PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Fundamentos de aplicaciones móviles

Dispositivos móviles

Estos dispositivos aparecieron por primera vez en **1917**, cuando su inventor, el finlandés Eric Tigerstedt, presentó su patente para un teléfono de bolsillo.

El primer teléfono móvil lo presentó **Motorola** en el año **1973**, aunque no se comercializó hasta los **años 80**.

Hay crear varias categorías:

- **Smartphones**: Estos son los teléfonos inteligentes que tenemos hoy en día.
- **PDA**: Estos dispositivos ya están en desuso, pero eran una especie de **asistente digital** diseñado para servir como una **agenda digital**.
- **Tablets**: Estos son dispositivos que realizan prácticamente las mismas operaciones que los *smartphones*, pero son de mayor tamaño.

Sistemas operativos móviles

Los sistemas operativos móviles son una especie de sistema operativo **como los que conocemos para ordenadores**, **pero** especialmente diseñados para los dispositivos móviles. Esto significa que no tienen la misma potencia de cálculo, ya que en los dispositivos móviles debemos optimizar los **recursos** (como pueden ser la RAM o la batería), puesto que son **más reducidos**.

Entre los sistemas operativos existentes para dispositivos móviles, tenemos el sistema operativo Android, desarrollado por Google y que está basado en un Kernel de Linux.

También destaca el sistema operativo iOS, desarrollado por Apple Inc. Fue desarrollado para el iPhone (iPhone OS), aunque después se ha usado en dispositivos como iPod touch y iPad.

Para desarrollar aplicaciones para iOS debemos tener forzosamente un ordenador Mac con MacOS, ya que el ***framework* de desarrollo para iOS** solamente está disponible **en el dicho sistema operativo**.

El desarrollo de aplicaciones para dispositivos con **iOS es más sencillo**.

*También existen **versiones de Ubuntu** que se pueden instalar en móviles, como es **Ubuntu Touch**.*

Evolución de Android

En el año **2003**, en **California** había una pequeña compañía dedicada al desarrollo de software llamada Android Inc. Por su nombre ya podemos deducir fácilmente que ellos fueron los creadores originales de la primera versión del sistema operativo Android. En torno al año 2005 Google compró Android Inc. para seguir el proyecto de este sistema operativo que revolucionaría el mercado de los dispositivos móviles.

Hoy en día Android es un sistema operativo libre de código abierto, pero hasta el año 2007 Google no liberó su sistema operativo bajo la **licencia de código abierto Apache**, lo cual lo convirtió en un software libre y de código abierto.

Las **versiones** de Android reciben en inglés el nombre de diferentes **postres** o dulces hasta la versión 9.0. En cada versión el postre o dulce elegido empieza por una letra distinta, conforme a un orden alfabético:

Nombre código	Número de versión	Fecha de lanzamiento	Nivel de API
Apple Pie58	1.0	23 de septiembre de 2008	1
Banana Bread58	1.1	9 de febrero de 2009	2
Cupcake	1.5	25 de abril de 2009	3
Donut	1.6	15 de septiembre de 2009	4
Eclair	2.0 – 2.1	26 de octubre de 2009	5 – 7
Froyo	2.2 – 2.2.3	20 de mayo de 2010	8
Gingerbread	2.3 – 2.3.7	6 de diciembre de 2010	9 – 10
Honeycomb59	3.0 – 3.2.6	22 de febrero de 2011	11 – 13
Ice Cream Sandwich	4.0 – 4.0.5	18 de octubre de 2011	14 – 15
Jelly Bean	4.1 – 4.3.1	9 de julio de 2012	16 – 18
KitKat	4.4 – 4.4.4	31 de octubre de 2013	19 – 20
Lollipop	5.0 – 5.1.1	12 de noviembre de 2014	21 – 22
Marshmallow	6.0 – 6.0.1	5 de octubre de 2015	23
Nougat	7.0 – 7.1.2	15 de junio de 2016	24 – 25
Oreo	8.0 – 8.1	21 de agosto de 2017	26 – 27
Pie	9.0	6 de agosto de 2018	28
10	10.0	3 de septiembre de 2019	29
11	11.0	8 de septiembre de 2020	30
12	12.0 - 12L	4 de octubre de 2021	31 - 32
13	13.0	15 de agosto de 2022	33
14 (Beta 4)	14.0	13 de julio de 2023	34

Limitaciones del uso de una aplicación móvil

• Estos dispositivos tienen un uso de energía limitado, ya que dependen de **baterías**, las cuales se descargan más rápido cuantas más aplicaciones se usen o porque una aplicación determinada consume muchos recursos, y por lo tanto, necesita más energía para ejecutarse.

• **Procesadores** tienen **baja capacidad** de cómputo en comparación a los de los ordenadores convencionales. Esto también está relacionado con el punto anterior: cuanto **más potente sea el microprocesador** que tenga nuestro dispositivo móvil, más **energía** necesitará para estar en funcionamiento.

• Memoria **RAM bastante reducida**, aunque es verdad que en los últimos modelos este problema se va solventando; pero, en consecuencia, el **precio** del dispositivo aumenta considerablemente.

• Reducido espacio de **almacenamiento interno**, por lo que no puede almacenar gran cantidad de datos como los ordenadores, aunque lo normal es que estos dispositivos tengan acceso a una **memoria auxiliar** en forma de tarjeta de memoria o «**nube**».

• **Teclado** integrado, normalmente táctil, con unas **funcionalidades muy reducidas**.

• Las **pantallas** son de **pequeñas** dimensiones.

• Programa de **malware** que ponga nuestros datos en peligro.

• Quedarnos sin **cobertura** y, por lo tanto, desconectados de cualquier operación importante que estemos realizando o necesitemos realizar.

Ciclo de vida de una aplicación móvil Android

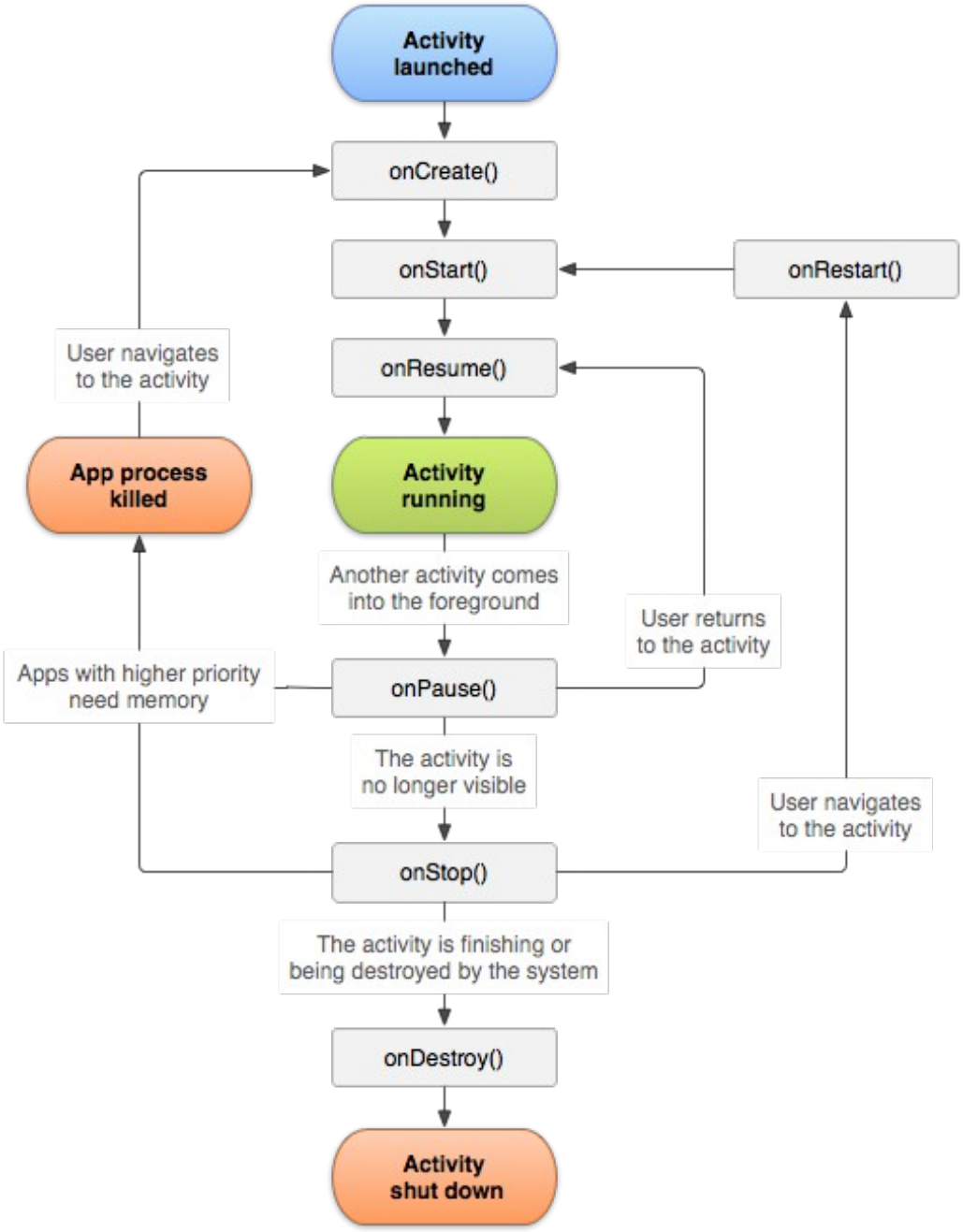
Conocer y controlar cuándo el usuario está utilizando la aplicación, cuándo la finaliza, cuándo le pasa el control a otra aplicación, etc.

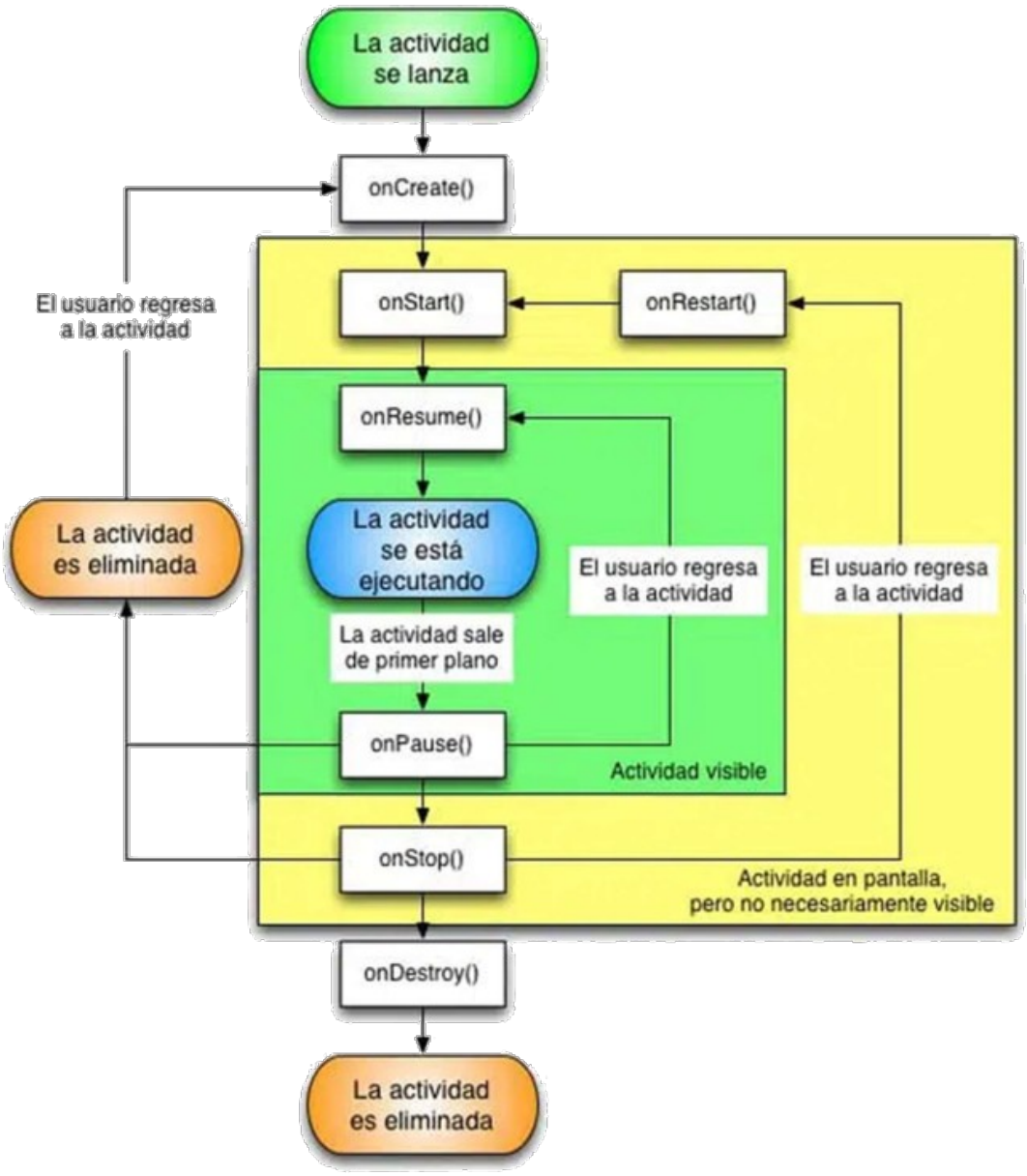
Estados de una aplicación móvil;

- **Activa**: La aplicación está siendo utilizada por el usuario.
- **Pausada**: La **actividad** está activa, aunque ha **perdido el foco** y se está ejecutando otra actividad. Entendemos una actividad como una **pantalla** de nuestra aplicación, pudiendo tener una aplicación varias actividades.
- **Parada**: La **actividad ya no es visible** y deberíamos **salvar los datos** de la misma para luego rescatarlos.

Eventos que van a representar el ciclo de vida de una aplicación Android:

- **onCreate(Bundle)**: Este evento representa el momento en el que **se crea** la actividad. Si antes hemos salvado los **datos** de la actividad, podremos utilizarlo para **regenerarla**.
- **onStart()**: Este evento representa el momento en que la actividad va a pasar a estar **en pantalla**, aunque no necesariamente de forma visible. Si venimos **de una parada**, pasaremos **antes por onRestart()**.
- **onRestart()**: Este evento es **anterior a onStart()**, cuando **procedemos de una llamada a onStop()**.
- **onResume()**: Este viene a indicar que la actividad va a **empezar** a responder a la **interacción** del usuario.
- **onPause()**: En este caso, la actividad va a **dejar** de responder a la **interacción** del usuario.
- **onStop()**: La actividad ha pasado completamente a **segundo plano**.
- **onDestroy()**: Indica que la actividad va a ser **destruida** y sus **recursos liberados**.


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down


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

Frameworks

Se pueden dividir en tres categorías:

• **Nativos**: Este tipo de frameworks son los entornos de desarrollo que nos van a permitir desarrollar aplicaciones móviles para un determinado sistema operativo de forma nativa, es decir, que podremos utilizar **todo el potencial** de las mismas. Un ejemplo de entorno de desarrollo para Android es **Android Studio**, mientras que para aplicaciones iOS es **XCode**. Para el desarrollo nativo es necesario tener **conocimientos en lenguajes** de programación como Java / Kotlin / XML para Android, y **Swift / Objective-C para iOS**.

• **Híbridos**: La tecnología híbrida nació para **reducir los costes y tiempos** y así facilitar el aprendizaje y desarrollo de aplicaciones móviles, además de poder **crear webs con estos entornos** de trabajo. Este tipo de desarrollo tiene una curva de aprendizaje bastante más suave que el desarrollo nativo. Ejemplos:

- Adobe **PhoneGap**: gratuito, código libre, html, javascript y css3. Trabaja sobre Apache y da acceso al hardware.
- **Angular**: funciona bajo aplicaciones web, trabaja de forma asíncrona sin recargar la página (angular.io)
- **React Native**: utiliza javascript y React (reactnative.dev).

• **Webs**: Este tipo de *frameworks* nos van a permitir, además de crear aplicaciones web, crear también aplicaciones móviles **mediante la tecnología web**.

Los ***frameworks* nativos** nos van a permitir aprovechar al máximo todas las características de los dispositivos móviles. En cambio, si utilizamos *frameworks* no nativos que nos permitan crear una aplicación, a la hora exportarla a diferentes sistemas operativos estos nos la limitarán bastante y se podrán ofrecer menos funcionalidades.

g

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Proyectos Android

Arquitectura Android

• Capa de **aplicaciones**: Aplicaciones que forman la base de Android. Están escritas en lenguaje **Java (ahora kotlin de preferencia)**.

• Capa del **marco de aplicaciones**: **Simplifica** la **reutilización** de **componentes** por el usuario.

• Capa de **bibliotecas**: Escrita en **C/C++,** la **usan muchos de los componentes** que forman el **sistema operativo** Android.

• Capa de **runtime**: Conforma un conjunto de **bibliotecas** que proporcionan muchas de las **funciones** del **sistema operativo**.

• **Núcleo Linux**: Es el núcleo o kernel el que va a proporcionar todo lo relacionado con la **gestión de los recursos** del dispositivo.

ESQUEMA

APLICACIONES

- INICIO
- CONTÁCTOS
- TELÉFONO
- EXPLORADOR

LIBRERÍAS

- ADMINISTRADOR DE SUPERFICIES
- ARMAZÓN DE MEDIA
- SQLITE
- OPENGL | ES
- FREETYPE
- WEBKIT
- SSL
- SGL
- LIBC

ARMAZÓN/MARCO DE APLICACIONES

- ADMINISTRADOR DE ACTIVIDAD
- ADMINISTRADOR DE VENTANAS
- PROVEEDOR DE CONTENIDOS
- VISTA DEL SISTEMA
- ADMINISTRADOR DE PAQUETES
- ADMINISTRADOR DE RECURSOS
- ADMINISTRADOR DE UBICACIONES
- ADMINISTRADOR DE NOTIFICACIONES
- ADMINISTRADOR DE TELEFONIA

ANDROID RUNTIME

- LIBRERÍAS DEL NÚCLEO
- MÁQUINA VIRTUAL DALVIK
- **KERNEL DE LINUX**
  - CONTROLADORES DE PANTALLA
  - CONTROLADORES DE LA CÁMARA
  - CONTROLADORES DE MEMORIA FLASH
  - CONTROLADORES BINDER (IPC)
  - CONTROLADOR DE TECLADO
  - CONTROLADOR DE WIFI
  - CONTROLADOR DE AUDIO
  - GESTIÓN DE ENERGIA

Permisos en Android

*Mediante éstos, las apps podrán **acceder a los elementos del dispositivo** móvil, como puede ser la cámara, el GPS, bluetooth o internet.*

Estos permisos tendrán que indicarse en cada app y tendrán que ser aceptados por el usuario cuando instale la misma.

Instalación de Android Studio

Android Studio es el **compilador oficial** para desarrollar aplicaciones para el sistema operativo Android. Se basó en **IntelliJ IDEA** de **JetBrains** y está publicado de forma gratuita a través de la **Licencia Apache 2.0.**

Tras instalarlo, tendremos que iniciar Android Studio para que se **instale toda la JDK** necesaria para crear aplicaciones Android.

Instalación de Android Studio: https://developer.android.com/studio/install.html?hl=es-419

Emulador

Tenemos tres formas de emular nuestra aplicación en un sistema android:

1. Mediante un **emulador integrado** en Android Studio.
2. Mediante un **emulador de terceros**.
3. Mediante nuestro **propio dispositivo**.

La creación de un emulador integrado en Android Studio es muy fácil y rápida y nos va a permitir probar nuestras aplicaciones sin necesidad de utilizar nuestro dispositivo como emulador ni emuladores de terceros.

Para ello, en la ventana de inicio de Android Studio seleccionamos la opción Configure y luego seleccionamos **AVD Manager**. En el botón **Create Virtual Device**… nos aparecerá el asistente de creación de un dispositivo virtual. Debemos elegir el dispositivo móvil que queremos emular.

El siguiente paso será elegir el **sistema operativo** Android que queramos, pudiendo elegir entre todos los sistemas operativos que han aparecido en Android.

En el caso de que no tengamos descargada la imagen del sistema operativo, debemos **descargarla**. Hay que tener en cuenta que tendremos que **elegir la imagen que tenga las Google Apis integradas**.

Emulador Android NOX

Emuladores de Android hay muchísimos: BlueStacks, ARChon, Genymotion, Bliss OS, etc. **NOX es fácil de usar**.

NOX es un emulador de dispositivos Android disponible para **Windows y para MacOS**. Este emulador se centra en el uso de videojuegos, lo cual nos permite jugar a juegos Android desde nuestro PC sin tener que instalarlos en los dispositivos móviles. Igualmente, se podrá utilizar NOX para emular a un dispositivo físico con el objetivo de probar el funcionamiento de las aplicaciones.

Podemos descargar NOX de su página web: https://es.bignox.com/

Tener cuidado de **no aceptar que nos instale ningún sistema antivirus**, ya que nosotros **únicamente queremos el emulador de Android**.

Creación de un proyecto

Para crear un proyecto nuevo en Android Studio debemos seguir los siguientes pasos:

1. Seleccionamos '**Start a new Android Studio proyect**'.

2. Indicamos el tipo de actividad que queremos, en este caso vamos a seleccionar una **actividad vacía**. Una actividad es una **pantalla** de una aplicación que se compone de la parte gráfica (escrita en **XML**) y la parte de código (escrita en **Java o en Kotlin**). Una aplicación Android tendrá varias actividades, cada una con su parte gráfica y su parte de código. Pulsamos 'siguiente'.

3. Seleccionamos las **SDK mínimas** para que nuestro proyecto funcione, lo cual implica que **cuando instalemos** nuestra aplicación en un dispositivo móvil este **deberá tener una SDK igual o superior** a la que indiquemos en este paso. Otra cosa a elegir es el lenguaje de programación: podemos elegir entre Kotlin o Java, y nosotros elegiremos Java. Aquí podremos seleccionar también la ruta donde se guardará nuestro proyecto. Pulsamos en el botón 'Finish' para crear el proyecto.

Tras seguir todos los pasos se creará nuestro proyecto. Después, se sincronizarán todos los ficheros del mismo, lo cual puede tardar unos minutos dependiendo de las características de nuestro ordenador. Hasta que no se terminen de sincronizar todos los ficheros del proyecto no podremos empezar a programar nuestra aplicación Android.

Ejecución de un proyecto

Para ejecutar y probar nuestra aplicación tenemos dos posibilidades:

1. Utilizar el **emulador** de Android que integra Android Studio o uno de terceros.
2. Utilizar **nuestro propio dispositivo** móvil para ejecutar la aplicación.

Para lanzar la prueba, tendremos que hacerlo en Android Studio, desde el botón de compilación y ejecución. Es el siguiente:

Cuando pulsemos el botón de compilar nuestra aplicación, nos aparecerá una lista con todos los dispositivos (tanto virtuales como físicos) que tenemos disponibles para elegir donde queremos probarla.

Una vez elegido el dispositivo, la aplicación se instalará y ejecutará automáticamente.

Cuando queramos volver a lanzar la aplicación porque hayamos **realizado ciertos cambios** en el código, no tendremos que cerrar el dispositivo donde estemos probando, ya que simplemente volviendo a **pulsar el botón de compilar** la aplicación se cerrará, se **compilará, se instalará y se ejecutará** automáticamente en el mismo dispositivo.

En el caso de que queramos cambiar de dispositivo, sí que tendremos que cerrar el que tengamos abierto y elegir el nuevo dispositivo.

Para ejecutar nuestra aplicación en **varios dispositivos**, podremos elegir la opción '**Run on Multiple Devices**'.

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Programación en Android

Pantalla

Características a tener en cuenta en el diseño de las pantallas:

- **Tamaño**: Longitud de la pantalla en diagonal y esta puede ser **small, normal, large, extra large**...
- **Densidad**: Cantidad de píxeles que tendrá la pantalla, y se mide en **puntos por pulgada** o **DPI**.

  Esta puede ser **baja o LDPI, media o MDPI, alta o HDPI, extra alta o XHDPI, etc.**
- **Resolución**: Se define como la **cantidad de píxeles** de la pantalla tanto en horizontal como en vertical.

Recursos

Utilizamos recursos para la creación de nuestras aplicaciones.

En la **categoría recursos** tenemos:

- Textos.
- Imágenes.
- Colores.
- Definición de estilos.
- Sonidos.

Son una parte fundamental y vamos a **separarlos del código** para un mejor **mantenenimiento.**

Si tenemos organizados nuestros recursos de APP el **sistema operativo** Android se encarga de **elegir por nosotros** cada elemento dependiendo de la **configuración** del dispositivo, como el tamaño de pantalla, el idioma del dispositivo, etc.

Los recursos estarán localizados en un punto específico dependiendo del tipo de recurso.

Todos y cada uno de los **recursos**, **salvo las imágenes**, van a estar dentro de un **fichero XML**, aprovechando así la simplicidad de este lenguaje de marcas para su configuración. Cada uno de ellos tiene, además, una sintaxis diferente. Todos los recursos de nuestra aplicación estarán dentro de la **carpeta /res** del proyecto, de la cual se desprenderán una serie de directorios para cada uno de ellos.

Textos

Los podremos poner de dos formas completamente opuestas:

- **Forma incorrecta** o menos aconsejada: Directamente en la configuración de texto del elemento que estemos creando: no es aconsejable ya que, si queremos cambiar un texto en concreto, tendremos que ir a dicha pantalla y cambiarlo.
- **Forma correcta** o aconsejada:

  La forma correcta de almacenar los textos es en un **fichero de recursos separado**. Ese fichero se llama **strings.xml** y su ruta completa dentro del proyecto es **res/values/strings.xml**.

  El fichero strings.xml tendrá un contenido similar a este:

```
<resources>
   <!-- ESTE SERÁ EL LENGUAJE POR DEFECTO, INGLÉS -->
   <string name="app_name">My Bus Capacity</string>
   <string name="actualiza_suben">Update getting on</string>
</resources>
```

  Cada texto tendrá dos propiedades:

  - **Name**: El identificador del recurso de texto que usaremos para acceder a él.
  - El propio **valor** del texto, que podrá contener todo tipo de caracteres.

Es posible poner comentarios en nuestros ficheros de recursos, ya que son ficheros XML que nos ayudarán a identificar a qué pantalla pertenecen dichos textos.

Para poder acceder al texto «actualiza_suben» pondremos:

```
android:text="@string/actualiza_suben"
```

Imágenes

Tenemos varias carpetas dentro del proyecto, como **res/drawable**, **res/mipmap-hdpi**, **res/mipmap-xhdpi** y similares. Los sufijos que siguen a **mipmap** se refieren a la **densidad de la pantalla**. Cada dispositivo Android tiene una densidad de píxeles por pantalla que se agrupa en varios grupos:

- **ldpi** (pequeña, x**0,75**)
- **mdpi** (media x1)
- **hdpi** (grande x**1,5**)
- **xhdpi** (extra grande x**2**)
- **xxhdpi** (extra extra grande x**3**),
- xxxhdpi...

En la carpeta **drawable** se ubicarán las **imágenes que no cambiarán** según el tamaño de la pantalla del dispositivo.

Colores

Forma incorrecta:

Estos colores se pueden colocar directamente sobre la configuración de los elementos gráficos, pero al querer cambiar un color que se use repetidamente tendremos que ir pantalla por pantalla cambiando manualmente el color...

Forma correcta:

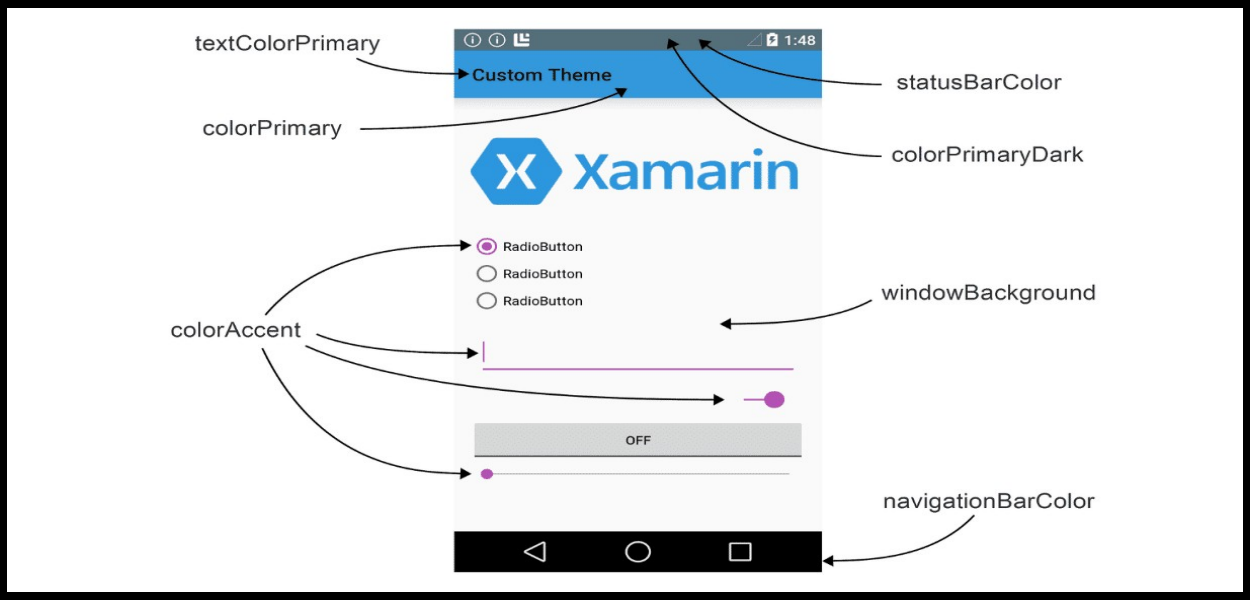
Almacenar los colores en un fichero de **recursos separado**. Ese fichero se llama colors.xml y su ruta completa dentro del proyecto es **res/values/colors.xml**.

El fichero colors.xml tendrá un contenido similar a este:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="colorPrimary">#FF6200EE</color>
  <color name="colorPrimaryDark">#FF3700B3</color>
   <color name="colorAcent">#FFBB86FC</color>

   <color name="gris_oscuro_fondo_progressbar">#434343</color>
   <color name="gris_deseleccionado_icono">#FF6C6C6C</color>
   <color name="gris_claro_fondo_progress_y_fondomenu">#AEAEAE</color>
   <color name="gris_muy_claro">#DADADA</color>
</resources>
```

Los colores los definimos mediante su **código RGB** y se muestra una **previsualización** a la izquierda del código. Hay varios colores que ya están **predefinidos** en la configuración de las pantallas y que podemos **sobrescribir**.


textColorPrimary
colorPrimary
statusBarColor
colorPrimaryDark
1:48
Custom Theme
Xamarin
RadioButton
RadioButton
RadioButton
colorAccent
windowBackground
OFF
navigationBarColor

Estilos

Definición: **colección de propiedades** que indican la **apariencia y el formato** de una pantalla.

Será similar a las de las **hojas de estilo CSS** del diseño web: permiten **separar el diseño del contenido**.

Forma correcta de usar los estilos:

En un fichero de recursos separado. Ese fichero se llama **styles.xml**, y su ruta completa dentro del proyecto es **res/values/styles.xml**.

El fichero styles.xml tendrá un contenido similar a este:

```
<resources>
    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light">
        <item name="colorPrimary">#FFB8EC</item>
        <item name="colorPrimaryDark">#FF63D6</item>
        <item name="colorAccent">#FD72D8</item>
        <item name="fontFamily">casual</item>
    </style>
    <style name="DialogTema" parent="Theme.AppCompat.Dialog.Alert">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">#8CDAD2</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
        <item name="textColorAlertDialogListItem">@android:color/white</item>
        <item name="fontFamily">casual</item>
        <item name="android:windowBackground">@color/colorRosaClaro</item>
    </style>
    <style name="AppTheme.AppBarOverlay" parent="ThemeOverlay.AppCompat.Dark.ActionBar" />
    <style name="AppTheme.PopupOverlay" parent="ThemeOverlay.AppCompat.Light" />
<resources>
```

Podremos crear tantos estilos como queramos dentro de nuestro proyecto, usando, también, otros recursos como pueden ser los colores.

En el ejemplo anterior estamos definiendo un estilo para un Dialog donde el color de fondo de la ventana de diálogo será un color ya definido por nosotros en res/values/colors.xml como **colorRosaClaro**.

Y así podemos cambiar otras propiedades (fontFamily,android:windowActionBar, etc.).

Para cambiar el estilo de un elemento gráfico lo haremos a través de su propiedad **style**.

```
        <style name="DialogTema" parent="Theme.AppCompat.Dialog.Alert">
```

Internacionalización (idiomas)

Conseguiremos traducir nuestra aplicación a cualquier lengua, aunque debemos realizar una buena traducción si queremos que tenga usabilidad nuestra APP.

Un requisito fundamental para conseguirlo se basa en **colocar todos los recursos en su fichero correspondiente**, como hemos visto en puntos anteriores. Si no hacemos esto, no podremos traducir nuestra aplicación.

Si queremos que nuestros textos estén disponibles en **varios idiomas**, tendremos que añadir una **carpeta diferente** para cada copia del fichero strings.xml. De este modo, si queremos traducir nuestra aplicación al español y al portugués, teniendo el idioma por defecto el inglés (muy aconsejable), tendríamos estos tres directorios:

- **res/values** (valores por defecto -inglés-)
- **res/values-es** (valores para el español)
- **res/values-pt** (valores para el portugués)

Códigos de idioma admitidos:

http://utils.mucattu.com/iso_639-1.html.

Es decir, a la carpeta se le añade un sufijo con el **código ISO del idioma concreto**, y dejaremos los textos por defecto en la carpeta **sin sufijo**. Cuando queramos un recurso del tipo que sea, Android buscará el que mejor encaje y, si no encuentra ninguno, podemos encontrarnos con cierres inesperados de la aplicación. Si el usuario tiene un idioma que no aparezca en nuestros strings.xml, verá el que tenga nuestra app por defecto, y posiblemente será el inglés el más cómodo de entender porque es el más usado a nivel mundial.

Una vez hecho esto, bastará con cambiar el idioma del dispositivo para que los textos aparezcan en dicho idioma.

Android Developers, estilos: https://developer.android.com/guide/topics/resources/providing-resources

Android Developers, recursos: https://developer.android.com/guide/topics/ui/themes. html?hl=es-419

TÉCNICO EN DESARROLLO DE APLICACIONESMULTIPLATAFORMA

Programación multimedia y dispositivos móviles

PROGRAMACIÓN EN ANDROID

LAYOUTS

Al realizar un maquetado de los elementos gráficos que utilicemos tendremos que **distribuirlos de cierta forma en la pantalla** para que se adapten al **diseño** que queramos conseguir.

En Android Studio vamos a utilizar Layouts para la maquetación de las interfaces gráficas de nuestras aplicaciones. Podemos definir los **Layouts** como elementos no visuales destinados a controlar la **distribución**, **posición** y **dimensiones** de los elementos gráficos que se insertan en su interior. Extienden la clase base **ViewGroup**, como muchos otros componentes **contenedores**, capaces de **contener a otros controles**.

Vamos a tener diferentes tipos de Layouts a nuestra disposición, cada uno de los cuales distribuirá de forma diferente los elementos que coloquemos en su interior. Vamos a poder utilizar los siguientes, entre otros:

- LinearLayout.
- TableLayout y RowLayout.
- FrameLayout.
- ConstraintLayout.

LinearLayout

Coloca los **elementos** que contenga de **forma consecutiva** en pantalla.

Propiedades

Propiedad android:orientation

Elementos en dos orientaciones diferentes: vertical y horizontal.

Propiedad android:layout_weight

Otorga a los elementos del Layout un **tamaño proporcionado** entre todos ellos.

Por ejemplo, si en un LinearLayout tenemos **dos elementos**, y uno de ellos tiene un layout_**weight="1"** y el otro un layout_**weight="2"**, todo el contenido del Layout quedará ocupado por estos dos elementos, y **el segundo ocupará el doble que el primero**.

Propiedad android:gravity

Podremos indicar la **alineación** de los elementos **dentro del mismo**. Esta propiedad puede tener los siguientes valores:

- **Center**: Centra los elementos dentro del Layout tanto de forma **horizontal como vertical**.
- **Top**: alinea **arriba** los elementos dentro del Layout.
- **Bottom**: Alinea **abajo** los elementos dentro del Layout.
- **Right**: Alinea a la **derecha** los elementos dentro del Layout.
- **Left**: Alinea a la **izquierda** los elementos dentro del Layout.
- **Center_horizontal**: Alinea al centro de forma **horizontal** los elementos dentro del Layout.
- **Center_vertical**: Alinea al centro de forma **vertical** los elementos dentro del Layout.

Propiedad setVisibility

Oculta automáticamente todos los elementos que contengan. Deberemos proporcionar un id al Layout y ligarlo a un objeto. De esta forma, y mediante el método, podremos aplicarle los siguientes valores:

- **View.GONE**: **Oculta** el Layout y sus elementos.
- **View.VISIBLE**: **Muestra** el Layout y sus elementos.

Ejemplo de diseño de LinearLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout 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"
        android:orientation="vertical"
        tools:context=".MainActivity">

        <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:layout_weight="1"
                android:gravity="center"
                android:orientation="horizontal">

                <Button
                        android:id="@+id/button"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content"
                        android:text="Button" />

                <Button
                        android:id="@+id/button2"
                        android:layout_width="wrap_content"
                        android:layout_height="wrap_content"
                        android:text="Button" />

                <Button
                        android:id="@+id/button3"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Button" />
        </LinearLayout>

        <LinearLayout
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:layout_weight="2"
                android:gravity="center_horizontal"
                android:orientation="vertical">
```

```
            <Button
                android:id="@+id/button4"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Button" />

                <Button
                android:id="@+id/button5"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Button" />

                <Button
                android:id="@+id/button6"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Button" />
        </LinearLayout>
</LinearLayout>
```

Nota: Dentro de un Layout, sea cual sea, podremos introducir otro Layout, para así conseguir una interfaz gráfica lo más compleja posible. Podemos **anidar Layouts**, incluso diferentes, sin ningún problema.

TableLayout

Distribuye los elementos que introduzcamos en él en forma de **tabla**, definiendo las **filas y columnas** necesarias, así como la posición de cada componente dentro de esta.

La estructura de la tabla creada se define de forma similar a como se hace en el lenguaje HTML: tendremos que indicar las **filas** que compondrán dicha tabla (mediante **TableRow**) y, dentro de cada una de las filas, las columnas necesarias, con la salvedad de que **no existe ningún objeto especial para definir una columna**, sino que iremos insertando directamente los componentes necesarios dentro del TableRow.

Cada **componente** corresponderá con una **columna** de la tabla, pudiendo ser un elemento simple u otro Layout. De esta forma, el **número final de filas** de la tabla se corresponderá con el **número de elementos TableRow** insertados; y el **número total de columnas** quedará determinado por el **número de componentes de la fila** que más componentes contenga.

Una característica que debemos tener bastante en cuenta es la posibilidad de **configurar celdas** para que **ocupen** el espacio que les corresponderían a **varias de las columnas** de la tabla (funcionando igual que el **atributo colspan de HTML**). Esto lo podremos configurar con la propiedad **android:layout_span** del **elemento que hayamos añadido** a nuestro TableLayout.

```
<!-- layout_span: permite que un control se expanda más de una celda.
    Button pasará a ocupar el espacio de dos botones:
-->
<Button
   android:id="@+id/button"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:layout_span="2"
   android:text="Button" />
```

Como norma general, el **ancho de cada una de las columnas** va a corresponder al ancho del **mayor componente** que haya dentro de la columna, aunque también tenemos unas propiedades que nos van a permitir modificar este comportamiento.

• **android:stretchColumns**: Podremos indicar qué columnas se van a poder **expandir** para ocupar el espacio libre que han dejado las demás columnas a la derecha. Estará asociado al atributo: android:layout_column="0", coincidiendo el número (en este caso “0”. 0 es el índice de columna).

• **android:shrinkColumns**: Nos permitirá indicar qué columnas se van a poder **contraer** para dejar espacio, y que se puedan salir por la derecha.

• **android:collapseColumns**: Con esta propiedad vamos a poder indicar qué columnas de nuestra tabla queremos **ocultar** de forma completa.

```
<TableLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:gravity="center_vertical"
    android:padding="16dp"
    android:stretchColumns="0">

    <TableRow>

        <Button
            android:layout_height="50dp"
            android:text="ind 0" />

        <Button
            android:layout_width="100dp"
            android:layout_height="50dp"
            android:text="ind 1" />

        <Button
            android:layout_width="100dp"
            android:layout_height="50dp"
            android:text="ind 2" />

    </TableRow>

</TableLayout>
```

```
<TableLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:layout_weight="1"
   android:gravity="center_vertical"
   android:padding="16dp"
   android:shrinkColumns="0">
   <!-- En el atributo android:shrinkColumns: Reduce el ancho de la columna seleccionada
hasta ajustar la fila al tamaño del padre.
   -->

   <TableRow>
      <!-- Esta columna (indice 0) se encogerá hasta ajustar la fila al tamaño del contenedor
           padre
           El resto de Buttons estarán con la misma anchura para apreciar el efecto.
           Este Button estará sin definir su ancho para que aplique shrinkColumns.
      -->
      <Button
         android:layout_height="50dp"
         android:text="ind 0" />

      <Button
         android:layout_width="149dp"
         android:layout_height="50dp"
         android:text="ind 1" />

      <Button
         android:layout_width="149dp"
         android:layout_height="50dp"
         android:text="ind 2" />

   </TableRow>
</TableLayout>
```

FrameLayout

Es el **más simple**. Este **alineará** automáticamente en la **esquina superior izquierda** del Layout **todos los elementos** que contenga, de forma que todos los elementos se irán **superponiendo** encima del anterior. Este Layout suele utilizarse mucho para **mostrar un único elemento** gráfico. Dentro de todos los elementos gráficos, incluidos Layouts, vamos a tener dos propiedades básicas para poder **gestionar su tamaño**:

- **android:layout_width**: Definirá el tamaño del **ancho** del elemento.
- **android:layout_height**: Esta propiedad va a definir el tamaño del **alto** del elemento.

En estas **propiedades podrán configurarse** los siguientes valores:

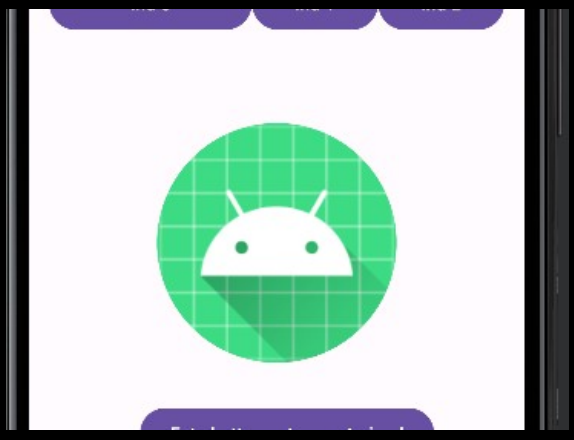
- **match_parent**: Con este valor indicamos que el elemento debe medir lo **mismo que su contenedor**.
- **wrap_content**: Con este valor indicamos que el elemento debe medir lo justo y necesario respecto a su propio contenido.

Podemos usar la paleta de elementos gráficos (pestaña Layouts).

```
<FrameLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:layout_weight="1">

      <ImageView
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         android:src="@mipmap/ic_launcher_round">
      </ImageView>

</FrameLayout>
```



*"**ScrollView**"*

*Existe otro elemento llamado ScrollView, el cual **hereda de FrameLayout**.*

*ScrollView agrega automáticamente una barra de desplazamiento, que puede ser tanto **horizontal** como **vertical**, la cual, nos ayudará a **desplazar el contenido para ambas direcciones**.*

Suele combinarse con otros Layouts y agregarse automáticamente.

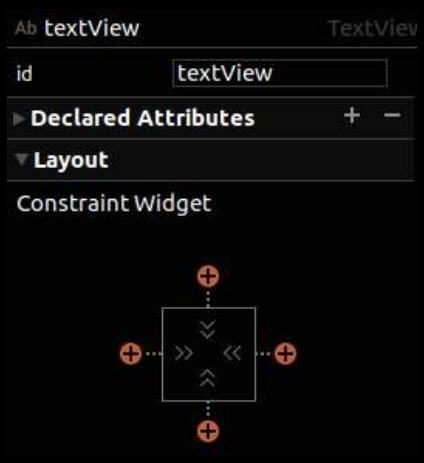
ConstraintLayout

Usado de forma **predeterminada** cuando creamos una actividad. **Diseños bastante complejos sin necesidad de tener que anidar Layouts**, (anidamiento ocasionaba problemas de memoria y eficiencia en dispositivos de poco rendimiento).

Es muy flexible y fácil de usar desde el editor visual de Android Studio. De hecho, es bastante recomendable crear los Layout de las interfaces gráficas con las herramientas **drag-and-drop**, en lugar de editar el código del fichero XML, aunque tienen más complejidad que los demás Layouts.

Las **posiciones** de las diferentes vistas dentro de este Layout se definen **usando constraint (restricción)**. Podemos definir una **restricción** en relación al elemento **contenedor**, a otra **vista**, o respecto a una línea de guía (elemento **GuideLine**). Deberemos definir **al menos un constraint horizontal y uno vertical.**

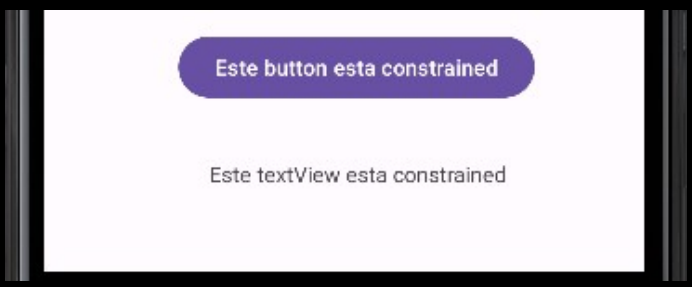
Como cada Layout, contemplará una serie de nuestras propiedades a los elementos que le agreguemos (distancias a los lados que tendrán los elementos con respecto a los demás...).




```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="88dp"
        android:layout_marginTop="32dp"
        android:text="@string/constraint_bt"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="108dp"
        android:layout_marginTop="36dp"
        android:text="@string/constraint_tv"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/button" />
</androidx.constraintlayout.widget.ConstraintLayout>
```



Pestañas (Tabs)

Veremos que es posible dividir una pantalla en varias, por medio de pestañas.

Disponemos de esta opción gracias a las pestañas (llamadas Tab) y a los Fragments.

- El trabajo con pestañas consistirá en **definir una única actividad**, y vamos a establecer varias **pestañas**, de forma que cada una de ellas tendrá un **contenido diferente** a las demás.
- Los **Fragments** nos van a permitir **dividir la interfaz de una pantalla de forma proporcional**.

Para integrar todo esto, Android Studio nos facilita el control **TabLayout** y los:

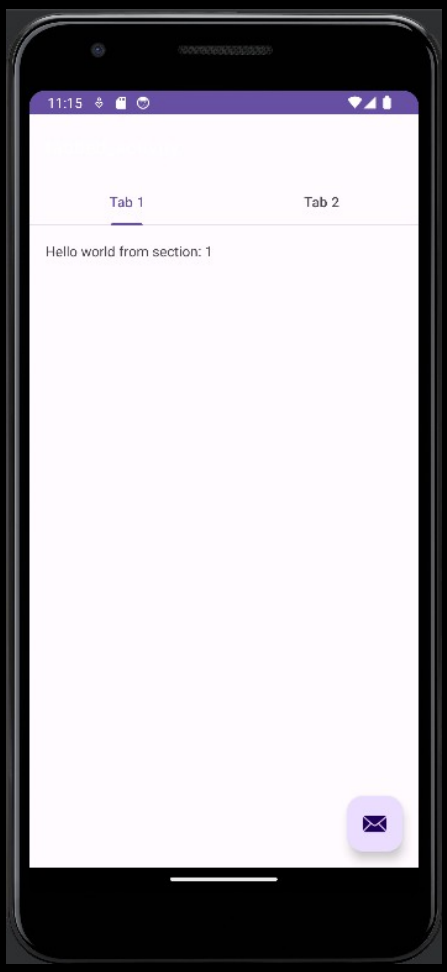
TabLayout: es el **Layout** que deberemos utilizar para usar pestañas.

TabItem: contiene las pestañas en sí mismas.

Internamente vamos a utilizar la **clase ViewPager**, que nos permitirá **cambiar de forma fluida** de pestañas.

Para **crear una aplicación** con pestañas, tendremos que indicar que deseamos este tipo de aplicación cuando creemos el proyecto en Android Studio. Por lo tanto, deberemos seleccionar **Tabbed Activity** como plantilla, en lugar de Empty Activity, tal como estábamos haciendo hasta ahora.

Una vez hecho esto, el propio Android Studio nos creará una **aplicación con la pantalla principal con varias pestañas**.



Máster en Desarrollo de Aplicaciones Android - Uso de ConstraintLayout. (s. f.)

Revelo, J. (29 de enero de 2020). TabLayout: ¿Cómo Añadir Pestañas En Android? Hermosa Programación: +50 Tutoriales Desarrollo Android.

Sgoliver. (17 de agosto de 2010). [Entrada de Blog]. Interfaz de usuario en Android: Layouts.

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Android

Eventos en botones

Listas optimizadas

Etiquetas y cajas de texto

TextView: Se encarga de mostrar texto en la pantalla mediante una **etiqueta**.

EditText: **Caja de texto** en la que podemos **escribir** el texto que deseemos y podremos obtenerlo o modificarlo. Existen **varios tipos de EditText** ya formateados que nos permitirán introducir únicamente un texto, un **email**, un **teléfono**, **números**, **contraseñas**, etc.

Deberemos crearlos en la **interfaz gráfica** y proporcionarles un **id**, para poder **ligarlos a código** y **trabajar** con ellos. Luego debemos **crear un objeto de la clase TextView o EditText**, según necesitemos, **y ligarlo** al elemento gráfico mediante el método **findViewById**.

Tras ligarlo, podremos **trabajar** con él con normalidad.

Consejo: llamar al **id** y al **objeto** de la **misma forma (**el **código más claro, o limpio)**.

Atributos genéricos a los elementos

Todos los **elementos gráficos** tienen una serie de **atributos**, de los cuales, **algunos son genéricos**. Algunos de estos son:

- **ID**: **identificador** del elemento para poder usarlo.
- **Heigth** y **Width:** dimensiones del elemento.
- **Paddin**: los márgenes a los lados.
- **Gravity**: proporcionar una **alineación** al elemento.

Código java:

```java
public class MainActivity extends AppCompatActivity {

    // Creo el objeto de TextView
    private TextView tvSaludo;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // asocio al elemento gráfico xml:
        tvSaludo = findViewById(R.id.tv_saludo);

        // Cambiamos el nombre al textView
        tvSaludo.setText("Hola Mundo");
        System.out.println(tvSaludo.getText());

    }
}
```

Elemento gráfico en xml:

```xml
<TextView
    android:id="@+id/tv_saludo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Botones

Gracias a ellos podremos proporcionar **interacción** entre el usuario y la aplicación.

La clase encargada de implementar los botones es la clase **Button**, de la cual hay varias **variantes**:

- **ImageButton**: Botón con una **imagen** en lugar de texto.
- **Switch**: Botón que podrá estar **activado o desactivado**.
- **FloatingActionButton**: Botón **flotante** que podrá estar en las **esquinas** de la pantalla.

Los botones mostrarán normalmente un **texto**, el cual deberá ir en el **fichero de strings.xml** y, entre otras cosas, podremos **cambiar y obtener el texto** mostrado, cambiar su **estilo** a uno definido en el fichero **styles.xml**, etc.

Para poder hacer que un botón realice una **acción** (**evento**) cuando hacemos clic sobre él, debemos seguir los siguientes **pasos**:

- Dar un **id** al botón y crear un **objeto** de tipo **Button** ligándolo con el método **findViewById** a su identificador.
- Hacer que la **clase de la actividad implemente** la interfaz **View.OnClickListener**.
- Implementar el método **onClick** y, usando un **switch** a la variable del tipo **View** con el método **getId()**, podremos implementar el código de cada uno de los **botones** que tengamos.
- Por último, asignar **Listener** al **botón** con el método **setOnClickListener**.

Podremos hacer esto con todos los botones que implementemos.

Ejemplo de implementación de **Button con escuchador de eventos**:

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener{
    // Creamos el objeto TextView y el objeto botón Button
    private TextView tMensaje;
    private Button bBoton1, bBoton2;

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

        // Ligamos los elementos gráficos
        tMensaje = findViewById(R.id.tMensaje);
        bBoton1 = findViewById(R.id.bBoton1);
        bBoton2 = findViewById(R.id.bBoton2);

        // Damos la funcionalidad a los botones
       // Trabajamos con los elementos normalmente
        bBoton1.setOnClickListener(this);
        bBoton2.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
            // Detectamos en qué botón se ha pulsado y asignamos una acción a cada botón:
        switch(v.getId())
        {
            // Hemos pulsado el botón con id bBoton1
            case R.id.bBoton1:
                String mensaje1 = getResources().getString(R.string.mensaje1);
                tMensaje.setText(mensaje1);
                break;
            // Hemos pulsado el botón con id bBoton2
            case R.id.bBoton2:
                String mensaje2 = getResources().getString(R.string.mensaje2);
                tMensaje.setText(mensaje2);
                break;
        }
    }
}
```

Forma correcta de implementar funcionalidades

Consiste en hacer que la **actividad** en la que nos encontramos **implemente el Listener** de la funcionalidad que queremos proporcionar, **implementando**, además, dicha **funcionalidad en la propia clase**.

A efectos de funcionalidad, esta forma de implementar es **igual que la opción de implementar los eventos en cada elemento gráfico**. No obstante, nos ofrece la ventaja de tener el **código ordenado**, de forma que podremos **identificar la funcionalidad de cada elemento mucho más rápido**, además de tener que escribir mucho menos código.

Imágenes

Para mostrar **imágenes** en una pantalla de nuestra interfaz gráfica podremos utilizar el **elemento ImageView**.

Con **ImageView** podremos mostrar de una forma muy sencilla una **imagen que esté dentro de nuestro proyecto**. Al agregar este elemento nos aparecerá una ventana con la que podremos elegir qué imagen queremos mostrar. La imagen **se mostrará con su tamaño original**.

Las imágenes que utilizaremos se guardarán en la carpeta **res/drawable** y, para cambiarlas, utilizaremos la propiedad **app:srcCompact="@drawable/unaimagen"**.

```
<ImageView
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="136dp"
    android:layout_marginTop="120dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:srcCompat="@drawable/perrito" />
```

Checkbox y Radiobutton

Dos de los elementos que no podían faltar en cualquier interfaz gráfica son los CheckBox y los RadioButton.

Estos elementos funcionan exactamente igual que cuando los estudiamos en la asignatura de programación. Los **CheckBox** podrán ser activados o no, y podremos tener **todos los que necesitemos**, mientras que los **RadioButton** se van a utilizar **en grupos**, para que podamos activar un único elemento entre ellos. Los grupos son los **RadioGroup**, e igualmente, podremos tener tantos como necesitamos, pudiendo **activar un único RadioButton en cada uno**. Los **grupos** los podremos poner tanto en posición **vertical** como en **horizontal**.

Podremos saber si los **CheckBox** están activados o no mediante el método **isChecked()**, que nos devolverá 'true' en caso de que esté activado o 'false' en el caso contrario. También podremos cambiar su activación mediante el método **setChecked()**, al que le pasaremos 'true' o 'false', según queramos activar o desactivar el elemento.

Sin embargo, para **saber qué RadioButton está activo** dentro de un grupo, tendremos que **implementar la interfaz RadioGroup**.**OnCheckedChangeListener** con la cual, mediante su método **onCheckedChanged**, podremos realizar una opción específica cuando se seleccione un RadioButton dentro de un grupo.

Deberemos utilizar el método **setOnCheckedChangeListener** del grupo para asignar la funcionalidad al mismo.

Al método **onCheckedChanged** le llegarán **dos parámetros**: el primero será el **id** del **grupo** que hayamos cambiado y el segundo será el **id** del **RadioButton** que ha cambiado.

Ejemplo de RadioButton y checkBox:

Nota: Dará un error "constant expression required" when trying to create a switch case block, y deberéis hacer lo siguiente: *https://stackoverflow.com/questions/76430646/constant-expression-required-when-trying-to-create-a-switch-case-block*

Para seguir este ejemplo deberemos de abrir el Logcat (desde la pestaña que está en la ventana inferior del IDE AdnroidStudio), una vez que lo tengamos implementado y listo para ejecutar:

En el archivo java de MainActivity:

```
public class MainActivity extends AppCompatActivity {

    // Declaramos los objetos gráficos:
    private RadioGroup rgGrupo;
    private RadioButton rb1;
    private RadioButton rb2;
    private CheckBox chb1;
    private CheckBox chb2;

    @Override

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // enlazamos objetos gráfico al código
        chb1 = findViewById(R.id.checkBox1);
        chb2 = findViewById(R.id.checkBox2);

        rgGrupo = findViewById(R.id.radioGroup);
        rb1 = findViewById(R.id.radioButton1);
        rb2 = findViewById(R.id.radioButton2);


        // damos funcionalidad a los RadioButtons:
        rgGrupo.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                switch (group.getId()) {
                    case R.id.radioGroup:
                        switch (checkedId) {
                            case R.id.radioButton1:
                                System.out.println("Has marcado el radiobutton 1");
                                break;
                            case R.id.radioButton2:
                                System.out.println("Has marcado el radiobutton 2");
                                break;
                        }
                }
            }
        });
```

```
    }

    /**
     *  Damos funcionalidad a los CheckBoxs:
     *  Esta vez usamos el método que enlazamos en el archivo xml del activity_main, asociado
a los checkboxs
     */

    public void onClickCheckBox(View v) {
        switch (v.getId()) {
            case R.id.checkBox1:
                System.out.println("Has marcado el checkbox 1");
                break;
            case R.id.checkBox2:
                System.out.println("Has marcado el checkbox 2");
                break;
        }
    }

}
```

En el xml de activity_main:

```
<CheckBox
    android:id="@+id/checkBox1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="68dp"
    android:layout_marginTop="64dp"
    android:onClick="onClickCheckBox"
    android:text="CheckBox 1"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<CheckBox
    android:id="@+id/checkBox2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="68dp"
    android:layout_marginTop="24dp"
    android:onClick="onClickCheckBox"
    android:text="CheckBox 2"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/checkBox1" />

<RadioGroup
    android:id="@+id/radioGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="65dp"
    android:layout_marginTop="32dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/checkBox2">

    <RadioButton
        android:id="@+id/radioButton1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="RadioButton 1" />
    <RadioButton
        android:id="@+id/radioButton2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="RadioButton 2" />

</RadioGroup>
```

Listas
Spinner

Otro elemento que vamos a usar mucho son las **listas desplegables**. En la asignatura de programación vimos que teníamos un elemento gráfico llamado JComboBox que nos permitía mostrar una lista de elementos para poder elegir de entre uno de ellos. El equivalente en Android será la clase **Spinner**, que nos va a permitir realizar exactamente la misma funcionalidad que los JComboBox en Java.

Los elementos que queramos mostrar en un Spinner deberemos colocarlos dentro del fichero **strings.xml** como un **array** y, mediante la propiedad **entries** del panel gráfico, podremos asignarle su valor como **@array/valores**.

Archivo **strings.xml**:

```
<resources>
   <string name="app_name">Ejemplo ImageView Spinner</string>
   <array name="valores_spinner">
      <item>Valor 1</item>
       <item>Valor 2</item>
       <item>Valor 3</item>
   </array>
</resources>
```

Listas optimizadas
Creación de los componentes

Las **listas** son elementos que nos van a permitir mostrar **varios elementos** en pantalla. Concretamente, es el componente **RecyclerView** el que nos va a permitir mostrar en pantalla **colecciones grandes de datos**, teniendo en consideración el **trato eficiente de la memoria** del dispositivo. El RecyclerView no va a hacer prácticamente nada por sí mismo, sino que se va a sustentar sobre otros **componentes complementarios** para determinar **cómo acceder a los datos** y **cómo mostrarlos**. Los más importantes serán los siguientes:

- **RecyclerView.Adapter**: Este representa un adaptador que gestionará **cómo se mostrarán** los elementos en pantalla.
- **RecyclerView.ViewHolder**: Esta clase se encargará de la **gestión de la memoria** del dispositivo a la hora de mostrar la lista optimizada. Nosotros únicamente le tendremos que **indicar qué elementos habrá**.
- **ItemDecoration**: Con estos elementos podremos **añadir efectos** a nuestras listas como, por ejemplo, una separación más grande entre los elementos.

Lo primero que tendremos que hacer es crear un elemento en la **carpeta layout**, que será la **interfaz** de los **elementos** de nuestra lista optimizada. Esto será una **pantalla normal y corriente**.

Creamos una clase que representará lo que queramos **mostrar en nuestra** lista. Esta clase tendrá los **métodos básicos**.

El siguiente paso será **crear un adaptador** mediante una clase Java a la que llamaremos AdaptadorEjemplo.

En el adaptador tendremos que crear los siguientes **elementos**:

Métodos de un adaptador básico

Elemento	Función
Crear clase **estática interna**	Hereda de **RecyclerView.ViewHolder** que representará un elemento de la lista
Método **onCreateViewHolder**	Obtendrá la **interfaz gráfica** de los elementos de la lista
Método **onBindViewHolder**	(Enlazador) Mostrará los **datos** de los elementos de la lista
Método **getItemCount**	**Devolverá los elementos** que tendrá la lista

Clase de un elemento de la lista AdaptadorEjemplo.java:

```
public class ElementoLista {
    private String texto1, texto2;

    public String getTexto1() {
        return texto1;
    }

    public void setTexto1(String texto1) {
        this.texto1 = texto1;
    }

    public void setTexto2(String texto2) {
        this.texto2 = texto2;
    }

    public String getTexto2() {
        return texto2;
    }

    public ElementoLista(String texto1, String texto2) {
        this.texto1 = texto1;
        this.texto2 = texto2;
    }
}
```

Adaptador de la lista completo:

```
public class AdaptadorEjemplo extends
RecyclerView.Adapter<AdaptadorEjemplo.HolderEjemplo> {

    /**
     * Clase interna equivalente al Holder de los elementos.
     * RecyclerView.ViewHolder: Esta clase se encargará de la gestión de la memoria del
dispositivo a la hora de mostrar la lista optimizada.
     * Nosotros únicamente le tendremos que indicar qué elementos habrá.
     */
    public static class HolderEjemplo extends RecyclerView.ViewHolder {
        TextView tEjemplo1, tEjemplo2;

        HolderEjemplo(View itemView) {
            /*
```

```
            En el constructor obtendremos los recursos del fichero
            de recursos xml que tengamos asociado a la clase, en
            este caso el fichero elemento.xml
        */
        super(itemView);
        tEjemplo1 = itemView.findViewById(R.id.tEjemplo1);
        tEjemplo2 = itemView.findViewById(R.id.tEjemplo2);
        // Si hubiera que sobrecargar eventos se haria aqui
    }
}

;

// Atributos de la clase AdaptadorEjemplo
private ArrayList<ElementoLista> elementos;
private Context contexto;

/**
 * Constructor de la clase
 *
 * @param contexto  Contexto de la aplicación
 * @param elementos Elementos de la lista
 */
public AdaptadorEjemplo(Context contexto, ArrayList<ElementoLista> elementos) {
    this.contexto = contexto;
    this.elementos = elementos;
}

/**
 * Agrega los datos que queramos mostrar
 *
 * @param datos Datos a mostrar, en este caso, los elementos básicos de la lista
 */
public void add(ArrayList<ElementoLista> datos) {
    elementos.clear();
    elementos.addAll(datos);
}

/**
 * Actualiza los datos del ReciclerView
 */
public void refrescar() {
    notifyDataSetChanged();
}
```

```
    @Override
    /**
     * Este método irá creando uno a uno los elementos de la lista optimizando el uso de
memoria del dispositivo
     */
    public HolderEjemplo onCreateViewHolder(ViewGroup parent, int viewType) {
        View v = LayoutInflater.from(parent.getContext()).inflate(R.layout.elemento, parent,
                false);
        HolderEjemplo pvh = new HolderEjemplo(v);
        return pvh;
    }

    @Override
    /**
     * Este método mostrará los datos de cada elemento que esté visible en la lista
     */
    public void onBindViewHolder(HolderEjemplo elementoactual, int position) {
        // Ponemos los datos correspondientes al titular de posicion position
        elementoactual.tEjemplo1.setText(elementos.get(position).getTexto1());
        elementoactual.tEjemplo2.setText(elementos.get(position).getTexto2());
    }

    @Override
    /**
     * Este método devolverá la cantidad de elementos que hay en la lista
     */
    public int getItemCount() {
        return elementos.size();
    }

    @Override
    public void onAttachedToRecyclerView(RecyclerView recyclerView) {
        super.onAttachedToRecyclerView(recyclerView);
    }

}
```

Crear la lista

Una vez que hemos...

- Creado la **interfaz gráfica** del **elemento** de la lista.
- **Creado una clase** para representar el **elemento** de la lista.
- Creado un **adaptador completo** con el **holder interno** para mostrar los elementos de la lista.

... podremos **crear la lista optimizada y agregar todos los elementos** que queramos.

Para esto, debemos seguir los siguientes **pasos** en el proyecto donde queramos mostrar nuestra lista optimizada:

1. Crear un **objeto** de tipo **RecyclerView** e integrarlo con su equivalente **elemento gráfico**.
2. A continuación, crear tantos **elementos** como queramos en nuestra lista.
3. No se nos puede olvidar crear un **adaptador de elementos para la lista**; en nuestro caso, podremos crear un **LinearLayoutManager** para que los elementos se muestren en forma de **lista normal** y corriente.
4. Creamos el **adaptador** al que le **pasaremos un array** con todos los **elementos** que queramos mostrar.

Como podemos observar, cuando se ejecute nuestra aplicación con la lista, se irán mostrando los elementos que únicamente quedan en pantalla, y tendremos que **hacer scroll** para poder **ver los demás**. Esto se debe a que estamos usando una **lista optimizada** en **memoria**. Únicamente estarán los elementos que estén en pantalla y cuando vayamos **mostrando elementos nuevos** esa **memoria se irá reutilizando**; de esta manera, **no se gastará memoria** de forma indebida.

Si quisiéramos mostrar los elementos en forma de **matriz de tres elementos**, podremos usar un **GridLayoutManager**.

Código en el **MainActivity.java**:

```
public class MainActivity extends AppCompatActivity {
    private RecyclerView listaEjemplo;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Ligo los elementos gráficos
        listaEjemplo = findViewById(R.id.listaEjemplo);

        // Creamos 20 elemento de la lista
        ArrayList<ElementoLista> elementos = new ArrayList<>();

        for (int i = 1; i <= 20; i++)
        {
            ElementoLista elemento = new ElementoLista("Elem " + i + " - T1", "Elem " + i + " -
T2");
            elementos.add(elemento);
        }

        // Con esto el tamaño del recyclerwiew no cambiará
        listaEjemplo.setHasFixedSize(true);
        // Creo un layoutmanager LinearLayoutManager para el recyclerview
        LinearLayoutManager llm = new LinearLayoutManager(this);
        // Creo un layoutmanager GridLayoutManager para el recyclerview
        // LinearLayoutManager llm = new GridLayoutManager(this,3);
        listaEjemplo.setLayoutManager(llm);
        // Creamos el adaptador
        AdaptadorEjemplo adaptador = new AdaptadorEjemplo(this, elementos);
        // Indicamos que el adaptador de la lista en el que hemos creado
        listaEjemplo.setAdapter(adaptador);
        adaptador.refrescar();

    }

}
```

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Funcionalidades Android

NavigationDrawer
SharedPreferences
Intent
Librerías de terceros

Media**nte los menús laterales izquierdos** podremos **navegar** por **diferentes pantallas** en nuestra aplicación.

Las preferencias nos ayudarán con la configuración de las aplicaciones.

Abriremos una pantalla desde otra mediante lo que conocemos como **Intents**.

Aprenderemos cómo podemos integrar una **biblioteca de terceros** en nuestra aplicación.

NavigationDrawer

Es un menú deslizante a la izquierda, que nos permite acceder a diferentes pantallas. Este tipo de menú deslizante se llama NavigationDrawer y, para poder crear una aplicación con este tipo de menú, deberemos seguir los siguientes **pasos**:

• En Android Studio, cuando pulsemos 'Crear una **nueva** aplicación', en la parte de **plantillas** deberemos seleccionar la plantilla llamada **Navigation Drawer Activity**, la cual nos creará un proyecto cuya pantalla principal tendrá este menú deslizante a la izquierda.

Una vez creado el **proyecto**, este se va a **dividir en varias partes**:

• Las **interfaces gráficas de cada una de las pantallas se van a encontrar en la carpeta res**/layout y se llamarán ' **fragment _nombredelfragment.xml**'.

• El **código** de cada una de las pantallas se ubicará en la **carpeta java/ui**, donde tendremos un paquete por cada uno de los menús disponibles. En cada paquete hay **dos clases**, un **Fragment** y un **ViewModel**. En este caso, ViewModel lo eliminaremos y utilizaremos únicamente Fragment.

• Los **elementos del menú desplegable** están definidos en el fichero **activity_main_drawer.xml**.

• En el fichero **mobile_navigation.xml** tendremos declarados los **fragments** junto con su funcionalidad, es decir, junto a la **pantalla que abrirán** cuando se pulse dicho menú al que están asociados.

En la **carpeta layout** tenemos un layout para el fragment, y para la **actividad de drawer layout** tenemos los siguientes ficheros XML:

• **activity_main.xml**: Es el fichero de interfaz gráfica **principal** que **contiene el NavigationDrawer**.

• **app_bar_main.xml**: Este fichero es el que contiene el **botón flotante**, la **ToolBar** principal y al fichero '**content_main.xml**':

• **content_main.xml**: Tiene el contenido que se mostrará en la actividad principal. Contiene un **nav_host_fragment**, que es el layout que nos va a permitir **navegar por los fragment** de los menús en nuestra aplicación. El nav_host_fragment **contiene el fragment_home**:

• **fragment_home.xml**: Este fragment contiene los **elementos del menú** principal de la actividad que es el que se va a mostrar por defecto en el **NavigationDrawer**.

Menú deslizante en otras pantallas

Es posible también crear una pantalla con un **NavigationDrawer**, **no** teniendo que **ser ésta la pantalla principal**, por ejemplo, podemos tener una pantalla de login, y que ésta redirija (si el login es correcto) a otra con un NavigationDrawer.

Para esto, al **crear una nueva actividad** podemos seleccionar la pantalla de **NavigationDrawer**, siendo ya el proceso de *creación de forma idéntica* a crear la aplicación directamente.

Agregar un nuevo menú en NavigationDrawer

Para crear un nuevo menú en NavigationDrawer y otorgarle la funcionalidad de abrir una nueva actividad, tenemos que hacer lo siguiente:

• Primero debemos crear un **nuevo paquete en ui** para almacenar las **clases** de la **nueva actividad**.

• Una vez creado el paquete, debemos **crear los fragments de la nueva actividad** que queremos; para ello, pulsamos con el botón derecho en el nuevo paquete creado y seleccionamos **nuevo → fragment →** fragment (**with ViewModel**). **Eliminamos el ViewModel**.

• En el código Java del nuevo fragment deberemos hacer **que se «infle» (cargue) su layout** y ya podremos **trabajar** como hasta ahora estamos acostumbrados. **Agregamos el nuevo menú** a '**activity_main_drawer.xml**'. Todos los menús deben tener estos **3 elementos obligatoriamente**:

• **android:id**: Es el **identificador** único del **menú**.

• **android:icon**: Es el icono que **aparecerá en el menú** desplegable junto al menú.

• **android:title**: Es el texto que **aparecerá en el menú** desplegable junto al icono. **Deberá ir en** '**strings.xml**'.

• **Agregamos el fragment a 'mobile_navigation.xml'** y el menú tendrá su funcionalidad de abrir su fragment.

Si nos fijamos, el **menú de 'home'** siempre va a estar en el DrawerLayout y **no va a tener una funcionalidad útil**, ya que las demás pantallas, al tener la flecha de volver hacia atrás, ya nos permiten volver a la pantalla principal. Por lo tanto, al pulsar sobre este elemento, se nos abre la pantalla principal, que ya tendremos abierta. Con la propiedad **android:visible = false** hacemos que el menú no aparezca. El código para que la pantalla de 'home' se oculte en el menú será:

Ocultar pantalla home:

```
<item
    android:id="@+id/nav_home"
    android:icon="@drawable/ic_menu_camera"
    android:title="@string/menu_home"
    android:visible="false"/>
```

Preferencias (SharedPreferences)

Cuando hablamos de «**preferencias**» de una aplicación nos referimos a una serie de **datos de configuración** que se **almacenarán dentro de nuestra aplicación** y que podremos **utilizar en el momento que necesitemos**. carga

El manejo de las preferencias es muy sencillo. Para ello, debemos utilizar la **clase SharedPreferences**, que va a representar una colección de preferencias, es decir, que podremos tener todas las que consideremos oportuno sin límite.

Las **colecciones** de preferencias se identificarán mediante un **identificador único**, por lo que una misma aplicación podrá gestionar **varias colecciones** de preferencias sin ningún problema, y **sin que se solapen** unas con otras.

Para obtener una **referencia a una colección** de preferencias, tendremos que usar el **método getSharedPreferences()**, al que tendremos que **pasar** como parámetro ese **identificador único**.

A las preferencias debemos **acceder con un modo de acceso**, el cual nos indicará **qué operación** podremos hacer con ellas. Existen **tres modos de acceso** diferentes:

1. **MODE_PRIVATE**: Solo nuestra aplicación tiene acceso a estas preferencias.
2. **MODE_WORLD_READABLE**: Todas las **aplicaciones pueden leer** estas preferencias, pero solo la **nuestra puede modificarlas**.
3. **MODE_WORLD_WRITABLE**: Todas las **aplicaciones** pueden **leer y modificar** estas preferencias.

Para agregar una preferencia, debemos **crear un objeto SharedPreferences.Editor** y, mediante los **métodos put**, podremos agregar las preferencias que queramos. Estas tendrán un **identificador único** y un **valor**.

Para **obtener** una preferencia podremos utilizar el **método get**, al cual le pasaremos el **identificador y** un **valor** por **defecto**, por si no encuentra dicha preferencia.

Usos de las preferencias:

El **usuario** que ha hecho login.

El **tipo** de usuario (ADMIN, USER).

El **tema** de la aplicación que esté utilizando el usuario.

Settings Activity

La **PreferenceActivity** es la típica pantalla de configuración que tienen todas las aplicaciones Android.

Para crear esta pantalla, tenemos una **plantilla** llamada **Settings Activity**.

En este tipo de pantalla se gestionará toda la configuración de nuestra aplicación y **se guardará en** las **preferencias** de la misma, las cuales podremos recuperar en el momento que queramos, tal y como hemos visto anteriormente.

Dentro de esta, podremos incluir una **lista de opciones de configuración** organizada por **categorías**, mediante el elemento **PreferenceCategory**, al que tendremos que dar un **texto descriptivo** mediante el atributo **android:title**.

Dentro de cada categoría podremos añadir cualquier elemento de configuración, entre los que podemos destacar:

- **CheckBoxPreference**: Marca **seleccionable**.
- **EditTextPreference**: Cadena simple de **texto**.
- **ListPreference**: Lista de valores **seleccionables** (**exclusiva**).
- **MultiSelectListPreference**: Lista de valores **seleccionables** (**múltiple**).

Intents

Debemos poder **movernos** de forma fluida **por una serie de pantallas interconectadas**.

Si queremos tener **más de una actividad** en nuestro proyecto, podremos crear las que necesitemos haciendo clic derecho sobre el **paquete Java** y seleccionando **New Activity**. De esta forma, nos aparecerán todas las **plantillas de actividades** de las que dispone Android Studio, entra las cuales la más usual es la actividad vacía.

Una vez hemos seleccionado la actividad que queremos crear, nos aparecerá el asistente de creación de actividades, donde podremos indicar el **nombre** y el **lenguaje** en el que la queremos crear (**Java** o **Kotlin**), pudiendo **mezclar los dos códigos** en un mismo proyecto sin problema alguno.

Cuando hayamos creado la actividad nueva, tendremos **dos ficheros adicionales** en el proyecto, uno con el código de la **funcionalidad** de la actividad y otro en la carpeta layout con el código XML de la **interfaz gráfica**.

Podemos **invocar otra actividad**:

> Necesitaremos crear un **objeto** de la clase **Intent**, al que **podremos agregar parámetros** para pasar **entre actividades**, de la siguiente forma:
>
> Objeto Intent:

```
//Iniciando la actividad Visor
Intent intent = new Intent(this, ActividadFinal.class);
// Agregamos parámetros al intent, agregamos de nuestra cadena
intent.putExtra("datos", "Este dato es de la anterior Activity");
// Iniciamos la actividad nueva
startActivity(intent);
```

Agregar bibliotecas de terceros a nuestra app

Una de las cosas que podemos hacer en nuestros proyectos es integrar una biblioteca de terceros, la cual nos ofrecerá **funcionalidades extra**. Este tipo de bibliotecas suele alojarse en sitios como Github, donde sus creadores suben su código fuente y las instrucciones para integrarlas en un proyecto de Android Studio.

El procedimiento para ello consiste en que una vez tengamos localizada la biblioteca que necesitamos, por norma general, el **autor nos va a proporcionar las instrucciones** para integrarla a nuestro proyecto de una forma relativamente sencilla, pudiendo así utilizarla sin problemas.

Como último apunte, existe una **aplicación** que integra **muchas** de estas **bibliotecas**. Se llama **Libraries for Developers** y se encuentra en la **Play Store** de forma totalmente gratuita.

Tendrás información acerca del autor, capturas , licencia, descripción, links de la librería y podrás ver un ejemplo de la librería en funcionamiento dentro de la propia aplicación.La mayoría de las aplicaciones están recogidas en github, otras son de google code y bitbucket.

https://play.google.com/store/search?q=Libraries%20for%20Developers&c=apps&hl=es_419&gl=US

En la siguiente tabla vemos algunas de las bibliotecas que podremos insertar en nuestros proyectos, junto a lo que hacen y su **página**, donde estarán las **instrucciones para insertarlas**.

Toasty: Para crear fácilmente Toast personalizados.

https://github.com/GrenderG/Toasty

Material TextField: Creación de cajas de texto personalizadas.
https://github.com/florent37/MaterialTextField

LoadingView: Con esta biblioteca se pueden crear pantallas de carga.
https://github.com/ldoublem/LoadingView

Qbutton: Elaboración de botones personalizados.

https://github.com/manojbhadane/QButton

About Page: Permite el desarrollo de una pantalla del tipo «Acerca de». Además, proporciona un menú de contacto.

https://github.com/medyo/android-about-page

Circle Image View: Permite montaje de imágenes redondas.
https://github.com/Shashank02051997/FancyAlertDialog-Android

FancyAlertDialog-Android: Elaboración de pantallas de confirmación personalizadas.
https://github.com/Shashank02051997/FancyAlertDialog-Android

Sweet Alert Dialog: Desarrollo de diálogos personalizados.

https://github.com/pedant/sweet-alert-dialog

Vamos a ver paso a paso cómo podemos integrar una, concretamente, la de los emoticonos de las aplicaciones de mensajería instantánea como WhatsApp o Telegram.

Para agregar la funcionalidad de Emojis en nuestra app, similar a whatsApp nos vamos a las instrucciones de esta página:

https://github.com/vanniktech/Emoji

PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TÉCNICO EN DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Sonidos y bases de datos

Reproducción de sonido

En Android tenemos la posibilidad de reproducir sonidos **sin tener que recurrir a bibliotecas de terceros** mediante dos clases:

1. **MediaPlayer**: Esta clase se usa para poder reproducir archivos de **audio largos**, es decir, para reproducir, por ejemplo, música de fondo.
2. **SoundPool**: En cambio, esta se usa para reproducir archivos de **audio muy cortos**, tales como efectos de botones. Con esta clase existe una restricción de tamaño del fichero a un **máximo de 1 Mb**.

Al igual que ocurría cuando teníamos recursos de texto, imágenes y demás, los sonidos de nuestras aplicaciones deberán ir en un **directorio específico**. Concretamente, deberemos crear una carpeta llamada raw dentro del directorio /res.

Para crear un objeto de la clase **MediaPlayer** utilizamos su método create:

```
MediaPlayer mp = MediaPlayer.create(this, R.raw.sonido);
```

Esta clase posee los siguientes **métodos** interesantes:

- **play()**: Inicia la reproducción del sonido.
- **pause()**: **Pausa o vuelve a iniciar** la reproducción del sonido.
- **stop()**: Detiene la reproducción del sonido.
- **release()**: **Libera** los recursos del sonido. Este método siempre hay que instanciarlo **después de llamar a stop()**.

Ejemplo de uso de MediaPlayer:

```
MediaPlayer mp;
mp = MediaPlayer.create(this, R.raw.cancion);
// Inicia la música
mp.start();
// pausa la música
mp.pause();
// reanuda la música
mp.pause();
// Para la música
mp.stop();
// Libero los recursos del sonido
mp.release()
```

En el caso de **SoundPool** podremos crear una función cuyo código lo vemos a continuación:

```
/**
 * Prepara un sonido para ser reproducido
 *
 * @param sonido Identificador del sonido a reproducir
 */
private void crearSoundPool(int sonido) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        AudioAttributes attributes = new AudioAttributes.Builder()
                .setUsage(AudioAttributes.USAGE_GAME)
                .setContentType(AudioAttributes.CONTENT_TYPE_SONIFICATION)
                .build();
        sp = new SoundPool.Builder()
                .setAudioAttributes(attributes)
                .build();
    } else {
        sp = new SoundPool(5, AudioManager.STREAM_MUSIC, 0);
    }
    flujodemusica = sp.load(this, R.raw.vida, 1);
    sp.setOnLoadCompleteListener(new SoundPool.OnLoadCompleteListener() {
        public void onLoadComplete(SoundPool soundPool, int sampleId, int status) {
            loaded = true;
        }
    });
}
```

Cuando utilizamos las clases MediaPlayer y SoundPool para la reproducción de sonido, **estas se encargarán de realizar todas las conversiones de datos multimedia necesarias** para que los ficheros puedan ser reproducidos de forma correcta por el dispositivo móvil que estemos utilizando.

Los formatos y clips de audio que nos permitirán reproducir estas clases son el .mp3, .wav, .midi, .mp4, ogg, etc.

Android también nos permitirá **reproducir clips de vídeo**, aunque este tema sí está muy delimitado, ya que **solo permite unos ciertos códecs de vídeo**.

Precaución con los sonidos

La transmisión de la información contenida en los archivos multimedia de las apps, se realiza a tiempo real a través de **internet** mediante un **protocolo** denominado **RTP**, que se corresponde a las siglas **Real-time Transport Protocol**, en inglés.

Esta transmisión a tiempo real, no nos debe confundir, ni alejar, de la **precaución** con respecto al **espacio en memoria** que debemos considerar para las aplicaciones.

Como hemos comentado en unidades anteriores, tenemos que tener cuidado con el tamaño de los recursos que introducimos en nuestros desarrollos en entornos móviles, ya que todo lo que se vaya añadiendo, se irá sumando al tamaño de la aplicación en sí.

Con los sonidos y/o archivos multimedia en general, tenemos que tener especial atención, ya que cualquier sonido que se precie ocupará bastante espacio a poco que queramos que tenga una mínima calidad, así que tendremos que ser bastante selectivos en este caso, o en su defecto, **comprimir** los **sonidos** intentando tener la mínima pérdida de calidad.

Bases de datos

En nuestros dispositivos Android, por razones obvias, no vamos a poder instalar un gestor de bases de datos como MySQL, así que las bases de datos que vamos a utilizar van a ser **embebidas**. Esto quiere decir que la base de datos será un **fichero dentro de nuestra aplicación**, con el inconveniente de que, si **desinstalamos** la aplicación, la **base de datos se perderá**.

Para la gestión de las bases de datos vamos a utilizar **SQLite**. Android incorpora de serie todas las herramientas necesarias (bibliotecas, librerías, etc.) para la **creación y gestión de bases de datos SQLite** y, entre ellas, una **completa API** para desarrollar de manera sencilla todas las tareas necesarias. Para poder utilizar una base de datos SQLite será necesario **heredar** de la clase **SQLiteOpenHelper**, la cual tiene únicamente **un constructor** y **dos métodos abstractos**, **onCreate()** y **onUpgrade()**, que deberemos sobrescribir.

En el método **onCreate** deberemos ejecutar **consultas del tipo 'create table'** para poder **crear** las tablas de nuestra base de datos, mientras que en el método **onUpgrade** deberemos realizar las operaciones necesarias para **actualizar** la base de datos, es decir, deberemos **realizar una copia de seguridad** de la misma para poder volver a volcarla, ya que este método se llamará automáticamente cuando actualicemos su versión.

También deberemos crear un método para **cerrar la conexión** de la base de datos. Para **almacenar** la base de datos deberemos crear un objeto de tipo **SQLiteDatabase**.

La clase de SQLite que nos va a permitir **insertar datos** es **SQLiteDatabase**.

Creación de una BD simple de usuarios:

```
public class BDUsuarios extends SQLiteOpenHelper {
    private Context contexto;
    // Sentencia SQL para crear la tabla de Usuarios
    private final String SQLCREATE = "CREATE TABLE Usuarios (codigo INTEGER, nombre
TEXT)";
    // Sentencia SQL para eliminar la tabla de Usuarios
    private final String SQLDROP = "DROP TABLE IF EXISTS Usuarios";
    // Base de datos
    private SQLiteDatabase bd = null;

    public static final int DATABASE_VERSION = 1;
    public static final String DATABASE_NAME = "DBUsuarios.db";

    /**
     * Constructor de la clase
     *
     * @param contexto Contexto de la aplicación
     */
```

```java
    public BDUsuarios(Context contexto) {
        super(contexto, DATABASE_NAME, null, DATABASE_VERSION);
        this.contexto = contexto;
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(SQLCREATE);
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int versionAnterior, int versionNueva) {
        db.execSQL(SQLDROP);
        db.execSQL(SQLCREATE);
    }
    public void cerrarBD() {
        if (bd != null)
            bd.close();
    }
    /**
     * Inserta un nuevo usuario en la base de datos
     *
     * @param codigo Código del usuario
     * @param nombre Nombre del usuario
     * @throws Exception Esta excepción se lenzará si ocurre algún error en la inserción
     */
    public void insertarUsuario(int codigo, String nombre) throws Exception {
        // Obtengo los datos en modos de escritura
        bd = getWritableDatabase();
        // Si hemos abierto correctamente la base de datos
        if (bd != null) {
            long newRowId;
            try {
                // Creo un ContentValues con los valores a insertar
                ContentValues values = new ContentValues();
                values.put("codigo", codigo);
                values.put("nombre", nombre);
                // Inserta la nueva fila, devolviendo el valor de la clave
                // primaria de la nueva fila
                newRowId = bd.insert("Usuarios", "", values);
                cerrarBD();
            } catch (Exception e) {
                throw new Exception(e.toString());
            }
        }
    }
}
```

Qué bases de datos podremos utilizar en las aplicaciones Android

El lenguaje SQL ya lo conocemos y sabemos que nos proporciona una gran funcionalidad. Además, es relativamente fácil de utilizar. El uso de las bases de datos en las aplicaciones Android **es algo diferente** que el de las aplicaciones de escritorio, ya que, en estas vamos a usar un **gestor de bases de datos embebidas**, concretamente, este gestor de bases de datos será **SQLite**.

SQLite nos va a permitir **trabajar de dos formas distintas**.

Una mediante **consultas SQL**,

Otra mediante una serie de **clases que ya están implementadas** e integradas y que nos facilitarán las tareas de inserción, borrado, actualización y consulta de datos, para lo que **no tenemos que usar SQL para esto**. Como es lógico, lo más recomendable es usar esta segunda forma, ya que estas clases estarán **optimizadas para dichos efectos**.

Insertar valores

Usaremos el método **getWritableDatabase()** para obtener una **referencia a la base de datos en modo escritura**. Esto es muy importante ya que, si no obtenemos la referencia que nos permita escribir en la base de datos, no lo podremos hacer.

Una vez tengamos la referencia en modo escritura a la base de datos, ya podemos realizar todas las acciones que queramos sobre ella. Para **insertar datos** podemos utilizar el método **execSQL()**, al que le podremos pasar las **sentencias insert** correspondientes, pero existe **otra forma mucho más fácil de hacerlo**.

Podemos utilizar el **método insert**, al que le pasaremos el nombre de la **tabla** y los **valores** que queramos insertar.

Estos **valores** los deberemos **almacenar** en un **objeto de tipo ContentValues**. Mediante el método **put**, podremos indicarle al objeto los **valores a almacenar**, indicando el nombre del **atributo** y su **valor** correspondiente.

Por último, **cerramos la conexión** con la base de datos llamando al método **close()**.

```
public void insertarUsuario(int codigo, String nombre) throws Exception {
    // Obtengo los datos en modos de escritura
    bd = getWritableDatabase();
    // Si hemos abierto correctamente la base de datos
    if (bd != null) {
        long newRowId;
        try {
            // Creo un ContentValues con los valores a insertar
            ContentValues values = new ContentValues();
            values.put("codigo", codigo);
            values.put("nombre", nombre);
            // Inserta la nueva fila, devolviendo el valor de la clave
            // primaria de la nueva fila
            newRowId = bd.insert("Usuarios", "", values);
        } catch (Exception e) {
            throw new Exception(e.toString());
        }
    }
}
```

Borrar valores

Al igual que para insertar valores en nuestras tablas de la base de datos, cuando necesitemos borrarlos **también** usaremos el método **getWritableDatabase()** para obtener una **referencia a la base de datos** en modo **escritura**.

Para poder eliminar datos, podemos utilizar el método **execSQL()**, al que le podremos pasar las **sentencias DELETE** correspondientes, pero, tal y como ocurría con la inserción, existe **otra forma mucho más fácil** de hacerlo.

SQLite nos provee de la **función delete**, que nos permite eliminar valores de una tabla de una forma muy sencilla y **sin** tener que **utilizar** una **sentencia DELETE de SQL**.

A esta función deberemos pasarle los siguientes parámetros:

- Nombre de la **tabla** de la que queremos eliminar los valores.
- Condición o **condiciones** para el borrado de datos (WHERE y atributos).
- **Valor** de las **condiciones** para el borrado de datos (valores de los atributos a borrar).

A la hora de **crear las condiciones** deberemos hacerlo de la siguiente forma:

Imaginemos que queremos eliminar los usuarios con un código mayor a 10, tendríamos:

codigo > ?

La **interrogación** le indica al método delete que hay un **valor** que deberá **obtener del parámetro** de **valores**.

Independientemente de si son valores numéricos o cadenas, **no deberán llevar comillas aquí**.

Podremos unir las condiciones con todos los **AND** y **OR** que necesitemos. No obstante, tendremos que poner todos esos valores en el **tercer parámetro**.

```
public void eliminarUsuario() throws Exception
{
    // Obtengo los datos en modos de escritura
    bd = getWritableDatabase();
    // Si hemos abierto correctamente la base de datos
    try
    {
        // Defino la parte del WHERE
        String selection = "codigo > ? AND nombre = ?";
        // Valores para borrar en orden
        String[] deleteArgs = { "10", "María" };
        bd.delete( table: "Usuarios", selection, deleteArgs);
    }
    catch (Exception e)
    {
        throw new Exception(e.toString());
    }
}
```

Actualizar valores

Al igual que para insertar o eliminar valores en nuestras tablas de la base de datos, cuando necesitemos actualizarlos, usaremos **también** el método **getWritableDatabase()** con el objetivo de obtener una **referencia** a la **base de datos** en **modo escritura**.

Para **eliminar** datos podemos utilizar el método **execSQL()**, al que le podremos pasar las **sentencias UPDATE** correspondientes, pero también existe **otra forma mucho más** fácil de hacerlo.

SQLite nos provee de la **función update**, la cual nos va a permitir actualizar los valores de una tabla de una forma muy sencilla y **sin** tener que **utilizar una sentencia UPDATE de SQL**. A esta función deberemos pasarle los siguientes **parámetros**:

- Nombre de la **tabla** de la que queremos actualizar los valores.
- **Nuevos valores** de los elementos en un objeto de tipo **ContentValues**.
- Condición o **condiciones** para la **actualización** de datos.
- **Valor** de las **condiciones** para la actualización de datos.

Las condiciones de la actualización de los datos van a funcionar exactamente **igual** que cuando se trata de **borrar datos**, y podremos unir **tantas como queramos** con los operadores **AND y OR**.

Al igual que cuando eliminamos valores, las **interrogaciones no** tendrán que contener ningún tipo de **comillas**, ni simples ni dobles, ya que eso lo **procesará automáticamente** la función **update** de SQLite.

```
public void actualizarUsuarioPorNombre(String nombreantiguo, int codigo, String
        nombrenuevo) throws Exception
{
    // Obtengo los datos en modos de escritura
    bd = getWritableDatabase();
    // Si hemos abierto correctamente la base de datos
    try
    {
        // Creo un ContentValues con los valores a insertar
        ContentValues values = new ContentValues();
        values.put("codigo", codigo);
        values.put("nombre", nombrenuevo);
        // Defino la parte del WHERE
        String selection = "nombre = ?";
        // Valores para borrar en orden
        String[] updateArgs = { nombreantiguo };
        bd.update( table: "Usuarios", values, selection, updateArgs);
    }
    catch (Exception e)
    {
        throw new Exception(e.toString());
    }
}
```

Consultar valores

Para poder consultar los valores de las tablas de la base de datos, debemos usar el método **getReadableDatabase()** con el objetivo de obtener una **referencia** a la **base de datos** en modo **lectura**.

SQLite nos provee de la **clase Cursor**, la cual nos permite **seleccionar los valores** de una tabla de una forma muy sencilla, y **sin** tener que **utilizar una sentencia SELECT de SQL**.

Esta clase nos provee de la **función query**, a la cual le tendremos que pasar los siguientes **parámetros**:

- Nombre de la **tabla** cuyos valores queremos consultar.
- **Columnas** que queremos que nos devuelva. Si las queremos **todas** dejaremos el valor **null**.
- **Columnas** de la cláusula **WHERE**.
- **Valores** de la columna de la cláusula **WHERE**.
- **Valores** de la cláusula **GROUP BY**.
- **Valores** de la cláusula **HAVING**.
- **Orden** de la cláusula **ORDER BY**.

Las condiciones de la actualización de los datos van a funcionar exactamente **igual** que cuando se trata de **borrar datos**, y podremos unir **tantas como queramos** con los operadores **AND y OR**.

Una vez ejecutada la consulta, nos devolverá un **objeto de tipo Cursor**, donde nos tendremos que posicionar en el **primer valor** mediante el método **moveToFirst**. Este lo podremos recorrer mediante el método **moveToNext**, que se moverá al **siguiente elemento** siempre que lo haya devuelto 'true'; en el caso de que **no haya más elementos**, devolverá '**false**'.

Para *obtener* los valores tenemos los métodos **getString**, **getInt**, etc., a los que les tendremos que pasar el **índice del campo** que queremos recuperar.

Si queremos saber la **cantidad** de **elementos** recuperados por la consulta, disponemos del método **getCount()** en la **clase Cursor**.

También podremos ejecutar **directamente una consulta SQL** con el método **rawQuery**, al que le pasaremos como argumento la **consulta a ejecutar**, y nos **devolverá** un objeto de la clase **Cursor**.

Ejemplo completo de consulta de información (recuperar y mostrar) de una base de datos:

```
/**
 * Obtiene todos los usuarios de la base de datos
 *
 * @return Array con todos los usuarios de la base de datos
 * @throws Exception Esta excepción saltará cuando ocurra algún error recuperando los datos
 */
public ArrayList<Usuario> obtenerTodosUsuarios() throws Exception {
    ArrayList<Usuario> usuarios = new ArrayList<>();
    // Obtengo los datos en modo de lectura
    bd = getReadableDatabase();
    // Si hemos abierto correctamente la base de datos
    try {
        // Indico como quiero que se ordenen los resultados
        String sortOrder = "codigo ASC";
        // Creo el cursor de la consulta
        Cursor c = bd.query
                (
                        "Usuarios",   // Tabla para consultar
                        null,       // Columnas a devolver
                        null,      // Columnas de la clausula WHERE
                        null,   // Valores de la columna de la clausula WHERE
                        null,       // Valores de la clausula GROUP BY
                        null,        // Valores de la clausula HAVING
                        sortOrder           // Orden de la clausula ORDER BY
                );
        // Muestro los datos
        c.moveToFirst();
        if (c.getCount() > 0) {
            do {
                int codigo = c.getInt(0);
                String nombre = c.getString(1);
                usuarios.add(new Usuario(codigo, nombre));
            } while (c.moveToNext());
        }

        cerrarBD();

        return usuarios;
    } catch (Exception e) {
        throw new Exception(e.toString());
    }
}
```

Compartiendo datos

Cuando estamos tratando datos, es posible que necesitemos cierta información que está presente en nuestro dispositivo.

Esto es totalmente posible gracias a los **proveedores de contenidos** o **ContenProvider**, por su nombre en inglés.

Los ContentProvider son un mecanismo que nos proporciona Android para **compartir datos entre aplicaciones**.

Concretamente, nos permite **compartir el acceso a la información contenida dentro de una base de datos SQLite de una aplicación**, aunque la información sea inicialmente privada y solo accesible por la propia aplicación.

Si quieres crear un ContentProvider básico paso a paso, puedes hacerlo en el siguiente tutorial:

https://www.sgoliver.net/blog/content-providers-en-android-i-construccion/