18              android:text="" />
19
20          <TextView android:id="@+id/lblNombreAlumno"
21              android:layout_width="wrap_content"
22              android:layout_height="wrap_content"
23              android:text="@string/alumno"
24              android:textColor="#0000FF" />
25
26      </LinearLayout>
```

Por su parte, si revisamos ahora el código de la clase SaludoActivity veremos que es análogo a la actividad principal:

```
package com.example.actividad_3_1

import ...

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

Por ahora, el código incluido en estas clases lo único que hace es generar la interfaz de la actividad. A partir de aquí nosotros tendremos que incluir el resto de la **lógica de la aplicación**.

Y vamos a empezar con la **actividad principal MainActivity.kt**, obteniendo una referencia a los diferentes controles de la interfaz que necesitemos manipular, en nuestro caso sólo el **cuadro de texto** y el **botón**. Para ello definiremos ambas referencias como **atributos de la clase** y para obtenerlas utilizaremos el método **findViewById()** indicando el ID de cada control. Todo esto lo haremos dentro del **método onCreate()** de la **clase MainActivity**, justo a continuación de la llamada a **setContentView()** que ya comentamos:

```
activity_main.xml x strings.xml x MainActivity.kt x AndroidManifest.xml x build.gradle
1 package com.example.actividad_3_1
2
3 import androidx.appcompat.app.AppCompatActivity
4 import android.os.Bundle
5 import android.widget.Button
6 import android.widget.EditText
7 import android.content.Intent
8 import android.widget.TextView
9
10 class MainActivity : AppCompatActivity() {
11     private lateinit var txtNombre: EditText
12     private lateinit var btnAceptar: Button
13
14     override fun onCreate(savedInstanceState: Bundle?) {
15         super.onCreate(savedInstanceState)
16         setContentView(R.layout.activity_main)
```



Como vemos, hemos añadido también varios import adicionales (los de las clases Button, EditText e Intent) para tener acceso a todas las clases utilizadas.

Una vez tenemos acceso a los diferentes controles, ya sólo nos queda **implementar las acciones a tomar** cuando pulsemos el botón de la pantalla. Para ello, continuando el código anterior, y siempre dentro del método onCreate(), implementaremos el evento **setOnClickListener** de dicho botón. Este botón tendrá que ocuparse de abrir la actividad SaludoActivity pasándole toda la información necesaria. Veamos cómo:

```
1 package com.example.actividad_3_1
2
3 import androidx.appcompat.app.AppCompatActivity
4 import android.os.Bundle
5 import android.widget.Button
6 import android.widget.EditText
7 import android.content.Intent
8 import android.widget.TextView
9
10 class MainActivity : AppCompatActivity() {
11     private lateinit var txtNombre: EditText
12     private lateinit var btnAceptar: Button
13
14     override fun onCreate(savedInstanceState: Bundle?) {
15         super.onCreate(savedInstanceState)
16         setContentView(R.layout.activity_main)
17
18         // Obtenemos una referencia a los controles de la interfaz
19         txtNombre = findViewById(R.id.txtNombre)
20         btnAceptar = findViewById(R.id.btnAceptar)
21
22         // Implementamos el evento click del botón
23         btnAceptar.setOnClickListener { it: View!
24
25             // Creamos el Intent
26             val intent = Intent(packageContext, this@MainActivity, SaludoActivity::class.java)
27
28             // Creamos la información a pasar entre actividades
29             val b = Bundle()
30             b.putString("NOMBRE", txtNombre.text.toString())
31
32             // Añadimos la información al intent
33             intent.putExtras(b)
34
35             // Iniciamos la nueva actividad
36             startActivity(intent)
37         }
38     }
39 }
```

Como ya indicamos en el apartado anterior, la comunicación entre los distintos componentes y aplicaciones en Android se realiza mediante **intents**, por lo que el primer paso es crear un objeto de

este tipo. Existen varias variantes del constructor de la **clase Intent**, cada una de ellas dirigida a unas determinadas acciones. En nuestro caso particular vamos a utilizar el intent para iniciar una actividad desde otra actividad de la misma aplicación, para lo que pasaremos a su constructor una referencia a la propia actividad llamada (`this@MainActivity`), y la clase de la actividad llamada (`SaludoActivity::class.java`).

```
// Creamos el Intent
val intent = Intent(this@MainActivity, SaludoActivity::class.java)
```

Si quisiéramos tan sólo mostrar una nueva actividad ya tan sólo nos quedaría llamar a **startActivity(intent)** pasándole como parámetro el intent creado. Pero en nuestro ejemplo queremos también pasarle cierta información a la actividad llamada, concretamente el **nombre que introduzca el usuario** en el cuadro de texto de la pantalla principal. Para hacer esto **creamos un objeto Bundle**, que puede contener una lista de clave-valor con toda la información a pasar entre actividades. En nuestro caso sólo añadimos un dato de tipo String mediante el método **putString(clave, valor)**. Como clave para nuestro dato **"NOMBRE"**. Por su parte, el valor de esta clave lo obtenemos consultando el contenido del cuadro de texto de la actividad principal, y convirtiendo este contenido a texto mediante `toString()`.

```
// Creamos la información a pasar entre actividades
val b = Bundle()
b.putString("NOMBRE", txtNombre.text.toString())
```

Tras esto **añadiremos la información al intent** mediante el método `putExtras()`. Si necesitáramos pasar más datos entre una actividad y otra no tendríamos más que repetir estos pasos para todos los parámetros necesarios.

```
// Añadimos la información al intent
intent.putExtras(b)
```

Con esto hemos finalizado ya la actividad principal de la aplicación, por lo que pasaremos ya a la **secundaria, SaludoActivity.kt**:

Comenzaremos de forma análoga a la anterior, **ampliando el método onCreate()** obteniendo las referencias a los objetos que manipularemos, esta vez sólo la etiqueta de texto. Tras esto viene lo más interesante, debemos **recuperar la información** pasada desde la actividad principal y asignarla como **texto de la etiqueta**.

Todo el código junto quedaría así:

```
1 package com.example.actividad_3_1
2
3 import android.os.Bundle
4 import android.widget.TextView
5 import androidx.appcompat.app.AppCompatActivity
6
7 new *
8 class SaludoActivity : AppCompatActivity() {
9     private lateinit var txtSaludo: TextView
10
11     new *
12     override fun onCreate(savedInstanceState: Bundle?) {
13         super.onCreate(savedInstanceState)
14         setContentView(R.layout.activity_saludo)
15
16         // Localizar los controles
17         txtSaludo = findViewById(R.id.txtSaludo)
18
19         // Recuperamos la información pasada en el intent
20         val bundle = intent.extras
21
22         // Construimos el mensaje a mostrar
23         txtSaludo.text = "Hola ${bundle?.getString( key: "NOMBRE")}"
24     }
25 }
```

Viendo por partes, accederemos en primer lugar al **intent** que ha originado la actividad actual:

```
// Recuperamos la información pasada en el intent
val bundle = intent.extras
```

Y hecho esto tan sólo nos queda construir el texto de la etiqueta mediante su método `setText(texto)` y recuperando el valor de nuestra clave almacenada en el objeto `Bundle` mediante `getString(clave)`.

```
// Construimos el mensaje a mostrar
txtSaludo.text = "Hola ${bundle?.getString( key: "NOMBRE")}"
```

Con esto hemos concluido la lógica de las dos pantallas de nuestra aplicación y tan sólo nos queda un paso importante para finalizar nuestro desarrollo. Como ya indicamos en un apartado anterior, toda aplicación Android utiliza un fichero especial en **formato XML (AndroidManifest.xml)** para definir, entre otras cosas, los diferentes elementos que la componen. Por tanto, todas las actividades de nuestra aplicación deben quedar convenientemente definidas en este fichero. En este caso, Android

Studio se debe haber ocupado por nosotros de definir ambas actividades en el fichero, pero lo revisaremos para así echar un vistazo al contenido.

Si abrimos el fichero **AndroidManifest.xml** veremos que contiene un elemento principal `<application>` que debe incluir varios elementos `<activity>`, uno por cada actividad incluida en nuestra aplicación. En este caso, comprobamos como efectivamente Android Studio ya se ha ocupado de esto por nosotros, aunque este fichero sí podríamos modificarlo a mano para hacer ajustes si fuera necesario:

```
activity_main.xml x strings.xml x MainActivity.kt x AndroidManifest.xml x build.gradle.kts (:app)
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     xmlns:tools="http://schemas.android.com/tools">
4
5     <application
6         android:allowBackup="true"
7         android:dataExtractionRules="@xml/data_extraction_rules"
8         android:fullBackupContent="@xml/backup_rules"
9         android:icon="@mipmap/ic_launcher"
10        android:label="Actividad_3_1"
11        android:roundIcon="@mipmap/ic_launcher_round"
12        android:supportsRtl="true"
13        android:theme="@style/Theme.Actividad_3_1"
14        tools:targetApi="34">
15        <activity
16            android:name=".SaludoActivity"
17            android:exported="false" />
18        <activity
19            android:name=".MainActivity"
20            android:exported="true">
21            <intent-filter>
22                <action android:name="android.intent.action.MAIN" />
23
24                <category android:name="android.intent.category.LAUNCHER" />
25            </intent-filter>
26        </activity>
27    </application>
28
29 </manifest>
```

Podemos ver como para cada actividad se indica entre otras cosas el nombre de su clase asociada como valor del atributo `android:name`, más adelante veremos qué opciones adicionales podemos especificar.

El último elemento que revisaremos de nuestro proyecto, aunque tampoco tendremos que modificarlo por ahora, será el fichero **build.gradle**. Pueden existir varios ficheros llamados así en nuestra estructura de carpetas, a distintos niveles, pero normalmente siempre accederemos al que está al nivel más interno, en nuestro caso el que está dentro del módulo "app". Veamos qué contiene:

```
android { this: BaseAppModuleExtension
    namespace = "com.example.actividad_3_1"
    compileSdk = 34

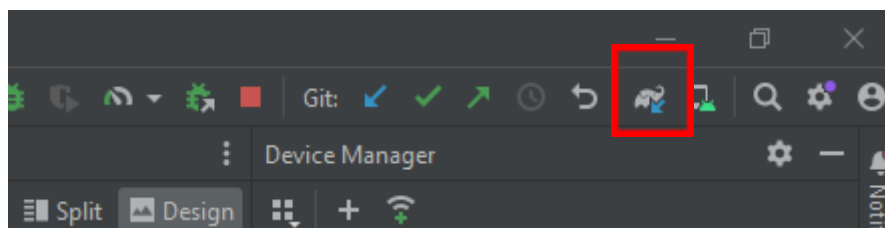
    defaultConfig { this: ApplicationDefaultConfig
        applicationId = "com.example.actividad_3_1"
        minSdk = 28
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
    }
}
```

Gradle es el sistema de compilación y construcción que tiene Google para Android Studio. Pero no es un sistema específico de Android, sino que puede utilizarse con otros lenguajes/plataformas. En él definiremos varias opciones específicas de Android, como las **versiones de la API mínima** (minSdk), **API objetivo** (targetSdk), la **versión tanto interna** (versionCode) como **visible** (versionName) de la aplicación,

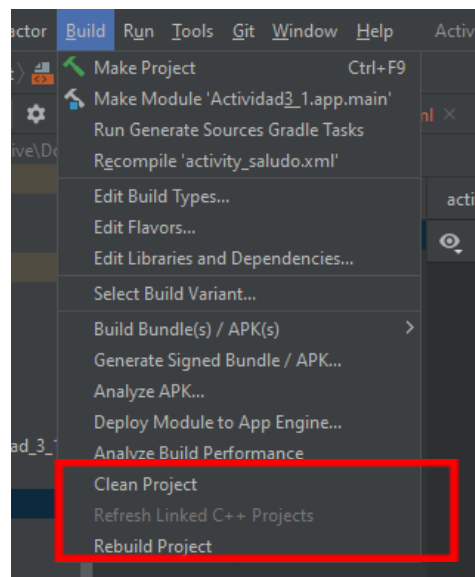
Otro elemento llamado "**dependencias**" también es importante y nos servirá entre otras cosas para definir las librerías externas que utilizaremos en la aplicación.

Llegados aquí, y si todo ha ido bien, deberíamos poder ejecutar el proyecto sin errores y probar nuestra aplicación en el emulador, pero para ello tendremos que definir primero uno. Antes vamos a realizar las acciones "**Sync Project with Gradle Files**" (Sincronizar Proyecto con Archivos Gradle), "**Clean Project**" (Limpiar Proyecto) y "**Rebuild Project**" (Reconstruir Proyecto) que son operaciones en Android Studio que nos ayudan a mantener la coherencia y la integridad del proyecto, especialmente en lo que respecta a la construcción y la gestión de dependencias, ayudándonos a determinar los errores que podamos tener en nuestra aplicación:



### Sync Project with Gradle Files

**Sync Project with Gradle Files** (Sincronizar Proyecto con Archivos Gradle) lo que hace es actualizar el proyecto para que coincida con la configuración definida en los archivos Gradle. Sincroniza las dependencias, configuraciones de compilación y otros aspectos del proyecto según lo definido en los archivos Gradle. Deberías utilizarlo después de realizar cambios en los archivos Gradle, como agregar o actualizar dependencias, para asegurarte de que los cambios se reflejen correctamente en el proyecto.

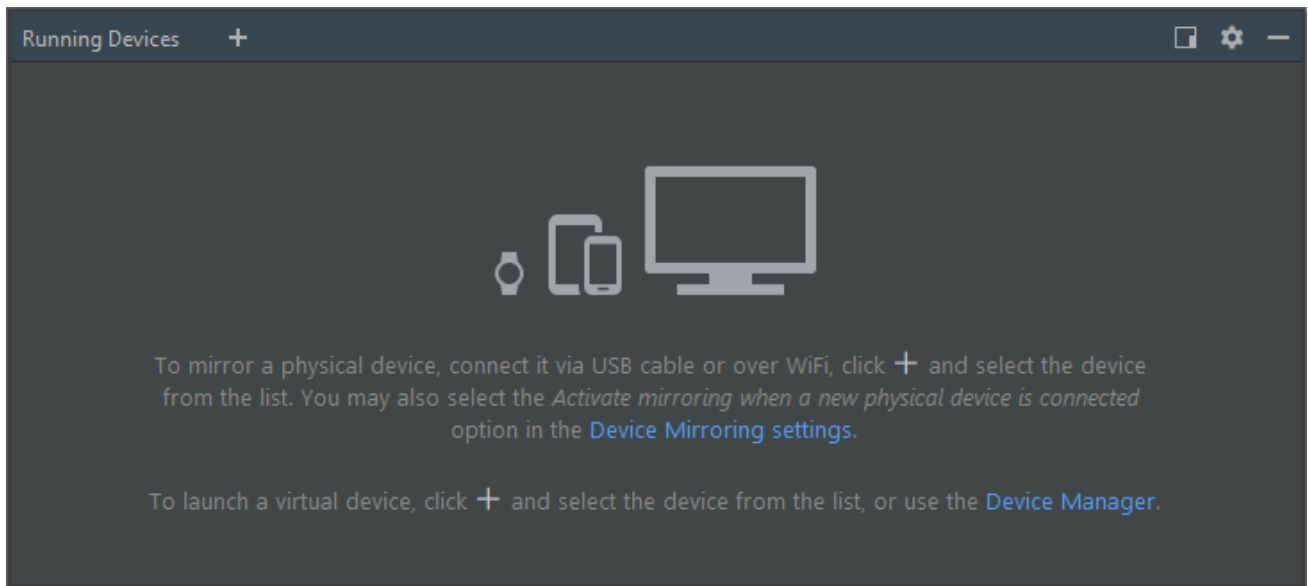


### Clean y Rebuild Project

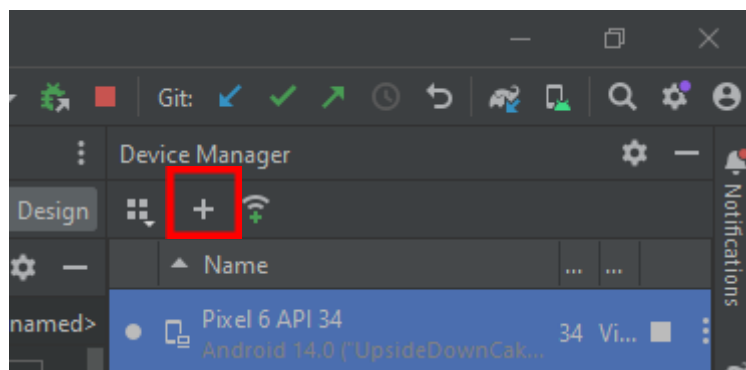
**Clean Project** (Limpiar Proyecto), limpia el proyecto eliminando todos los archivos compilados y los artefactos generados en la compilación anterior. Se utiliza cuando hay problemas en la compilación o cuando se quieren eliminar los archivos generados para comenzar desde cero. Puede ser útil después de realizar cambios significativos en el proyecto.

**Rebuild Project** (Reconstruir Proyecto), realiza la acción de "Clean Project" y luego realiza una compilación completa desde cero. Elimina todos los archivos compilados y regenera todo, incluyendo los archivos de clase, recursos y otros artefactos de construcción. Se utiliza cuando los problemas persisten incluso después de realizar la "Clean Project". Puede ser beneficioso cuando se sospecha que hay problemas profundos en la configuración del proyecto.

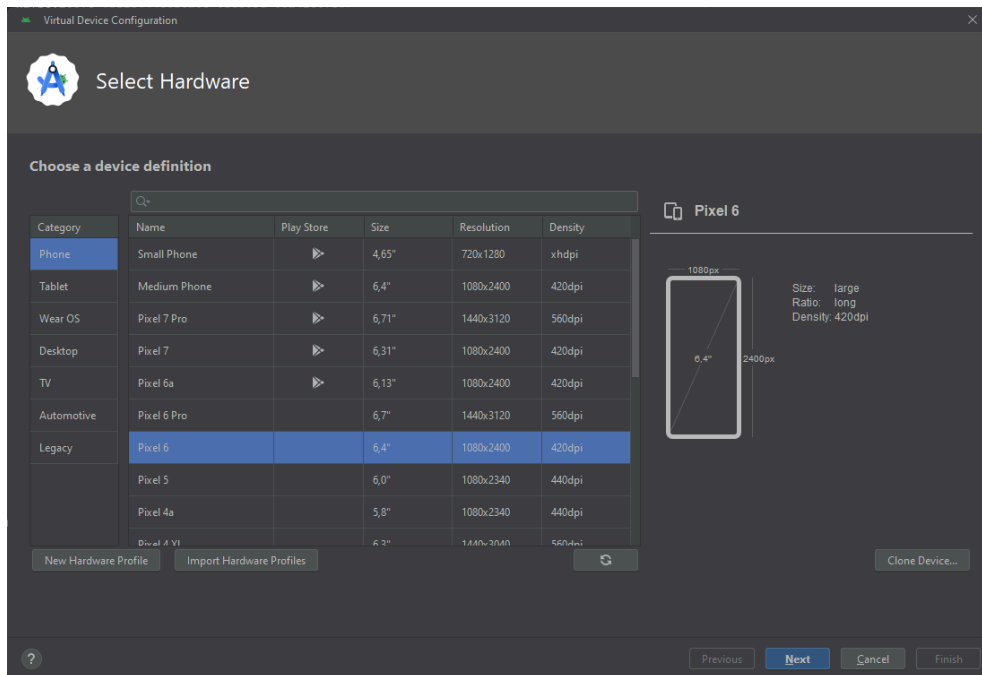
Si todos estos pasos se han ejecutado sin errores, guarda el proyecto y ahora vamos a **probar la aplicación Android** en nuestro PC sin tener que recurrir a un dispositivo físico, por lo que tenemos que definir lo que se denominan **AVD (Android Virtual Device)**. Para crear un AVD seleccionaremos el menú Tools / Android / AVD Manager. Si es la primera vez que accedemos a esta herramienta veremos la pantalla siguiente:



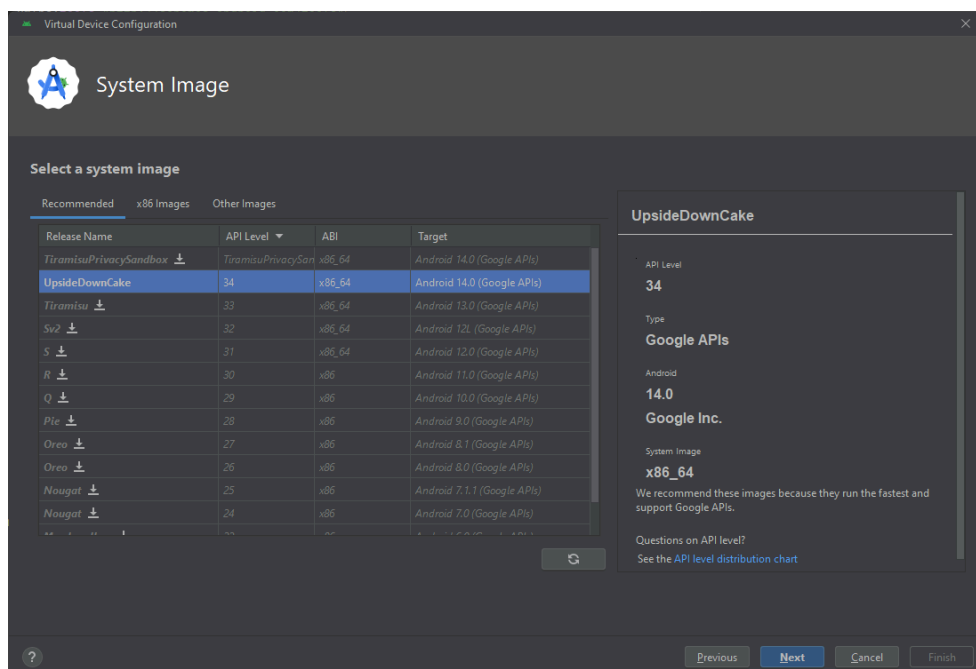
Si no te ha aparecido esta pantalla, puedes hacerlo desde aquí:



Pulsando el botón central "Create a virtual device" de la primera pantalla o dándole al "+" en el "Device Manager" accederemos al asistente para crear un AVD. En el primer paso tendremos que seleccionar a la izquierda qué tipo de dispositivo queremos que "simule" nuestro AVD (teléfono, tablet, reloj, ...) y el tamaño, resolución, y densidad de píxeles de su pantalla. En mi caso seleccionaré por ejemplo las características de un Pixel 6 y pasaremos al siguiente paso pulsando "Next".

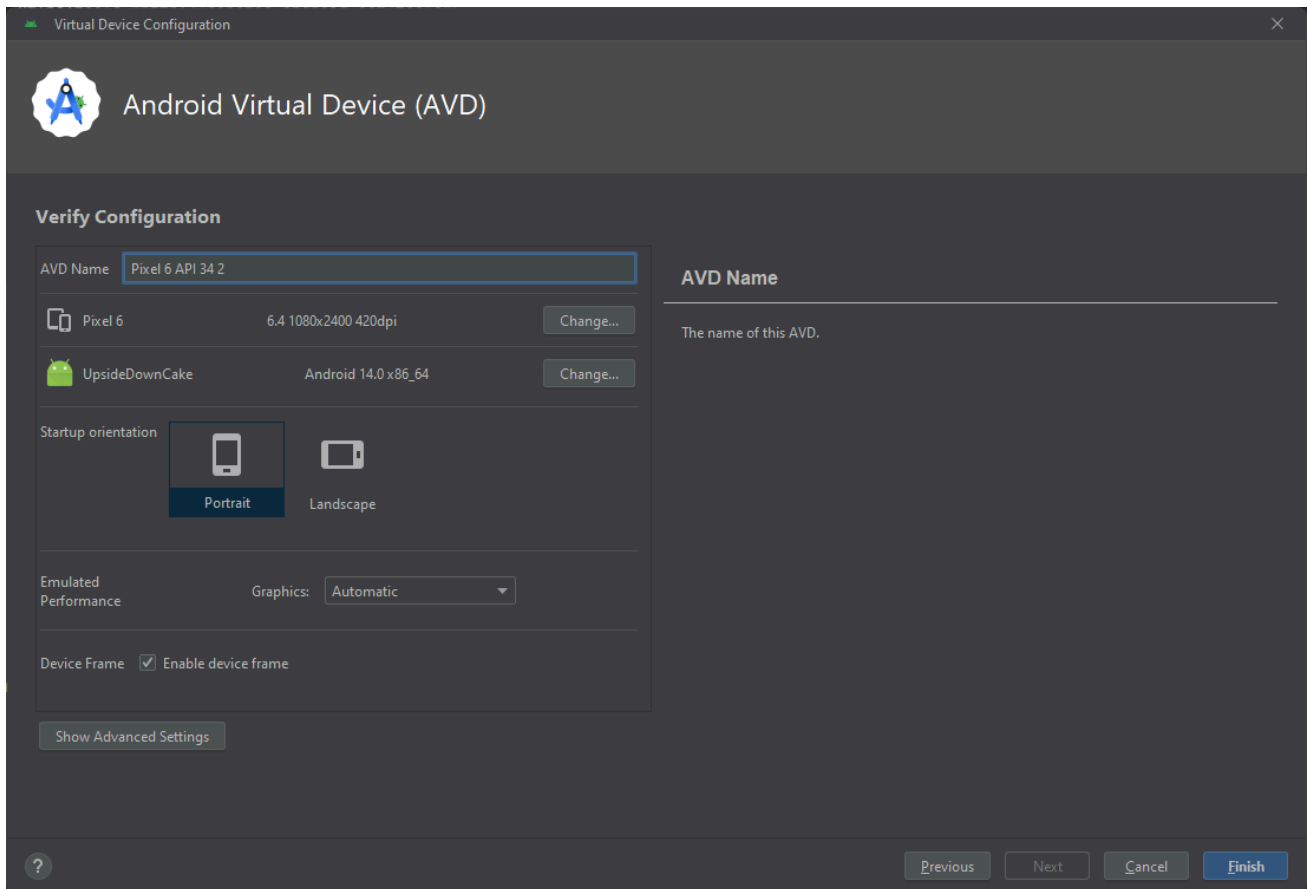


En la siguiente pantalla seleccionaremos la versión de Android que utilizará el AVD. Aparecerán directamente disponibles las que instalamos desde el SDK Manager al instalar el entorno, aunque tenemos la posibilidad de descargar e instalar nuevas versiones desde esta misma pantalla. En mi caso utilizaré la última versión de Android (UpsideDownCake) (API Level 34) para este primer AVD (podemos crear tantos como queramos para probar nuestras aplicaciones sobre distintas condiciones).

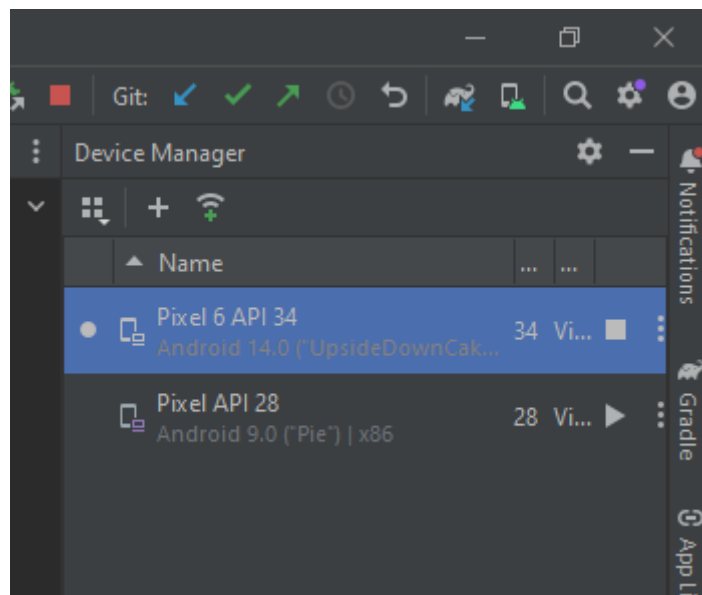


En el siguiente paso del asistente podremos configurar algunas características más del AVD, si simulará tener cámara frontal y/o trasera, teclado físico, ... Recomendando pulsar el botón "Show Advanced Settings" para ver todas las opciones disponibles. Si quieres puedes ajustar cualquiera de estos parámetros, pero por el momento os recomiendo dejar todas las opciones por defecto.



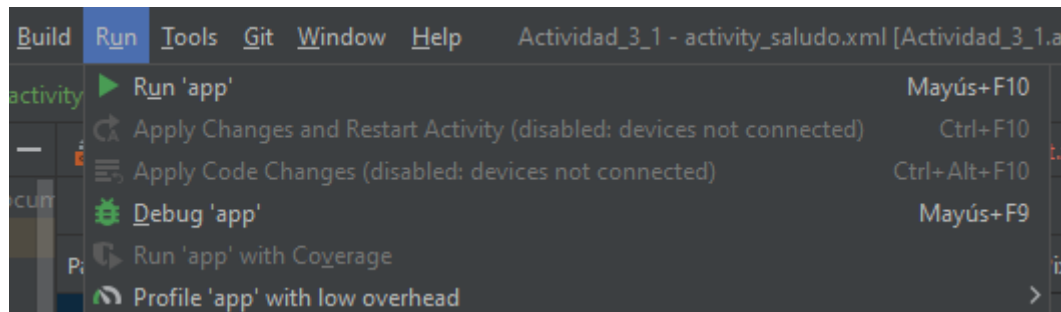


Tras pulsar el botón Finish tendremos ya configurado nuestro AVD, por lo que podremos comenzar a probar nuestras aplicaciones sobre él. En el Device Manager veremos los dispositivos que tenemos:

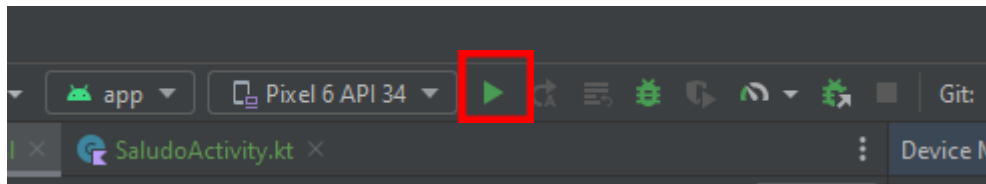


Para ejecutar la App en el dispositivo que hemos creado podemos hacerlo de varias opciones:

1. Pulsaremos simplemente el menú **Run / ejecutar Run 'app'** (o la tecla rápida Mayús+F10).  
Android Studio.



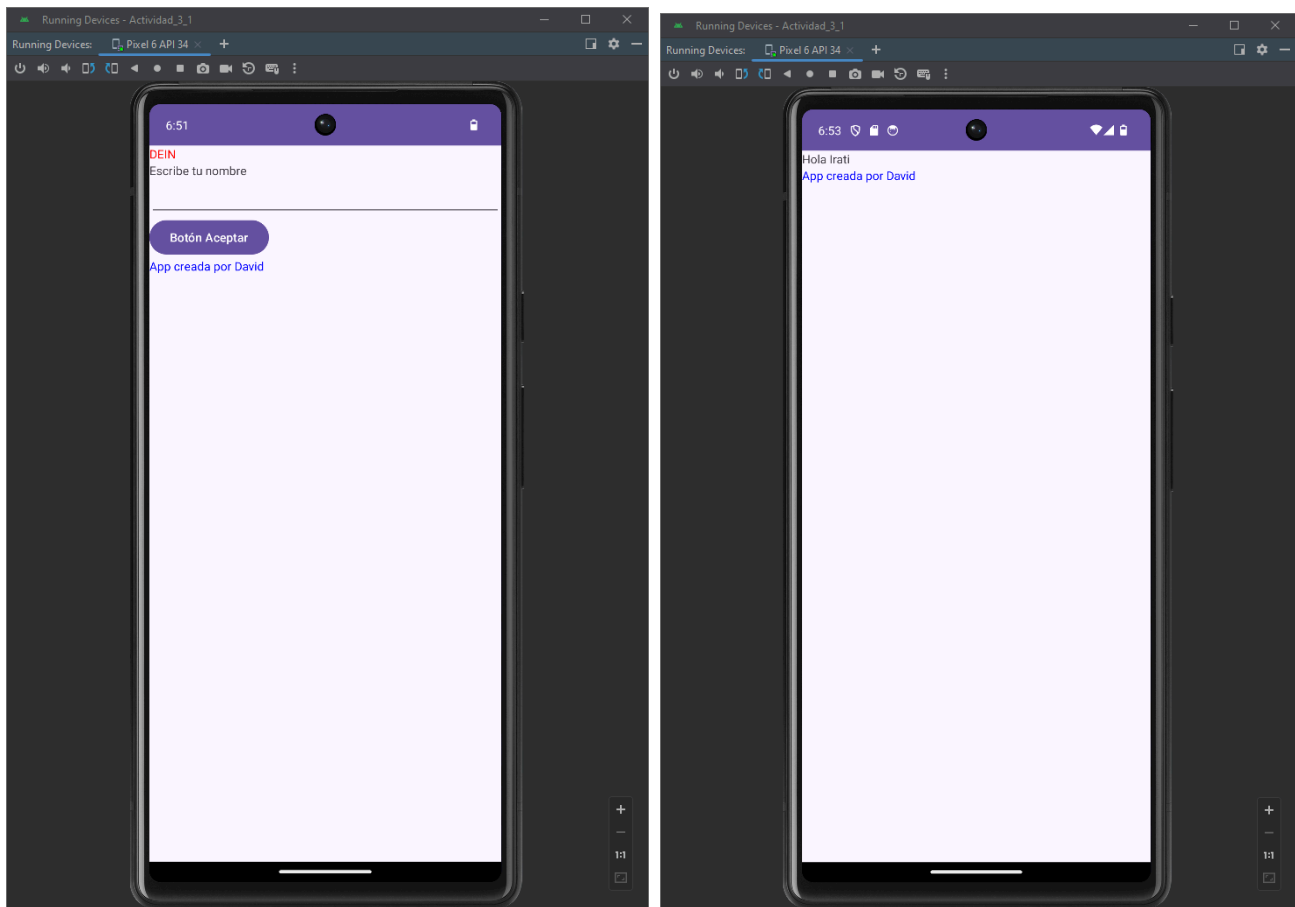
2. Ejecutarlo desde la ventana de navegación de la parte superior izquierda:



Donde en caso de tener más dispositivos, podemos seleccionar el que queramos en ese momento. Le damos al botón de **“Run App”**

Si todo va bien, tras una pequeña (o no tan pequeña) espera aparecerá el emulador de Android y se iniciará automáticamente nuestra aplicación (si se inicia el emulador pero no se ejecuta automáticamente la aplicación podemos volver a ejecutarla desde Android Studio, mediante el menú Run, sin cerrar el emulador ya abierto).

Prueba a escribir un nombre y pulsar el botón "Aceptar" para comprobar si el funcionamiento es el correcto.



Guarda el proyecto. Exporta todo a “to Zip file,” y genera el APK