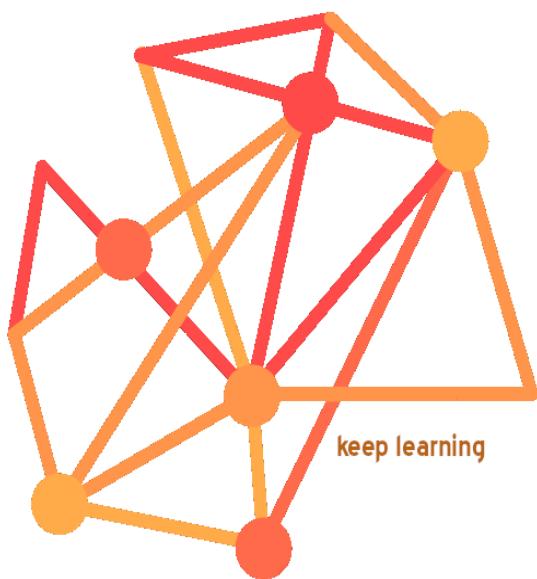


Android con Java



Juan Carlos Pérez Rodríguez

Sumario

Introducción.....	4
Instalación del entorno de desarrollo.....	4
Estructura del proyecto.....	4
Interfaz de usuario.....	5
Hacer zoom con CTRL+rueda ratón:.....	6
Sistema de compilación de Gradle.....	6
Creación de un dispositivo virtual Android (AVD).....	7
Crear estructura proyecto.....	8
Ficheros y carpetas de un proyecto Android.....	10
AndroidManifest.xml.....	11
Componentes de una aplicación.....	13
Ciclo de vida de una actividad.....	14
Toast.....	15
Creando Layouts.....	17
Atributos De Un Layout.....	19
Viewgroups.....	20
Frame Layout.....	21
Linear Layout.....	24
Table Layout.....	27
Constraint Layout.....	28
Cargando diferentes idiomas.....	30
Creando código.....	32
Activities.....	33
Escuchar y manejar eventos onClick.....	34
Intents.....	38
Introducción.....	38
Intents disponibles en Android.....	38
Intent Explícito. Lanzar una activity de nuestra app.....	40
Paso de información con el Intent.....	41
Mediante los putExtra() (clave,valor) de tipos nativos del intent.....	41
Mediante putExtra de cualquier objeto (Parcelable) y un helper.....	42
Paso de información al recrear por girar la pantalla.....	44
Uso de Databinding en lugar de view.findViewById.....	46
Uso de fragments.....	48
Uso de Databinding con Fragment.....	51
Uso de ViewModels.....	52
Tomar un ViewModel desde un Fragment.....	54
Compartir un ViewModel entre Fragments.....	55
Navegación.....	55
RecyclerView.....	58
Cargar imágenes en un imageview para el recycler. Glide.....	61
Drawer Navigation.....	63
Bottom Tab Navigation anidada en Drawer Navigation.....	64

Persistencia.....	66
Acceso a ficheros.....	66
Ficheros en memoria interna.....	66
Room.....	68
Estructura de paquetes con room (buenas prácticas).....	68
Clase para la creación/mantenimiento de la DDBB.....	69
Creación de las Entities.....	70
Relaciones con foreign key.....	71
Creación de los DAO.....	73
Uso de Repositorio.....	74
Uso de AsyncTask en consultas asíncronas de inserción y modificación.....	76
Ejecutando los métodos del repositorio en AndroidViewModel.....	78
Accediendo a un servicio REST con Retrofit.....	80
Repositorio accediendo a API y en caso de fallo a DDBB Local.....	87

Introducción

Android Studio es el entorno de desarrollo integrado oficial para la plataforma Android desde 2014. Reemplazó a Eclipse como el IDE oficial para el desarrollo de aplicaciones para Android. Está basado en el software IntelliJ IDEA de JetBrains y ha sido publicado de forma gratuita a través de la Licencia Apache 2.0. Las aplicaciones android nativas que desarrollamos con Android Studio pueden ser tanto con Java como con Kotlin

Instalación del entorno de desarrollo

La instalación está documentada en: <https://developer.android.com/studio/install?hl=es-419>

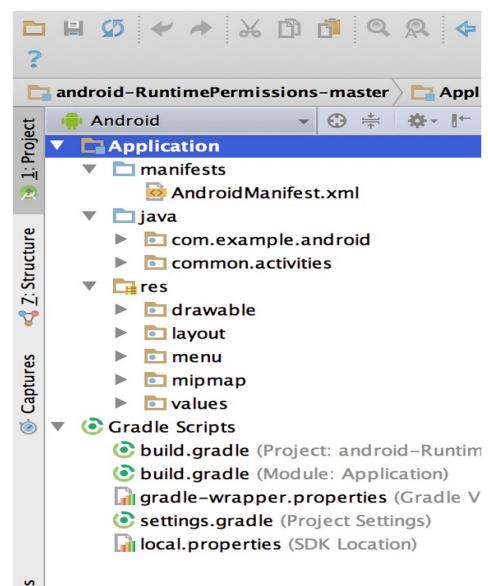
En linux no precisa permisos de administrador. Basta con descargar el comprimido, descomprimirlo y luego ejecutar el fichero: studio.sh (está dentro de android-studio/bin)

Estructura del proyecto

En el ejemplo. La vista se muestra organizada en módulos

Todos los archivos de compilación son visibles en el nivel superior de Secuencias de comando de Gradle y cada módulo de la aplicación contiene las siguientes carpetas:

- **manifests:** contiene el archivo **AndroidManifest.xml**. (definimos permisos, cuál es la primera pantalla que se muestra de nuestra app, etc)
- **java:** contiene los archivos de código fuente de Java, incluido el código de prueba JUnit.



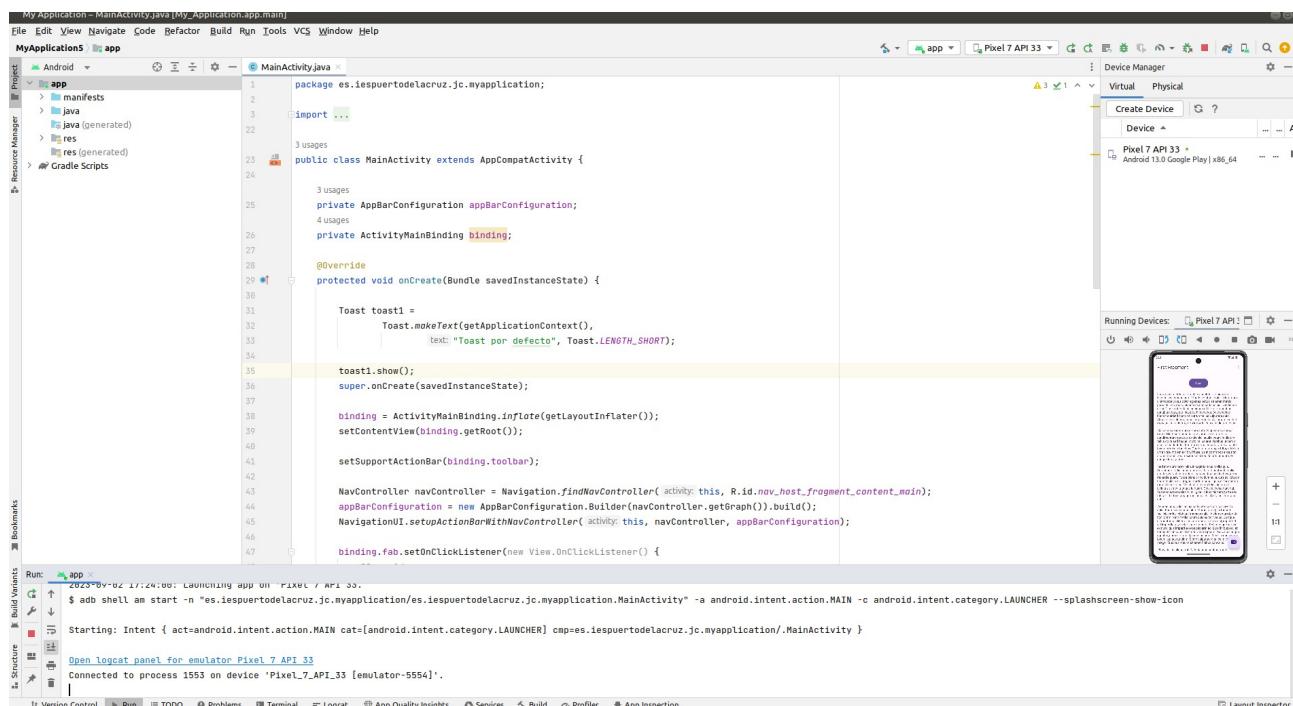
- **res:** Contiene todos los recursos, como diseños XML, cadenas de IU e imágenes de mapa de bits.

La estructura del proyecto para Android en el disco es un poco diferente. Siempre podemos ubicarnos encima de un archivo, botón derecho y elegir el explorar de archivos del sistema operativo para verla

Interfaz de usuario

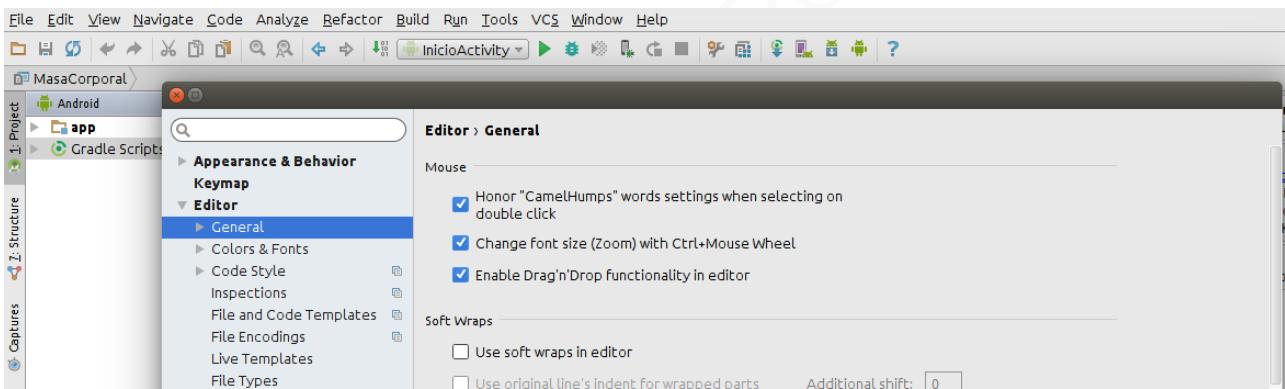
La ventana principal de Android Studio consta de varias áreas lógicas:

1. La barra de herramientas: En la parte superior de la pantalla. Permite realizar la mayor parte de las acciones. Como iniciar y hacer debug al proyecto
2. La barra de navegación: En el lateral izquierdo. Ayuda a explorar el proyecto y abrir archivos para editar.
3. La ventana del editor: En el centro de la pantalla. Es el área donde se crea y modificar código.
4. En la barra de estado: En la parte de abajo de la pantalla. Se muestra el estado de tu proyecto y el IDE, además de advertencias o mensajes.
5. Device Manager: En el lateral derecho. Permite ver e interactuar con los dispositivos virtuales o reales donde lancemos/probemos nuestras aplicaciones



Hacer zoom con CTRL+rueda ratón:

File- → Settings- → Editor- → General- → Change font size(Zoom) with Ctrl+Mouse Wheel



Otra opción es en esa ventana pero donde dice Keymap elegir increase Font Size y decrease font size poniendo las combinaciones de teclas que se quiera

Sistema de compilación de Gradle

Android Studio usa Gradle como base del sistema de compilación, y proporciona más características específicas de Android a través del Complemento de Android para Gradle. Este sistema de compilación se ejecuta en una herramienta integrada desde el menú de Android Studio, y lo hace independientemente de la línea de comandos. Los archivos de compilación de Android Studio se denominan **build.gradle**. Cada proyecto tiene un archivo de compilación de nivel superior para todo el proyecto y archivos de compilación de nivel de módulo independientes para cada módulo.

Creación de un dispositivo virtual Android (AVD)

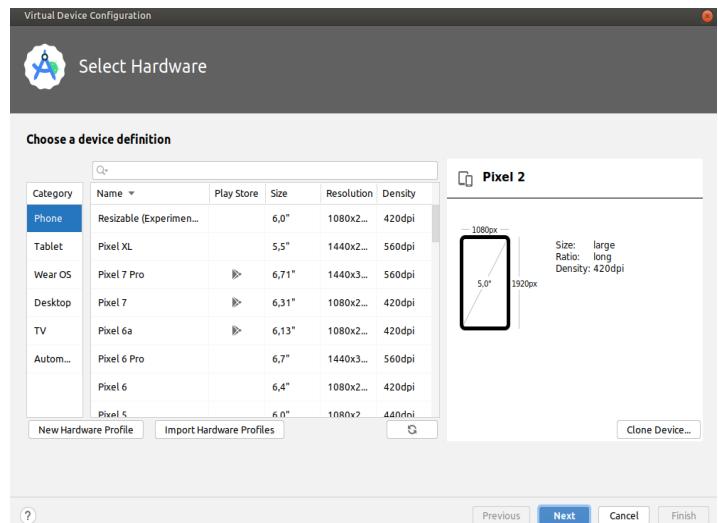
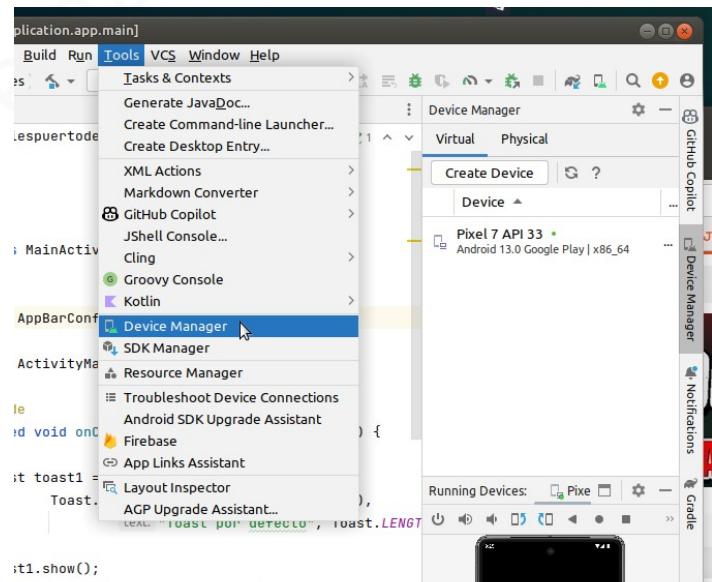
Un dispositivo virtual Android (AVD) te va a permitir emular en tu ordenador diferentes tipos de dispositivos basados en Android. De esta forma podrás probar tus aplicaciones en una gran variedad de teléfonos, tabletas, relojes o TV con cualquier versión de Android, tamaño de pantalla o tipo de entrada.

Hay un panel lateral que está preparado ya. Pero podemos abrirlo mediante: tools → device manager

Encontraremos allí un botón: “create device” que abre un wizard para elegir el dispositivo a emular

El procedimiento es sencillo, lo más importante es cuándo llegamos a la ventana que se elige la imagen del sistema (observar que si se elige un sistema antiguo no se podrá ejecutar con las últimas novedades de android)

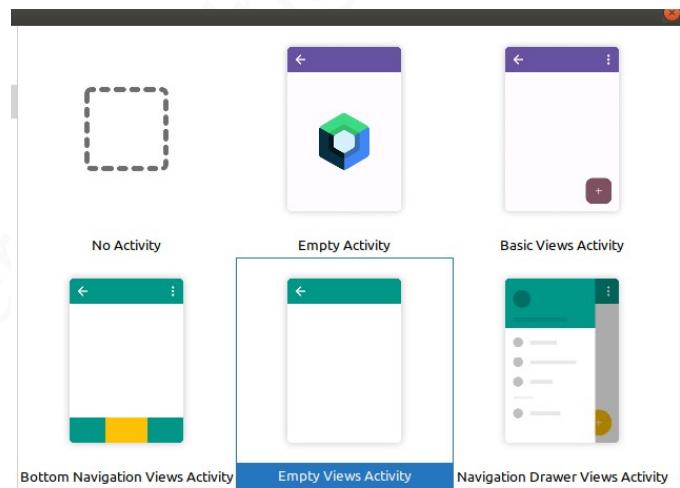
En la última ventana. Se nos mostrará un resumen con las opciones seleccionadas, además podremos seleccionar el tamaño inicial del AVD, si queremos usar el coprocesador gráfico (GPU) de nuestro ordenador o si queremos guardar una imagen congelada del emulador para que arranque más rápido las próximas veces.



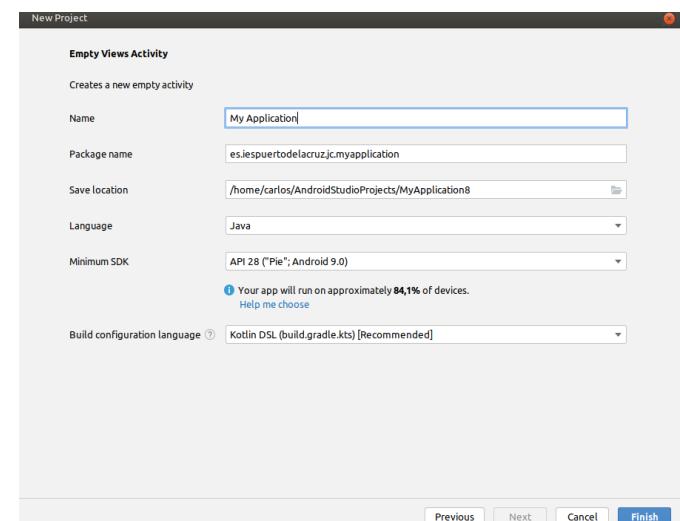
Crear estructura proyecto

Al pulsar para crear un nuevo proyecto se nos oferta que elijamos como será la pantalla inicial del proyecto:

Elegiremos “empty views activity”

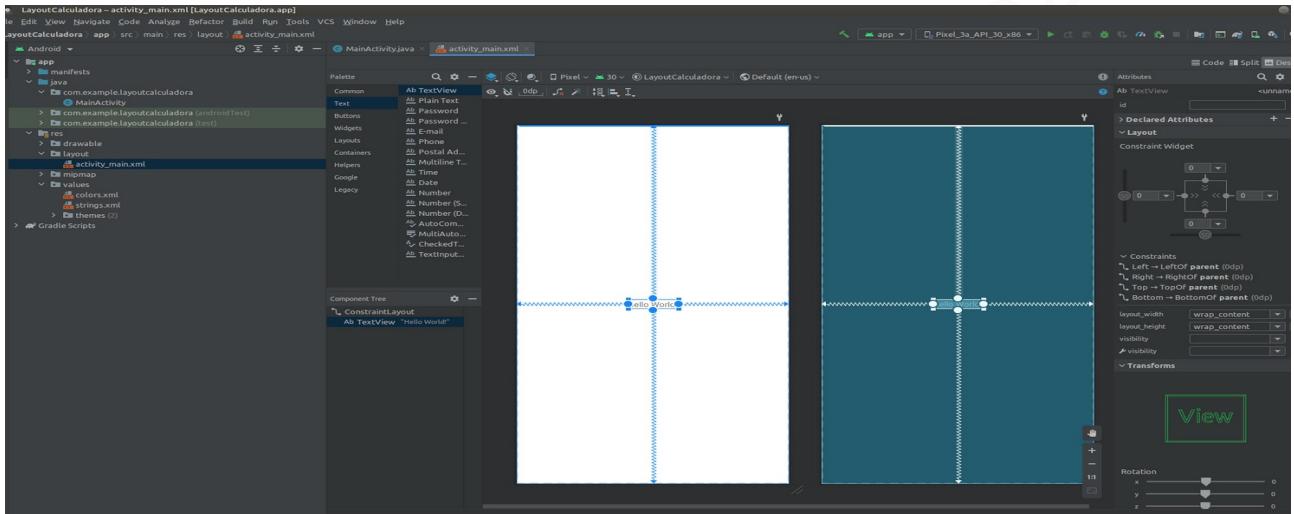


Luego toca especificar la versión de Android mínima donde se ejecutará la aplicación que desarrollamos. Si queremos usar java o kotlin y el nombre del proyecto



Android Studio nos genera todos los directorios y archivos básicos para iniciar nuestro proyecto

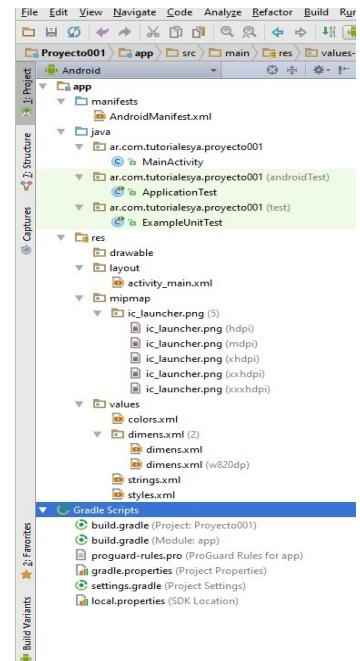
En res/layout están las interfaces gráficas. Se muestra res/layout/activity_main.xml:



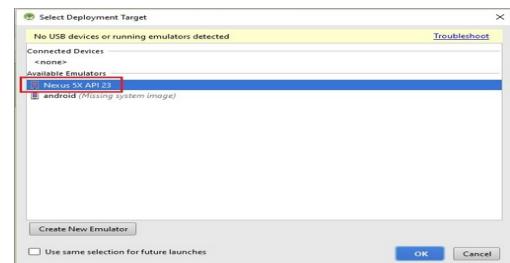
Observar que generó constraint layout (hay marcas de fijación-líneas del textView tanto en horizontal como vertical) El layout (nuestra interfaz gráfica xml) está en la carpeta res → layout → activity_main.xml

Veamos la estructura de ficheros generada:

- **AndroidManifest.xml** guardamos información de la aplicación importante para su ejecución y publicación
- En la carpeta java ponemos los fuentes de código
- En **res → layout** va la parte de modelado de pantallas gráficas
- En **values → strings** las cadenas de texto (esta parte es importante para hacer traducciones de nuestra aplicación en diferentes idiomas con facilidad)



Para ejecutar la aplicación el triángulo verde o "Run -> Run app" y presionamos el botón "OK" (si no hay emulador se puede crear en ese momento):

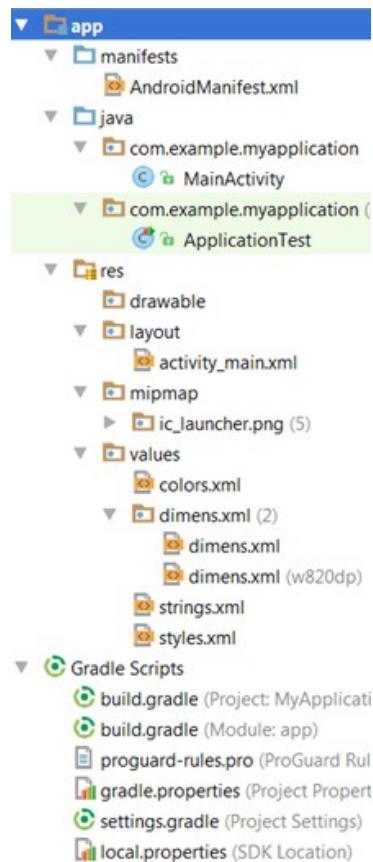


Pasado un rato aparecerá el emulador de Android en pantalla. No cerrar este emulador ya que es más lento en la carga. También se puede ejecutar en un terminal real pero hay que disponer de los driver. En el móvil se deberá activar la opción para desarrolladores: Ajustes->Información del teléfono y pulsar siete veces Luego activar Depuración de USB

Ficheros y carpetas de un proyecto Android

Cada módulo en Android está formado por un descriptor de la aplicación (*manifests*), el código fuente en Java (*java*), una serie de ficheros con recursos (*res*) y ficheros para construir el módulo (*Gradle Scripts*).

- *AndroidManifest.xml*: Describe la aplicación Android. Se define su nombre, paquete, etc. Se indican las *actividades*, las *intenciones*, los *servicios* y los *proveedores de contenido* de la aplicación. También se declaran los permisos que requerirá la aplicación. Se indica la versión mínima de Android para poder ejecutarla, el paquete Java, la versión de la aplicación, etc.
- *java*: Carpeta que contiene el código fuente de la aplicación. Como puedes observar los ficheros Java se almacenan en carpetas según el nombre de su paquete.
- *MainActivity*: Clase Java con el código de la actividad inicial.
- *ApplicationTest*: Clase Java pensada para insertar código de testeo de la aplicación utilizando el API Junit.
- *res*: Carpeta que contiene los recursos usados por la aplicación.
- *drawable*: En esta carpeta se almacenan los ficheros de imágenes (JPG o PNG) y descriptores de imágenes en XML.
- *mipmap*: Es una carpeta con la misma finalidad que *res/drawable*, pero más apropiado para iconos
- *layout*: Contiene ficheros XML con vistas de la aplicación. Las vistas nos permitirán configurar las diferentes pantallas que compondrán la interfaz de usuario de la aplicación. Se utiliza un formato similar al HTML usado para diseñar páginas web.
- *values*: Ficheros XML para indicar valores usados en la aplicación, En el fichero *strings.xml*, tendrás definir todas las cadenas de caracteres resultará muy sencillo traducir una aplicación a otro idioma.



AndroidManifest.xml

- package: nombre del paquete raíz para las clases
- versionCode: número que se incrementa cada vez que cambiamos la versión para google play
- versionName: es el nombre que se les mostrará a los usuarios en google play
- installLocation: donde debe instalarse la aplicación (ejemplo en la tarjeta sd)
- la etiqueta application:

```
    android:icon="@drawable/icon"
    android:label="@string/app_name"
```

la @ significa que queremos referenciar un recurso que está en algún lado

@drawable/icon dice que dentro de res/drawable está el archivo icon (observar que no lleva extensión)

label="@string/app_name" está en carpeta values (res/values/strings.xml Aquí se almacenan todos los strings de la aplicación. Esto es todo texto que se muestra. Es cómodo así para cambiar de idioma)

se puede poner directamente aquí la cadena de título de la aplicación en lugar de referenciarlo a string.xml (no se recomienda por las traducciones

- android:debuggable="true" (evidente su utilidad)

Etiqueta activity (aparece dentro de application en manifest)

atributos:

android:name="nombre de la actividad"

android:label="titulo" (si no se especifica se usa label de application)

android:screenOrientation="portrait" (orientación de la pantalla al mostrar la actividad En este caso únicamente podrá mostrarse en portrait. De no poder nada entonces se muestra en ambas formas. Cuidado con que al girar te destruye la activity y te la vuelve a crear)

etiqueta intent-filter (está dentro de activity)

Para relacionar las activities con los intent

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

cada vez que queramos una activity como la principal es la primera linea con MAIN

Si no se establecen los intent-filter únicamente accederas con la activity desde dentro dela aplicación Si se quiere nos dice de esta forma cuál es la activity que se lanzará al iniciar la aplicación

Permisos(ya no estamos en application.

Hijo directo de manifest):

```
<uses-permission>
    a qué puede acceder ( internet, tarjeta sd,
... )
```

Ej.

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

```
1 <?xml version="1.0" encoding="utf-8"?>
2 ②manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.treslines.phonegapfirstapp"
4     android:versionCode="1"
5     android:versionName="1.0" 
6
7     <supports-screens
8         android:anyDensity="true"
9         android:largeScreens="true"
10        android:normalScreens="true"
11        android:resizeable="true"
12        android:smallScreens="true"
13        android:xLargeScreens="true" />
14
15     <uses-permission android:name="android.permission.CAMERA" />
16     <uses-permission android:name="android.permission.VIBRATE" />
17     <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
18     <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
19     <uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
20     <uses-permission android:name="android.permission.INTERNET" />
21     <uses-permission android:name="android.permission.RECEIVE_SMS" />
22     <uses-permission android:name="android.permission.RECORD_AUDIO" />
23     <uses-permission android:name="android.permission.RECORD_VIDEO" />
24     <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
25     <uses-permission android:name="android.permission.READ_CONTACTS" />
26     <uses-permission android:name="android.permission.WRITE_CONTACTS" />
27     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
28     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
29     <uses-permission android:name="android.permission.GET_ACCOUNTS" />
30     <uses-permission android:name="android.permission.BROADCAST_STICKY" />
31
32     <uses-sdk
33         android:minSdkVersion="7"
34         android:targetSdkVersion="17" />
35
36     <application
37         android:allowBackup="true"
38         android:icon="@drawable/ic_launcher"
39         android:label="@string/app_name"
40         android:theme="@style/AppTheme" >
41         <activity
42             android:name="com.treslines.phonegapfirstapp.MainActivity"
43             android:label="@string/app_name"
44
45             android:configChanges="orientation|keyboardHidden"
46         >
47     </application>

```

Componentes de una aplicación

Vista (View)

Las vistas son los elementos que componen la interfaz de usuario de una aplicación: por ejemplo, un botón o una entrada de texto. Todas las vistas van a ser objetos descendientes de la clase View, y por tanto, pueden ser definidas utilizando código Java.

Layout

Un layout nos muestra como se van a ver nuestras diferentes pantallas. Es un conjunto de vistas agrupadas de una determinada forma. Vamos a disponer de diferentes tipos de layouts para organizar las vistas de forma lineal, en cuadrícula o indicando la posición absoluta de cada vista. Los layouts también son objetos descendientes de la clase View. Igual que las vistas, los layouts pueden ser definidos en código, aunque la forma habitual de definirlos es utilizando código XML (hay un asistente gráfico para esta acción si se prefiere)

Actividad (Activity)

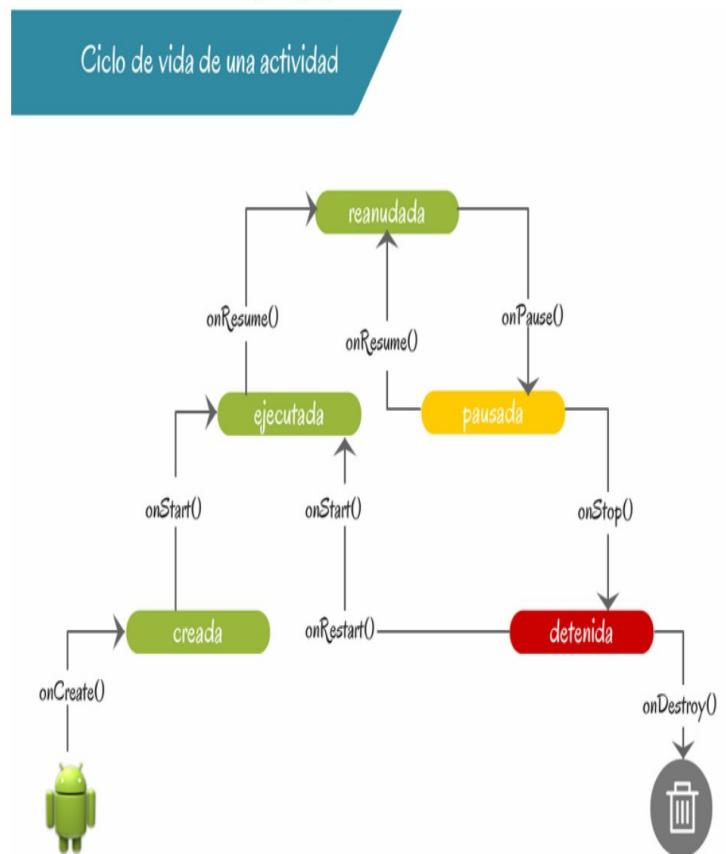
Una aplicación en Android va a estar formada por un conjunto de elementos básicos de visualización, coloquialmente conocidos como pantallas de la aplicación. En Android cada uno de estos elementos, o pantallas, se conoce como actividad. Su función principal es la creación de la interfaz de usuario. Una aplicación suele necesitar varias actividades para crear la interfaz de usuario. Las diferentes actividades creadas serán independientes entre sí, aunque todas trabajarán para un objetivo común.

De cara a crear el código java no olvidar que: En el ciclo de vida de una activity Siempre existe el método `onCreate()` Es un lugar apropiado para empezar a trabajar nuestro código

Ciclo de vida de una actividad

- **Creada:** Una actividad se ha creado cuando su estructura se encuentra en memoria, pero esta no es visible aun. Cuando el usuario presiona sobre el icono de la aplicación en su dispositivo, el método `onCreate()` es ejecutado inmediatamente para cargar el layout de la actividad principal en memoria.

- **Ejecutada:** Despues de haber sido cargada la actividad se ejecutan en secuencia a el método `onStart()` y `onResume()`. Aunque `onStart()` hace visible la actividad, es `onResume()` quien le transfiere el foco para que interactúe con el usuario.



- **Pausada:** Una actividad está en pausa cuando se encuentra en la pantalla parcialmente visible. Un ejemplo sería cuando se abren diálogos que toman el foco superponiéndose a la actividad. El método llamado para la transición hacia la pausa es `onPause()`

- **Detenida:** Una actividad está detenida cuando no es visible en la pantalla, pero aún se encuentra en memoria y en cualquier momento puede ser reanudada. Cuando una aplicación es enviada a segundo plano se ejecuta el método `onStop()`.

Al reanudar la actividad, se pasa por el método `onRestart()` hasta llegar a el estado de ejecución y luego al de reanudación.

- **Destruida:** Cuando la actividad ya no existe en memoria se encuentra en estado de destrucción. Antes de pasar a destruir la aplicación se ejecuta el método `onDestroy()`. Es común que la mayoría

de actividades no implementen este método, a menos que deban destruir procesos (como servicios) en segundo

Toast

Para mostrar un aviso tenemos un ejemplo en android developer:

```
CharSequence text = "Hello toast!";
int duration = Toast.LENGTH_SHORT;
Toast toast = Toast.makeText(this /* MyActivity */, text, duration);
toast.show();
```

Vemos que para crearlo (`Toast.makeText()`) le damos la ubicación de pantalla-activity donde se mostrará (`this` en este caso), le pasamos el texto a mostrar y la duración del aviso

● Práctica 1

Usar toast para mostrar en los diferentes métodos asociados al ciclo de vida de la app cuando aparecen. Observar que nosotros podemos sobreescribir `onCreate()` y otros métodos. Generar un documento/informe pdf que detalle que acciones has realizado para disparar el evento (al menos para 2-3 eventos)

ejemplo para `onCreate`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.content_main);
    Toast.makeText(getApplicationContext(), "First onCreate() calls", Toast.LENGTH_SHORT).show();
}
```

Servicio (*Service*)

Un *servicio* es un proceso que se ejecuta “detrás”, sin la necesidad de una interacción con el usuario. Es algo parecido a un *demonio* en Unix o a un *servicio* en Windows. En Android disponemos de dos tipos de servicios: servicios locales, que son ejecutados en el mismo proceso y servicios remotos, que son ejecutados en procesos separados.

Intención (*Intent*)

Una *intención* representa la voluntad de realizar alguna acción; como realizar una llamada de teléfono, visualizar una página web. Se utiliza cada vez que queramos:

- Lanzar una actividad
- Lanzar un servicio
- Enviar un anuncio de tipo broadcast
- Comunicarnos con un *servicio*

Los componentes lanzados pueden ser internos o externos a nuestra aplicación. También utilizaremos las *intenciones* para el intercambio de información entre estos componentes.

Fragment

Un *fragment* está formado por la unión de varias vistas para crear un bloque funcional de la interfaz de usuario. Una vez creados los *fragments*, podemos combinar uno o varios *fragments* dentro de una actividad, según el tamaño de pantalla disponible. Esto es cómodo especialmente en tabletas.

Receptor de anuncios (*Broadcast receiver*)

Un *receptor de anuncios* recibe anuncios *broadcast* y reacciona ante ellos. Los anuncios *broadcast* pueden ser originados por el sistema (por ejemplo: *Batería baja*, *Llamada entrante*) o por las aplicaciones. Las aplicaciones también pueden crear y lanzar nuevos tipos de anuncios *broadcast*.

Proveedores de Contenido (*Content Provider*)

En muchas ocasiones las aplicaciones instaladas en un terminal Android necesitan compartir información. Android define un mecanismo estándar para que las aplicaciones puedan compartir datos sin necesidad de comprometer la seguridad del sistema de ficheros. Con este mecanismo podremos acceder a datos de otras aplicaciones, como la lista de contactos, o proporcionar datos a otras aplicaciones.

Creando Layouts

Un Layout es un elemento que representa el diseño de la interfaz de usuario de componentes gráficos como una actividad, fragmento o widget.

Los layouts pueden ser creados a través de archivos XML o con código de forma programática.

Las definiciones XML para los layouts se guardan dentro del subdirectorio **res/layout**.

Para crear un layout nuevo:, Sobre el directorio layout botón derecho --> New > Layout resource file.

Cada recurso del tipo layout debe ser un archivo XML, donde el elemento raíz solo puede ser un ViewGroup o un View. Dentro de este elemento puedes incluir hijos que definan la estructura del diseño.

Algunos de los view groups más populares son:

LinearLayout

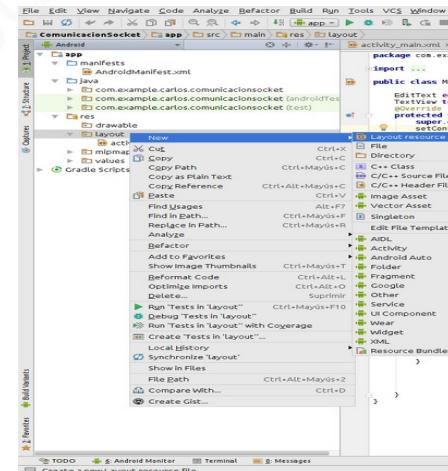
FrameLayout ← el más ligero

RelativeLayout ← poco uso

TableLayout

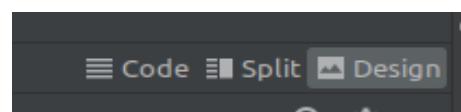
GridLayout ← en desuso

ConstraintLayout ← el más habitual

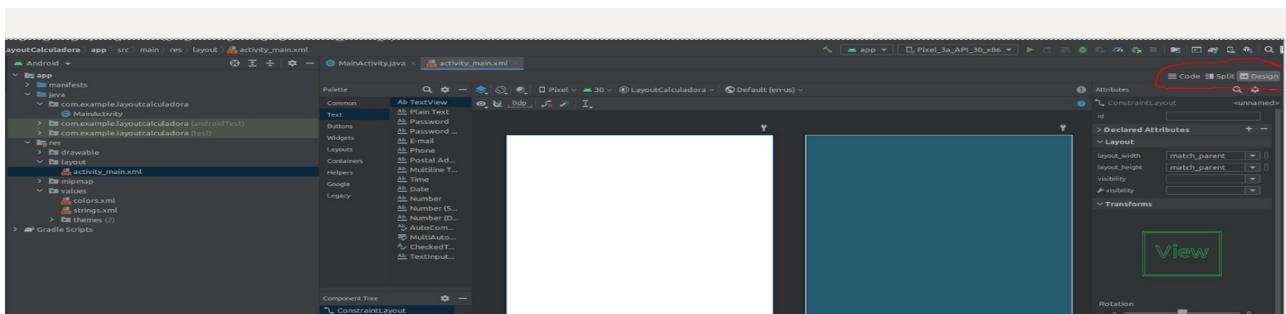


Observar el layout que se generó al crear el primer programa. El layout que generó se puede ver su código xml. Para ello abrimos el fichero correspondiente dentro de la carpeta layout: **activity_main.xml**

Aparecerá un renderizado de la posible presentación del layout en un dispositivo.



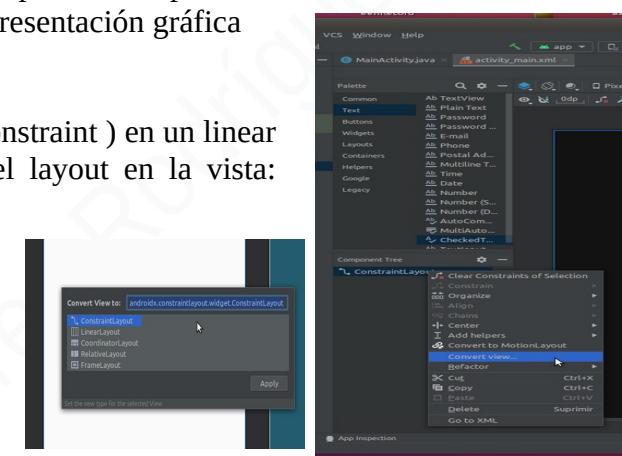
En la parte superior derecha tenemos:



Al pulsar en “Code” nos muestra el xml Al pulsar en Split nos muestra tanto xml como la representación gráfica y Design únicamente la representación gráfica

Podemos convertir un layout por defecto (constraint) en un linear layout gráficamente así: (botón derecho sobre el layout en la vista: component tree) → convert view

Aparecerá un popup y elegimos linear layout



Ejemplo de un layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".com.example.carlos.comunicationsocket.MainActivity"
    android:orientation="vertical">
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/editTextDireccionIP"
        android:text="8.8.8.8" />
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/buttonPing"
        android:id="@+id/buttonPing"
        android:onClick="ping" />
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:id="@+id/textViewResultado"
        android:layout_weight="1" />
</LinearLayout>
```



Agrupado en una estructura lineal por filas (se van introduciendo los objetos uno encima de otro) Nos encontramos EditText (para introducir texto), Button (Un botón), y un TextView (salida de texto)

Cargar layout XML En Android— Al tener definido tu recurso, ya es posible inflar su contenido en la actividad. Esto podemos ver que nos lo ha generado ya el IDE cuando creamos nuestra primera aplicación. Mirar en MainActivity se puede ver este código:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    R.layout.activity_main es nuestro layout
```

Atributos De Un Layout

Identificador de un view— Existe un atributo que heredan todos los elementos llamado id. Este representa un identificador único para cada elemento. Lo que permite obtener una referencia de cada objeto de forma específica. La sintaxis para el id sería la siguiente:

`android:id="@+id/nombre_identificador"`

Donde el símbolo '@' indica al parser interno de XML, que comienza un nuevo identificador para traducir. Y el símbolo '+' declara que dicho id no existe aún. Por ello se da la orden para crear el identificador dentro del archivo R.java a partir del string que proporcionamos.

Este atributo es útil para acceder al elemento de diseño xml luego desde nuestro código java y usarlo.

Sea por ejemplo:

```
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textAppearance="?android:attr/textAppearanceLarge"  
    android:id="@+id/textViewResultado"  
    android:layout_weight="1" />
```

Acceder a ese elemento desde código sería:

```
textViewResultado = (TextView) findViewById(R.id.textViewResultado);
```

La estrategia es sencilla. Simplemente se declara un objeto del tipo buscado y luego asignar el resultado que produce findViewById(). Como se puede ver se ha usado el **id** que está en el **recurso R**. También vemos el cast (TextView). Haremos siempre el cast correspondiente al tipo que se precise. Otros atributos que aparecen son los padding, las dimensiones de los objetos (width y height) etc

Viewgroups

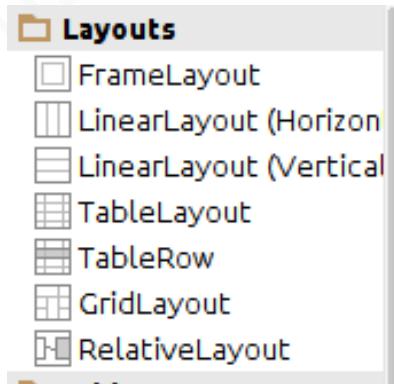
Para todo diseño que hagamos nos apoyaremos en una estructura para ir ubicando nuestros elementos.

En la paleta se nos muestran las diferentes opciones:

Vamos a trabajar con varios de ellos y practicar

Por defecto el ide nos creará un ConstraintLayout Podemos transformarlo así:

botón derecho sobre constraintlayout → Convert view- → FrameLayout



Frame Layout

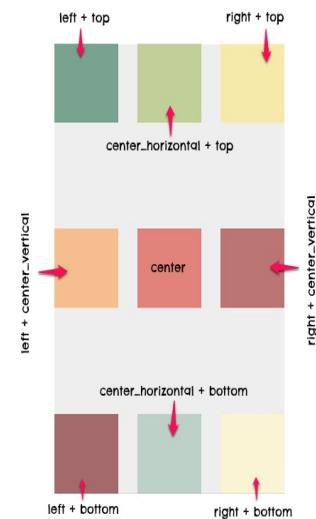
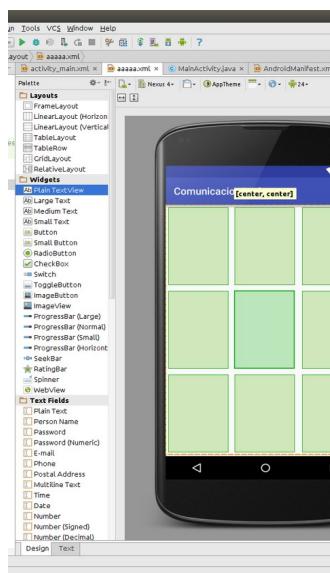
Este no realiza ninguna distribución de las vistas, simplemente las coloca unas encima de otras. Esto le evita tener que relacionar los tamaños de unas vistas con los de las demás, por lo que se ahorra recorridos del árbol de vistas, tardando menos en mostrar su contenido.

Observemos las opciones que nos da cuando arrastramos un objeto a la pantalla que tiene un framelayout:

Disponemos de unas pocas posiciones básicas donde ubicar. Este viewgroup tiene la circunstancia especial que permite poner objetos encima uno de otro y eso pudiera ser interesante en alguna ocasión.

Ej. Se quiere poner un dibujo y que su comportamiento sea el mismo que un botón.

Bastará con poner la imagen debajo, un botón encima y hacer este último transparente. De cara al usuario tendremos que actúa la imagen como un botón



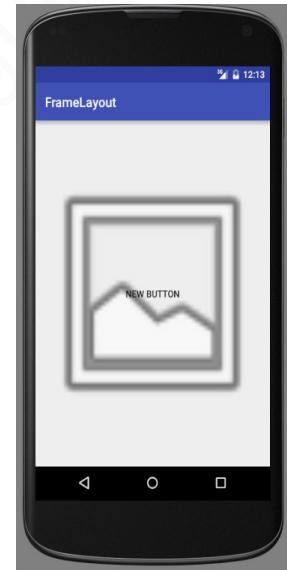
Nota: si los atributos no se muestran todos (por ejemplo las opciones para bottom etc, podemos buscarlos directamente con el icono que es como una lupa: -search-)

Vamos a hacer una operación sencilla en la que ponemos una imagen y un botón encima transparente

Crear un proyecto nuevo llamado FrameLayout Y en la vista diseño de texto del layout modificamos el constraintlayout que pone por defecto por el framelayout

En el código xml nos aparece (modificamos lo coloreado)

```
<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=".UnaActivity">  
</androidx.constraintlayout.widget.ConstraintLayout>
```



También podemos hacerlo de forma gráfica, como ya se ha explicado

Y nosotros pondremos:

```
<?xml version="1.0" encoding="utf-8"?>  
<FrameLayout 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=".UnaActivity">  
</FrameLayout>
```

Lo anterior es una alternativa escribiendo directamente en el XML en lugar de elegir la transformación de View que ya hemos explicado

El botón tendrá que estar transparente:
mediante la propiedad background y escoger en color: transparent

Una vez regresamos al modo diseño Agregamos un ImageView en la propiedad “src” pulsamos sobre el ícono de los tres puntos y nos desplegará una ventana

Elegimos la pestaña Drawable y elegimos ic_menu_gallery

De esta forma estamos poniendo ya una imagen

Observar que no está ocupando toda la pantalla. Vamos a introducir el concepto de dos valores que se usan mucho en nuestros diseños:

match_parent

wrap_content

Estas propiedades se usan en width y height habitualmente

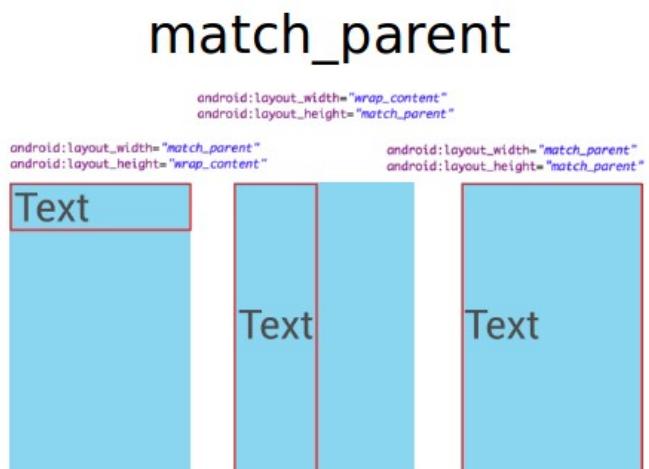
wrap_content hace que se ajuste al contenido, match_parent que ocupe el espacio que le permita el padre

Ahora, establecemos:

```
layout:width match_parent  
layout:height match_parent
```

y podemos observar que nos ocupa toda la pantalla la imagen

Ahora arrastramos un Button y lo ubicamos en la misma posición que hicimos con el ImageView hacemos lo mismo de antes con width y height



Recordar que hay multitud de dispositivos con pantallas diferentes. Usar tamaños fijos mediante **dp** nos puede dar renderizados inapropiados. **match_parent** y **wrap_content** evitan esa problemática

Ya que introducimos las imágenes, tener en cuenta que **la forma que procede habitualmente para incorporar una imagen es mediante imageView** y para poner una imagen externa disponible para nuestra aplicación:

Buscar la carpeta de androidStudio (por ejemplo)
~/AndroidStudioProjects/MyApplication/app/src/main/res/drawable

poner la imagen (nombre con únicamente minúsculas y números)

Ahora en el editor una vez se ha elegido el imageview nos aparece en propiedades la opción: srcCompat pulsando en el ícono para seleccionar mediante browser vamos a la pestaña Drawable y nos aparece la imagen que hemos incorporado antes



Práctica 2: Realizar la actividad que acabamos de describir

Linear Layout

Este es un viewgroup sencillo que permite ir colocando los objetos uno detrás de otro según la dirección escogida (vertical, horizontal)

Vamos a usar el LinearLayout vertical para mostrar como ocupar los espacios.

Recrear el layout de la imagen para conseguir que el textView ocupe el mismo espacio que el button vamos a hacer uso de una propiedad muy útil:
layout:weight

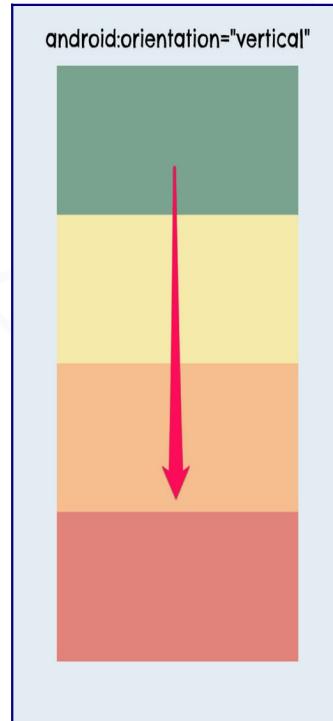
En esa propiedad establecemos el peso que tiene el elemento respecto a los demás dentro de nuestro ViewGroup

Si establecemos ambos elementos a 1 le estamos diciendo que queremos que cojan el mismo espacio ambos

para que esto funcione apropiadamente en layout:height le ponemos: wrap_content

● **Práctica 3:** Realizar la actividad que acabamos de describir

● **Práctica 4:** Poner en un linear layout horizontal un editText, un button y un text hacer que el text tenga el doble de crecimiento que los otros dos

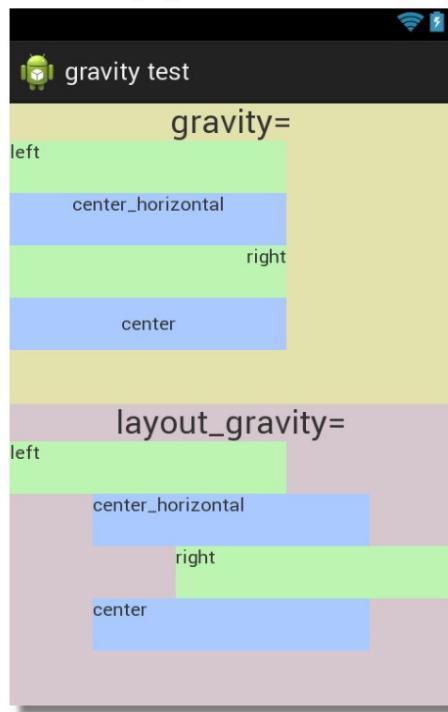


Otra propiedad importante es `gravity` y `layout_gravity`. Ambas nos ayudan a ubicar los elementos. Para entenderlo mejor ver un ejemplo:

Como se puede observar `gravity` afecta a la posición interna. Así pues el texto nos queda ubicado según lo establecido en esa propiedad.

`Layout_gravity` viene a determinar la posición respecto al padre. Observar que los `textview` se alinean ellos mismos no su contenido (el texto)

Nota: recordar que si no aparecen los atributos que queremos siempre los podemos buscar directamente: `button`, `top`, etc desde la lupa (comando search de las propiedades)



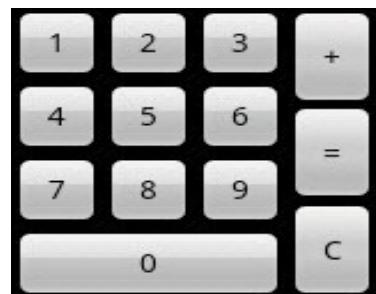
Los `viewgroup` (`frameLayout`, `LinearLayout`,...) se pueden anidar unos dentro de otros para conseguir los diseños deseados.

Es conveniente usar el mínimo número de layouts posibles porque supone un importante gasto de recursos en el renderizado

Aún con ello para practicar su uso vamos a realizar el siguiente layout utilizando únicamente `LinearLayout` (habrá varios anidados)

● **Práctica 5:** Renderizar una tabla con linear layouts: Conseguir un layout lleno de botones como si fuera una tabla de 3x2. El primer botón dirá: 1, el segundo: 2,... (pensar en poner un `linear layout horizontal` dentro de otro `vertical`)

Sea el siguiente renderizado:



Una alternativa para hacerlo:

Para hacer ese render se va a tener que usar las siguientes propiedades:

- layout_margin (todos los botones se les ha aplicado lo tienen)

- a veces hay que usar en los objetos: wrap_content y a veces match_parent

- hay que establecer “peso” a los objetos. Es con la propiedad: layout_weight

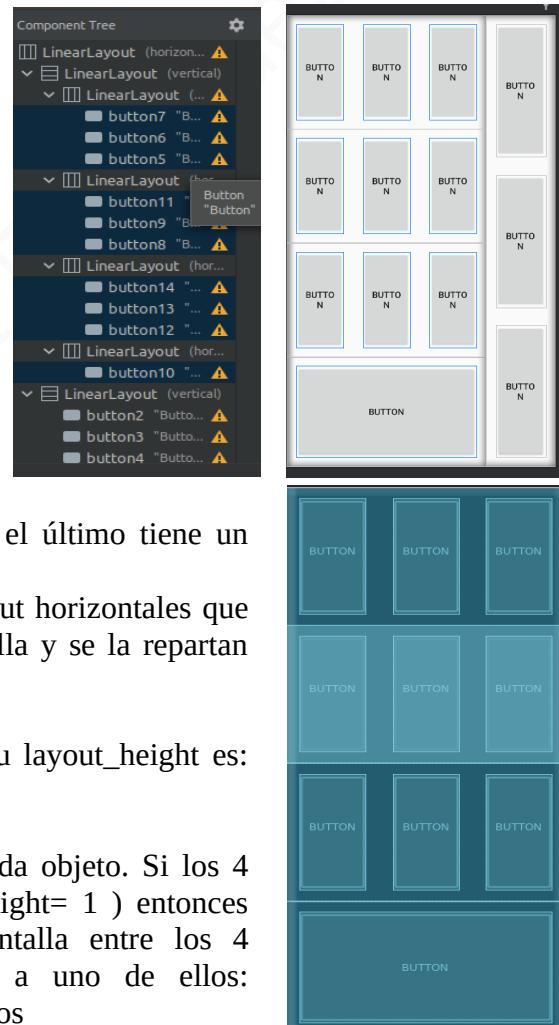
Como ejemplo los 4 linear layout horizontales:

Vemos que los tres primeros tienen 3 botones y el último tiene un único botón

¿cómo hacemos para que esas 4 filas (los 4 layout horizontales que hemos dicho) ocupen el total de la altura de la pantalla y se la repartan equitativamente ?

La solución es poner en cada uno de ellos que su layout_height es: wrap_content y que su layout_weight es 1

El layout weight viene a significar el peso de cada objeto. Si los 4 layout tienen el mismo peso (todos pesan layout_weight= 1) entonces tratan de repartirse el total de la altura de la pantalla entre los 4 equitativamente. Si en lugar de eso le ponemos a uno de ellos: layout_weight=2 ese será el doble de grande que los otros



Práctica 6: Realizar el ejercicio descrito

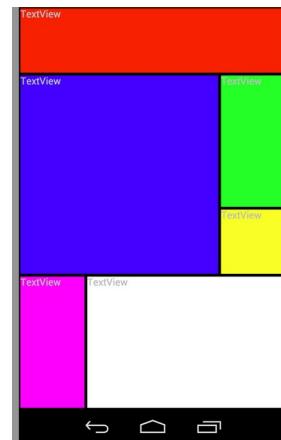
Table Layout

En TableLayout indicaremos el número de filas y columnas como propiedades del layout, mediante **android:rowCount** y **android:columnCount**.

No hace falta TableRow, sino que los diferentes elementos hijos se irán colocando ordenadamente por filas o columnas (dependiendo de la propiedad android:orientation) hasta completar el número de filas o columnas indicadas en los atributos anteriores.

Adicionalmente, también tendremos disponibles las propiedades **android:layout_rowSpan** y **android:layout_columnSpan** para conseguir que una celda ocupe el lugar de varias filas o columnas.

- **Práctica 7:** Realizar el layout de la figura mediante tablelayout o una combinación de linear layouts y table layouts



- **Práctica 8:** Hacer el layout para la siguiente imagen:



Constraint Layout

Hay muchísimos dispositivos diferentes que ejecutan Android. Eso implica muchos tamaños de pantalla, y cuándo hacemos un renderizado, muchas veces no se ve tan bien en un dispositivo como en otro. De igual manera, en el mismo dispositivo cuándo lo giramos la pantalla también tiene impacto en la visualización.

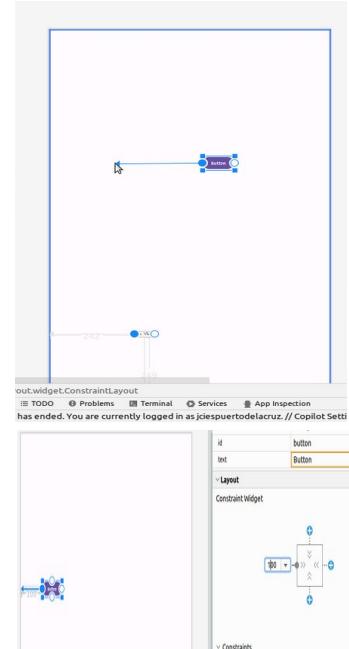
Hay ocasiones en las que el reparto proporcional del incremento de pantalla no se ajusta lo suficientemente bien a lo que queremos. El constraint layout está pensado para ser más flexible. Ya que delimitamos los elementos en relación a otros.

Un detalle importante es que, estamos obligados a poner “ligaduras/ataduras” a nuestras views en el layout o se renderizarán mal

● **Práctica 9:** En layout constraint Arrastrar un button y un text al lugar que se quiera de la pantalla. Ejecutar la app ¿ se visualizan dónde las ubicaste ?
Pulsar el icono de “infer constraints” 
Ejecutar de nuevo la app ¿ qué ha ocurrido ?

Cuándo pulsamos encima del botón que hemos puesto en el layout, observamos que aparecen 4 círculos que hacen referencia a las posibles “constraints” que podemos poner.

Vamos a arrastrar desde el izquierdo (aparecerá una flecha) hasta el lateral izquierdo de la pantalla. Observaremos que se queda pegado al lado izquierdo. Esto ocurre porque le estamos diciendo que tiene que estar ligado a ese borde. Si queremos podemos establecer una separación ahora mediante un margen

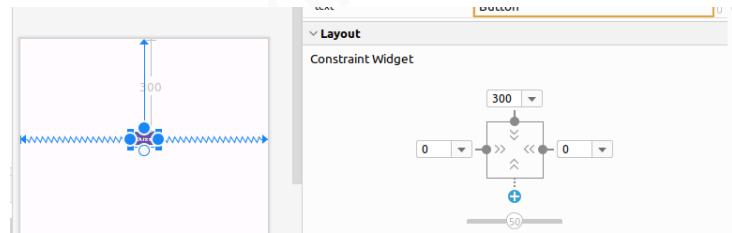


En “Constraint Widget” podemos establecer ese margen. En el ejemplo le hemos puesto un “100”

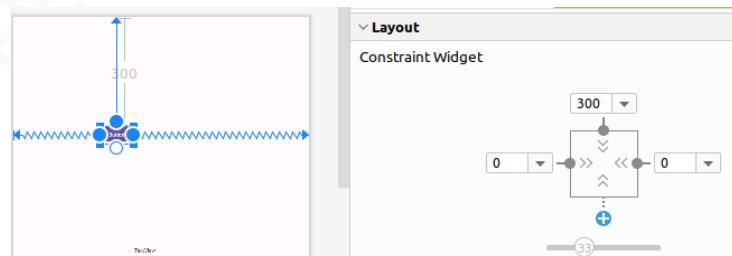
● **Práctica 10:** Poner una restricción del botón al lateral derecho con un margen de 100 y otra del botón al borde superior con un margen de 300. Ejecutar la app ¿ se muestra dónde esperamos el botón ?

Hay una restricción especial cuando atamos a un lado y al contrario (ponemos la restricción a la izquierda y a la derecha o por ejemplo arriba y abajo)

Vamos a arrastrar del lado derecho del botón hasta el borde derecho de la pantalla Debe quedar parecido a esto:



Ahora aparece abajo: “horizontal bias” y está en 50, viiniendo a significar que las dos ataduras están en equilibrio. Ahora bien, podemos decirle que esté a $\frac{1}{3}$ del lado izquierdo y a $\frac{2}{3}$ del derecho, poniendo el horizontal bias a ese punto



Podemos poner nuestras ligaduras de un objeto a otro. Así podemos hacer una ligadura arrastrando desde la parte superior del textView a la parte de abajo del botón y decirle que queremos una separación de 100. De esa forma estará separado en vertical siempre 100dp el textView respecto al botón. Si subimos el botón en la pantalla el textView lo seguirá a una distancia de 100



● **Práctica 11:** Poner una restricción del textView al botón de 100dp. En horizontal pondremos la restricción mediante horizontal bias que esté en 75 (un 75% lejos del borde izquierdo y un 25% del derecho)

● **Práctica 12:** Hacer un layout de formulario persona con constraint layout. Campos: nombre, apellidos, edad (edittext de número), y dos radiobutton (para hombre y mujer) NO pondremos label en los edittext sino un “hint” que es equivalente a placeholder

Cargando diferentes idiomas

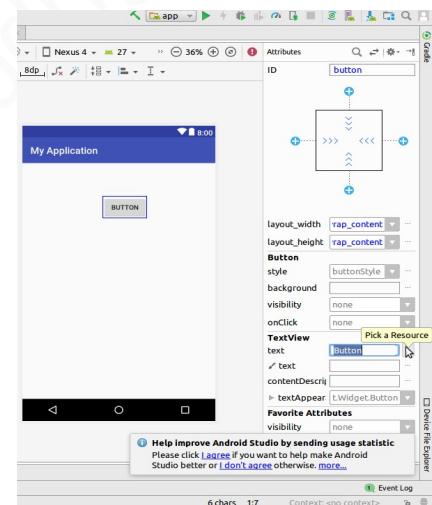
Para cargar diferentes idiomas tenemos que crear el fichero strings.xml pertinente

Por ej. para crea carpeta en castellano-España:

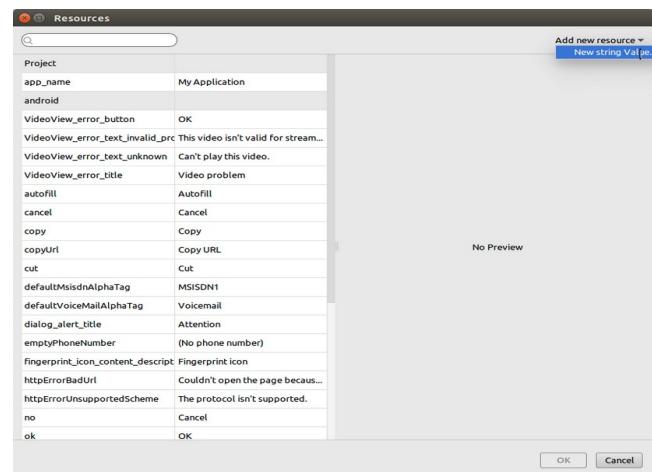
MyProject/res/values-es-rES

y ahí dentro estará el strings.xml en castellano

Para que funcione previamente habremos puesto los textos como un resource:

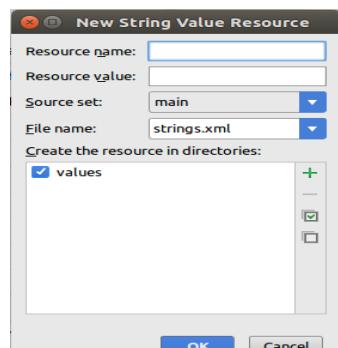


Pulsamos en el icono de los ... que aparece en la imagen anterior con el globo amarillo: “Pick a resource” y nos aparece la siguiente pantalla:



Seleccionando: “New string value” nos mostrará:

Es en esa ventana donde ponemos el nombre/id del recurso y el texto para el idioma



Una vez establecidas todas las cadenas de texto únicamente las copiamos y sustituimos en la carpeta de idioma que comentábamos antes. De esta forma cuando se detecte en el sistema operativo un cambio de idioma también cambiará la aplicación

Otra forma de hacerlo:

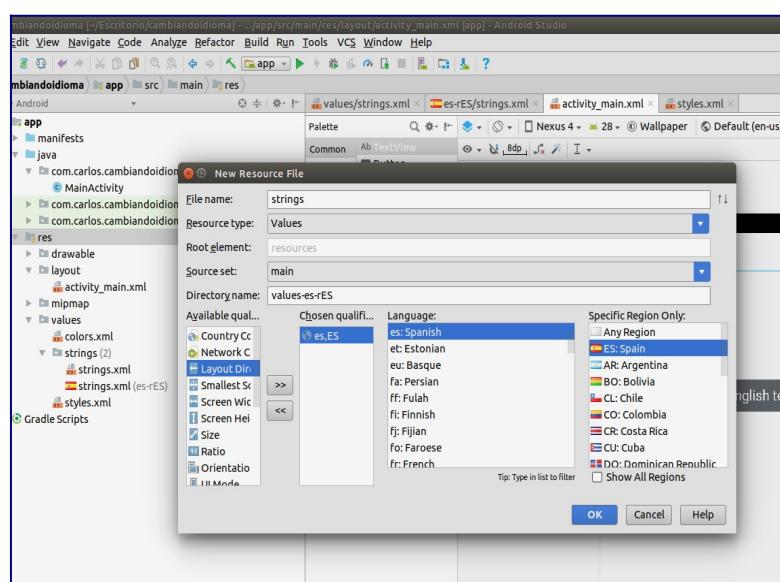
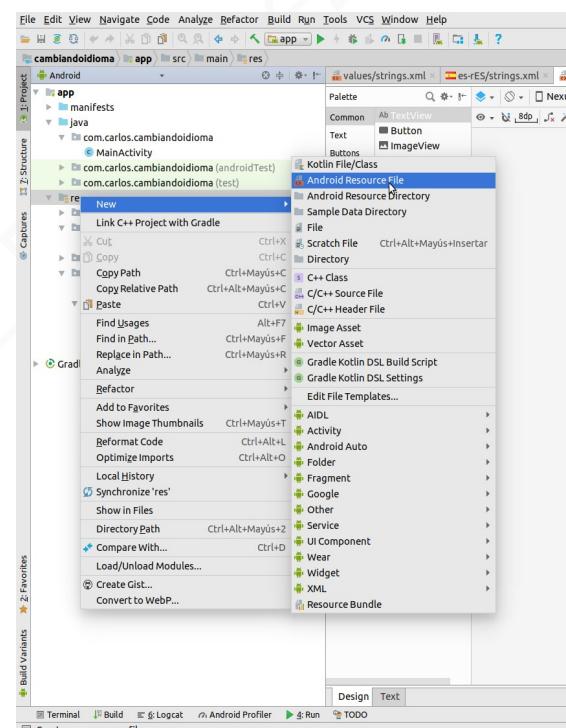
App > Res > Values y elegir “New Values Resource”.

Se nos abrirá una nueva ventana para crear el recurso. En esta ventana ponemos el nombre (strings) y el tipo (locale) del menú de la izquierda y pulsaremos sobre la flecha para agregarlo al menú de la derecha.

Automáticamente veremos cómo se nos habilitan dos columnas nuevas: Language y Region. Aquí debemos elegir el idioma y la región que correspondan con el idioma al que vamos a traducir, por ejemplo, “es: Spanish” y “Any Region” de manera que esa traducción será “al español” para todas las regiones de habla hispana.

Pulsamos sobre “Ok” y listo, se creará un nuevo strings con el icono de la bandera de España, indicando que dicho fichero almacena la traducción al español.

Abrimos este fichero y lo rellenamos igual que el strings.xml que hemos visto al principio de este tema, aunque cambiando el texto original por el texto traducido.



Práctica 13: En el proyecto que estamos haciendo una calculadora (ya hemos hecho el layout) se observa que aparece como título “Calculadora” Tenemos que tener dos ficheros xml de strings para que el título corresponda en inglés o a castellano y se muestre correctamente al cambiar el idioma del teléfono

Creando código

Una buena forma de aprender es mirando códigos de ejemplo. Un lugar que concentra mucha información es:

<https://developer.android.com/develop/index.html>

Hay una sección que nos informa de los diferentes controles etc

También nos referencia ejemplos. Aquí el enlace en github:

<https://github.com/googlesamples/>

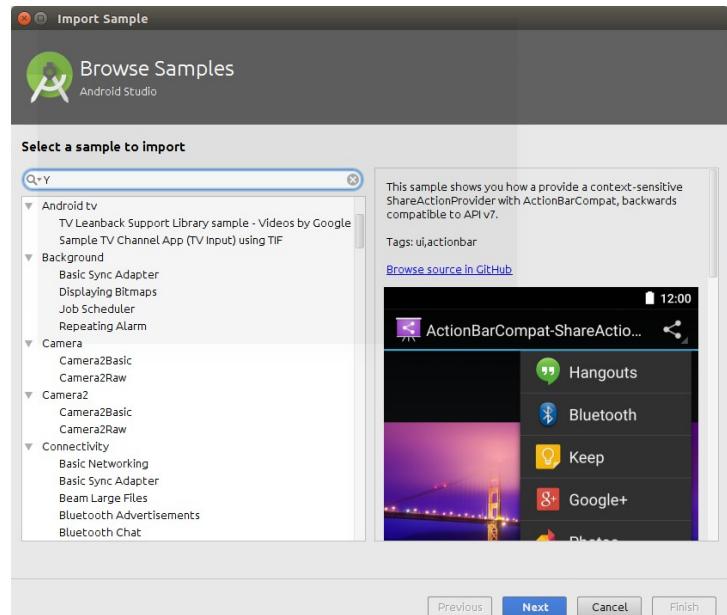
¿ Cómo importar proyectos de GitHub?

File>New>Project from Version Control>Git

Por ejemplo:

https://github.com/codepath/intro_android_demo.git

En android studio ya tenemos ejemplos que podemos cargar:
File- → New- → Import sample

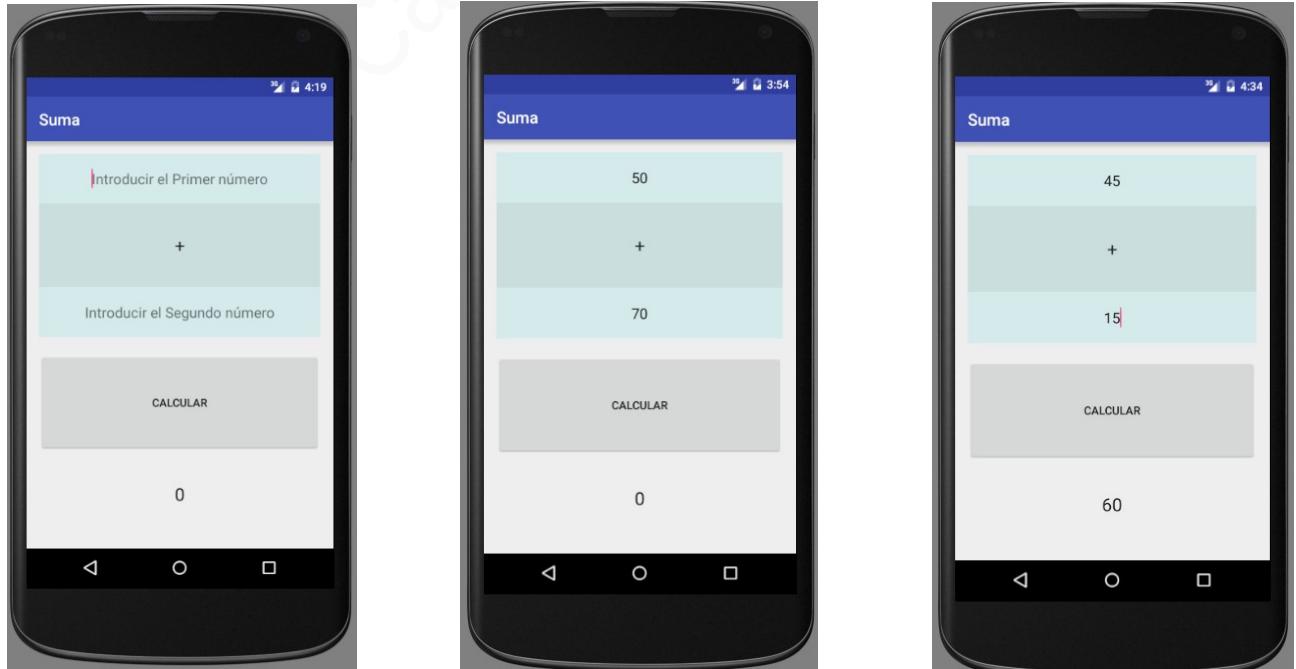


Activities

Como una primera aproximación una definición de Activity sería cada una de las pantallas (y su código asociado) que se muestra en una aplicación

Así pues habrá un fichero .java para el código y un .xml para el layout de la Activity

Práctica 14: Realizar la siguiente aplicación (el layout ahora y luego el evento)
Debe tener soporte para idiomas (crear un recurso para el texto del botón)



La parte de layout ya sabemos como hacerla. En cuanto a ejecutar una acción de botón, hablaremos de los listener

Escuchar y manejar eventos onClick

Varias formas de hacerlo:

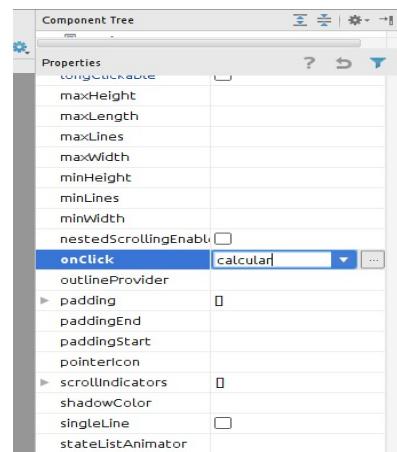
1. Declarando el método en el fichero xml

```
<Button  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="@string/calcular"  
    android:id="@+id/buttonCalcular"  
    android:longClickable="false"  
    android:onClick="calcular" /> <!-- observar el onClick -->
```

El equivalente del código XML se puede hacer mediante la interfaz gráfica, poniendo el nombre del método en el atributo: onClick Como se ve en la imagen:

De esta forma en el código podemos llamar al método llamado: **void onClick(View view)** .

¡¡ojo !! el prototipo debe ser que **devuelva void y reciba View**



Veamos un ejemplo en código:

```
public class MainActivity extends AppCompatActivity {  
    TextView resultado;  
    Button boton;  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
  
    void calcular(View view){  
        int num1 =  
        Integer.parseInt(((EditText) findViewById(R.id.editTextNumero1)).getText().toString());  
        int num2 =  
        Integer.parseInt(((EditText) findViewById(R.id.editTextNumero2)).getText().toString());  
        int res = num1 + num2;  
        resultado.setText(" " + res);  
    }  
}
```

● **Práctica 15:** Aparte de acabar la actividad anterior con los conocimientos de como crear un listener, agregar un segundo botón que sirva para limpiar los input y el resultado

2. Agregando un listener en el código Java:

```
Button b = (Button) findViewById(R.id.button1);
b.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        //aquí el código. O también podemos llamar al método anterior:
        calcular(v);
    }
});
```

● **Práctica 16:** Cambiar el botón de la actividad anterior que limpiaba los input y el resultado para que ahora el método no esté declarado en el xml sino que sea desde código con: setOnClickListener()

Esta forma es la más potente ya que podemos establecer otros listener (longclick, drag, etc)

Observemos por ejemplo onLongClick():

```
btnHola.setOnLongClickListener(new View.OnLongClickListener() {
    no usages
    @Override
    public boolean onLongClick(View view) {
        Toast toast = Toast.makeText(
            context: MainActivity.this, //hay dos posibles this
            "yeeeepa!!!",
            Toast.LENGTH_LONG);
        toast.show();
        return true;
    }
});
```

El return true se hace para indicar que no se quiere que se disparen los otros listener posibles al evento (por ejemplo, si la view padre del botón también tiene este listener y el return está en false, se dispararían los dos)

Práctica 17: Agregar un listener para longClick y que muestre tu nombre el toast. Observa que al usar un método de una clase anónima tenemos que llamar a this mediante MainActivity.this Convertir en una lambda y escribir this sin MainActivity, Comprobar que así se puede llamar a this sin especificar MainActivity

3. Agregando una interfaz a la activity

Se le agregar el listener mediante una interfaz a la activity y luego cuando se dispare el evento se discrimina la acción según el objeto que haya disparado la acción:

```
public class MainActivity extends AppCompatActivity implements View.OnFocusChangeListener{

    1 usage
    @Override
    public void onFocusChange(View view, boolean b) {
        if( view == editText1 && b){
            Toast toast = Toast.makeText(
                context: MainActivity.this,
                text: "establecido foco en editText1",
                Toast.LENGTH_LONG);
            toast.show();
        }
    }
    3 usages
    EditText editText1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        editText1 = (EditText) findViewById(R.id.editTextText);
        editText1.setOnFocusChangeListener(this);
    }
}
```

En el ejemplo se usa un interfaz para poner el listener para FocusChange. Eso implicaría que se desencadenaría tanto si se toma como se pierde el foco. El boolean del método: onFocusChange() nos indica con: true si ha tomado el foco y con false que no lo tiene

● **Práctica 18:** Agregar mediante implementar interfaz en la activity el listener FocusChange a los dos edittext La propiedad editText.setTextSize(23) permite establecer el tamaño de letra en 23 al edittext. Hacer que la letra del edittext que tenga el foco se haga más grande y si no tiene el foco se haga más pequeña

● **Práctica 19:** Realizar la siguiente Activity usando las tres formas que hemos visto de implementar un listener (xml, con código y con interfaz) para el listener onClick

Observar el alineado. Se tiene que conseguir que quede centrada la tabla en la pantalla



● **Práctica 20:** Realizar una calculadora sencilla con la interfaz que se muestra en la imagen



Intents

Introducción

Un intent sirve para invocar componentes: Activities, Services, AsyncTask (hilos en background), ... Los Intents nos permiten llamar a aplicaciones externas a la nuestra, lanzar eventos a los que otras aplicaciones puedan responder, lanzar alarmas etc.

Intents disponibles en Android

En developer.android.com/guide/appendix/g-app-intents.html se puede encontrar una lista con las aplicaciones disponibles en Android junto con los intents que las invocan.

Por ejemplo, para el navegador web, tenemos dos acciones, VIEW ** y **WEB_SEARCH, que abren el navegador en una url específica o realizan una búsqueda.

En el caso del dialer (marcador), tenemos las acciones CALL y DIAL, que vienen dadas por la URI tel:numero_de_teléfono, la diferencia entre estas dos acciones, es que CALL realiza la llamada al número de la URI, y DIAL solo lo marca, pero no realiza la llamada.

Ejemplos:

```
// estamos dentro de una Activity y queremos que se abra el navegador url google
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("http://www.google.com"));
startActivity(intent);
```

```
// estamos dentro de una Activity y queremos llamar al número: 555-555-555
// se abrirá la app vinculada para llamadas
Intent intent = new Intent(Intent.ACTION_CALL);
intent.setData(Uri.parse("tel:555-555-555"));
startActivity(intent);
```

```
// estamos dentro de una Activity y queremos busqueda en maps restaurantes
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("geo:0,0?z=4&q;=restaurantes"));
activity.startActivity(intent);
```

● **Práctica 21:** Crear una pantalla con tres botones. Cada uno de los botones ejecutará las acciones que acabamos de describir. Observar que puede haber circunstancias de permisos (por ejemplo para navegar hace falta el de internet) Buscar los que corresponda y ponerlos en AndroidManifest.xml

De los ejemplos anteriores se desprende que Intent.ACTION_VIEW puede desencadenar diferentes acciones según la uri que se quiera (así una url nos abre el navegador. Si se pone geolocalización abrirá maps) Luego están otros intent que ya delimitan claramente la acción. Por ejemplo, Intent.ACTION_CALL está pensado para hacer una llamada de teléfonos

Los anteriores intents los llamamos **implícitos**, el sistema determinará que app es la que se encarga de ejecutar la acción que hemos solicitado. En los explícitos nosotros le decimos quién se encarga de la acción

● **Práctica 22:** Realizar una aplicación que al pulsar el botón nos lleve a la página escrita por el usuarios



Intent Explícito. Lanzar una activity de nuestra app

Primero una nota importante a tener en cuenta: Cuando creamos una Activity mediante el IDE este nos modifica convenientemente AndroidManifest.xml para que nos aparezca la nueva activity. De no hacer esto NO podrá ser llamada con un intent explícito. Es pues algo que tenemos que tener en cuenta si lo hacemos nosotros a mano

En el caso de los Intent explícitos no hay que registrarlos en AndroidManifest. Esto es porque estamos llamando a la actividad en el intent directamente con el nombre de la clase

Supongamos la siguiente activity(botón derecho sobre carpeta java- → New-->Activity):

```
public class SegundaActivity extends AppCompatActivity {  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_segunda);  
    }  
}
```

Ahora se lanza la SegundaActivity con el Intent:

```
//estamos dentro de una Activity  
Intent intent = new Intent(getApplicationContext(),SegundaActivity.class);  
//Intent intent = new Intent(this,SegundaActivity.class);  
  
startActivity(intent);
```

Nota: podemos hacer uso de: **getApplicationContext()** o **this** en ambos casos nos funcionará

Se puede tomar desde la Activity lanzada por el Intent la información de ese propio intent. Esto se hace así:

```
//Para que funcione poner dentro de onCreate() de la Activity  
Intent intent = this.getIntent();  
if (intent == null){  
    Log.d("Tag", "No hubo llamada desde Intent.");  
}
```

Práctica 23: Crear una app con dos pantallas En la inicial aparecerán dos botones. Un botón dirá: calculadora y abrirá la activity de la calculadora que ya hemos hecho. El otro botón dirá: descomposición y abrirá una activity donde ponemos un número y nos obtiene el número descompuesto en primos

Paso de información con el Intent

Mediante los putExtra() (clave,valor) de tipos nativos del intent

La forma de hacer esto es mediante clave/valor: `intent.putExtra("clave", valor);`

Ejemplo:

```
Intent intent = new Intent(this,verResultados.class);
intent.putExtra("edad", persona.edad);
intent.putExtra("altura", persona.alturaEnCm);
startActivity(intent);
```

En el trozo anterior se está suponiendo que estamos dentro de una Activity. Si estuviéramos a su vez dentro de una clase anónima de un listener de un botón en lugar de Intent(this, verResultados.class) sería:

```
Intent(getApplicationContext(),verResultados.class); o: Intent(NombreActivity.this, verResultados.class)
```

Así se lanzaría la Activity: verResultados que recibirá dos datos: edad y altura

Veamos la recepción de esos datos en la activity verResultados:

```
int variableInteger = getIntent().getIntExtra("edad",0);
double variableDouble = getIntent().getDoubleExtra("altura",0);
```

Una forma útil y elegante es hacer uso de los Bundle. Donde agrupamos lo que queremos enviar:

```
Bundle bundle = new Bundle();
bundle.putString("nombre","Alfonso Hernández");
intent.putExtras(bundle)
```

Bundle permite múltiples tipos de datos. Buscar en la documentación oficial

Ahora bien, ¿ cómo se recibe la información desde el otro lado ? (desde la actividad que ha sido lanzada por el Intent):

```
Intent intent = getIntent();
Bundle bundle = intent.getExtras();
```

Finalizar una Activity

Cerrar una Activity y regresar a la origen es tan sencillo como llamar al método: finish()

● **Práctica 24:** Modificar la práctica anterior, para que ahora haya un input en la activity inicial que permita poner el número que luego al pulsar en el botón nos envíe a la segunda activity y haga la descomposición en números primos. En la activity que realiza la descomposición habrá un botón para finalizar y regresar a la activity inicial

Mediante putExtra de cualquier objeto (Parcelable) y un helper

Lo anterior tiene varios problemas:

- Las String que hacen de clave, para el par clave/valor de putExtra() provocarán errores si no coincide el string que se envía con el que se recibe

Para resolverlo podemos crear una clase pública accesible para las dos activities que nos garantice que las etiquetas serán las mismas siempre:

```
public class ActivityHelper {
    public static final String USER_NAME = "key_user_name";
    public static final String USER_EMAIL = "key_user_email";
}
```

Así en el putExtra ahora pondríamos: putString(ActivityHelper.USER_NAME,"Ana")

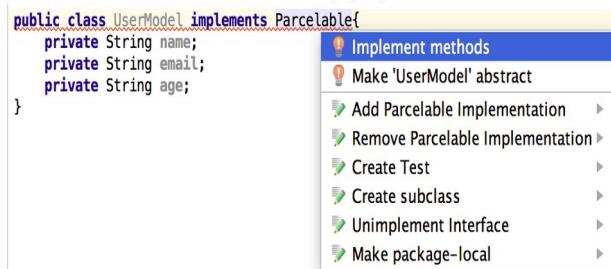
- Segundo problema: ¿ qué ocurre si queremos pasar objetos personalizados ?

Para eso existe una opción ya incluida en android llamada: Parcelable

Es una interfaz que nos permite determinar como vamos a serializar nuestros objetos y como los reconstruimos (otra opción es mediante serializable de java pero se demuestra que es mucho mejor en gasto computacional Parcelable)Ej. Tenemos una clase: UserModel(name:string, email:string, age:int)

Le diremos que implemente Parcelable y luego el ide nos generará los métodos y un CREATOR

El ide nos ayuda para crear la clase. Nosotros lo que debemos fijarnos especialmente es en el writeToParcel() (ahí se especifica el orden en el que se serializan los atributos y el tipo de dato serializado) y luego observar que en el constructor que recibe un Parcel los atributos se lean en el mismo orden que ese han escrito



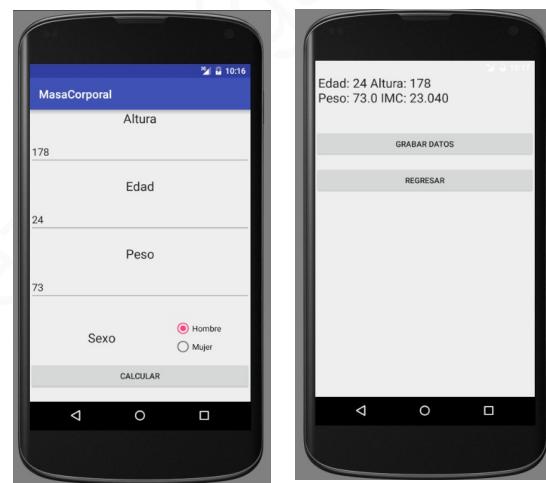
```
public class UserModel implements Parcelable {
    private String name;
    private String email;
    private String age;
    public UserModel(Parcel in){
        // El constructor de UserModel aquí
    }
    protected UserModel(Parcel in) {
        name = in.readString();
        email = in.readString();
        age = in.readString();
    }
    public static final Creator<UserModel> CREATOR = new Creator<UserModel>() {
        @Override
        public UserModel createFromParcel(Parcel in) {
            return new UserModel(in);
        }
        @Override
        public UserModel[] newArray(int size) {
            return new UserModel[size];
        }
    };
    @Override
    public int describeContents() {
        return 0;
    }
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(name);
        dest.writeString(email);
        dest.writeString(age);
    }
}
```

Los putExtra() tienen soporte para Parcelable así que basta enviar el objeto como un putExtra más: `putExtra(ActivityHelper.KEY_USER, miObjetoUserModel);`

Recibirlo en la otra Activity mediante getParcelableExtra():

```
UserModel user = (UserModel) getIntent().getParcelableExtra(ActivityHelper.KEY_USER);
```

● **Práctica 25:** Sea una aplicación que calcule los datos de índice de masa corporal. En la primera Activity se introducen los datos y al pulsar el botón calcular nos muestra la segunda Activity con los resultados. En esta segunda Activity el botón regresar ejecuta el finish(). El botón grabar datos quedará en el layout pero no lo vamos a implementar en el código de momento.



Paso de información al recrear por girar la pantalla

Algo que se parece al problema de pasar información entre pantallas pero es un poco diferente es que al girar la pantalla podemos perder la información cargada ya que se regenera toda la pantalla. Para solucionar ese problema sobreescrivimos el método: onSaveInstanceState()

Cuando giramos la pantalla se recrea la activity (es una orden que ejecuta el sistema operativo) y se pierde la información que había previamente.

Podemos ejecutar el siguiente ejemplo:



Crear en la activity un edittext con id: **tvShowNumbers** y un button con id: **btnNumberCreate**

Pondremos en la activity un atributo de tipo ArrayList llamado datos. Luego en el onCreate() al poner el código del listener del botón le pedimos que nos agregue el número aleatorio generado al arraylist y después visualizamos el contenido del arraylist en el view

```
datos = new ArrayList<>();
binding.btnNumberCreate.setOnClickListener(v -> {
    int aleatorio = (int) (Math.random() * 100);
    datos.add(aleatorio);
    binding.tvShowNumbers.setText(datos.toString());
});
```

● **Práctica 26:** Crear la activity descrita y comprobar que ocurre al girar la pantalla

Al recrear la pantalla se ha perdido la información almacenada en el ArrayList

La solución históricamente es el Bundle savedInstanceState:

```
@Override  
protected void onCreate(Bundle savedInstanceState) { //se recibe el bundle con datos
```

```
@Override  
protected void onSaveInstanceState(Bundle outState) { //se guarda el bundle con datos
```

En el onSave del ejemplo estamos guardamos en el bundle outState el contenido de un edittext y en el onCreate recuperamos la información de ese bundle.

onSaveInstanceState() se dispara automáticamente antes de recrear la pantalla

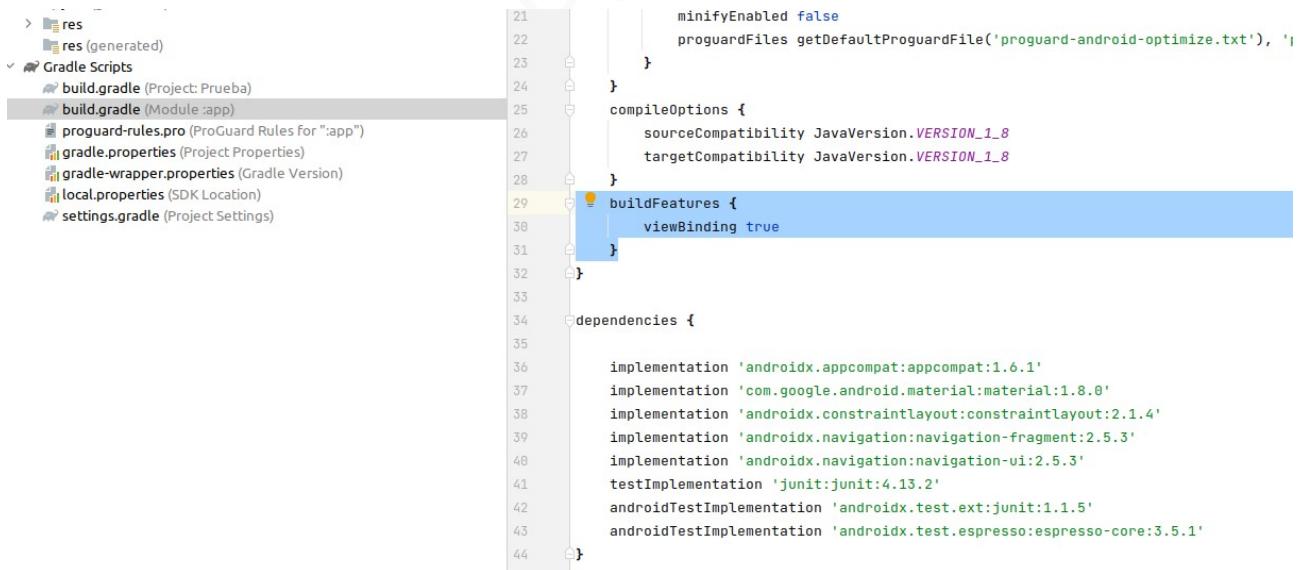
```
@Override  
protected void onSaveInstanceState(Bundle outState) {  
    super.onSaveInstanceState(outState);  
    String strIntroDatos = txtIntroDatos.getText().toString();  
    outState.putString("txtIntroDatos", strIntroDatos);  
}  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    txtIntroDatos = (EditText) findViewById(R.id.txtIntroDatos);  
    if(savedInstanceState != null){  
        txtIntroDatos.setText(savedInstanceState.getString("txtIntroDatos", ""));  
    }  
}
```

● **Práctica 27:** Usar onSaveInstanceState() para guardar el arraylist de la actividad anterior y recuperarlo en el onCreate() Comprobar que ya no hay problema con el giro de pantalla

Uso de Databinding en lugar de view.findViewById

Podemos usar Databinding para localizar nuestras Views. Tiene por ventaja que se genera de forma dinámica para el Layout específico y es más fácil localizar la view que queremos

Para activar el databinding en gradle:



```
21 minifyEnabled false
22 proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), ''
23 }
24 }
25 }
26 compileOptions {
27     sourceCompatibility JavaVersion.VERSION_1_8
28     targetCompatibility JavaVersion.VERSION_1_8
29 }
30 buildFeatures {
31     viewBinding true
32 }
33
34 dependencies {
35
36     implementation 'androidx.appcompat:appcompat:1.6.1'
37     implementation 'com.google.android.material:material:1.8.0'
38     implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
39     implementation 'androidx.navigation:navigation-fragment:2.5.3'
40     implementation 'androidx.navigation:navigation-ui:2.5.3'
41     testImplementation 'junit:junit:4.13.2'
42     androidTestImplementation 'androidx.test.ext:junit:1.1.5'
43     androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
44 }
```

Observar que es en el build.gradle app y debemos incluir:

```
buildFeatures {
    viewBinding true
}
```

Aprovechamos para activar adicionalmente la navegación en las dependencias:

```
dependencies {
    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.8.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
    implementation 'androidx.navigation:navigation-fragment:2.5.3'
    implementation 'androidx.navigation:navigation-ui:2.5.3'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.5'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
}
```

El import para el databinding es incluyendo el paquete base de nuestro proyecto agregando: databinding.ActivityMainBinding

(el nombre cambiará según el nombre de nuestra activity)

```
import es.iespuertodelacruz.jc.prueba.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {
...
```

Hagamos un ejemplo rápido:

```
import es.iespuertodelacruz.jc.prueba.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        binding.btnPrueba.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                binding.textView.setText("Hola mundo");
            }
        });
    }
}
```



Práctica 28: Crear una nueva app de Hola Mundo usando Databinding. Cuándo se pulse el botón aparecerá: “Soy nombrealumno. Hola mundo! “

Uso de fragments

Los fragments surgieron con la idea de reutilizar trozos de pantalla y código (por ejemplo en una tablet el nav de una aplicación aparece en todas las pantallas). En 2021 se empezó a orientar el desarrollo a una única Activity y que todas las pantallas fueran fragment (evita algunos problemas de rendimiento de la UI en las transiciones de pantallas por ejemplo) La idea es entonces una Activity que dentro tiene un FragmentContainer y es ahí donde se mostrarán los diferentes fragment-pantallas de la app

Empecemos por un ejemplo sencillo, en el que intercambiaremos en una misma activity dos fragment

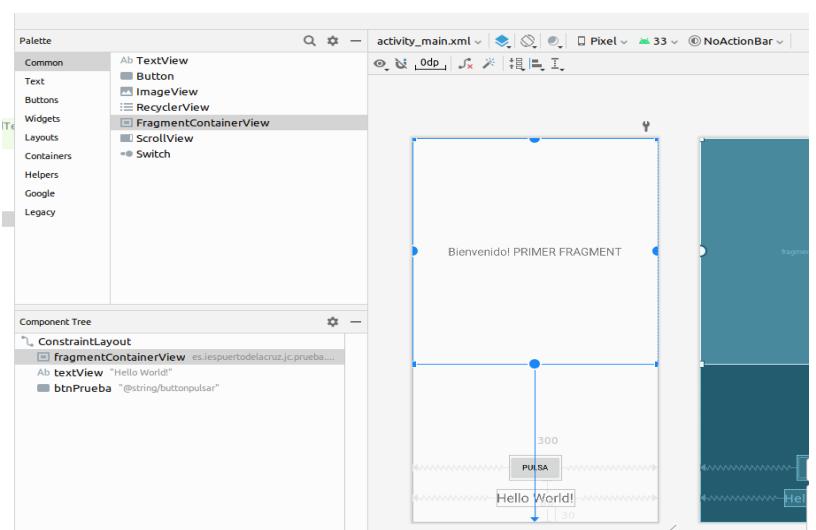
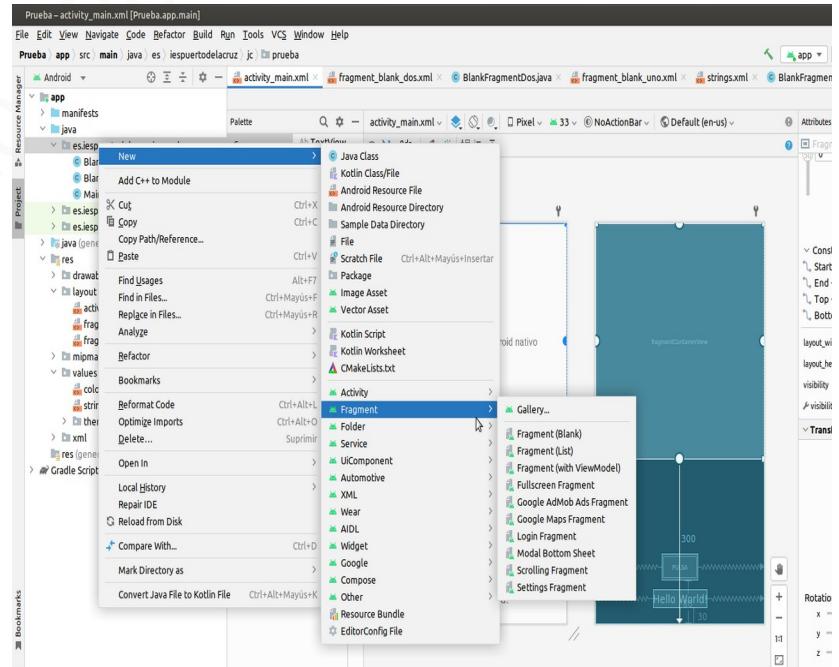
Crearemos un proyecto con un main activity vacío.

En el layout queremos incorporar el objeto: FragmentContainerView pero ese objeto nos solicita que haya ya un fragmento para renderizarlo por defecto. Así que vamos a crear ese objeto

botón derecho → new fragment → fragment(Blank)

Nos pedirá un nombre. Por ejemplo podemos llamarlo: BlankFragmentUno. Nos generará el código necesario y un layout: fragment_blank_uno.xml

Adaptaremos el layout para que tenga un mensaje propio distinguible. Por ejemplo: "bienvenido al primer fragment!!"



Arrastraremos a: activity_main.xml un objeto: FragmentContainerView nos pedirá que asociemos un fragment para que renderize por defecto. Seleccionaremos nuestro fragment

Es muy conveniente que este objeto FragmentContainerView tenga un id. ya que así podemos referenciar con facilidad donde queremos que se muestren los fragment. El propio ide ya propone uno. Veamos un ejemplo del xml que nos genera:

```
<androidx.fragment.app.FragmentContainerView  
    android:id="@+id/fragmentContainerView"  
    android:name="es.iespuertodelacruz.jc.prueba.BlankFragmentUno"  
    android:layout_width="0dp"  
    android:layout_height="0dp"  
    android:layout_marginBottom="300dp"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    tools:layout="@layout/fragment_blank_uno" />
```

Es en ese objeto: FragmentContainerView (que no es más que un Framelayout con características de fragment) donde renderizaremos los fragments que queramos. Inicialmente tomará BlankFragmentUno (así que sabemos que desde el inicio ya tendrá un fragment que se visualice en esa ubicación).

Este tipo de objeto es mejor que las etiquetas: <fragment> que se usaban durante bastante tiempo, ya que permite el reemplazo de los objetos que hemos establecido estáticamente en el xml (**FragmentContainerView** permite reemplazar el fragmento inicial que hemos puesto en el xml por otro en runtime sin problema)

Luego crearemos un segundo fragmento, con un fondo de color diferente para distinguir con facilidad uno de otro.

Para pasar de un fragmento a otro se propone el siguiente código (se presupone que hemos arrastrado un button a nuestro layout activity_main.xml y que tiene por id: **btnPrueba**

Nuestra ActivityMain queda:

```
int contador = 0;
private ActivityMainBinding binding;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());

    binding.btnPrueba.setOnClickListener(new View.OnClickListener() {
        Fragment f = null;
        @Override
        public void onClick(View v) {
            contador++;
            if(contador % 2 == 0){
                f = new BlankFragmentUno();
            }else{
                f = new BlankFragmentDos();
            }

            getSupportFragmentManager()
                .beginTransaction()
                .replace(R.id.fragmentContainerView, f)
                .commit();
        }
    });
}
```

Observar que intercambiamos un fragment por otro mediante: `getSupportFragmentManager()` y que hacemos uso de transacciones

- **Práctica 29:** Crear una activity que incorpore dos fragment que se reemplacen mediante pulsaciones de botón. Una será en modo oscuro y otra en modo día. En ambos casos habrá un input para introducir el nombre de la persona y un botón para que muestre un saludo: “buenos días nombrepersona”

Uso de Databinding con Fragment

Hay un pequeño cambio cuando usamos binding (en lugar de findbyviewid) con fragments:

```
import com.example.myapplication.databinding.FragmentFirstBinding;

public class FirstFragment extends Fragment {

    private FragmentFirstBinding binding;

    @Override
    public View onCreateView(
        LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState
    ) {
        binding = FragmentFirstBinding.inflate(inflater, container, attachToParent: false);
        return binding.getRoot();

    }
}
```

Observar que es muy parecido a con activity. La principal diferencia es que **recibe tres parámetros** el método: inflate() . Comparemos en el caso de activity con el de fragment y lo vemos mejor:

```
//con activity:
binding = ActivityMainBinding.inflate(getApplicationContext());

//con fragment:
binding = FragmentFirstBinding.inflate(inflater, container, false);
```

Adicionalmente en los fragment se devuelve getRoot():

```
return binding.getRoot();
```

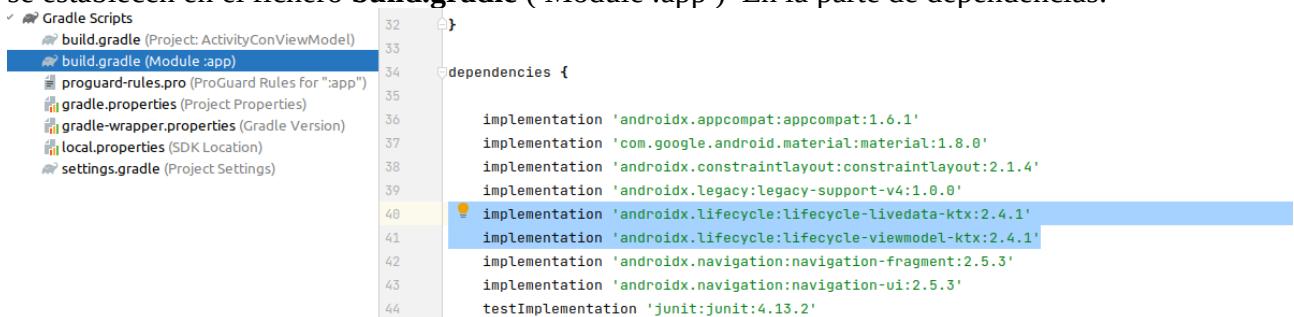
● **Práctica 30:** Hacer la actividad de IMC con dos fragment en lugar de dos activity. Se debe hacer uso de binding en los fragment en lugar de findbyid()

Uso de ViewModels

La solución de Bundle cuando giramos la pantalla para no perder la información tiene problema: Si estás con imágenes de mapa de bits y demás datos pesados, la serialización que implica para luego recuperar es pesado

Ahí es donde entran los ViewModel, que adicionalmente nos ayudan a separar los datos de nuestra interfaz y poder compartir esos datos entre fragments, activities, etc

Nota: Para usar viewmodel hay que tener las dependencias de lifecycle. Tales dependencias se establecen en el fichero **build.gradle** (Module :app) En la parte de dependencias:



```
dependencies {
    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.8.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
    implementation 'androidx.legacy:legacy-support-v4:1.0.0'
    implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.4.1'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1'
    implementation 'androidx.navigation:navigation-fragment:2.5.3'
    implementation 'androidx.navigation:navigation-ui:2.5.3'
    testImplementation 'junit:junit:4.13.2'
}
```

En formato texto las tenemos aquí:

```
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.4.1'  
implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1'
```

Vamos a ver el código del arraylist de números aleatorios usando ViewModel en lugar del bundle y el método onSaveInstanceState()

Debemos crear una nueva clase que extienda de ViewModel. En este caso al ser un viewmodel de ámbito la mainactivity se propone: MainActivityViewModel

```
public class MainActivityViewModel extends ViewModel {
    public ArrayList<Integer> datos;
    public MainActivityViewModel() {
        datos = new ArrayList<>();
    }
}
```

Y ahora en MainActivity:

```

+ usages
MainActivityViewModel viewModel;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(binding.getRoot());
    viewModel = new ViewModelProvider(owner: this).get(MainActivityViewModel.class);
    binding.tvShowNumbers.setText(viewModel.datos.toString());

    binding.btnNumberCreate.setOnClickListener(v -> {
        int aleatorio = (int) (Math.random() * 100);
        viewModel.datos.add(aleatorio);
        binding.tvShowNumbers.setText(viewModel.datos.toString());
    });
}
}

```

Práctica 31: Crear la actividad de arraylist aleatorio usando viewmodel y comprobar que no hay problema al girar la pantalla

Observar que lo determinante está en ViewModelProvider:

```
viewModel = new ViewModelProvider(this).get(MainActivityViewModel.class);
```

Ahí estamos asociando el ViewModel a la activity.

Observar que si hacemos nuestro código con ViewModel podemos separar un poco más la interfaz de los datos que maneja la interfaz. Podrían estar las llamadas a una api, a la base de datos, etc. Y nuestro código es más testeable y mantenable

Adicionalmente hay una gran ventaja si lo usamos asociado a los fragment: podemos tener un espacio común donde compartir información entre todos los fragment. En un modelo de aplicación que hace uso de una única activity, consigues tener un espacio común para todas tus pantallas.

Tomar un ViewModel desde un Fragment

Haremos uso en onCreate() de: ViewModelProvider(requireActivity()):

```
@Override  
public void onCreate(@Nullable Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    mViewModel = new  
    ViewModelProvider(requireActivity()).get(BlankFragmentViewModel.class);  
}
```

onCreate() se ejecuta para la primera ocasión. Así cuando se regresa de background no se vuelve a llamar. Sin embargo onCreateView() y onViewCreated sí son llamados.

Para agregar listener o tomar/poner info en objetos del layout es mejor usar onViewCreated(). En ese momento el layout del fragment ya está creado (el método recibe un View que es ya el fragment inflado)

```
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {  
    super.onViewCreated(view, savedInstanceState);  
  
    TextView txtEstadisticas = (TextView)view.findViewById(R.id.txtEstadisticas);  
}
```

Práctica 32: Crear una app que contabilice el número de letras y palabras en un texto que introduzca el usuario. Esta app tendrá una única Activity y un único fragment, que es el que contendrá la aplicación. La texto que haya en el edittext y los resultados asociados (cantidad de letras y palabras) estarán guardados en el ViewModel
Nota: Hay un wizard específico para crear un fragment con un viewmodel

Compartir un ViewModel entre Fragments

Crearemos un ViewModel y lo enlazaremos a la Activity contenedora de los fragment. Para ello usaremos: requireActivity() que es un método muy usado con fragments para obtener la Activity a la que pertenecen:

```
mViewModel = new ViewModelProvider(requireActivity()).get(ViewModelCompartido.class);
```

En el código anterior, que se ejecuta en el onCreateView() de los fragment se usa requireActivity() para obtener la Activity que será la propietaria. El ViewModel (aquí se le dio el nombre: ViewModelCompartido)

● **Práctica 33:** Modificar la app anterior para que sean dos fragment dentro de la misma activity. En el primero fragment se pone el texto y habrá un botón “guardar” En el segundo se leen los datos del ViewModel compartido y los muestra. Habrá un botón que estará en la activity padre que permitirá cambiar de un fragment a otro
Observar que al pulsar el botón de cambiar y se regrese al fragment donde introducimos el texto se vacía. Tomar la información del ViewModel compartido para que se muestre en el edittext

Navegación

Dependencia Gradle en build.gradle:

```
implementation 'androidx.navigation:navigation-fragment:2.5.3'  
implementation 'androidx.navigation:navigation-ui:2.5.3'
```

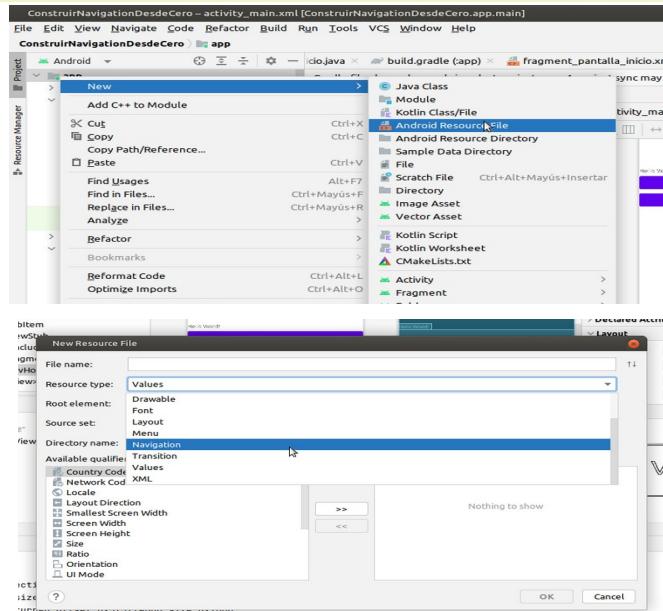
Ahora hay que crear las rutas de navegación:

botón derecho sobre :

app → new → android resource file

En el wizard elegimos en
“Resource type” la opción Navigation

Es habitual poner en “file name” nav_graph

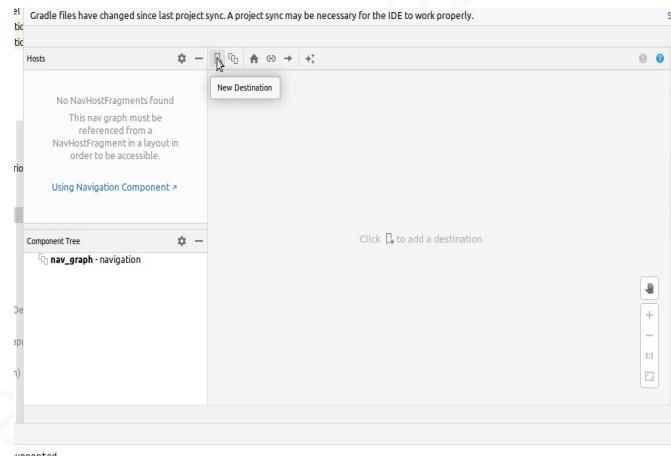


Nos abre el fichero xml en modo diseño y aquí podemos ir creando fragmentos o agregando otros fragmentos o activities ya creados al gestor de navegación

Hay un icono que dice: “**New destination**”

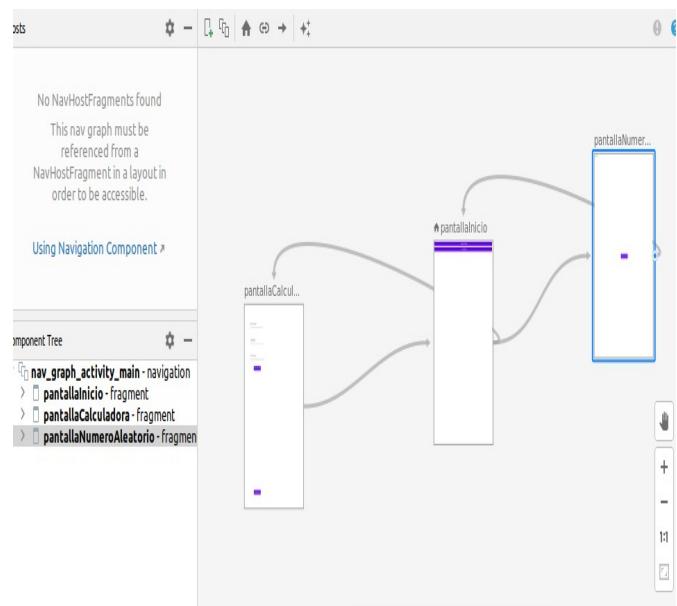
Al pulsarlo nos permite agregar/crear fragmentos

Podemos ir usando el wizard para crear las pantallas que queramos



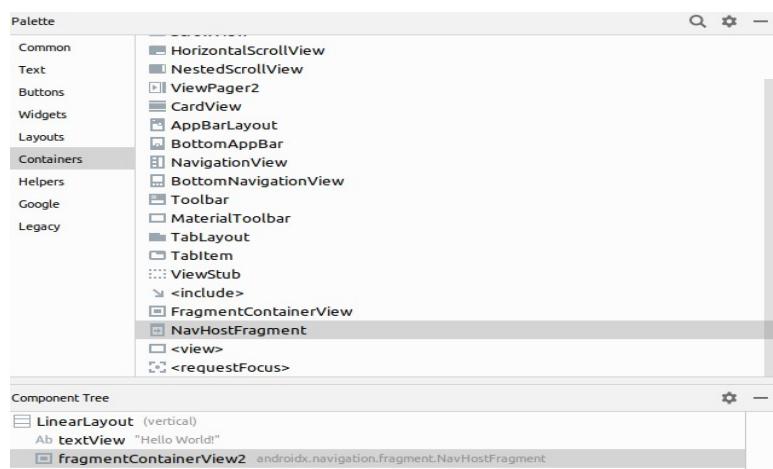
La siguiente pantalla refleja tres fragmentos creados con líneas de navegación

Las líneas de navegación se obtienen arrastrando desde una pantalla origen hacia la pantalla destino

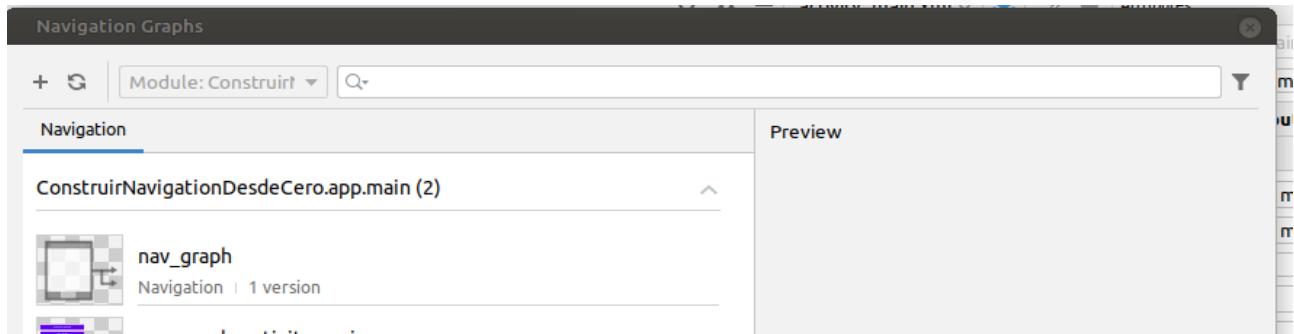


Una vez realizados los fragmentos y puestas las líneas de navegación, pondremos en el layout de la activity el contenedor de fragmentos que necesitamos

Tomamos el elemento que está en:
Containers → NavHostFragment



Al arrastrarlo nos abrirá un popup para que seleccionemos el fichero xml de navegación. Introducimos el nav_graph que creamos antes:



Con lo anterior ya podemos navegar. Imaginemos que estamos en el Listener de un botón del fragment de inicio y queremos ir a un fragment que hemos llamado: PantallaCalculadora, entonces lo que tenemos que hacer es:

```
myButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Navigation.findNavController(view).navigate(R.id.action_pantallaInicio_to_pantallaCalculadora);
    }
});
```

Vemos que usamos: **Navigation.findNavController()**

```
Navigation.findNavController(view).navigate(R.id.action_pantallaInicio_to_pantallaCalculadora);
```

Esa línea está usando el R.id autogenerado en el nav_graph.xml cuando arrastrábamos destinos entre una pantalla y otra. Esos ids autogenerados siempre dicen: action y nombran el origen y el destino de la navegación

Práctica 34: Crearemos una app con una única activity y un navcontroller. Tendremos tres fragment. Uno será la descomposición en números primos. Otro será un contador de palabras y letras de un texto (el edittext y los resultados en el mismo fragment) y un tercero el generador de números aleatorios mostrados en un arraylist. Cada uno de los fragment tendrán dos botones que permiten ir desde allí a los otros dos fragment

RecyclerView

Si hay muchos objetos UI de un mismo tipo, recurrimos a una lista. Cargar esos objetos puede ser costoso. RecyclerView carga solo los objetos en pantalla en un momento dado

Paso 1: Creamos una clase de datos que representa cada elemento de la lista. Por ejemplo si vamos a hacer una lista de personas , creamos una clase Persona. Lo haremos como un POJO-JAVA BEAN (getter, setter, constructor vacío)

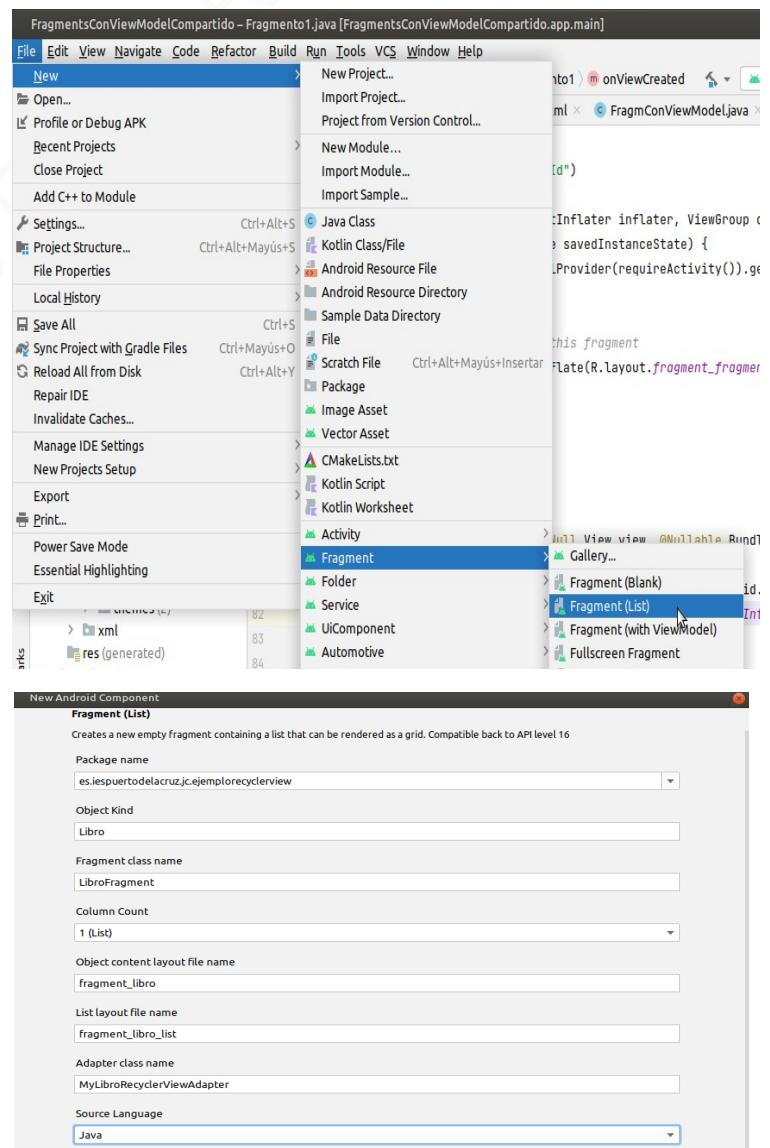
Paso 2: Creamos un fragment de tipo list que nos genera el RecyclerView y lo demás que necesita. Para ello nos apoyaremos en el ide que nos generará parte del código.

File → new → fragment →
→ Fragment(list)

En la ventana que surge reemplazamos item por la clase de modelo que queramos vincular (Libro en el ejemplo)

Paso 3: Enlazamos el fragment generado con un FragmentContainer

Al ejecutar la app aparecerán datos de prueba en el fragment



Práctica 35: Crear una app desde cero para mostrar personas (nombre y edad). En el MainActivity pondremos el FragmentContainer pero para que no se queje de que no tiene fragments, seguir los pasos de arriba (primero creamos la clase Persona como un Java Bean, luego crearemos el FragmentList y finalmente agregaremos el FragmentContainer al layout del ActivityMain y lo enlazaremos al fragmento generado) Al ejecutar la app deben verse datos de prueba

Paso 4: Reemplazar por datos propios:
Borramos los datos de prueba (el ide habrá generado una carpeta y un fichero para eso. Típicamente su nombre es: placeholder)

Al hacerlo el ide mostrará una queja allí donde se están usando.

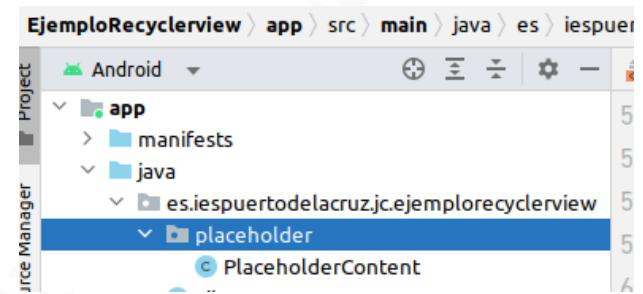
Primero corregiremos el Fragment (posiblemente el nombre sea: PersonaFragment). En el onCreateView vemos que hay una queja en la línea con:

```
recyclerView.setAdapter(new MyPersonasRecyclerViewAdapter(PlaceholderContent.ITEMS));
```

Podemos crear de momento una lista de Personas que le pasemos en lugar del objeto: PlaceholderContent.ITEMS. El siguiente ejemplo es para una lista de libros:

```
// Set the adapter
if (view instanceof RecyclerView) {
    Context context = view.getContext();
    RecyclerView recyclerView = (RecyclerView) view;
    if (mColumnCount <= 1) {
        recyclerView.setLayoutManager(new LinearLayoutManager(context));
    } else {
        recyclerView.setLayoutManager(new GridLayoutManager(context, mColumnCount));
    }
    List<Libro> libros = Arrays.asList(
        new Libro( titulo: "uno" , autor: "unoautor" , portada: null , numeroPaginas: 200),
        new Libro( titulo: "dos" , autor: "dosautor" , portada: null , numeroPaginas: 150)
    );
    recyclerView.setAdapter(new MyLibroRecyclerViewAdapter(libros));
}
return view;
```

Observar que sigue generando error, porque hay que modificar aún el RecyclerViewAdapter.



Paso 5: Vamos al fichero del MyPersonaRecyclerViewAdapter generado y reemplazamos allí donde esté el: PlaceholderItem por Persona (Libro en el ejemplo anterior) . Una vez hecho se reducirán bastante los errores. Se mantendrán en el método: onBindViewHolder() porque allí se usan los atributos del objeto de prueba Placeholder generado por el id. Reemplazamos por los atributos propios de nuestra clase Persona (o Libro en el ejemplo) aquellos que generan error (recordar también quitar los import del Placeholder)

```
@Override  
public void onBindViewHolder(final ViewHolder holder, int position) {  
    holder.mItem = mValues.get(position);  
    holder.mIdView.setText(mValues.get(position).titulo);  
    holder.mContentView.setText(""+mValues.get(position).numeroPaginas);  
}
```

● **Práctica 36:** Seguir los pasos descritos para reemplazar los datos de prueba que generó el ide por unos propios nuestros (una lista de personas)

Paso 6: Vamos a personalizar la presentación de cada item. En los layout nos habrá generado dos ficheros. Uno que representa el layout contenedor de la lista de personas (o libros en el ejemplo) nombre: fragment_persona_list.xml y otro para el layout de cada elemento de la lista individual: fragment_persona.xml Si abrimos ese fichero en mode “Code” vemos que aparece como LinearLayout Podemos modificarlo cómo queramos

● **Práctica 37:** Le pediremos al ide un nuevo fragment list, esta vez para tener una lista de libros Se quiere que de cada libro(titulo:string,autor:string,portada:string,paginas:int) . El layout de cada elemento de la lista sea de tipo constraint
Pondremos un imageview que cargará únicamente el icono de android u otro de la librería para emular la portada del libro (por tanto todos los libros mostrarán la misma portada)

Cargar imágenes en un imageview para el recycler. Glide

Vamos a visualizar imágenes desde url de internet así que tenemos que poner en AndroidManifest.xml el permiso:

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

Utilizaremos una librería llamada Glide: <https://github.com/bumptech/glide> Que nos permite cargar una imagen de la web directamente en un imageview

En la página establecen los pasos actualizados. Debe aparecer en nuestros repositorios del proyecto (fichero **settings.gradle**) **google()** y **mavenCentral()**. Probablemente ya estén en el proyecto cargados.

```
pluginManagement { this: PluginManagementSpec
    repositories { this: RepositoryHandler
        google()
        mavenCentral()
    }
}
```

Luego ponemos la dependencia de Glide en el fichero: **build.gradle** (Module:app)

```
implementation("com.github.bumptech.glide:glide:4.16.0")
```

Glide precisa de un contexto, así que debemos recibir en nuestro RecyclerViewAdapter ese contexto. Eso significa que tenemos que modificar el constructor de nuestro RecyclerViewAdapter y ejecutar Glide en **onBindViewHolder()**

El ejemplo muestra el constructor modificado recibiendo un contexto y luego en el **OnBindViewHolder()** el uso de Glide para tomar con la url la imagen de la web y ponerla en el imageview (**holder.imgPortada**).

```
public MyLibroRecyclerViewAdapter(Context ctx, List<Libro> items) {
    mValues = items;
    this.ctx = ctx;
}

new *

@Override
public void onBindViewHolder(final ViewHolder holder, int position) {
    holder.mItem = mValues.get(position);
    holder.txtAutor.setText(mValues.get(position).autor);
    holder.txtTitulo.setText(mValues.get(position).titulo);
    holder.txtPaginas.setText(""+mValues.get(position).numeroPaginas);
    String imageUrl = mValues.get(position).portada;
    Glide.with(ctx) RequestManager
        .load(imageUrl) RequestBuilder<Drawable>
        .into(holder.imgPortada);
}
```

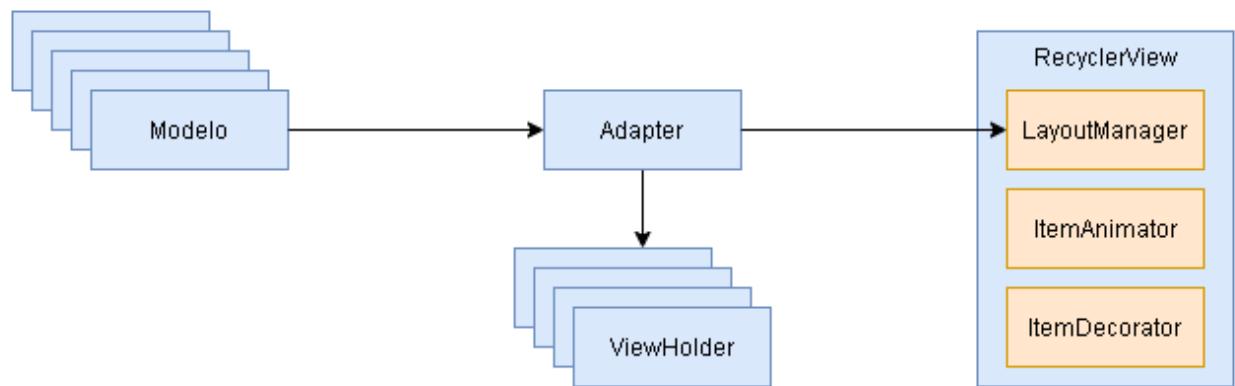
Observar que al cambiar el constructor del adapter, ahora en el fragmento hay que enviar el contexto:

```
recyclerView.setAdapter(new MyLibroRecyclerViewAdapter(view.getContext(), libros));
```



Práctica 38: Modificar la actividad anterior. Ahora reemplazaremos el icono del imageview por una imagen de internet.

Ahora que ya hemos hecho un ejemplo, apoyados por el ide, podemos entender mejor el siguiente gráfico de funcionamiento de recyclerview:



Donde **Modelo** sería nuestra lista de libros. El **Adapter** sería nuestro **MyLigroRecyclerViewAdapter** y el **ViewHolder** es la clase anidada del adapter. Finalmente el objeto **RecyclerView** nos aparece en el layout del fragment que se generó para la lista (`fragment_libro_list.xml`) Allí podemos ver que está declarado el **LayoutManager** (aparece `LinearLayoutManager` en ese atributo)

Los **RecyclerView** admiten bastantes opciones (en lugar de una columna que sea múltiples columnas etc) pero queda como autoformación

Práctica 39: Crear una app que muestre una lista de equipos de fútbol. Donde por cada equipo se muestre su nombre, su escudo y el año de fundación del club

Drawer Navigation

Al crear una activity tenemos un wizard (Navigation Drawer Views) que nos permite crear un Drawer. Si da problemas agregamos en build.gradle (Module:app) en la parte de dependencies:

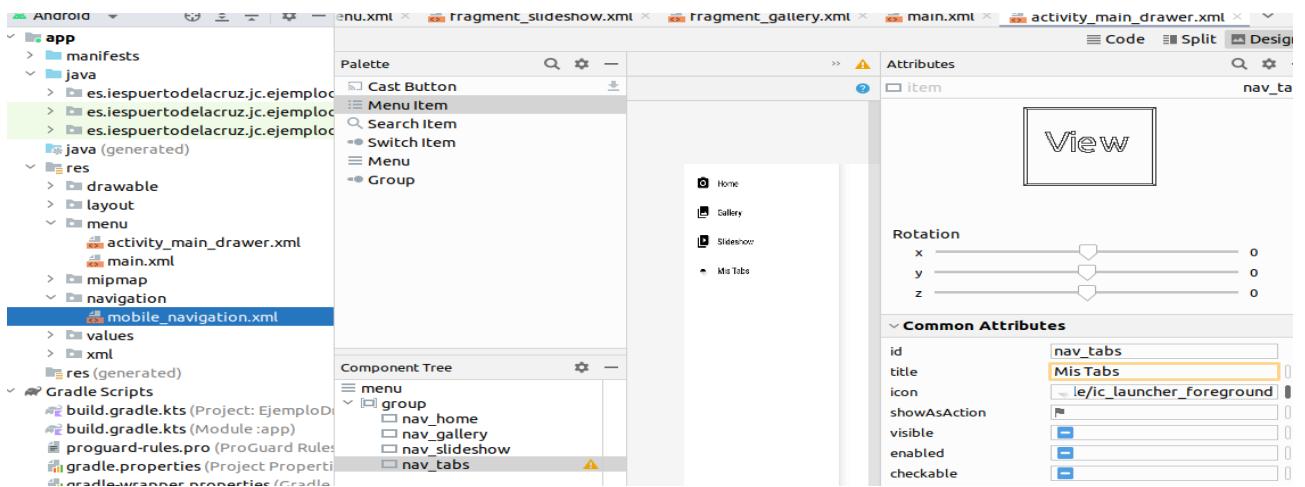
```
constraints {
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.8.0") {
        because("kotlin-stdlib-jdk7 is now a part of kotlin-stdlib")
    }
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.8.0") {
        because("kotlin-stdlib-jdk8 is now a part of kotlin-stdlib")
    }
}
```

Agregar un elemento nuevo es sencillo. Vamos al navigation (mobile_navigation.xml) y desde allí, como con otros navigation, pulsamos en el icono para crear un nuevo destino (new destination) y elegimos “create new destination”. Le ponemos un id al fragment nuevo generado

Luego en el MainActivity agregamos al AppBarConfiguration.Builder el id del fragmento de la navegación que hemos creado

```
mAppBarConfiguration = new AppBarConfiguration.Builder(
    R.id.nav_home, R.id.nav_gallery, R.id.nav_slideshow, R.id.nav_tabs)
    .setOpenableLayout(drawer)
    .build();
```

Finalmente ponemos en: res → menu → activity_main_drawer.xml un nuevo: Menu Item, le damos un título y si se quiere un ícono. Lo importante es que el id apunte al id que establecimos en el navigation.



● **Práctica 40:** Crear una app con Drawer Navigation. Tendrá 4 elementos: home, aboutme, descomposición primos, contador texto calculadora y contador texto corresponden a las ya realizadas. Aboutme hablará de tu curso y nombre

Bottom Tab Navigation anidada en Drawer Navigation

Cuando tenemos la necesidad de submenus, una alternativa recomendable es la de anidar navegaciones. Así por ejemplo para el modelo de una única Activity, podemos poner un Drawer Navigation para las opciones principales a modo de apartados, y al acceder a una de esos apartados que se muestren tabs y el usuario navegue mediante tabs dentro de ese apartado.

Para conseguir lo anterior usaremos dos NavController. Uno para la navegación principal (controlará la navegación en los fragments que se acceden desde Drawer Navigation) y el otro para la navegación secundaria

Por ejemplo, en el Drawer tenemos las opciones: AboutMe, Home, Maths al entrar en la opción de Maths queremos poder elegir entre una calculadora y descomposición en números primos. Entonces el: MathsFragment debe contener otro navcontroller que gestione a las dos tabs: calculator, prime numbers, así que habrá que crear toda una nueva navegación

Primer paso: Creamos la app con drawer

Segundo paso: Creamos un nuevo: Navigation Resource File (botón derecho sobre: res → navigation, y elegir el recurso) Le podemos poner un nombre acorde: maths_navigation.xml

Tercer paso: Creamos desde el resource file los fragments que necesitamos (calculadora y primos) . Les ponemos ids significativos para la navegación: nav_calculadora, nav_primos

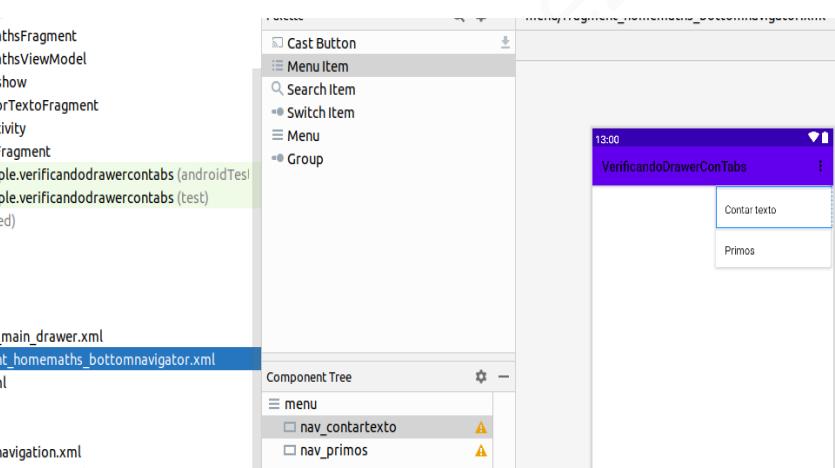
Cuarto paso: Seguimos los pasos que ya conocemos para crear una nueva opción del drawer (o renombramos alguno de los autogeneratedados). El fragment correspondiente lo podemos llamar MathsFragment

Quinto paso: Agregamos un nav_host_fragment_maths al layout de nuestro mathsfragment.

Sexto paso:

Ahora hay que crear un BottomNavigation para tener tabs. Ese objeto precisa de un menú (nombre de cada tab, icono) Así que primero generamos el menu en res → menu → botón derecho → new menu resource file. Esto nos generará un menú. Nosotros le vamos agregando diferentes Menu item. Es importante que cada menu item TENGA UN ID COINCIDENTE CON EL DEL NAVIGATION (en nuestro caso maths_navigation.xml). Se propone un nombre significativo para ese menu: fragment_hOMEMATHS_bottomnavigator.xml

Séptimo paso: Ahora creamos en el layout del fragment maths, en la parte de abajo el BottomNavigation. Este objeto tiene una propiedad: menu Estableceremos en esa propiedad el menu que acabamos de crear



```
<com.google.android.material.bottomnavigation.BottomNavigationView  
    android:id="@+id/bottomNavigationMaths"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center"  
    android:layout_marginLeft="10dp"  
    android:layout_marginRight="10dp"  
    android:layout_marginBottom="10dp"  
    app:menu="@menu/fragment_hOMEMATHS_bottomnavigator" />
```

Finalmente en el MathsFragment ponemos las siguientes líneas (recomendable quitar las opciones de binding porque vamos a acceder a los elementos por id)

```
public View onCreateView(@NotNull LayoutInflater inflater,  
    ViewGroup container, Bundle savedInstanceState) {  
    View view = inflater.inflate(R.layout.fragment_maths, container, false);  
  
    NavHostFragment localNavHost =  
(NavHostFragment) getChildFragmentManager().findFragmentById(R.id.nav_host_fragment_maths);  
    NavController navControllerLocal = localNavHost.getNavController();  
    BottomNavigationView bottomNav = view.findViewById(R.id.bottomNavigationMaths);  
    NavigationUI.setupWithNavController(bottomNav, navControllerLocal);  
  
    return view;  
}
```

Con lo anterior estamos haciendo que el navcontroller de nuestro navigation maths se encargue de la navegación: `NavigationUI.setupWithNavController(bottomNav, navControllerLocal);`

● **Práctica 41:** Crear la app descrita con un Drawer y un Bottom navigation

Persistencia

Acceso a ficheros

En android se diferencia entre la escritura en fichero cuando es en la memoria interna del dispositivo a cuando es en la tarjeta SD (tener en cuenta que muchos dispositivos no disponen de esa tarjeta)

Ficheros en memoria interna

Podemos utilizar las librerías habituales de Java. Lo único a tener en cuenta es el uso de `getFilesDir()` para que nos devuelva el directorio donde tenemos permisos.

`getContext().getFilesDir()` nos da la dirección apropiada.

El ejemplo crea un descriptor `File` para el fichero: `personas.csv` en el directorio con permisos de la aplicación. Escribe una lista csv de personas y luego la recupera del fichero creado mostrando mediante: `Log.d()` cada una de las líneas (visibles en logcat)

```
File file = new File(getContext().getFilesDir(), "personas.csv");
try{
    BufferedWriter bw = Files.newBufferedWriter(file.toPath());
}{

    List<String> listaEscribir = Arrays.asList(
        "ana; pérez; 27",
        "lidia;Hernández; 21",
        "Rigoberto;Afonso;37"
    );
    for( String linea: listaEscribir){
        bw.write(linea);
        bw.newLine();
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}

try{
    BufferedReader br = Files.newBufferedReader(file.toPath());
}{

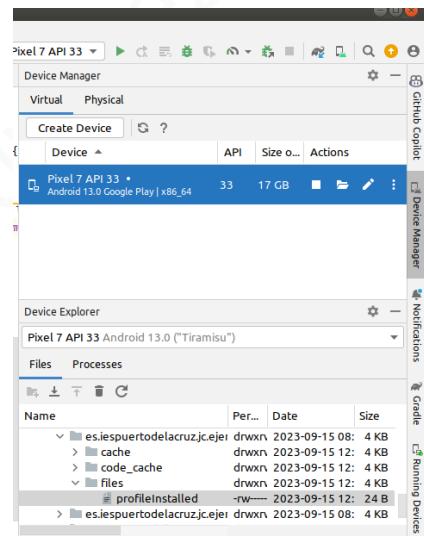
    List<String> listaLeida = br.lines().collect(Collectors.toList());
    listaLeida.forEach( linea -> Log.d( tag: "linea: ", linea));

} catch (IOException e) {
    throw new RuntimeException(e);
}
```

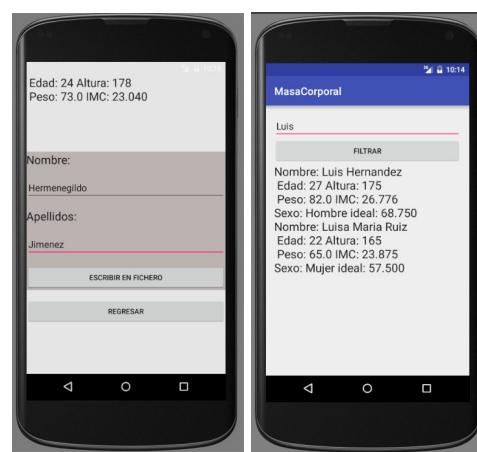
Práctica 42: Crear una app que permita hacer un editor de textos sencillo. El usuario escribe/modifica el contenido de un edittext multiline y luego al pulsar botón se guarda. Hay otro botón que permite recuperar lo guardado

Tenemos acceso a los ficheros de la aplicación en Android Studio en Device Explorer (observar las opciones de la barra lateral derecha)

Para llegar a la ruta donde la aplicación tiene permiso de escritura en el dispositivo android: **data → data → nombre del paquete del proyecto → files**



● **Práctica 43:** Para la aplicación de masa corporal guardar en fichero (agregando cada nueva persona, no borrar anteriores) Luego acceder al fichero desde Android studio para recuperarlo. Agregar una tercera pantalla que permita ver los datos del fichero guardado filtrados por nombre



Room

La librería Room es el ORM que tenemos disponible en Android

Dependencias gradle en: Gradle Scripts → build.gradle(Module:app)

```
dependencies{
    ...
    implementation "androidx.room:room-runtime:2.5.0"
    annotationProcessor "androidx.room:room-compiler:2.5.0"
    androidTestImplementation "androidx.room:room-testing:2.5.0"

    // Lifecycle components
    implementation "androidx.lifecycle:lifecycle-viewmodel:2.5.1" //es posible que sobre
    implementation "androidx.lifecycle:lifecycle-livedata:2.5.1"
    implementation "androidx.lifecycle:lifecycle-common-java8:2.5.1"
    ...
}
```

Estructura de paquetes con room (buenas prácticas)

La documentación oficial de Android nos recomienda:



Clase para la creación/mantenimiento de la DDBB

El siguiente código nos puede valer (observar que hay varias líneas comentadas, que a medida que, hagamos las clases, iremos descomentando)

```
@androidx.room.Database(  
    entities = {  
        //MonedaEntity.class,  
        //HistoricoEntity.class  
    }, version = 1  
    , exportSchema = false  
)  
@TypeConverters({ /*Converters.class */})  
public abstract class DatabaseMonedas extends androidx.room.RoomDatabase{  
    //abstract public MonedaDAO monedaDAO();  
  
    //abstract public HistoricoDAO historicoDAO();  
  
    private static volatile DatabaseMonedas INSTANCE;  
    private static final int NUMBER_OF_THREADS = 4;  
    static final ExecutorService databaseWriteExecutor =  
        Executors.newFixedThreadPool(NUMBER_OF_THREADS);  
  
    public static DatabaseMonedas getDatabase(final Context context) {  
        if (INSTANCE == null) {  
            synchronized (DatabaseMonedas.class) {  
                if (INSTANCE == null) {  
                    INSTANCE = Room.databaseBuilder(context.getApplicationContext(),  
                        DatabaseMonedas.class, "monedas.db")  
                        .allowMainThreadQueries()  
                        .fallbackToDestructiveMigration()  
                        .build();  
                }  
            }  
        }  
        return INSTANCE;  
    }  
}
```

En la anotación: `@androidx.room.Database` declaramos las entities que vayamos a usar (debemos registrar todas las que tengan: `@Entity`) y adicionalmente el número de la versión (para que regenere la base de datos ante un cambio del schema)

`@TypeConverters` lo usaremos para conversión de tipos de datos específicos: Típicamente se usa para convertir un tipo java fecha a algo que maneje la base de datos sqlite (un número estilo unix epoch por ejemplo)

Las clases dao, también las registramos aquí (observar que son abstractas): `abstract public MonedaDAO monedaDAO();`

Por lo demás, el código es básicamente la creación de un Singleton para acceder a la base de datos con protección ante concurrencia mediante el uso de synchronize()

Estableceremos aquí el nombre de la base de datos. Por ejemplo: "monedas.db"

Creación de las Entities

Estamos ante una base de datos local, que únicamente es accesible desde el propio dispositivo. La creación de las tablas, nos la generará Room con la declaración de nuestras Entity. Eso implica conocer algunas anotaciones.

Ejemplo MonedaEntity, que registra el nombre de una moneda y su país: (no se muestran los getter y los setter)

```
@Entity(tableName = MonedaEntity.TABLE_NAME)
public class MonedaEntity {
    public static final String TABLE_NAME = "monedas";
    public static final String ID = "id";
    public static final String NOMBRE = "nombre";
    public static final String PAIS = "pais";

    @PrimaryKey(autoGenerate = true)
    @ColumnInfo(name = ID)
    public Long id;

    @ColumnInfo(name = NOMBRE)
    public String nombre;

    @ColumnInfo(name = PAIS)
    public String pais;
}
```

Los `@ColumnInfo` son únicamente necesarios si vamos a poner un nombre diferente en el nombre de la columna en la tabla de la base de datos que en el nombre de nuestro atributos

Mediante `tableName` decimos el nombre que queremos que tenga la tabla en la base de datos

Para declarar una clave primaria autoincremental: `@PrimaryKey(autoGenerate = true)`

Relaciones con foreign key

Veamos el caso de los valores históricos de una moneda (con referencia a una fija, por ejemplo el euro). En el modelo relacional, sabemos que hay que registrar una foreign key para moneda. Ejemplo:

```
@Entity(tableName = HistoricoEntity.TABLE_NAME,
    foreignKeys = @ForeignKey(entity = MonedaEntity.class,
        parentColumns = MonedaEntity.ID,
        childColumns = HistoricoEntity.ID_MONEDA
    )
)
public class HistoricoEntity {
    public static final String TABLE_NAME = "historico";
    public static final String ID = "id";
    public static final String ID_MONEDA = "id_moneda";
    public static final String VALOR = "valor";
    public static final String FECHA = "fecha";

    @PrimaryKey(autoGenerate = true)
    public Long id;

    @ColumnInfo(name = ID_MONEDA, index = true)
    public Long id_moneda;

    @ColumnInfo(name = VALOR)
    public Double valor;

    @ColumnInfo(name = FECHA)
    public Date fecha;
}
```

Para decirle que debe haber una restricción de clave foránea en la tabla lo hacemos mediante:

```
foreignKeys = @ForeignKey(entity = MonedaEntity.class,
    parentColumns = MonedaEntity.ID,
    childColumns = HistoricoEntity.ID_MONEDA
)
```

donde:

entity hace referencia a la entity de la que procede la clave foránea.

parentColumns es el nombre del campo en la tabla de procedencia

childColumns es el nombre del campo foráneo en la tabla de nuestra entity actual

También vemos que en el **ColumnInfo** de moneda le pedimos que nos ponga un índice:
@ColumnInfo(name = ID_MONEDA, index = true)
para que las consultas por foreign key sean más rápidas

Si compiláramos ahora, entre los errores que nos mostraría diría:
error: Cannot figure out how to save this field into database. You can consider adding a type converter for it.
public Date fecha;

Sqlite no entiende los tipos de datos Date. Le tenemos que decir que lo convierta, por ejemplo, en un número que refleje unix epoch. Para eso está la anotación: @typeconverter

Anotación @TypeConverter

Cuando tengamos un tipo de datos personalizado debemos decirle a room como queremos que nos lo transforme. Veamos un ejemplo para convertir un Date en java en un Long en la base de datos y viceversa:

```
public class Converters{  
    @TypeConverter  
    public static Date fromTimestamp(Long value) {  
        return value == null ? null : new Date(value);  
    }  
  
    @TypeConverter  
    public static Long dateToTimestamp(Date date) {  
        return date == null ? null : date.getTime();  
    }  
}
```

Cuando Room se encuentre un tipo de dato Date, sabrá así que debe guardarlo como un Long

Observar que ahora nuestra clase: DatabaseMonedas tiene esta forma en su declaración:

```
@androidx.room.Database(  
    entities = {  
        MonedaEntity.class,  
        HistoricoEntity.class  
    }, version = 1  
)  
@TypeConverters({ Converters.class })  
public abstract class DatabaseMonedas extends androidx.room.RoomDatabase{
```

Le hemos informado de nuestros tipos de datos específicos para conversión

Creación de los DAO

Veamos un ejemplo para MonedaDAO y explicamos:

```
@Dao
public abstract class MonedaDAO {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    public abstract Long insert(MonedaEntity monedaEntity);

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    public abstract void insertAll(List<MonedaEntity> monedasEntity);

    @Query("SELECT * FROM " + MonedaEntity.TABLE_NAME + " WHERE " + MonedaEntity.ID + " = :id")
    public abstract MonedaEntity getMonedaById(Long id);

    @Query("SELECT * FROM " + MonedaEntity.TABLE_NAME)
    public abstract List<MonedaEntity> getAll();

    @Delete
    public abstract void delete(MonedaEntity monedaEntity);

    @Query("DELETE FROM " + MonedaEntity.TABLE_NAME + " WHERE " + MonedaEntity.ID + " = :id")
    public abstract void delete(Long id);

    /*
    @Query("DELETE FROM " + MonedaEntity.TABLE_NAME)
    public abstract void deleteAll();
    */

    @Query("SELECT * FROM " + MonedaEntity.TABLE_NAME + " WHERE " + MonedaEntity.NOMBRE + " LIKE :nombre")
    public abstract List<MonedaEntity> getMonedasByNombre(String nombre);

    @Query("DELETE FROM " + HistoricoEntity.TABLE_NAME + " WHERE " + HistoricoEntity.ID_MONEDA + " = :id")
    public abstract void deleteHistoricoByIdMoneda(Long id);

    @Transaction
    public void deleteWithHistorico(Long id){
        deleteHistoricoByIdMoneda(id);
        delete(id);
    }

    @Query("UPDATE monedas SET nombre = :nombre, pais = :pais WHERE id = :id")
    public abstract void update(Long id, String nombre, String pais);

    @Update
    public abstract void update(MonedaEntity monedaEntity);

}
```

Lo primero que observamos es que debe estar anotado con: `@Dao`

Fijarse que estamos usando clases abstractas. También podríamos usar un interfaz (y así no tenemos que ponerle a los métodos el: `public abstract` que aquí estamos poniendo en todos)

Pero hacerlo como una clase abstracta nos permite crear métodos protegidos por una transacción que hagan varias ejecuciones en la DDBB

Podemos observar que las anotaciones `@Insert`, `@Update`, `@Delete` directamente resuelven el objeto entity a su correspondiente acción en la base de datos.

Si queremos personalizar cualquier consulta (select, delete, update,...) lo haremos con la anotación: `@Query` En el código de ejemplo observamos varios casos, como un borrado por id, etc

Y ahora nos fijaremos en la transacción:

```
@Transaction  
public void deleteWithHistorico(Long id){  
    deleteHistoricoByIdMoneda(id);  
    delete(id);  
}
```

Observar que en este caso no hemos hecho un método abstracto ni le hemos asociado una query. Lo que hace es proteger en una única transacción el borrado (no se está usando borrado en cascada, que también Room lo permite)

Primero se borran todos los históricos y después ya se borra el objeto moneda. Si algo fallara, se revertería todo

Uso de Repositorio

Imaginemos que construimos una clase: MonedaRepository que incluye un método: getAll() Quizás no le veamos interés si lo único que hace es llamar al DAO:

```
public List<MonedaEntity> getAll() {  
    return monedaDAO.getAll();  
}
```

Pero si en ese método primero hacemos la consulta a una API en red y si no tenemos respuesta tomamos la información de la base de datos local ?

Ese comportamiento nos permitiría tener la aplicación funcionando sin tanta dependencia de la red. Ese es uno de los motivos para introducir la capa repositorio

Veamos un ejemplo de un repositorio sencillo para monedas (de momento sin consultas a APIs remotas):

```

public class MonedaRepository {
    private MonedaDAO monedaDAO;
    public MonedaRepository(Application application){
        DatabaseMonedas database = DatabaseMonedas.getDatabase(application);
        monedaDAO = database.monedaDAO();
    }

    public void insert(MonedaEntity monedaEntity){
        new insertAsyncTask(monedaDAO).execute(monedaEntity);
    }

    private static class insertAsyncTask extends android.os.AsyncTask<MonedaEntity, Void, Void> {

        private MonedaDAO mAsyncTaskDao;

        insertAsyncTask(MonedaDAO dao) {
            mAsyncTaskDao = dao;
        }

        @Override
        protected Void doInBackground(final MonedaEntity... params) {
            mAsyncTaskDao.insert(params[0]);
            return null;
        }
    }

    public LiveData<List<MonedaEntity>> getAll() {
        return monedaDAO.getAll();
    }
}

```

El uso de Room (nuestra clase singleton DatabaseMonedas) implica que tenemos que tener un contexto, así que necesitamos recibirla en el constructor de nuestro repositorio:

```

public MonedaRepository(Application application){
    DatabaseMonedas database = DatabaseMonedas.getDatabase(application);
    monedaDAO = database.monedaDAO();
}

```

Para obtener todas las monedas no estamos devolviendo una lista sino un LiveData:

```

public LiveData<List<MonedaEntity>> getAll() {
    return monedaDAO.getAll();
}

```

Eso lo hacemos para poder hacer consultas asíncronas a la base de datos (puede ralentizar la aplicación, así que si las hacemos asíncronas, no nos quedaremos esperando a la respuesta, sino que seguirá la ejecución de la aplicación normal y cuando termine actualizaremos los datos) Es en esto en lo que nos es útil LiveData. De hecho se queda escuchando por cambios en la DDBB y actualiza la información cuando corresponde

Para el uso de LiveData en el repositorio hay que cambiar los Dao que ahora nos queda:

```
@Dao
public abstract class MonedaDAO {

    @Query("SELECT * FROM " + MonedaEntity.TABLE_NAME)
    public abstract LiveData<List<MonedaEntity>> getAll();

    ...
}
```

Observar que eso implica que Room ya nos realiza el asincronismo . Nos devuelve un LiveData que disparará un aviso cuando haya cambios en la consulta en la base de datos, y así nosotros actualizar la UI de nuestra aplicación

● **Práctica 44:** Crear una app para guardar datos de personas (una única tabla) y luego recuperar en otra pantalla todos los datos de la tabla

● **Práctica 45:** Crear una app para guardar y leer la información de categorías y libros (relación 1:N) habrá una pantalla para guardar libro, otra para guardar categoría y una pantalla para recuperar libros (mostrando su categoría)

● **Práctica 46:** Crear una app para guardar propietarios y casas (relación N:M) donde se guardará la info de la fecha de compra (atributo propio de la relación) Una pantalla permite crear propietarios, otra crear casas y una tercera permite vincular propietario con casa con la fecha. Habrá otra pantalla que permite visualizar los propietarios con sus casas

Uso de AsyncTask en consultas asíncronas de inserción y modificación

Antes se vio el uso de LiveData, para recibir los resultados de una query de forma asíncrona.

Las consultas de inserción y modificación también pueden ser costosas. Pero no disparan un aviso de cambio, simplemente devuelven la cantidad de filas afectadas, o el autoincremental de una inserción.

Veamos el código y veamos lo más relevante:

```
public void insert(MonedaEntity monedaEntity){  
    new insertAsyncTask(monedaDAO).execute(monedaEntity);  
}  
  
private static class insertAsyncTask extends android.os.AsyncTask<MonedaEntity, Void, Void> {  
  
    private MonedaDAO mAsyncTaskDao;  
  
    insertAsyncTask(MonedaDAO dao) {  
        mAsyncTaskDao = dao;  
    }  
  
    @Override  
    protected Void doInBackground(final MonedaEntity... params) {  
        mAsyncTaskDao.insert(params[0]);  
        return null;  
    }  
}
```

la orden: **execute()** de AsyncTask ejecuta lo que se haya establecido en el método: **doInBackground()** de forma asíncrona y lo que recibe como parámetro en la posición cero **doInBackground()** es lo que envía **execute()** En este caso **execute()** envía **monedaEntity** y por tanto se recibe en: **params[0]** ese dato



Práctica 47: Modificar la app anterior de personas para que se haga con async task los guardados. Cuando se visualizan las personas y se pulsa sobre una de ellas que abra una pantalla de edición de la persona y se pueda hacer el update en la base de datos con asynctask

Ejecutando los métodos del repositorio en AndroidViewModel

Como sabemos, debemos hacer uso de ViewModel para mantener los datos que maneja la UI. Ahora bien, como queremos pasar a Room el contexto, hemos hecho un repositorio que precisa de la información de contexto:

```
public class MonedaRepository {  
    private MonedaDAO monedaDAO;  
    public MonedaRepository(Application application){  
        DatabaseMonedas database = DatabaseMonedas.getDatabase(application);  
        monedaDAO = database.monedaDAO();  
    }  
}
```

Eso hace que no podamos usar un ViewModel normal. Necesitamos uno que maneje la información de contexto. Para eso podemos usar: **AndroidViewModel**, que es un ViewModel (extiende de la clase ViewModel) que maneja contexto

Veamos un AndroidViewModel posible:

```
public class FirstFragmentViewModel extends AndroidViewModel {  
    public FirstFragmentViewModel(Application application){  
        super(application);  
        monedaRepository = new MonedaRepository(application);  
    }  
    private MonedaRepository monedaRepository;  
  
    public LiveData<List<MonedaEntity>> getLiveDataMonedas() {  
        return monedaRepository.getAll();  
    }  
}
```

Ahora **desde un fragmento o una activity** haríamos:

```
viewModel= new ViewModelProvider(this).get(FirstFragmentViewModel.class);  
viewModel.getLiveDataMonedas().observe(getViewLifecycleOwner(), monedas -> {  
    binding.tvMonedas.setText(monedas.toString());  
});
```

Lo primero destacable es que con esta instrucción estamos obteniendo el Singleton del ViewModel y le estamos pasando el contexto:

```
new ViewModelProvider(this).get(FirstFragmentViewModel.class);
```

Lo siguiente que vemos es que los Observers (el LiveData devuelve un observer) recibe un método para el Callback (cuando termina la ejecución asíncrona) Ese método lo hemos destacado en amarillo:

```
viewModel.getLiveDataMonedas().observe(getViewLifecycleOwner(), monedas -> {  
    binding.tvMonedas.setText(monedas.toString());  
});
```

En este caso, está poniendo en un TextView llamado: tvMonedas la lista de monedas recibida



Práctica 48: Modificar la app de categorías y libros para que se use viewmodel con uso de repositorios

Accediendo a un servicio REST con Retrofit

En el fichero de dependencias de gradle:

```
// Retrofit  
implementation "com.squareup.retrofit2:retrofit:2.9.0"  
//implementation "com.squareup.retrofit2:converter-jackson:2.9.0"  
//implementation "com.fasterxml.jackson.core:jackson-databind:2.12.5"  
implementation "com.google.code.gson:gson:2.8.8"  
implementation "com.squareup.retrofit2:converter-gson:2.9.0"
```

Hay que dar permiso en **manifests.AndroidManifest.xml** acceso a internet:

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

Y si nuestra api está sobre http en lugar de https también agregar:

```
android:usesCleartextTraffic="true"
```

Quedando el manifiesto así:

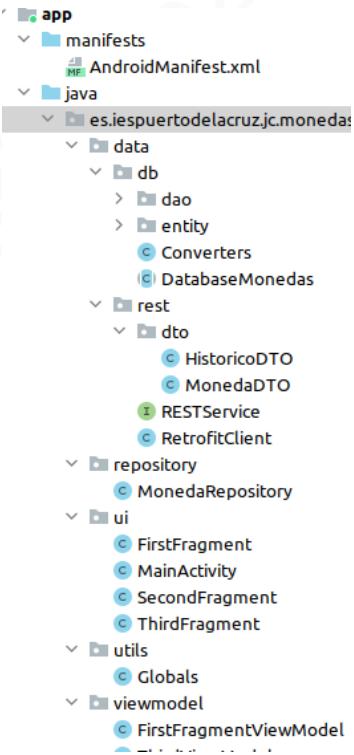
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools">  
    <uses-permission android:name="android.permission.INTERNET"></uses-permission>  
    <application  
        android:usesCleartextTraffic="true"
```

Haremos una estructura de paquetes similar a la siguiente:

Como ahora hay dos accesos a datos crearemos un package data y dentro estará lo que teníamos para acceder a la base de datos Room (package db) y otro package para la comunicación rest con retrofit: package rest

Crearemos un package para los dto (o con más granularidad podemos subdividir los dto en request y response)

Un package utils para poner utilidades como la clase de nombres constantes: Globals.java (ahí pondremos la url de la api, por ejemplo)



Para trabajar con Retrofit, primero generaremos los DTO haciendo uso del json que devuelva la api y la página: (Para tomar un json y obtener el equivalente en java)

<https://www.jsonschema2pojo.org/>

Tener en cuenta que estamos usando gson para marcarlo en la página

Nos quedarán unos DTO parecidos a estos:

```
public class MonedaDTO {  
    @SerializedName("idmoneda")  
    @Expose  
    private Long id;  
    @SerializedName("nombre")  
    @Expose  
    private String nombre;  
    @SerializedName("pais")  
    @Expose  
    private String pais;  
    @SerializedName("historicos")  
    @Expose  
    private List<HistoricoDTO> historicos;  
    ...  
}
```

Después crearemos un fichero con un INTERFACE con las rutas que vamos a consultar en el servicio dentro del package: data.rest

```
public interface RESTService {  
    @GET("v1/monedas")  
    Call<List<MonedaDTO>> doGetMonedasDTO();  
}
```

Observar que le estamos dando una ruta relativa (no aparece la ruta base de la api)

Ahora crearemos un Singleton para controlar retrofit en el package: data.rest

```
public class RetrofitClient {  
  
    private Retrofit retrofit = null;  
    private static RetrofitClient instance = null;  
    private RESTService restService;  
  
    private RetrofitClient() {  
  
        retrofit = new Retrofit.Builder()  
            .baseUrl(Globals.API_MONEDAS_URL_BASE)  
            .addConverterFactory(GsonConverterFactory.create())  
            .build();  
        restService = retrofit.create(RESTService.class);  
    }  
    public static synchronized RetrofitClient getInstance() {  
        if (instance == null) {  
            instance = new RetrofitClient();  
        }  
        return instance;  
    }  
  
    public RESTService getRestService() {  
        return restService;  
    }  
}
```

Observar: `baseUrl(Globals.API_MONEDAS_URL_BASE)` ahí es donde ponemos la ruta base **Tener en cuenta que se ejecutará en un emulador la ruta base no puede ser localhost**, sino la ruta donde esté nuestra api

Ahora, veamos los cambios en el repositorio: MonedaRepository:

```
public class MonedaRepository {
    private MonedaDAO monedaDAO;

    RESTService restService;
    private DatabaseMonedas database;

    public MonedaRepository(Application application){
        database = DatabaseMonedas.getDatabase(application);
        monedaDAO = database.monedaDAO();

        restService = RetrofitClient.getInstance().getRestService();
    }

    public LiveData<List<MonedaEntity>> getLiveDataMonedas() {
        MutableLiveData<List<MonedaEntity>> mutableMonedas = new MutableLiveData<>();
        Call<List<MonedaDTO>> callMonedas = restService.doGetMonedasDTO();
        callMonedas.enqueue(new retrofit2.Callback<List<MonedaDTO>>() {
            @Override
            public void onResponse(Call<List<MonedaDTO>> call,
retrofit2.Response<List<MonedaDTO>> response) {
                if(response.isSuccessful()) {
                    List<MonedaDTO> monedas = response.body();
                    ArrayList<MonedaEntity> monedasEntityList = new ArrayList<>();
                    for (MonedaDTO monedaDTO : monedas) {
                        MonedaEntity monedaEntity = new MonedaEntity();
                        monedaEntity.setId(monedasDTO.getId());
                        monedaEntity.setNombre(monedasDTO.getNombre());
                        monedaEntity.setPais(monedasDTO.getPais());
                        monedasEntityList.add(monedaEntity);
                    }
                    mutableMonedas.setValue(monedasEntityList);
                }
            }
            @Override
            public void onFailure(Call<List<MonedaDTO>> call, Throwable t) {
                System.out.println("Error en la llamada");
                System.out.println(t.getMessage());
            }
        });
        return mutableMonedas;
    }
}
```

Veamos lo destacable:

El constructor toma el servicio para atacar la api:

```
public MonedaRepository(Application application){  
    database = DatabaseMonedas.getDatabase(application);  
    monedaDAO = database.monedaDAO();  
  
    restService = RetrofitClient.getInstance().getRestService();  
}  
}
```

El método getLiveDataMonedas() es el que tiene más peculiaridades. Volvamos a verlo:

```
public LiveData<List<MonedaEntity>> getLiveDataMonedas() {  
    MutableLiveData<List<MonedaEntity>> mutableMonedas = new MutableLiveData<>();  
  
    Call<List<MonedaDTO>> callMonedas = restService.doGetMonedasDTO();  
  
    callMonedas.enqueue(new retrofit2.Callback<List<MonedaDTO>>() {  
        @Override  
        public void onResponse(Call<List<MonedaDTO>> call,  
retrofit2.Response<List<MonedaDTO>> response) {  
            if(response.isSuccessful()) {  
                List<MonedaDTO> monedas = response.body();  
                ArrayList<MonedaEntity> monedasEntityList = new ArrayList<>();  
                for (MonedaDTO monedaDTO : monedas) {  
                    MonedaEntity monedaEntity = new MonedaEntity();  
                    monedaEntity.setId(monedaDTO.getId());  
                    monedaEntity.setNombre(monedaDTO.getNombre());  
                    monedaEntity.setPais(monedaDTO.getPais());  
                    monedasEntityList.add(monedaEntity);  
                }  
  
                mutableMonedas.setValue(monedasEntityList);  
            }  
        }  
        @Override  
        public void onFailure(Call<List<MonedaDTO>> call, Throwable t) {  
            System.out.println("Error en la llamada");  
            System.out.println(t.getMessage());  
        }  
    });  
    return mutableMonedas;  
}
```

Si nos fijamos ese método devuelve un livedata:

```
public LiveData<List<MonedaEntity>> getLiveDataMonedas()
```

Sin embargo lo que devuelve en su interior es un: MutableLiveData (que es un LiveData que permite cambios):

```

public LiveData<List<MonedaEntity>> getLiveDataMonedas() {
    MutableLiveData<List<MonedaEntity>> mutableMonedas = new MutableLiveData<>();
    // hemos borrado código para verlo mejor
    return mutableMonedas;
}

```

La llamada al interface donde declaramos las consultas a la api devuelve un Call:

```
Call<List<MonedaDTO>> callMonedas = restService doGetMonedasDTO();
```

Este tipo de datos al estar pensados para asincronía tendrá un callback para cuando hay éxito y otro cuando hay error (métodos onResponse() y onFailure())

```

callMonedas.enqueue(new retrofit2.Callback<List<MonedaDTO>>() {
    @Override
    public void onResponse(Call<List<MonedaDTO>> call,
retrofit2.Response<List<MonedaDTO>> response) {

        // se ejecuta este código si hay éxito
    }

    @Override
    public void onFailure(Call<List<MonedaDTO>> call, Throwable t) {
        // se ejecuta este otro si hay fallo
    }
});
```

Veamos ahora el código de si hay éxito con detalle:

```

@Override
public void onResponse(Call<List<MonedaDTO>> call,
retrofit2.Response<List<MonedaDTO>> response) {
    if(response.isSuccessful()) {
        List<MonedaDTO> monedas = response.body();
        ArrayList<MonedaEntity> monedasEntityList = new ArrayList<>();
        for (MonedaDTO monedaDTO : monedas) {
            MonedaEntity monedaEntity = new MonedaEntity();
            monedaEntity.setId(monedaDTO.getId());
            monedaEntity.setNombre(monedaDTO.getNombre());
            monedaEntity.setPais(monedaDTO.getPais());
            monedasEntityList.add(monedaEntity);
        }
        mutableMonedas.setValue(monedasEntityList);
    }
}
```

Vemos que convierte entre el DTO a la Entity, ya que es lo que usábamos hasta ahora en nuestra aplicación y así no se modifica el resto. Y finalmente se establece lo recibido en el mutable(recordar que mutable es un livedata)

```
mutableMonedas.setValue(monedasEntityList);
```

Repositorio accediendo a API y en caso de fallo a DDBB Local

La documentación oficial ha dejado unos recursos específicos para este caso que copiaremos en nuestro proyecto:

NetworkBoundResource, Resource, Status

```
public abstract class NetworkBoundResource<ResultType, RequestType> {
    private final MediatorLiveData<Resource<ResultType>> result = new MediatorLiveData<>();

    @MainThread
    public NetworkBoundResource() {
        result.setValue(Resource.loading(null));
        LiveData<ResultType> dbSource = loadFromDb();
        result.addSource(dbSource, data -> {
            result.removeSource(dbSource);
            if (shouldFetch(data)) {
                fetchFromNetwork(dbSource);
            } else {
                result.addSource(dbSource, newData -> result.setValue(Resource.success(newData)));
            }
        });
    }

    private void fetchFromNetwork(final LiveData<ResultType> dbSource) {
        result.addSource(dbSource, newData -> result.setValue(Resource.loading(newData)));
        createCall().enqueue(new Callback<RequestType>() {
            @Override
            public void onResponse(Call<RequestType> call, Response<RequestType> response) {
                result.removeSource(dbSource);
                saveResultAndRelInit(response.body());
            }

            @Override
            public void onFailure(Call<RequestType> call, Throwable t) {
                onFetchFailed();
                result.removeSource(dbSource);
                result.addSource(dbSource, newData -> result.setValue(Resource.error(t.getMessage(),
newData)));
            }
        });
    }

    @MainThread
    private void saveResultAndRelInit(RequestType response) {

        new AsyncTask<Void, Void, Void>() {

            @Override
            protected Void doInBackground(Void... voids) {
                saveCallResult(response);
                return null;
            }
        };
    }
}
```

```

@Override
protected void onPostExecute(Void aVoid) {
    result.addSource(loadFromDb(), newData -> result.setValue(Resource.success(newData)));
}
}.execute();

@WorkerThread
protected abstract void saveCallResult(@NonNull RequestType item);

@MainThread
protected boolean shouldFetch(@Nullable ResultType data) {
    return true;
}

@NonNull
@MainThread
protected abstract LiveData<ResultType> loadFromDb();

@NonNull
@MainThread
protected abstract Call<RequestType> createCall();

@MainThread
protected void onFetchFailed() {
}

public final LiveData<Resource<ResultType>> getAsLiveData() {
    return result;
}
}

```

```

public class Resource<T> {
    @NonNull
    public final Status status;
    @Nullable
    public final T data;
    @Nullable public final String message;
    private Resource(@NonNull Status status, @Nullable T data, @Nullable String message) {
        this.status = status;
        this.data = data;
        this.message = message;
    }

    public static <T> Resource<T> success(@NonNull T data) {
        return new Resource<>(SUCCESS, data, null);
    }

    public static <T> Resource<T> error(String msg, @Nullable T data) {
        return new Resource<>(ERROR, data, msg);
    }

    public static <T> Resource<T> loading(@Nullable T data) {
        return new Resource<>(LOADING, data, null);
    }
}

```

```
public enum Status { SUCCESS, ERROR, LOADING }
```

Ahora en el repositorio, cuando queramos obtener la lista de monedas el método cambia bastante:

```
public LiveData<Resource<List<MonedaEntity>>> getLiveDataMonedas() {  
    return new NetworkBoundResource<List<MonedaEntity>, List<MonedaDTO>>() {  
  
        @Override  
        protected void saveCallResult(@NonNull List<MonedaDTO> monedasDTO) {  
            //este método se ejecuta adicionalmente cuando hay éxito en la consulta a la api remota  
            //así que se ejecuta justo después de createCall()  
            //y guarda la información en la base de datos local para futuras consultas  
            ArrayList<MonedaEntity> monedasEntityList = new ArrayList<>();  
            for (MonedaDTO monedaDTO : monedasDTO) {  
                MonedaEntity monedaEntity = new MonedaEntity();  
                monedaEntity.setId(monedasDTO.getId());  
                monedaEntity.setNombre(monedasDTO.getNombre());  
                monedaEntity.setPais(monedasDTO.getPais());  
                monedasEntityList.add(monedaEntity);  
            }  
            monedaDAO.insertAll(monedasEntityList);  
        }  
  
        @NonNull  
        @Override  
        protected LiveData<List<MonedaEntity>> loadFromDb() {  
            //se acude a este método en caso de fallo en la consulta a la api remota y  
            //devuelve la lista de monedas de la base de datos local  
            return monedaDAO.getAll();  
        }  
  
        @NonNull  
        @Override  
        protected Call<List<MonedaDTO>> createCall() {  
            //Devuelve la lista de monedas de la base de la api remota.  
            //primero se ejecuta este método, si falla, se ejecuta el método loadFromDb.  
            //si tiene éxito, se ejecuta el método saveCallResult  
  
            return restService.doGetMonedasDTO();  
        }  
    }.getAsLiveData();  
}
```

Miremos que devuelve el método: `LiveData<Resource<List<MonedaEntity>>>`

Ahora ya no se devuelve un `LiveData<List>` sino que se devuelve: `LiveData<Resource<List>>` El motivo es porque según el caso que sea, el Resource será la base de datos local, o la api remota. Eso afectará a nuestro fragment donde “observábamos” el `LiveData`

En el new NetworkBoundResource le tenemos que establecer en el primer diamond: <> el tipo de objeto para la base de datos local. Y en el segundo diamond el tipo de objeto de la api remota:

```
new NetworkBoundResource<List<MonedaEntity>, List<MonedaDTO>>()
```

Por lo demás ya el propio código lo comenta:

El **createCall()** se intenta primero. Siendo una llamada a la api remota

El **saveCallResult()** se desencadena justo después, si hay éxito en la consulta, y se encarga de guardar en la base de datos local la información actualizada de la api

El **loadFromDb()** se ejecuta si no hubo éxito al llamar a la api y retorna la información en la base de datos local como sustituto

Finalmente veamos como tenemos que modificar en el Fragment los datos livedata “observados” del repositorio:

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {  
    super.onViewCreated(view, savedInstanceState);  
  
    viewModel= new ViewModelProvider(this).get(FirstFragmentViewModel.class);  
    viewModel.getListaMonedas().observe(getViewLifecycleOwner(), monedasResource -> {  
        List<MonedaEntity> monedas = monedasResource.data;  
        System.out.println("Monedas: " + monedas);  
        if( monedas != null)  
            binding.tvMonedas.setText(monedas.toString());  
    });
```

La línea nueva es:

```
List<MonedaEntity> monedas = monedasResource.data;
```

que lo que nos está diciendo es que, como ahora obtenemos un Resource, debemos tomar la información almacenada en el atributo data: monedasResource.data

Así tenemos la posibilidad dinámica de que el Resource devuelva en: data un recurso u otro (si api o base de datos) según lo que haya ocurrido