Programación multimedia y dispositivos móviles

ILERNA

Introducción

A lo largo de los temas de este libro veremos el desarrollo de aplicaciones móviles en entorno Android con lenguaje Kotlin. El enfoque de este es completamente práctico desde el primer momento, aportando los conceptos teóricos que aplicamos en el desarrollo de una aplicación real.

A este enfoque práctico lo acompaña la simulación de ponerse en el papel del desarrollador de una empresa desarrollo de aplicaciones móviles, a la cual uno de sus clientes le ha pedido un proyecto para desarrollar una aplicación de gestión de tareas llamada IlernaTodo.

Como desarrollador, te enviarán diferentes casos de uso donde se indican los requisitos del cliente y qué se espera que haga la aplicación.

Comencemos este proyecto por los conceptos básicos y preparando el entorno para trabajar.

1. Análisis de tecnologías para aplicaciones en dispositivos móviles

En este primer tema veremos las principales características y los tipos de dispositivos móviles que podemos encontrar, sus limitaciones y su arquitectura, a la vez que exploraremos las herramientas necesarias para desarrollar aplicaciones para dispositivos móviles y cómo podemos ejecutar las aplicaciones sin necesidad de disponer de un depósito físico.

Al finalizar el tema siguiendo las prácticas, tendremos nuestro ordenador preparado para continuar con el desarrollo de aplicaciones móviles.

1.1. Limitaciones que plantea la ejecución de aplicaciones en dispositivos móviles: desconexión, seguridad, memoria, consumo de batería, almacenamiento

Los dispositivos móviles plantean serias limitaciones a la hora de ejecutar aplicaciones que debemos tener en consideración durante el desarrollo de la aplicación para tener un producto de calidad y robusto.

Estas limitaciones hoy en día no vienen dadas tanto por la capacidad de procesamiento o memoria como por el entorno en el que se utilizan estos dispositivos. Entre ellas, definimos las más relevantes:

Desconexión

Al tratarse de dispositivos que podemos llevar con nosotros en cualquier momento, lugar y permanentemente conectados, estos pueden sufrir desconexiones tanto de forma total como parcial, lo que, en el caso de las aplicaciones que utilicen datos de un servidor, provocará que se vean afectadas por la variabilidad de la conexión.

La aplicación que desarrollemos debe controlar esta situación y poder ofrecer un mecanismo que asegure, en la medida de lo posible, la disponibilidad de los datos.

Seguridad

En lo que se refiere seguridad física, la limitación viene dada por la propia virtud del dispositivo: al ser ligeros y fáciles de llevar, son a la vez susceptibles de ser más fácilmente sustraídos.

En el ámbito de la seguridad de la información, este tipo de dispositivos son más vulnerables, al permitir conectarse a redes poco seguras o la posibilidad de instalar aplicaciones de dudosa procedencia, unido a la gran cantidad de sensores, como cámaras, GPS o micrófonos, de que disponen.

En el desarrollo de aplicaciones debemos considerar usar los permisos mínimos necesarios para el funcionamiento de la aplicación. En el caso de que desarrollemos una aplicación con uso de datos, siempre deben viajar bajo una conexión segura o cifrada para evitar que los datos sean expuestos.

Consumo de batería

Por su propia naturaleza de ser lo más ligeros posible, la batería suele ser uno de los elementos más comprometidos, y el diseño de las aplicaciones debe tener en consideración qué recursos utiliza en cada momento, liberando aquellos que no sean necesarios para ahorrar los tan preciados miliamperios de la batería.

Memoria y almacenamiento

A día de hoy, los dispositivos móviles cada vez incorporan más memoria y capacidad de almacenamiento, aunque no por ello debemos desaprovecharla, ya que es un recurso

limitado que no podemos ampliar sin adquirir un nuevo dispositivo. Por ello, las aplicaciones móviles deben optimizar el uso de los recursos, ya que también debe compartirlos con otras aplicaciones.

1.2. Tecnologías disponibles

En el desarrollo de aplicaciones móviles, a lo largo de los últimos años han aparecido tecnologías que nos ayudan a la creación de aplicaciones según las necesidades del proyecto que debamos afrontar. Normalmente, los creadores de cada sistema operativo móvil han designado el lenguaje y el entorno de desarrollo que se debe utilizar para la creación de aplicaciones en su plataforma. Por ejemplo, Java era el único estándar para Android hasta que en 2017 Google anunció Kotlin, y el IDE era Eclipse hasta que en el 2013 pasó a ser Android Studio, basado en IntelliJ IDEA de JetBrains. De igual modo, en iOS se programaba en Objective-C hasta que se creó el nuevo lenguaje Swift en sus múltiples versiones, y el entorno de desarrollo es Xcode. Ya apenas hay terminales Windows Phone o Blackberry, pero si necesitásemos desarrollar algo para esas u otras plataformas, tendríamos que saber que cada una de ellas normalmente utilizará un lenguaje y un entorno diferentes.

¿Qué interés hay en utilizar otras tecnologías diferentes a las de cada plataforma? El motivo es ahorrar recursos. Imaginemos una empresa dedicada a programar juegos para Android e iOS. Tiene dos opciones: la primera es contratar un equipo de desarrolladores Android nativos y otro de desarrolladores iOS nativos que paralelamente desarrollen el mismo juego pero en plataformas diferentes, multiplicando por dos el coste en horas; la segunda es utilizar una tecnología híbrida que se programe una vez y funcione igualmente en iOS y Android, con lo que solo se necesita un equipo de desarrollo híbrido. La última opción parece prometedora, pero conlleva algunos inconvenientes: la mayoría de las tecnologías híbridas están basadas en web, es decir, se crea una especie de aplicación web que se mostrará en el dispositivo como si fuese nativa, utilizando algunas librerías para el acceso al *hardware*, como la cámara, etcétera. Esto hace que sean aplicaciones menos potentes, que no puedan acceder a ciertos tipos de *hardware* y que sean más lentas. Quizá por ello, estas tecnologías son populares durante un tiempo y luego dejan de usarse, lo que supone que los programadores tengan que cambiar constantemente de tecnología, impidiendo que ganen experiencia.

Otras tecnologías como Unity para juegos o Xamarin para aplicaciones son más potentes. Sin embargo, no son públicas e implican un gasto de licencia. Otro aspecto para tener en cuenta es que las plataformas móviles evolucionan rápidamente y no tienen ningún interés en ser compatibles con los *frameworks* híbridos ya existentes. Si eres un programador nativo, podrás adaptarte rápidamente a las nuevas interfaces o

funcionalidades, tendrás documentación y ejemplos, pero si utilizas un *framework*, tendrás que esperar a que los creadores del *framework* saquen una actualización compatible con la nueva versión del sistema operativo.

1.2.1. Desarrollo de aplicaciones nativas

Como hemos mencionado antes, las aplicaciones nativas son aquellas que se implementan en el lenguaje específico que facilita la plataforma/fabricante para el desarrollo. Por ejemplo, para Android tenemos Java y Kotlin sobre Android Studio. En la plataforma de Apple para iPhone nos encontramos con Objective-C en las aplicaciones antiguas y Swift en las nuevas, sobre el entorno Xcode.

1.2.2. Desarrollo de aplicaciones híbridas

Las aplicaciones híbridas son aquellas en las que un mismo código pueda compilarse para dos o más plataformas diferentes, ahorrando recursos de desarrollo. Podemos mencionar algunas de ellas:

Plataforma	Lenguaje	Compañía	Plataformas de destino
Xamarin	C#	Microsoft	Android, iOS y Windows.
Unity	C#	Unity Technologies	La mayoría de plataformas de escritorio, móviles y de juegos.
Fluter	Dart	Google	Android, iOS, Windows, Mac, Linux, Web.
Ionic	JavaScript	Open source	Android, iOS, Windows, Web.
React Native	JavaScript	Facebook	Android, iOS, UWP, Web.
PWA	JavaScript	Google	Android, iOS.

1.3. Entornos de trabajo integrado

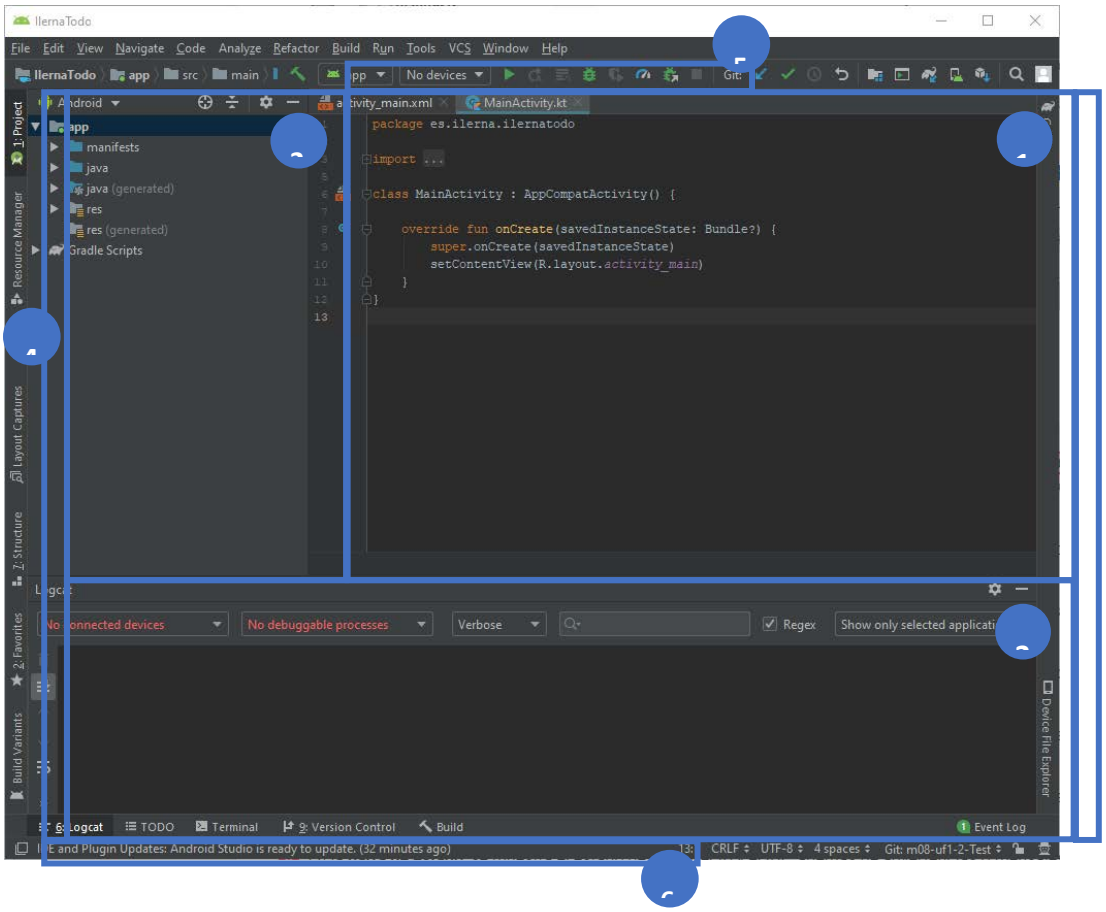
En este apartado presentamos qué es un entorno de trabajo integrado o, como más comúnmente se suele mencionar con sus siglas del inglés, un **IDE** *(integrated development environment)*.

Un **IDE** no es más que una aplicación informática que agrupa un conjunto de herramientas para facilitar el desarrollo de aplicaciones, acompañado de una interfaz gráfica que facilite su uso con la finalidad de aumentar la productividad.

El IDE oficial para el desarrollo de aplicaciones en Android es Android Studio, basado en la herramienta de desarrollo IntelliJ que se ha modificado incluyendo aquellas funciones específicas que mejoran la productividad en el desarrollo de aplicaciones para Android. Algunas de estas funciones son:

- Integración con el emulador, que veremos en unidades posteriores.
- *Instant run*, que permite ver cambios de código sin reiniciar la aplicación.
- Desarrollo unificado para todos los dispositivos Android (móvil, TV, *smartwatch*, Android Auto).
- Editor de código inteligente, con autocompletado.
- Control de versiones del código integrado.

Todas las herramientas que ofrece Android Studio se agrupan en una interfaz gráfica, las más importantes las podemos identificar en la siguiente imagen:



1. Editor de código: en esta sección podremos escribir y modificar el código de los diferentes archivos que componen la aplicación. Esta sección puede cambiar dependiendo del tipo de archivo que editemos.

2 y 3. Ventanas de herramientas: cambian en función de la herramienta que hemos seleccionado, en ellas podemos encontrar la administración del proyecto.

4. Barra de ventana de herramientas: permite expandir o contraer las ventanas de herramientas que podemos ver en las secciones 2 y 3.

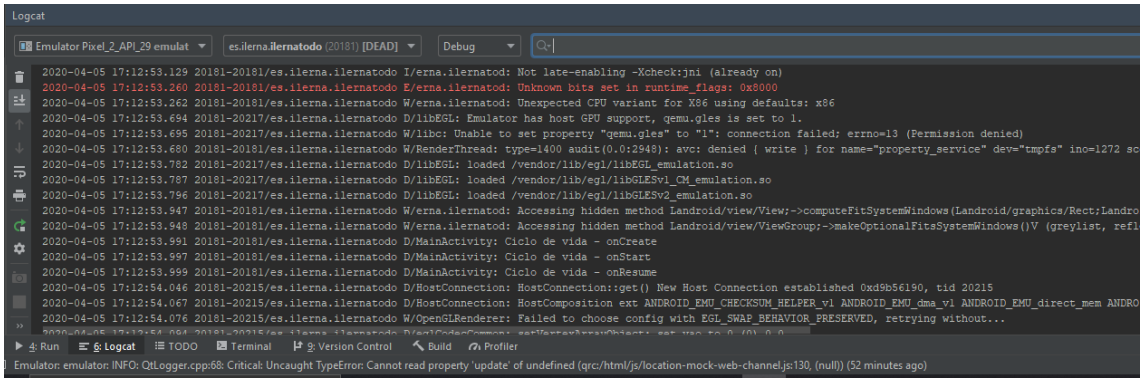
5. Herramientas de compilación y ejecución de la aplicación Android: en esta barra podemos ver en qué dispositivo ejecutamos la aplicación.

6. Barra de estado: nos mostrará los mensajes de información y advertencias que se muestran en los diferentes procesos que ejecuta el IDE.

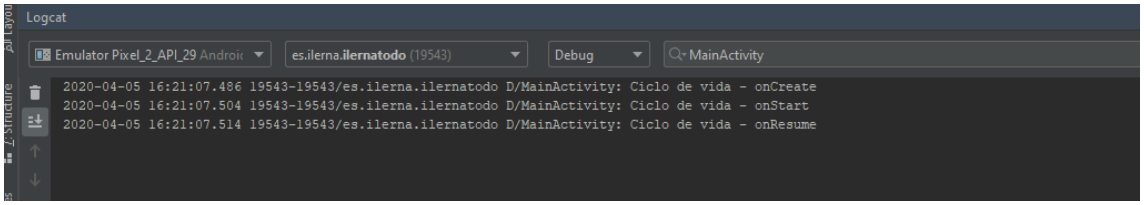
1.3.1. Registros de la aplicación, Logcat

El IDE de Android Studio tiene multitud de herramientas que descubriremos a lo largo de las siguientes lecciones, según las necesitemos, aunque en particular la herramienta de registro de aplicación o Logcat tienen una utilidad fundamental durante el desarrollo de las aplicaciones, para solucionar y averiguar posibles errores.

La ventana de Logcat que podemos encontrar en la sección inferior muestra los mensajes del sistema a la vez que los mensajes propios de la aplicación que podemos añadir con la clase *Log*.



Durante la ejecución de la aplicación se pueden producir una gran cantidad de mensajes, de modo que podemos aplicar filtros para localizar los mensajes que son de nuestro interés: en la barra superior de esta misma ventana podemos filtrar por el paquete que identifica a la aplicación, por nivel (*debug*, info, error, *warning*) e incluso aquellos mensajes que contengan una palabra concreta.



1.3.2. Poniendo en práctica: instalación de entorno de desarrollo

Ya conocemos qué es un IDE y que es nuestra herramienta principal para desarrollar aplicaciones, de modo que os invito a instalar Android Studio para poder seguir los ejemplos que mostraremos a lo largo de los temas.

Descargar Android Studio desde la web oficial en el siguiente enlace:

https://developer.android.com/studio

Su instalación es muy sencilla: desde en enlace de descarga seleccionamos el tipo de descarga que corresponda a nuestro sistema, que debe cumplir con los requisitos mínimos para poder ejecutarlo:

Sistema operativo	Memoria RAM	Espacio en disco
Windows 7/8/10 64 bits	4 GB mín. / 8 GB recom.	2 a 4 GB libres
Mac OSX 10.10 o superior	4 GB mín. / 8 GB recom.	2 a 4 GB libres
Linux basado en Debian	4 GB mín. / 8 GB recom.	2 a 4 GB libres

Es importante contar con bastante memoria RAM y espacio en disco para poder trabajar de una forma fluida, teniendo en cuenta que, al utilizar dispositivos virtuales, consumirá también bastante memoria y disco.

1.4. Módulos para el desarrollo de aplicaciones móviles

Los proyectos de Android se dividen en módulos que contienen los ficheros de código, de recursos y de compilación necesarios para la construcción de la aplicación, de modo que un proyecto Android al menos constará de un módulo.

Los módulos permiten dividir el proyecto en funcionalidades, permitiendo que podamos crear bibliotecas de código que se comparten en otros módulos del proyecto o incluso de otros proyectos.

Como ejemplo supongamos que desarrollamos una aplicación que es compatible con móviles y con *smartwatch:* la base de funcionalidades de la aplicación será la misma para ambas aplicaciones, pero la interfaz de usuario cambiaría. Podemos crear tres módulos: uno contendrá las funciones compartidas de ambos dispositivos y crearemos dos módulos más para cada dispositivo, de modo que compartimos código en ambas aplicaciones, facilitando el desarrollo, test y ampliaciones futuras.

Un proyecto puede tener varios módulos, nos permiten dividir el código de la aplicación y reutilizarlo.

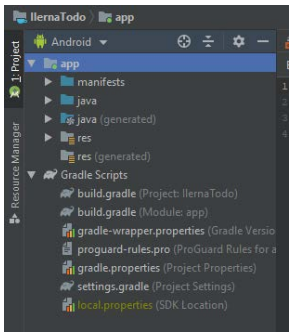
En Android Studio podemos crear diferentes tipos de módulos siguiendo la ruta *File > New > New Module*, de los cuales algunos son:

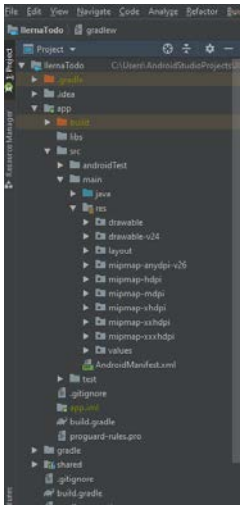
- **Módulos de aplicación**: por defecto, al crear un nuevo proyecto, automáticamente se nos creará un módulo de aplicación. Según el tipo de dispositivo pueden ser: Modulo Wear, Android TV, Android Things, Phone o Tablet. Este tipo de módulo contiene los ficheros y código específico para el tipo de dispositivo. Dentro del proyecto podemos tener varios módulos de aplicación, dependiendo de si estamos desarrollando una ampliación compatible con varios tipos de dispositivos.
- **Módulo de biblioteca *(library)***: creamos módulos de este tipo cuando el código que contienen se comparte con otros módulos, de modo que podamos reutilizar el código. Este tipo de módulos deben ir acompañados o utilizarse como dependencia en módulos de tipo aplicación, ya que por sí solos no son ejecutables en los dispositivos.

1.4.1. Android Studio: tipos de vista

Para visualizar los módulos y los ficheros que los componen, Android Studio nos ofrece varios tipos de vistas que podemos cambiar con el menú desplegable de la parte superior de la ventana de proyecto.

Vista de Android: muestra una vista simplificada de la jerarquía de los ficheros del proyecto, con los elementos más relevantes en el desarrollo de una aplicación Android.





Vista de proyecto: esta ofrece un mayor detalle de la jerarquía completa de las carpetas y archivos que compone el proyecto. Esta vista detallada no es la más cómoda para trabajar durante el desarrollo, pero es útil conocerla cuando necesitamos editar un fichero concreto que no podemos localizar en la vista anterior.

1.4.2. Archivos y carpetas de los módulos

Como hemos comentado anteriormente, los módulos contienen los archivos de código y los recursos que componen la aplicación. En función del tipo de módulo, podemos encontrar diferentes carpetas que organizan el contenido. Los más relevantes son:

- **Java**: contiene el código de la aplicación tanto en lenguaje Java como en Kotlin. Dentro del código podemos encontrar las clases de controladores, funciones e incluso las pruebas unitarias, todo ello organizado en paquetes Java.

- **Res**: dentro de este módulo encontramos aquellos archivos referentes al diseño de las pantallas, traducciones de textos, estilos, gráficos e imágenes que componen la aplicación. Este módulo se divide en carpetas para organizar el contenido y que sea fácilmente localizable. Según el tipo de recurso, encontramos las siguientes:

  - *drawable:* almacena recursos del tipo imágenes o gráficos en diferentes formatos (PNG, JPG, BMP, etcétera) o vectoriales en formato XML.

  - *layout:* encontramos los ficheros que definen el diseño de las pantallas de la aplicación, en formato XML.

  - *values:* en esta carpeta tenemos varios ficheros que almacenan los estilos y códigos de colores usados en la aplicación, así como los textos que se utilizan en la aplicación para su posterior traducción.

  - *minimap:* contiene los archivos gráficos del icono de la aplicación, podemos ver que existen varios en función de la resolución de pantalla del dispositivo donde se ejecute la aplicación.

- **Manifests**: encontramos el archivo AndroidManifest, en este archivo se definen la configuración y otros aspectos de la aplicación que estamos desarrollando, así como los permisos que utilizará el dispositivo.

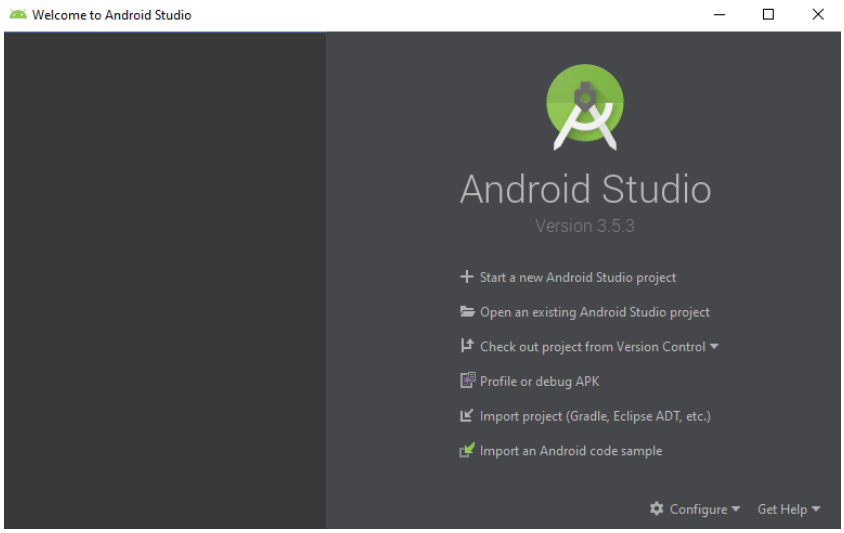
- **Gradle Scripts**: esta sección agrupa los ficheros necesarios para la compilación de la aplicación, en la que podemos encontrar al menos dos: uno para la compilación del proyecto y otro para cada módulo que contenga el proyecto.

Estos ficheros los veremos con mayor detalle en los siguientes capítulos, donde hablaremos de la fase de compilación.

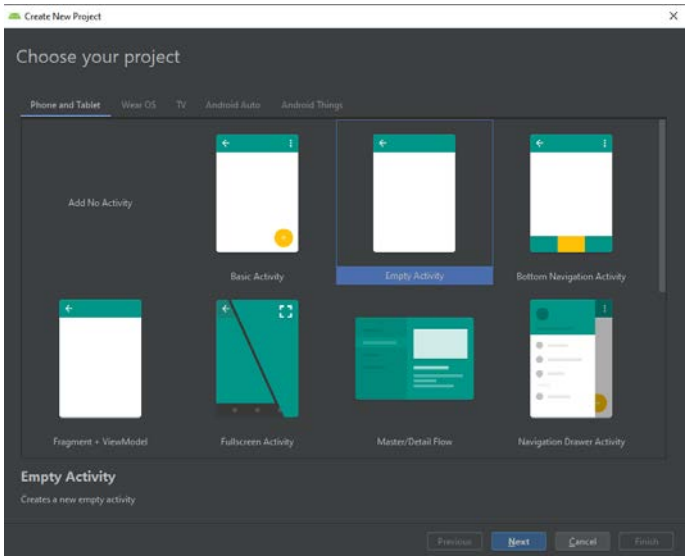
1.4.3. Poniendo en práctica: creando un proyecto

Para comenzar el desarrollo de cualquier aplicación, deberemos crear un proyecto para familiarizarnos con el entorno y la estructura. Comenzamos por crear un proyecto llamado IlernaTodo al que, a lo largo de los siguientes capítulos, le añadiremos funcionalidad hasta crear una aplicación real y funcional.

Al arrancar Android Studio, en la página de bienvenida seleccionamos la opción *Start a new Android Studio Project.*

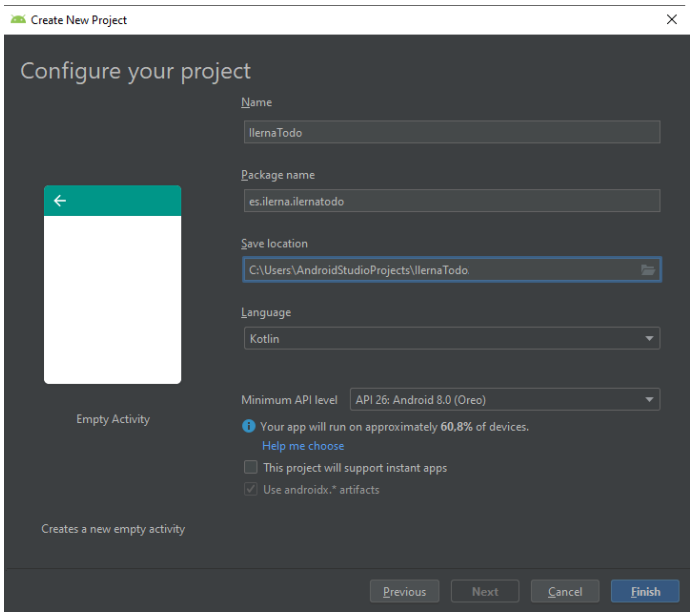


En el siguiente paso nos muestra una serie de plantillas predefinidas, según el tipo de aplicación que vamos a desarrollar. Para este proyecto seleccionaremos *Empty Activity*, esto creará un proyecto con una única pantalla en blanco.

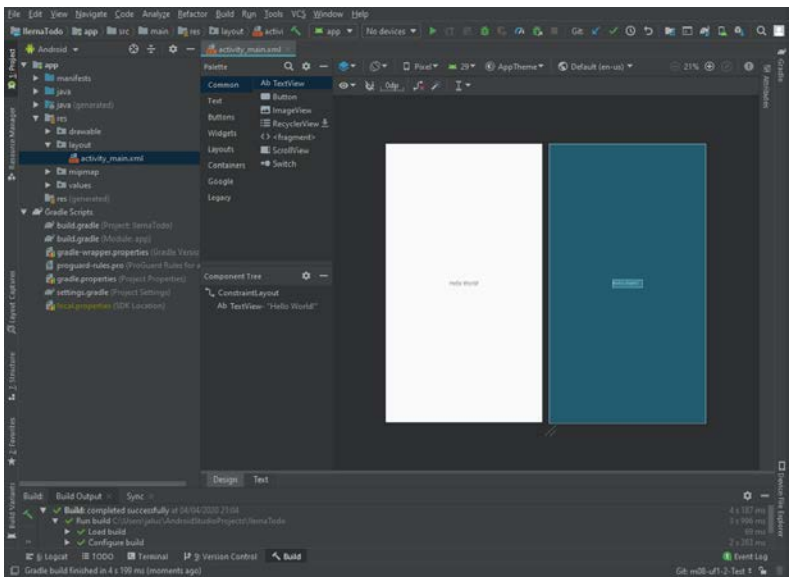


En este paso indicaremos el nombre de la aplicación y el *package*, así como el directorio donde se guardarán los archivos. En este punto también seleccionaremos el lenguaje que utilizaremos para el desarrollo: en este caso, será Kotlin.

Por último, indicaremos el *API Level* mínimo con que será compatible la aplicación. Veremos este concepto en los siguientes temas con más detalle y la importancia de su selección.



Tras un tiempo de preparación y ejecución de varias tareas que realiza Android Studio, tendremos el proyecto preparado para comenzar a desarrollar.



Llegados a este punto, os invitamos a navegar por la estructura del proyecto que ha generado Android Studio, localizar los ficheros más relevantes y dónde se guarda el código de la aplicación y el diseño de las pantallas.

Una vez familiarizados con esta estructura, para mantener el código organizado debéis crear un módulo de tipo biblioteca y responder a las siguientes preguntas:

¿Qué diferencias hay entre la estructura del módulo aplicación y del módulo que hemos creado?

¿En qué carpeta y módulo debemos colocar el código que queremos compartir con otros módulos?

¿Cuántos archivos build.gradle tenemos después de crear el módulo y por qué?

1.5. Emuladores

En el ciclo de desarrollo de cualquier tipo de aplicación necesitaremos probar el funcionamiento y corregir los posibles errores que surjan. Cuando desarrollamos aplicaciones de escritorio o web, estas pruebas las podemos realizar en nuestro propio ordenador.

En el ámbito de los dispositivos móviles es algo más complicado al existir multitud de dispositivos con diferentes, configuraciones, resoluciones de pantallas, sensores, etcétera. A esta variedad de dispositivos le añadimos el factor económico, ya que probar en un dispositivo real implica el gasto de compra del dispositivo y, en el mejor de los casos de que dispongamos de presupuesto, solo podremos probar en un número finito de estos.

El emulador incluido en Android Studio simula multitud de dispositivos junto a los sensores que incorporan, nos permite probar la aplicación sin necesidad de invertir en dispositivos físicos, de un modo rápido y ágil. No solo podemos probar en diferentes dispositivos, sino también en diferentes versiones de sistemas operativo Android, de modo que podemos comprobar si nuestra aplicación funciona correctamente en versiones más antiguas o modernas del sistema.

Un ejemplo típico donde el uso de emuladores es fundamental son aplicaciones con el uso de GPS. Para probar este tipo de aplicación, necesitaremos simular ubicaciones concretas o rutas: por supuesto, no podemos salir a cada momento a la calle para comprobar que la aplicación funciona correctamente, el emulador nos permite simular el funcionamiento del GPS incluyendo rutas o posiciones en cualquier lugar del mundo.

El emulador puede simular diferentes dispositivos, versiones y sensores como el GPS, acelerómetro, brújula, nivel de batería, rotación de la pantalla, etcétera.

1.6. Integración en el entorno de desarrollo

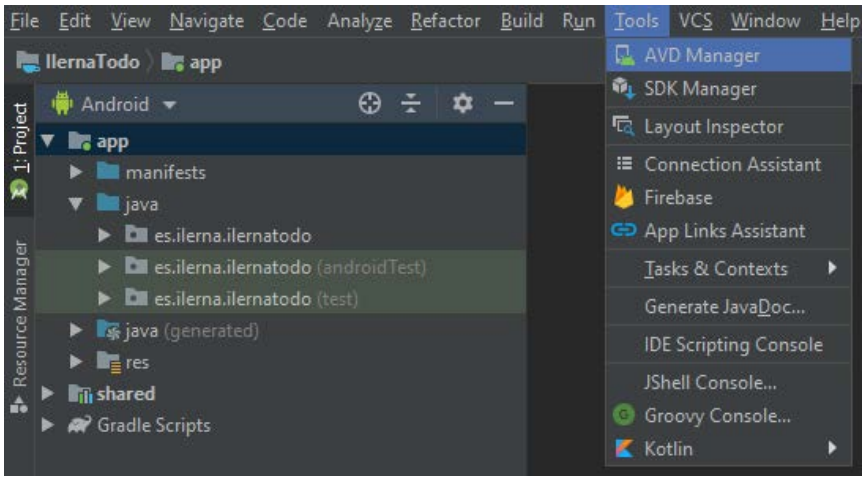
Como veremos en este apartado, el emulador de Android está perfectamente integrado en el entorno de desarrollo de Android Studio, aunque en los primeros pasos del desarrollo de *app* no fue así. Antes de la llegada de Android Studio, el IDE de desarrollo por excelencia estaba basando en Eclipse, y era necesario una configuración como *plugin* para poder integrar el emulador dentro del flujo de desarrollo de una aplicación.

Con la aparición de Android Studio, el emulador se integró perfectamente dentro del IDE, permitiendo ejecutar las aplicaciones que tenemos en desarrollo y depurarlas paso a paso. Para ver esta integración, vamos a ponerlo en práctica.

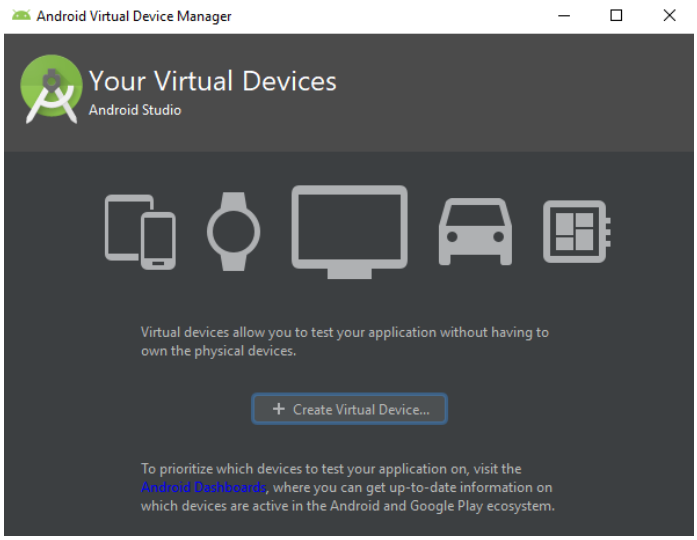
1.6.1. Poniendo en práctica: crear un dispositivo virtual con el emulador de Android Studio

A continuación, crearemos un dispositivo virtual que usaremos a lo largo del libro para probar los ejemplos y el desarrollo de la aplicación IlernaTodo.

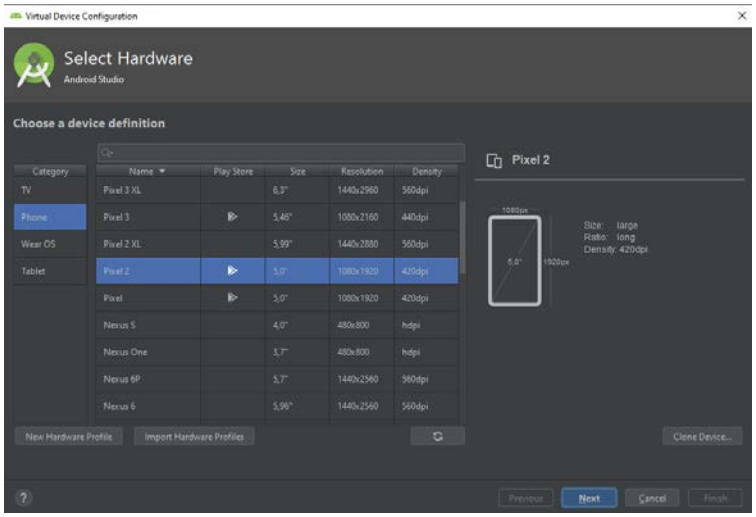
Aunque no es necesario tener abierto el proyecto, si hemos seguido los puntos anteriores podemos abrir el *AVD Manager*, que es la herramienta para administrar los múltiples dispositivos virtuales que tenemos configurados. La encontramos en el menú *Tools > AVD Manager*.



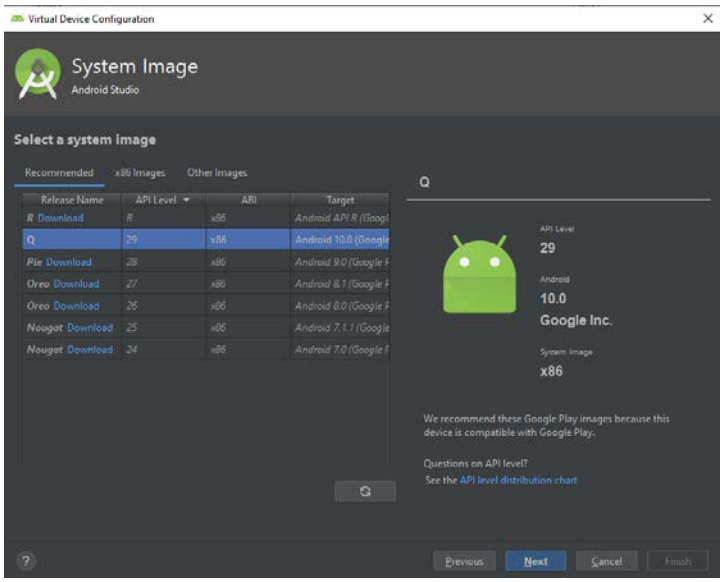
La primera vez no tendremos ningún dispositivo virtual y el asistente nos propone crear uno nuevo.



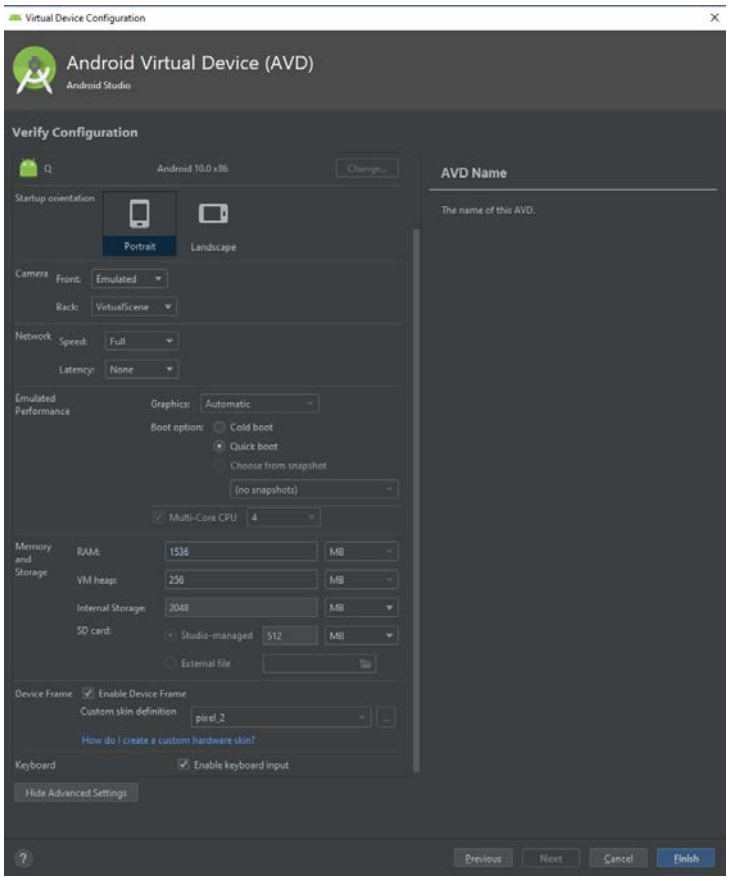
En el siguiente paso nos aparece una lista de posibles dispositivos que podemos simular: televisores, móviles, *smartwatch* o tabletas. Para este caso seleccionamos un *Píxel 2*, es un dispositivo de pantalla media o habitual a día de hoy y un buen punto de partida para probar las aplicaciones.



Como comentamos en las explicaciones previas, podemos crear dispositivos virtuales con diferentes sistemas operativos. En este paso seleccionamos qué versión de Android se instalará en el dispositivo. La selección de este dependerá de las pruebas que queremos hacer y del API mínimo que seleccionamos al crear el proyecto, lo que veremos con más detalle en lecciones posteriores. De momento, en este paso seleccionaremos *API Level 29*, que corresponde a Android 10.

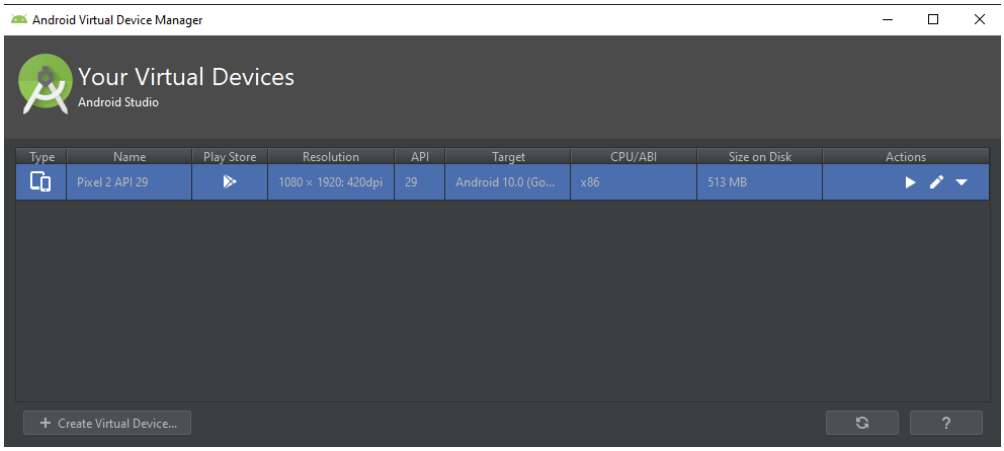


El siguiente paso nos muestra la configuración del dispositivo. Clicando sobre el botón *Show Advance Settings* vemos en detalle la cantidad de memoria RAM destinada al dispositivo virtual.



Al finalizar, el asistente nos muestra una lista de todos los dispositivos virtuales que tenemos configurados, y podemos encender o apagar los que consideremos para probar la aplicación en los diferentes dispositivos. En este punto ya podemos ver que disponer

de dispositivos virtuales nos facilita poder probar la aplicación con varias configuraciones.



Para encender el dispositivo virtual que queremos, tendremos que clicar sobre el botón con el icono de *play*. Tras unos minutos, tendremos un dispositivo virtual arrancado que funciona prácticamente igual que un dispositivo real.

1.7. Configuraciones. Tipos y características. Dispositivos soportados

En la actualidad, los dispositivos móviles se han extendido de forma exponencial, dando lugar a variantes de estos que se incorporan en nuestro día a día. Podemos encontrarlos con diferentes configuraciones, tipos y características según el uso al que estén destinados. Podemos encontrar los siguientes tipos:

- ***Smartphones* y móviles**: este tipo de dispositivo es el más común al que estamos habituados. Entre las características que ofrecen los diferentes fabricantes, podemos encontrar:
  - Tamaño de pantalla.
  - Tipo y número de cámaras.
  - Sensores (GPS, brújula, acelerómetros).
  - Memoria, CPU y batería.
- ***Smartwatch**:* los relojes inteligentes han visto cómo sus funcionalidades han aumentado gracias a la miniaturización y el desarrollo de procesadores más potentes y de bajo consumo, lo que ha permitido crear sistemas con características como Bluetooth, Wifi, acelerómetros, sensores de ritmo de cardiaco, GPS o brújulas. Este tipo de dispositivos se ha abierto camino rápidamente en el mundo del deporte y la salud, gracias a los sensores que incorporan es posible crear aplicaciones que monitorizan nuestra actividad física, como la calidad del sueño.

  Google ha creado un sistema operativo específico para este tipo de dispositivos llamado **WearOS**: al estar basado en Android, nos permite crear aplicaciones desde el mismo ecosistema de herramientas.
- **Android Auto**: la tecnología móvil no tardaría en llegar a nuestros coches, de modo que nos permite integrar sistemas de navegación y de entretenimiento. Google ha desarrollado un sistema adaptado a las necesidades particulares de los automóviles para evitar las distracciones, de modo que la interfaz de las aplicaciones es minimalista e integrada con el reconocimiento de voz.

  En este aspecto, los dispositivos soportados están limitados a la integración por parte del fabricante en los modelos, aunque hoy en día es muy común que estas estén en los modelos de serie.
- ***Smart TV* y Android TV**: aunque no es un dispositivo móvil, se usa el mismo concepto y sistema para los televisores, consiguiendo que se integren con el resto de los dispositivos, lo que permite recibir contenido, tanto vídeo como música, desde dispositivos móviles a la vez que se pueden ejecutar aplicaciones en nuestro televisor.

El ecosistema de dispositivos que utilizan tecnologías móviles ha crecido en los últimos años, llegando a más dispositivos. Compartir sistema y herramientas de desarrollo hace fácil que los desarrolladores adopten estos nuevos dispositivos, creando aplicaciones para los usuarios y proliferando su crecimiento.

1.8. Perfiles. Características. Arquitectura y requerimientos. Dispositivos soportados

El tipo de aplicación que estemos desarrollando y el público al que vaya dirigida marcará qué tipo de dispositivo utilizaremos, según las características que nos ofrecen, de modo que podemos establecer unos perfiles de aplicación, sus características y dispositivos soportados.

Aplicaciones deportivas, salud y/o notificaciones

En el caso de que desarrollemos una aplicación para monitorizar la actividad y/o salud de las personas, buscaremos ciertas características:

- Dispositivo ligero.
- Que podamos usar en el exterior mientras realizamos una actividad.
- Acceso rápido y fácil.
- Sensores de tipo ritmo cardiaco, GPS, brújula.

Para este tipo de aplicaciones, los dispositivos idóneos son los de tipo *smartwatch*, que disponen de las características indicadas y tienen ventajas sobre los otros dispositivos para cumplir con este tipo de aplicaciones.

Aplicaciones de visualización de contenidos y juegos

Para las aplicaciones de visualización de contenidos y videojuegos, podemos ofrecer como dispositivo *smart TV* con Android TV. Podemos considerar que son una plataforma alternativa, ya que este tipo de aplicaciones también pueden desarrollarse para móviles, pero cierto contenido puede ser más agradable de disfrutar en una gran pantalla.

Aplicaciones de localización, rutas, tráfico o infoentretenimiento

Las aplicaciones destinadas para automoción son muy específicas y limitadas por lo que implica el uso de sistemas que puedan distraer al conductor, de modo que podemos definir un perfil de aplicaciones destinadas a su uso en automóvil con una interfaz reducida e integrada con el reconocimiento de voz.

Al igual que con las *smart TV*, si desarrollamos una aplicación en el ámbito de rutas, de información del tráfico o de contenidos de música, los dispositivos Android Auto pueden marcar la diferencia gracias a su adaptación e integración con el vehículo.

1.8.1. Jerarquía de clases de perfil

Android Studio pone a nuestra disposición muchas utilidades que nos facilitan el desarrollo y la depuración de nuestras aplicaciones. Es necesario analizar no solo el código, sino también las interfaces gráficas. Imagina un proyecto con *layouts* complejos en el que las vistas se solapan unas sobre otras, o cuya posición es relativa a otra. En una aplicación así, es difícil saber por qué una vista ha quedado fuera de la pantalla, por qué está mal situada o por qué la velocidad de presentación es baja y crea cuellos de botella en la interfaz.

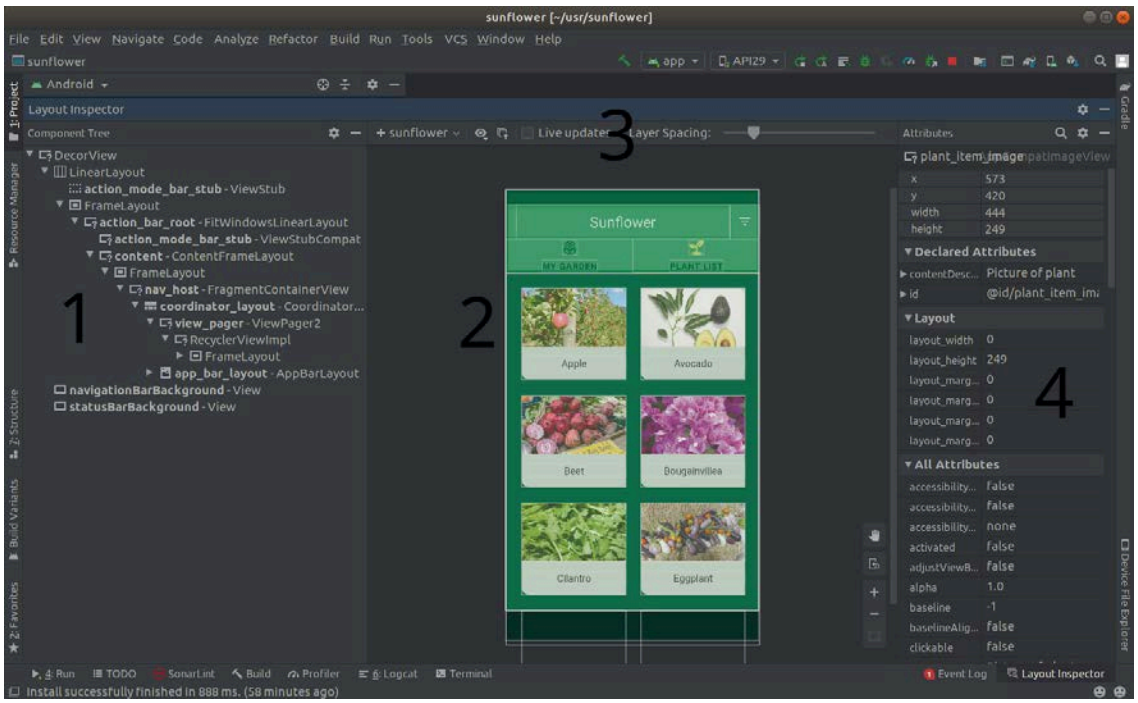
En versiones anteriores a la 3.1 de Android Studio, disponíamos del ***Hierarchy Viewer***. Con esta herramienta podíamos averiguar qué parte de la interfaz causaba el retardo de la presentación. Analizaba la aplicación conectada al emulador o a un dispositivo real, mostrando algunas medidas de tiempo y algunas valoraciones sobre el rendimiento, de este modo podíamos descubrir las partes del *layout* más lentas, sus relaciones con las otras, etcétera. Sin embargo, esta herramienta dejó de estar disponible en Android Studio desde la versión 3.1, así que es poco probable que podamos utilizarla en proyectos recientes.

Por fortuna, a cambio tenemos el ***Layout Inspector***, o inspector de diseños. Esta utilidad nos analizar la interfaz gráfica por completo. Podemos comparar nuestro *layout* con el prototipo de diseño o *mockup*. Así seremos capaces de comprobar que la alineación de las vistas sea la que deseamos, o la que nos piden los diseñadores el departamento de UI & UX. El *Layout Inspector* es de extremada importancia en aplicaciones cuya interfaz de usuario se crea durante la ejecución, pues se tiene menor control sobre el resultado que cuando diseñamos la interfaz estáticamente con código XML. El *Layout Inspector* nos permite comprobar qué aspecto tendrá nuestra interfaz gráfica en cualquier momento del desarrollo. Además, disponemos de otra herramienta llamada ***Layout Validation*** con la que podemos observar cómo nuestro *layout* se adapta a terminales móviles con diferentes tamaños de pantalla.

Veamos a continuación cómo podemos utilizar la herramienta *Layout Inspector*. Hagamos la prueba en una aplicación completa con una interfaz compleja, como, por ejemplo, Sunflower, de Google, que podemos encontrar en GitHub:

> Repositorio de la aplicación de ejemplo Sunflower en GitHub:
>
> https://github.com/android/sunflower

Descargamos el proyecto o lo clonamos desde el repositorio, y lo abrimos en Android Studio. Compilamos y lanzamos la aplicación a un emulador o dispositivo. Navega por la aplicación para tener una idea de cómo funciona. Vuelve a Android Studio, y haz clic en la opción de menú *Tools > Layout Inspector*. Puede que aparezca el diálogo *Choose Process*; si es así, selecciona el proceso de la *app* que desees inspeccionar y haz clic en *OK*. Es posible que no se muestre el diálogo si Sunflower es la única aplicación depurable arrancada en ese momento en el emulador o terminal. Verás una pantalla como la que mostramos a continuación:

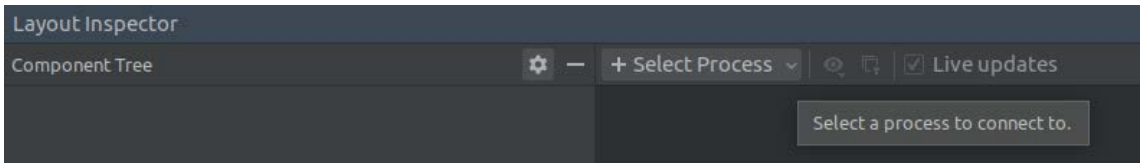


Podemos observar los diferentes paneles del *Layout Inspector:*

1. **Component Tree o árbol de vistas**: muestra la lista jerárquica de vistas y componentes gráficos que componen el *layout*.
2. **Layout Display o visor de pantalla**: renderiza el *layout* tal como se ve en la pantalla del terminal o emulador, señalando además los límites de cada componente. Es una especie de captura de pantalla de lo que muestra el dispositivo en un momento dado.

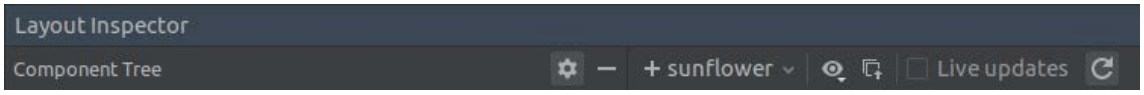
3. **Layout Inspector toolbar o barra de herramientas del inspector**: presenta algunas herramientas disponibles para analizar el *layout*.
4. **Attributes o tabla de propiedades**: muestra las propiedades gráficas del componente seleccionado actualmente.

En cualquier momento puedes cambiar la aplicación que estás depurando, mediante el selector *Select Process*. Recuerda que el proceso debe ser depurable, es decir, tener activada la depuración. Muy posiblemente, una aplicación descargada desde el *store* no te va a permitir que la depures.

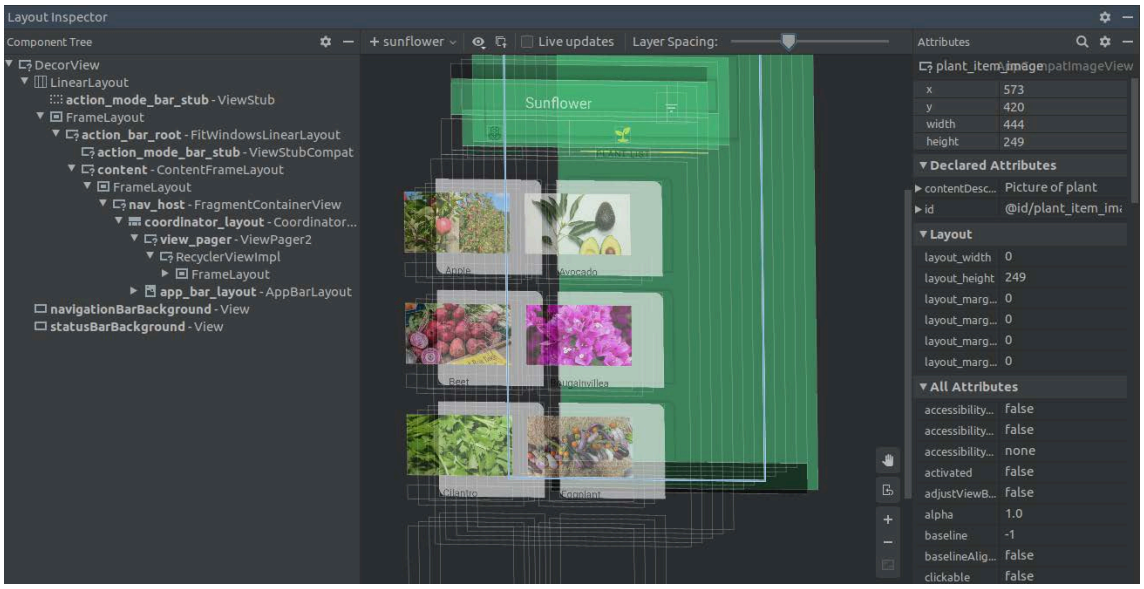


Si la aplicación se está ejecutando en un terminal o emulador con *API Level 29* o superior, puedes hacer clic en *Live Updates*, de modo que cualquier cambio de vista que realices en la aplicación será mostrado en la vista de pantalla, lo que te facilitará la depuración:

Si la aplicación rueda sobre un terminal más antiguo, cuando cambies la vista de la *app* podrás actualizar la vista de pantalla del *Inspector* mediante el icono de recarga, que es una flecha circular:

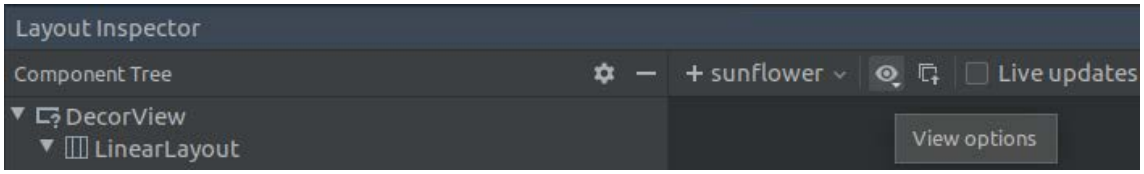


Puedes seleccionar cualquier componente gráfico haciendo clic sobre él, tanto en el árbol de vistas como en el visor de pantalla. En ese momento se mostrarán los atributos de la vista en la tabla de propiedades. Si quisieras seleccionar una vista que esté bajo otro componente, podrías hacer clic sobre ella en el árbol de vistas, o rotando el *layout* en el visor de pantalla y haciendo luego clic sobre la vista. La rotación permite una visión 3D de la superposición de las vistas una encima de otra, lo que puede resultar muy útil para depurar y diseñar la interfaz gráfica:



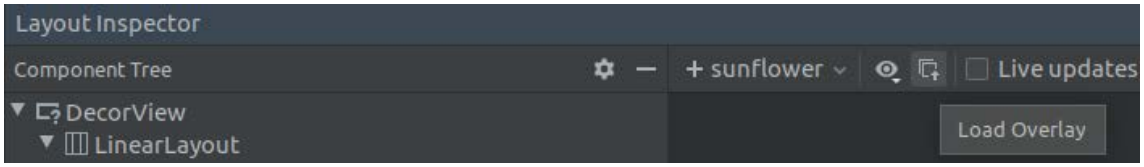
Si el *layout* es grande y complejo, quizá sea más sencillo centrarse solo en una parte del diseño cada vez. Para ello, puedes aislar un conjunto de vistas de la jerarquía, para que sean los únicos que veas en el árbol de vistas y en el visor de pantalla. Para aislar las vistas, el dispositivo tiene que estar aún conectado, para que pueda tomarse otra captura y análisis de pantalla para el visor. Para aislar una vista, pulsa con el botón derecho del ratón sobre la vista en el árbol de vista o en el visor y escoge *Show Only Subtree* (mostrar solo el subárbol). Para volver a ver todos los componentes del *layout*, haz clic con el botón derecho sobre la vista y escoge *Show All* (mostrar todo).

Quizá te confundan los márgenes que dibuja el visor de pantalla para delimitar las vistas. Si quieres ocultar esas líneas, haz clic en el icono de *View Options* con forma de ojo y haz clic sobre *Show Borders*, el *check* desaparecerá. La otra opción, *Show View Label*, te permite mostrar u ocultar las etiquetas de cada componente. Las etiquetas te facilitan comprender qué vista se corresponde con qué elemento en el árbol, pero si hay demasiadas y te molestan, puedes ocultarlas con esta opción.



Imagina que desde el departamento de producto te entregan un *mockup* o diseño gráfico de referencia, más o menos realista, de cómo quieren que luzca una de las pantallas de la aplicación que estás desarrollando. Tu trabajo consiste en programar la interfaz gráfica de Android a partir de ese diseño gráfico. En un punto en el que ya te has acercado bastante, puede que quieras comprobar cuánto se parece tu diseño al *mockup*, o en qué difiere. Para hacer dicha comparación, el *Layout Inspector* te permite cargar un *bitmap* que se muestre solapado sobre tu diseño. En la barra de herramientas pulsa la opción *Load Overlay*, que es un icono con cuadrados y una flecha hacia arriba.

La imagen solapada se adaptará automáticamente al *layout*. Para ajustar la transparencia de la imagen, utiliza la barra *Overlay Alpha*. Cuando hayas terminado, pulsa de nuevo el icono, que ahora se llama *Clear Overlay* y tiene forma de cuadrados y una equis roja.



Para más información, consulta la documentación oficial del *Layout Inspector:*

https://developer.android.com/studio/debug/layout-inspector

1.9. Modelo de estados de una aplicación para dispositivos móviles. Activo, pausa y destruido

A diferencia de otros sistemas operativos, en Android cada aplicación es un proceso separado, el usuario no decide cuándo se finaliza una aplicación, esta decisión reside en la gestión de la pila de aplicaciones del propio sistema Android, en función de las necesidades de memoria en cada momento.

Del mismo modo, cuando el usuario vuelve a solicitar la aplicación, el sistema se encarga de crear el proceso para mostrar la aplicación en pantalla.

Dependiendo del estado en el cual se encuentre el proceso, el sistema asigna una prioridad que determina la probabilidad de que la aplicación sea cerrada. Podemos verlo en la siguiente tabla.

Prioridad	Estado de proceso	Estado de la actividad
Bajo o nula	Primer Plano, tiene el foco	*Created* *Started* *Resumed*
Media	Segundo plano, perdió el foco	*Paused*
Alta	Segundo plano, no visible	*Stoped* *Destroyed*

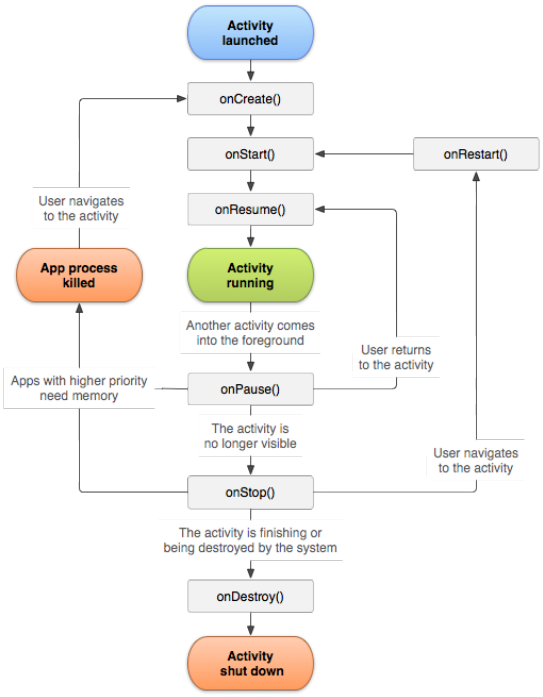
Vemos que el sistema no cerrará nunca la aplicación que tengan el foco, es decir, aquella aplicación que se está mostrando en la pantalla. Si la aplicación pierde el foco debido a que el usuario cambia de aplicación o está leyendo un mensaje o atendiendo una llamada, pasa a segundo plano y puede ser cerrada por el sistema si este necesita los recursos.

Es vital entender el funcionamiento del ciclo de vida de las aplicaciones para desarrollar aplicaciones robustas, ya que tenemos que manejar el ciclo de vida para determinar en qué momento recuperamos los datos de la aplicación.

En la tabla hemos incluido el estado de la actividad, puesto que el proceso de la aplicación está fuertemente vinculado al estado de la actividad, de modo que, para manejar el ciclo de vida desde nuestra aplicación, tenemos disponibles un conjunto de hasta seis métodos que el sistema invoca y podemos gestionar desde la actividad de la aplicación.

En el siguiente esquema podemos ver el ciclo vida representado de forma gráfica y el momento en el que se ejecuta cada método.

Es vital comprender el ciclo de vida de una aplicación, para crear aplicaciones robustas, durante el desarrollo tendremos debemos ejecutar el código en el momento indicado para que la aplicación funcione correctamente.



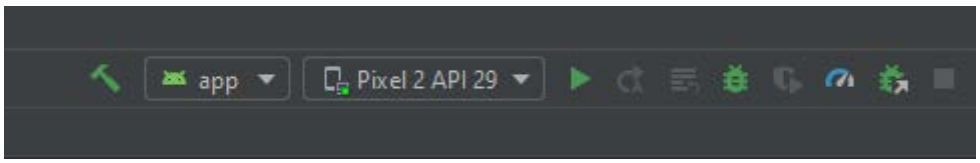
Vamos a explicar brevemente los diferentes métodos:

- ***onCreate()****:* este método siempre se debe implementar. Es el primero en ejecutarse al crear la actividad, en este método se ejecutará aquel código que solo debe ejecutarse una vez en el ciclo de la aplicación. Normalmente, solemos encontrar inicializaciones de la representación interfaz de usuario. En este estado la aplicación no es visible por el usuario.
- ***onStart()****:* en este estado la actividad se prepara para ser visible mientras entra en primer plano y recibe el foco.
- ***onResume()****:* en este estado la actividad ya es visible para el usuario, es el último método que se ejecuta mientras la actividad permanezca en primer plano.
- ***onPause()****:* este método se ejecuta ante cualquier indicio de que el usuario va a abandonar la aplicación o ha perdido el foco, aunque puede que finalmente no se abandone la aplicación. En este estado la aplicación suele ser parcialmente visible.
- ***onStop()****:* si la aplicación pasa a no estar visible para el usuario, se ejecuta este método indicando que la aplicación no está siendo utilizada. En este momento, la aplicación debe liberar los recursos que no son necesarios.
- ***onDestroy()****:* este método se llama antes de finalizar la actividad, ya sea porque el usuario descartó por completo la aplicación o porque el sistema cerró la aplicación para liberar recursos. Este método también se llama al rotar la pantalla.

1.9.1. Poniendo en práctica: ejecuta la aplicación desde Android Studio

En el apartado anterior hemos hablado de que las aplicaciones Android se empaquetan en un archivo APK para ser instaladas o distribuidas en los dispositivos, pero durante el desarrollo de la aplicación necesitaremos probar continuamente la aplicación. Android Studio automatiza todos los pasos de compilación, empaquetado e instalación en el depósito virtual con un solo clic.

Abre el proyecto que hemos creado previamente, debes tener creado un dispositivo virtual y situarte en el icono *play* (flecha verde),



Automáticamente, Android Studio arrancará el dispositivo virtual, compilará la aplicación, la empaquetará y la instalará en el dispositivo.

1.9.2. Poniendo en práctica: comprendiendo el ciclo de vida de una aplicación

Para comprender mejor el ciclo de vida, vamos a ponerlo en práctica en el proyecto que hemos creado previamente. Vamos al archivo MainActivity.kt; si no puedes localizarlo, repasa el punto 1.3, en el que habla sobre la estructura del proyecto.

Una vez localizado, sobrescribe los métodos *onCreate()*, *onRestart()*, *onStart()*, *onResume()*, *onPause()*, *onStop()* y *onDestroy()*, en cada uno de los métodos utiliza la función *Log.d()* con un texto descriptivo del método. La clase *android.util.Log* dispone de diferentes funciones para escribir en un *log*, es decir, un archivo en el que aparecen mensajes relevantes sobre la ejecución de la aplicación. *Log.d()* sirve para mensajes de depuración, que tienen menos prioridad que los mensajes *Log.e()*, que son errores y se muestran en rojo en las salida de mensajes Logcat. Entonces, veamos cómo sobrescribimos la función *onCreate()* para mostrar un mensaje:

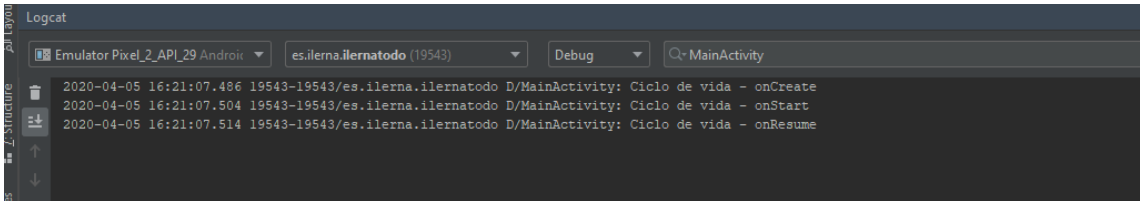
```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    Log.d(TAG, "Ciclo de vida - onCreate")
}
```

Por fortuna, mientras escribimos, el propio editor inteligente de Android Studio completará el código por nosotros. Observamos cómo utilizamos *override* para informar al compilador de que no estamos creando un método nuevo, sino sobrescribiendo el de la clase padre, que tiene igual nombre y parámetros. Al sobrescribir funciones de las clases del sistema Android, recuerda que en la mayoría de los casos será conveniente llamar primero al método original. Lo haremos al principio del método, mediante el objeto *super*, que es la palabra clave para identificar al objeto padre. Luego hemos añadido la llamada *Log*. La constante *TAG* sirve ver en el *log* desde qué clase imprimimos el mensaje. Podríamos definirla de la siguiente manera:

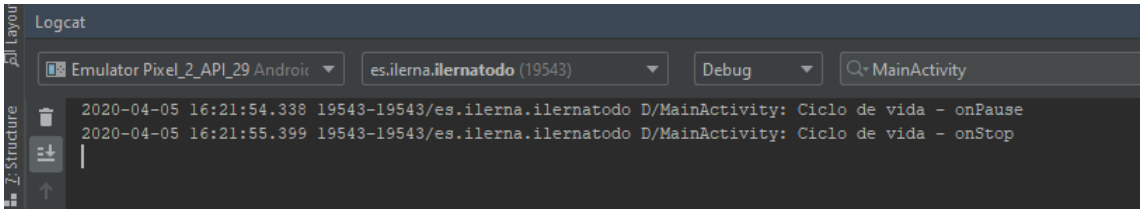
```
companion object {
    private const val TAG = "MainActivity"
}
```

Ejecuta la aplicación en el emulador y comprueba qué resultados obtienes en el Logcat. Luego cambia a otra aplicación y comprueba qué métodos del ciclo de vida se han ejecutado. Cierra la aplicación. ¿Qué métodos se ejecutan?

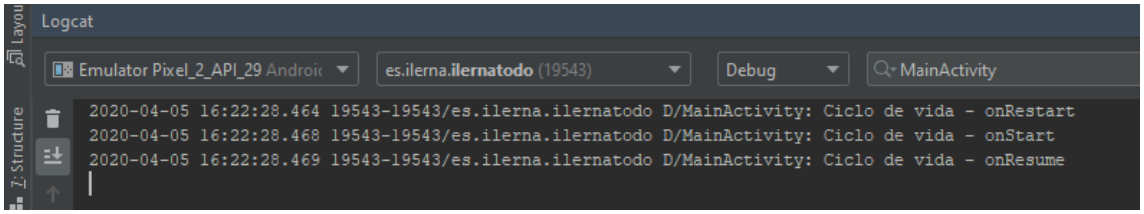
Al iniciar la aplicación, tenemos un resultado similar al siguiente: podemos ver que ha pasado por los métodos *onCreate()*, *onStart()*, *onResume()*.



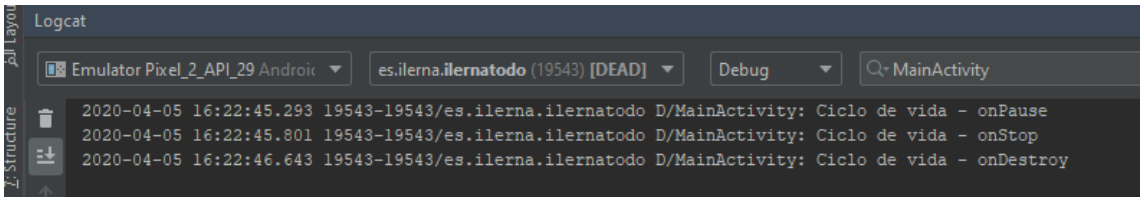
Al cambiar de aplicación, se ejecuta el método *onPause()* al perder el foco y un poco después se ejecuta el método *onStop()*.



Al volver a la aplicación, vemos que se ejecuta *onRestart()* en lugar de *onCreate()*, *onStart()* y *onResume()*.



Si cerramos la aplicación, se ejecutan *onPause()*, *onStop()* y *onDestroy()*.



¿Qué métodos crees que se ejecutarán al abrir la aplicación de nuevo?

¿Qué métodos se ejecutan si rotamos la pantalla del dispositivo con la aplicación abierta?

1.9.3. Ciclo de vida de una aplicación: descubrimiento, instalación, ejecución, actualización y borrado

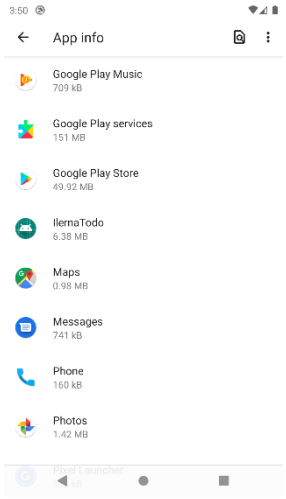
A diferencia de un sistema operativo de escritorio, el ciclo de vida de las aplicaciones Android es distinto. Pueden tener similitudes en la fase de instalación, actualización y borrado, pero la etapa de ejecución es distinta. Veamos cada una de las etapas en detalle.

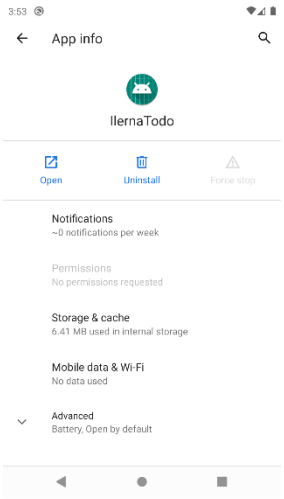
Las aplicaciones Android se empaquetan en un fichero APK, este archivo se genera en la fase de compilación y contiene todos los archivos necesarios para ejecutar la aplicación: el código compilado, los ficheros de recursos, imágenes, *manifest*, etcétera.

Este fichero APK se puede instalar en un dispositivo Android y nos puede servir para probar o realizar una demo, pero no es la forma más común de distribuir las aplicaciones a los usuarios. Para realizar la distribución, Google pone a nuestra disposición Google Play Store, donde podemos subir las aplicaciones, previo registro y pago del alta como desarrollador. Una vez subida, nuestra aplicación estará disponible al público para poder instalarla desde la Google Play Store.

Para la actualización de las aplicaciones es necesario disponer de un nuevo APK o actualizar la aplicación en el Google Play Store, de modo que este se encargara de distribuir las actualizaciones a los dispositivos donde se encuentre instalada la aplicación.

Si el usuario ya no quiere utilizar más una aplicación, puede ser eliminada del sistema de forma permanente junto con todos sus archivos y recursos desde las opciones de configuración del dispositivo, en esta sección disponemos de una lista de aplicaciones instaladas.





Como podemos ver en la captura de pantalla, estas pueden variar según la versión del sistema operativo. Tenemos el detalle de los recursos, notificaciones, permisos, espacio utilizado de la aplicación y una opción de desinstalar.

1.10. Modificación de aplicaciones existentes

En la mayor parte de situaciones en las que nos encontraremos, el desarrollo de las aplicaciones está comenzado y tendremos que realizar mantenimiento o añadir nuevas funcionalidades, por eso es importante entender la estructura de los proyectos, para localizar de forma rápida los archivos que tenemos que modificar o dónde debemos crear los nuevos archivos, ya sean clases o recursos.

Lo más habitual es que encontramos el código de la aplicación en un repositorio de control de versiones como puede ser Git en alguna de sus variantes (Gitlab, Github, etcétera) y que esté organizado en diferentes ramas. Una vez descargado, el primer paso que debemos hacer es descargarlo dentro de Android Studio y estudiar su estructura.

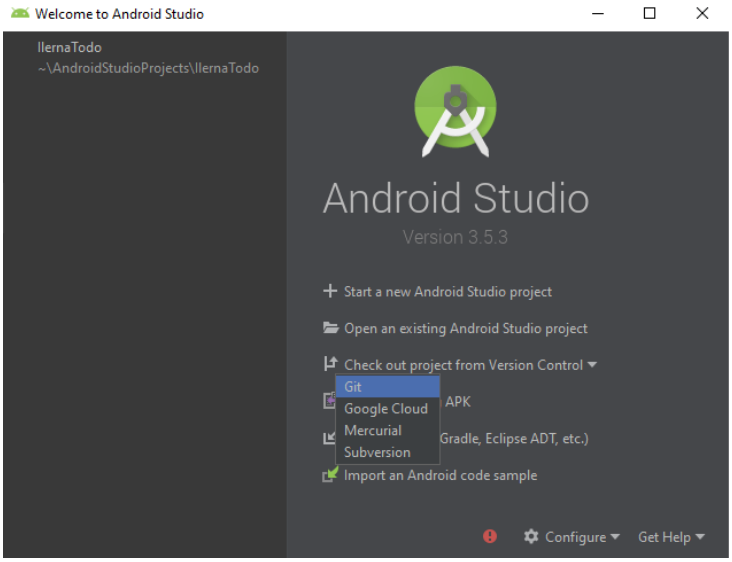
1.10.1. Poniendo en práctica: descarga el código y realiza una modificación

En este apartado comenzaremos a trabajar en el papel como desarrollador de la aplicación: vamos a simular que hemos llegado a una empresa de desarrollo donde tienen un proyecto para realizar una aplicación de gestión de tareas para un cliente.

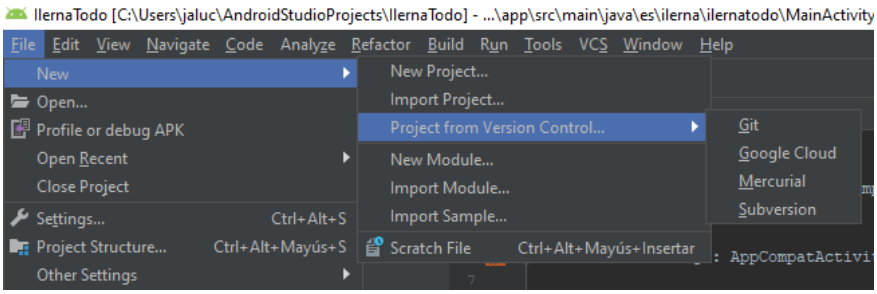
La empresa de desarrollo tiene el código en un repositorio Git, concretamente Gitlab, del cual nos han facilitado la dirección de acceso.

> URL del repositorio de Git:
>
> https://gitlab.com/ilerna/common/kotlin.git

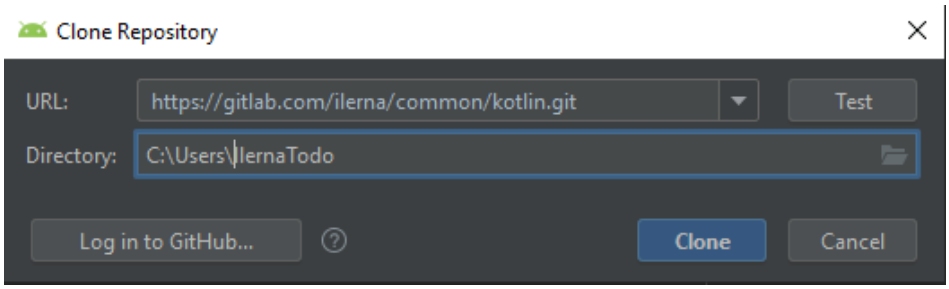
Nuestra primera tarea es descargar el código en Android Studio, podemos hacerlo desde la opción *Check out project from Version Control*.



O también desde el menú *File*, ambas opciones son igual de válidas.



En la siguiente ventana indica la URL del repositorio de Git y pulsa sobre el botón *Test* para comprobar que puedes acceder correctamente.



Tras clonar el repositorio y que Android Studio realice las tareas necesarias para importar el proyecto, muévete a la rama *01_Creando_Proyecto*. Lo podrás hacer desde el menú *VCS > Git > Branches*, donde aparecerá una ventana con las ramas disponibles en el repositorio.

Revisa el código que has descargado y prueba a ejecutarlo en el emulador. Si todo ha ido correcto, el responsable del proyecto nos informa de nuestra primera tarea, que es la siguiente:

#01 Caso de Uso:

Historia de usuario

"Como usuario, me gustaría ver un mensaje de bienvenida al entrar en la aplicación, del tipo 'Bienvenido, no tienes tareas a realizar'".

Tarea

Identifica el diseño de la pantalla de entrada a la aplicación y modifica el texto que aparece inicialmente por el que requiere el cliente.

Test

Comprueba que al ejecutar la aplicación se muestra correctamente el texto modificado.

1.11. Utilización de entornos de ejecución del administrador de aplicaciones

En las secciones anteriores hemos visto y hemos puesto en práctica cómo ejecutar una aplicación en un emulador que simula el funcionamiento de un dispositivo real.

Pero la prueba de fuego de una aplicación es cuando llega a los dispositivos reales, donde podemos encontrar situaciones, características del dispositivo o capas de *software* del fabricante que pueden influir en la aplicación.

Con Android Studio podemos ejecutar las aplicaciones que tenemos en desarrollo en dispositivos reales, de una manera fácil e integrada en el flujo de desarrollo.

1.11.1. Poniendo en práctica: ejecuta la aplicación en un dispositivo real

Partiendo de que tienes el proyecto anterior llamado IlernaTodo, ejecutaremos la aplicación en un dispositivo real. Necesitas tener un dispositivo Android con una versión Android Oreo (8.0) o superior y un cable USB para conectarlo al ordenador.

Paso 1. Conecta el móvil al ordenador. Si usas un sistema Windows, puede ser necesario instalar el *driver* si no lo reconoce automáticamente.

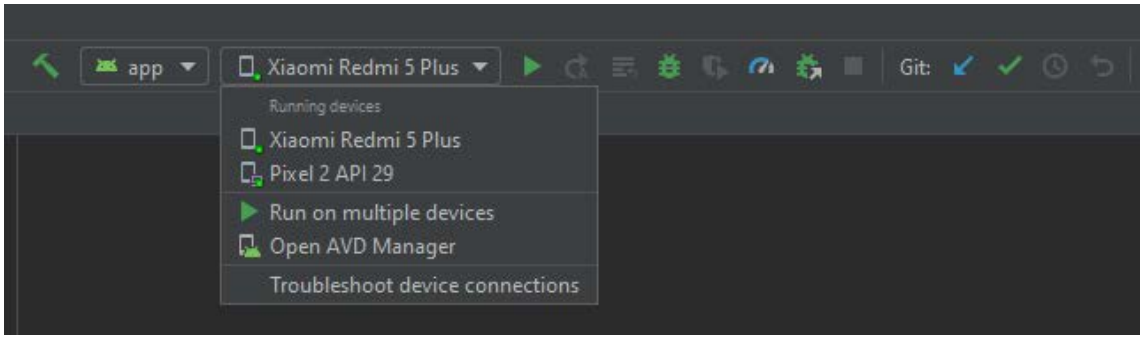
Visita la siguiente web si necesitas instalar el *driver:*

https://developer.android.com/studio/run/oem-usb?hl=es-419

Paso 2. Habilitar las opciones de depuración USB en la sección de *Opciones para desarrolladores*, para ello tienes que:

- Abrir la aplicación de *Configuración* de tu móvil.
- Selecciona la opción de *Sistema*.
- Desplázate hasta la parte inferior y selecciona *Acerca del teléfono*.
- Regresa a la pantalla anterior, desplázate hasta la parte inferior y presiona *Opciones para desarrolladores*.
- En la ventana *Opciones para desarrolladores*, desplázate hacia abajo para buscar y habilitar la depuración de USB.

Paso 3. En la barra de herramientas nos aparecerá nuestro dispositivo físico, en este ejemplo se trata de un Xiaomi Redmi 5 Plus. Una vez seleccionado, podemos pulsar sobre el botón *play* y Android Studio instalará la aplicación en el dispositivo físico.



Recapitulando

Llegado al final de este tema, hemos cubierto los siguientes puntos:

- Conceptos sobre dispositivos móviles, características y perfiles.
- Qué es un IDE y la instalación de Android Studio.
- Cómo funciona un emulador, configuración y ejecución.
- Estructura y creación de un proyecto Android.
- Modificar una aplicación desde un repositorio Git.
- Ejecutar la aplicación en un dispositivo real.
- Un primer caso de uso para construir la aplicación de IlernaTodo.

Con lo hemos visto en este capítulo de introducción, estamos preparados para continuar el desarrollo de la aplicación IlernaTodo y convertirnos en auténticos desarrolladores móviles.

2. Programación de aplicaciones para dispositivos móviles

Los móviles cambiaron el mundo, y continúan haciéndolo. Su potencia y versatilidad nos ayudan y nos mejoran. Con ellos somos capaces de comunicarnos con cualquiera desde cualquier parte, podemos caminar por cualquier ciudad del mundo sin llegar a perdernos, nos identificamos con huella dactilar o reconocimiento facial, jugamos, escribimos, navegamos por internet, conducimos por rutas calculadas en tiempo real teniendo en cuenta el tráfico, escuchamos música, hacemos fotos y vídeos, y los compartimos en la nube, etcétera.

Para los programadores, el desarrollo de aplicaciones móviles es un gran desafío, no solo porque la tecnología evoluciona rápida y constantemente, sino también porque los sistemas son cada vez más complejos y tenemos que conocerlos perfectamente si queremos que nuestras *apps* funcionen a la perfección. Además, el gran número de terminales diferentes, tamaños de pantalla, capacidades, los diferentes sistemas operativos, los *frameworks* y los lenguajes de programación hacen de este un mundo exigente, pero también divertido y lleno de posibilidades. Por otro lado, los entornos de desarrollo han ido evolucionando; son cada vez más inteligentes, rápidos y potentes, y nos ayudan más. Sigamos adelante y pronto veremos cómo desarrollar, de principio a fin, aplicaciones que puedan publicarse en los diferentes *stores* de aplicaciones.

En este tema, empezaremos viendo las fases fundamentales para el desarrollo de aplicaciones Android nativas en lenguaje Kotlin. Poco a poco, iremos adentrándonos en el mundo Android, estudiando el sistema operativo, sus mecanismos y todas sus opciones. Así, las *apps* que desarrollemos podrán sacarle el máximo partido a la potencia del terminal. Llegaremos a comprender los conceptos más complejos, como la programación asíncrona, las bases de datos, las conexiones a internet y los servicios web. ¡Vamos a por ello!

2.1. Herramientas y fases de construcción

2.1.1. Herramientas

En realidad, para empezar a programar aplicaciones Android, solo necesitamos una herramienta, que, además, es gratuita: el entorno de desarrollo Android Studio. El instalador de Android Studio se asegurará de que nuestro sistema dispone de una versión adecuada del JDK de Java y el SDK de Android. Podríamos elegir otros IDE, pero Android Studio es el estándar de Google, y quizá el mejor. Tiene tantas opciones que es

posible que al principio solo necesitemos una pequeña parte. Ya vimos algunas de sus funcionalidades en el tema anterior, y, mientras lo usamos, aprenderemos otras tantas.

Pero aparte del entorno de desarrollo, o integrado en él, podemos hacer uso de otras herramientas muy útiles y, dependiendo del proyecto, imprescindibles. Hagamos un listado rápido:

- **Git**: un gestor de versiones podría considerarse imprescindible para cualquier proyecto, salvo para una aplicación de prueba sin importancia para mantenerla. El cualquier otro caso, un gestor de versiones te permite llevar la cuenta de todos los cambios que se han producido en el proyecto. De este modo, si en un momento dado queremos volver atrás o unir el código de varios programadores, será sencillo. Un gestor de versiones permite desarrollar una *app* entre varios programadores, asegurando la integridad del código, es decir, que dos personas no modifican el mismo código y lo dejen inservible. Hay muchas herramientas de este tipo, pero Git es el control de versiones más usado hoy en día. Es potente y sus comandos son relativamente sencillos, solo hace falta un poco de práctica y entender bien la arquitectura de gestión de versiones que se monte en cada caso. Además, si no queremos usar la línea de comandos, existen aplicaciones gráficas, o incluso el propio Android Studio, que tienen incorporadas muchas funcionalidades de gestión de versiones.

- **Genymotion**: un emulador es una pequeña máquina virtual que emula el *hardware* de terminal real, al que podemos instalar una versión de Android y probar en él nuestras *apps*. Si nuestro ordenador soporta virtualización, la velocidad de ejecución de la máquina aumentará. De todos modos, los emuladores actuales son (afortunadamente) mucho más rápidos que antes. También son más realistas con respecto a las características de *hardware*. Si necesitamos comprobar si nuestra *app* se ve y funciona perfectamente en la mayoría de terminales del mercado pero no disponemos de ellos físicamente, usaremos emuladores. Desde Android Studio podemos crear un dispositivo virtual fácilmente con las características de memoria, pantalla y versión del sistema, entre otras. Sin embargo, si el ordenador en el que desarrollamos no es muy potente y dispone de virtualización, el emulador de Android Studio puede ser algo lento. Si es así, puedes utilizar los de Genymotion, que parecen ser los más potentes. Lo mejor es descargarlo y comprobarlo en cada caso.

- **Postman**: en un capítulo posterior, veremos qué es un servicio web y como acceder a él desde nuestra *app*, pero básicamente no es más que un programa instalado en algún servidor conectado a internet que nos permite obtener y/o enviar datos para ser procesados y almacenados. Para poder probar esos servicios web con facilidad, necesitamos una aplicación que permita definirlos y hacer pruebas sobre ellos. Estas aplicaciones pueden resultar de gran ayuda,

sobre todo cuando los servicios no están bien documentados o quizá aún están en fase de desarrollo. Si queremos tener una idea de los que nos devolverá el servicio cuando lo usemos en nuestra *app*, necesitamos estas aplicaciones. Postman es una de las más conocidas, pero existen otras, como SoapUI, Widzler o SoapSonar.

- **Axure**: para diseñar nuestra interfaz gráfica antes de empezar con el desarrollo, podemos usar papel y lápiz, pero si queremos un resultado más profesional y guardar los diseños junto con el resto del código, podemos utilizar una aplicación de diseño de pantallas, como Axure, MockPlus, Adobe XD, Mock Flow, JustInMind, Cacoo, Frame Box o muchas otras. Estas aplicaciones nos permiten colocar los *widgets* de cada ventana, moverlos, cambiarlos de tamaño, etcétera. Algunas de ellas también nos permiten diseñar un flujo de ventanas, que será el camino que tome el usuario de una pantalla a otra mientras utiliza nuestra *app*.
- **Jira**: principalmente cuando se trabaja en equipo, una aplicación de gestión de trabajo como Jira, Redmine o Trello puede ser de gran ayuda. Estas herramientas nos permiten hacer un listado de tareas, de errores que depurar y de nuevas funcionalidades que desarrollar. De ese modo, podemos calcular el tiempo que se tardará en terminar el desarrollo y asignar las tareas entre los programadores en el tiempo disponible.

Estas son algunas herramientas básicas y es conveniente conocerlas, pero no significa que sean de utilidad en todos nuestros proyectos, y probablemente tendremos que utilizar otras diferentes para otros fines. Como comentamos, el desarrollo de aplicaciones en general, y móviles en particular, avanza a pasos agigantados, y pronto surgirán nuevas necesidades y nuevas herramientas que nos ayuden en cada uno de nuestros proyectos. Para estar al día, no hay más que leer blogs de desarrolladores de vez en cuando.

2.1.2. Fases del desarrollo

Como en todos los oficios, la producción ha pasado de artesanal a técnica y científica. Los primeros programadores luchaban como podían con los unos y los ceros, sin utilidades ni sistemas que les permitiesen hacerlo mejor o más eficiente. La calidad del producto final dependía en gran medida de la inspiración y profesionalidad de los programadores.

Pero el desarrollo de *software* ha evolucionado rápidamente. Las empresas compiten ferozmente por entregar los mejores productos al menor coste posible, y para ello es necesario un proceso productivo enfocado a asegurar la calidad con los recursos

necesarios. Es estupendo que el programador tenga una vena artística, pero eso no es suficiente. Hoy en día, el ingeniero de *software* debe conocer las técnicas de programación, las mejores prácticas, los patrones de diseño, las pruebas de calidad, el análisis de los casos de uso, etcétera.

Si únicamente necesitamos una pequeña aplicación que tenga alguna utilidad para nosotros, o simplemente queremos programar para aprender, quizá entonces no necesitamos más que un poco de artesanía, pero si necesitamos llevar a cabo un gran proyecto profesional con éxito, debemos conocer y aplicar las técnicas de la **ingeniería del *software***, es decir, debemos seguir un camino bien marcado y medido durante el desarrollo; de otro modo, terminaremos malgastando tiempo y recursos en una aplicación llena de fallos que nadie desea descargar en su móvil. Para entender toda la ingeniería del *software* necesitaríamos una biblioteca entera, así que no vamos a entrar en ello, pero es imprescindible al menos conocer algunos puntos básicos, como las fases del desarrollo *software*, también conocido como el ciclo de vida del *software:*

Planificación

El cliente, el jefe de producto o nosotros mismos cuando se nos ocurre una idea para una *app* tenemos solo un concepto abstracto de lo que queremos. Sin embargo, antes de cortar, necesitamos medir. Debemos especificar cada una de las tareas que nuestro programa debe realizar, qué hará y cómo. Antes de arrancar Android Studio, debemos tener preparado un análisis funcional, es decir, debemos listar las funcionalidades que ofrecerá nuestra *app*. También es de gran utilidad diseñar con antelación la interfaz gráfica, o al menos una aproximación: las pantallas, los campos y cómo navegará el usuario por la *app*. Al definir todos los parámetros que influirán en el desarrollo, podremos tener una idea real del tiempo y los recursos técnicos y financieros que serán necesarios. Si existe un cliente, debemos repasar con él que la *app* hará lo que dice en el informe de requisitos y funcionalidades, de modo que podamos acotar el trabajo necesario y repartirlo entre equipos de desarrollo.

Implementación, pruebas y documentación

En esta fase, convertimos el diseño inicial en realidad ejecutable.

La **implementación** significa que codificaremos todas las funcionalidades requeridas. Utilizaremos todos los recursos del sistema operativo necesarios para que la *app* pueda llevar a cabo su funcionalidad de forma eficaz. Es posible que el desarrollo de la aplicación requiera de varios equipos diferentes, que deberán estar sincronizados para que el desarrollo progrese rápida y eficientemente. Dentro de la fase de implementación, el jefe de proyecto podría implementar también una metodología de

trabajo para incrementar la eficacia de los desarrolladores. Actualmente, las más habituales son las metodologías ágiles, como XP, Scrum o Kanvan.

Algunos tipos de **pruebas** pueden realizarse al mismo tiempo que se codifican las funcionalidades. En Android tenemos las pruebas unitarias y las pruebas instrumentales, que nos permiten asegurar que progresamos sin fallos. Las pruebas unitarias o *unit tests* permiten probar el funcionamiento de módulos de código puro, es decir, que no tienen llamadas al sistema operativo o que, de tenerlas, pueden emularse. Las pruebas instrumentales permiten probar el código del módulo directamente en un terminal para asegurarnos de que funcionan incluso las llamadas al sistema. Cuando varios sistemas de la misma aplicación están terminados, pueden hacerse pruebas de integración para asegurar que la coordinación entre ellos funciona como debería. Normalmente, en proyectos profesionales, tendremos también un equipo de QA o *quality assurance*, en el que los técnicos son especialistas en probar y encontrar los posibles fallos del código antes de que la aplicación llegue al usuario. Existen también otros tipos de pruebas que pueden realizarse cuando la aplicación está completa, por ejemplo, test de seguridad para impedir que la aplicación deje al descubierto información personal del usuario, o datos de conexión con los que pueda emularse al usuario en el servidor. Por defecto, en un proyecto de Android Studio, al compilar en *release* el código pasará por ProGuard, que es un ofuscador y optimizador de código. Este filtro reducirá al mínimo el tamaño del código y los recursos, y cambiará el nombre de todas las funciones, variables y clases. De este modo, al *hacker* que desempaquete el APK y estudie sus archivos le resultará casi imposible de entender. Existen otras herramientas para comprobar la seguridad de nuestra aplicación, como Zed Attack Proxy (ZAP), Quick Android Review Kit (QARK), Drozer o Mobile Security Framework (MobSF).

Por último, la **documentación** es imprescindible para disponer de una descripción del funcionamiento interno de los diferentes módulos del programa. Así, la transmisión de conocimiento entre equipos durante el desarrollo y el mantenimiento futuro de la *app* serán rápidos y eficientes. Pongamos el caso de que nuestra *app* utiliza unos servicios web que aún están en desarrollo por el equipo de *backend*. No sabremos cómo llamarlos hasta que el departamento de *backend* nos pase la documentación de la API. Si los de *backend* son unos desarrolladores eficientes, tras el análisis de requisitos de los servicios habrán redactado una documentación suficiente para posteriormente comenzar el desarrollo. El departamento de *backend* nos pasará la información, de modo que podemos saber cómo usar sus servicios antes incluso de que los hayan terminado.

Despliegue y mantenimiento

El **despliegue**, o lanzamiento a producción, significa que el programa está completo y ha pasado todas las pruebas de calidad. Puede distribuirse a los *stores* para su posterior descarga en los dispositivos compatibles, o directamente a los usuarios como un archivo

de instalación APK. El formato Android Application Package (APK o paquete de aplicación Android) ha sido durante mucho tiempo la forma estándar de distribución de una *app* Android. En realidad, no es más que un archivo comprimido que contiene todos los módulos ejecutables, los recursos y certificados necesarios para su funcionamiento. Podríamos utilizar cualquier aplicación de compresión para ver los archivos que lo componen. Actualmente, Google intenta imponer los App Bundles como método de distribución en APK, seguramente por dos motivos distintos. El primero sería por eficiencia, y el segundo por mantener el control de la distribución de aplicaciones en Android. La eficiencia de los Bundles reside en que su en su instalación solo se utiliza el código que cada terminal necesita. En los APK, por el contrario, se necesitaba incluir todas las librerías compiladas para los diferentes procesadores de cada terminal, lo que hace la descarga más lenta y consume mayor espacio en el terminal de destino. Con cualquiera de los dos tipos de paquetes, subir una aplicación al *store* de Google es muy sencillo. Pero antes necesitaremos crear una cuenta como desarrollador para tener acceso a la consola.

Web de la consola para desarrolladores de Google Play Store:

https://developer.android.com/distribute/console

El **mantenimiento** de una app consiste en detectar los fallos que aparecen cuando los usuarios ya están utilizando la aplicación. Para detectar esos fallos, podemos escuchar diferentes fuentes. Una sería el contacto directo con los clientes, que nos comentan los errores que han encontrado. Pero hay otros mecanismos mucho más eficientes, como introducir un código en la *app* que nos avise cada vez que algo falla. Por ejemplo, con unas pocas líneas de código, nuestra *app* podría llamar a los servicios de Firebase Crashlytics cada vez que detecte una excepción. El servidor de Firebase, por su parte, nos enviará una alerta al *email* informándonos del error. Entonces podríamos entrar en la web de Crashlytics, ver los detalles del error, la línea y el archivo en el que ocurrieron, y programar un parche de código para nuestra próxima versión.

Documentación sobre Firebase Crashlytics:

https://firebase.google.com/docs/crashlytics/get-started-android

2.2. Desarrollo del código

Una vez que tenemos el análisis funcional de nuestra aplicación, podemos empezar a codificar. Debemos conocer bien el lenguaje y las tecnologías con las que vamos a trabajar antes de empezar si queremos obtener un código legible y eficiente. Si aprendemos a sacarle el máximo rendimiento al entorno de desarrollo, trabajaremos más cómodamente, y eso influirá en el resultado. Además, debemos seguir una lista de recomendaciones que los desarrolladores de *software* han aprendido con el tiempo y que nos evitarán cometer errores clásicos de arquitectura y programación.

2.2.1. Lenguajes de programación

En el desarrollo de *apps* Android nativas podemos utilizar tanto Java como Kotlin, pero hoy en día preferimos este último, al ser un lenguaje moderno, potente y con mayores perspectivas. Java lleva años siendo uno de los lenguajes más versátiles, debido a la potencia de su *bytecode* y su máquina virtual, que más tarde muchos otros lenguajes han imitado. La máquina virtual de Java o JVM es una especie de procesador virtual que interpreta los *bytecodes* y los ejecuta como el código máquina de cada procesador físico en el que está instalada. Gracias al JVM, Java ha sido un lenguaje capaz de conquistar los dispositivos más inverosímiles con gran éxito, como, por ejemplo, lavadoras o frigoríficos inteligentes. En el caso de los dispositivos Android, las versiones previas a Android 5, Lollipop, disponían del Dalvik, una máquina virtual que traducía el *bytecode* de Java al del procesador del móvil. El formato del archivo APK y los archivos .dex que incluye en su interior vienen de esa época, y aún se utilizan. Pero a partir de Android 5 el sistema operativo venía integrado con otra máquina más potente para la ejecución de aplicaciones Android llamada Android Runtime (ART), que dejó obsoleto al Dalvik.

Con el paso de los años y el avance de la tecnología, Java ha encontrado dificultades para adaptarse a las nuevas necesidades, y un sustituto parecía necesario. Entonces llegó Kotlin, con una sintaxis aún más sencilla y potente. Kotlin es un lenguaje orientado a objetos que también permite la programación funcional. Dispone de potentes tipos de datos que nos facilitarán el desarrollo de los módulos más complejos, empleando menos líneas de código. Además, es totalmente compatible con Java, pues se traduce a los mismos *bytecodes*. Por ello, si encontramos una aplicación en Java, podríamos continuar el desarrollo en Kotlin: los objetos en uno y otro lenguaje se comunicarán entre sí perfectamente. De hecho, el increíble editor de código de Android Studio nos permite copiar código de un archivo Java y pegarlo en nuestro nuevo archivo Kotlin, que será automáticamente traducido a la sintaxis de Kotlin.

2.2.2. Entorno de trabajo

El editor de código del Android Studio presenta potentes funciones de autocompletado, con las que teclearemos menos y codificaremos más rápido. Además, podemos pedirle que mantenga un formato de código determinado y el editor corregirá mientras escribimos o pegamos código. Por otro lado, disponemos del generador de código; cuando creamos una nueva clase, implementamos un interfaz o llamamos a una función con un objeto, el editor se dará cuenta de lo que buscamos y escribirá automáticamente el esqueleto de lo que vamos a construir.

Cuando creamos nuestro proyecto, Android Studio genera una estructura de directorios que debemos entender. Pero más allá de lo que nos ayude el IDE, es nuestra obligación mientras desarrollamos ir creando y manteniendo una estructura de directorios lógica y fácil de navegar. Por ejemplo, si nuestra *app* hace uso de una base de datos y de servicios web, podemos crear un directorio llamado "repositorio", y dentro dos subdirectorios llamados "local" (para la base de datos) y "remoto" (para las llamadas a los servicios web). Cada equipo o desarrollador puede tener unas preferencias distintas a la hora de crear la estructura de directorios, pero antes de empezar a programar, debemos llegar a un acuerdo con el resto para que las clases, interfaces y objetos estén ordenados y sean de fácil acceso. En todos los procesos de desarrollo debemos ser ordenados si queremos que nuestro código sea legible y libre de errores.

Si el proyecto es muy grande, podría incluso dividirse en módulos distintos. El IDE comprenderá la posición de nuestras clases en la estructura de directorios, y creará los nombres de paquete de forma automática. Igualmente, hará que cualquier refactorización del nombre o la localización de una clase lleve consigo el cambio de las referencias que otro código tiene de la clase. Android Studio es un IDE estupendo, y mediante su uso iremos aprendiendo toda las funciones que nos presta.

2.2.3. Buenas prácticas

Aunque el desarrollo de *software* tiene una historia relativamente reciente, hemos aprendido mucho de nuestros errores pasados a la hora de codificar nuevas aplicaciones. Son muchos los gurús del desarrollo *software* que han buscado solución a los errores clásicos que cometemos los programadores. Vamos a destacar algunas de las soluciones más brillantes, estéticas y utilizadas en las mayores empresas de *software*. Si queremos llegar a ser buenos programadores, debemos ser conscientes de su existencia y ponerlas en práctica siempre que nos sea posible:

Patrones de diseño

Algunos expertos en programación comprendieron la importancia de encontrar estructuras de código que resultasen eficaces e impidiesen que programadores no experimentados las usasen de manera incorrecta, generando errores. Los patrones de diseño *software* pretenden dar solución a problemas típicos de arquitectura del código que se repiten en casi todas las aplicaciones, estandarizando el diseño de la aplicación. Suelen agruparse por su funcionalidad, y existen tres: creacionales, estructurales y de comportamiento.

Veamos un ejemplo con un patrón de diseño sencillo, por ejemplo, el llamado Singleton, se trata de un patrón de diseño creacional. Imaginemos que tenemos una clase *FileUtil* con métodos y variables que utilizamos para copiar y pegar archivos, abrir y leer datos de ellos, etcétera. Es necesaria solo una instancia de la clase para todas las operaciones de la *app*. Pero si al crear nuestra clase *FileUtil* no nos aseguramos de controlar su instanciación, puede que terminemos creando varios objetos *FileUtil* en diversas partes del código. Si utilizamos el patrón Singleton, haremos que *FileUtil* solo pueda crearse una vez. En Java, la clase sería algo así:

```
public class FileUtil {
   private static FileUtil INSTANCE = null;
   // Construcor privado para impedir instanciación directa
   private FileUtil(){}
   // Función sincronizada para evitar problemas con código asíncrono
   private synchronized static void createInstance() {
      if(INSTANCE == null) {
         INSTANCE = new FileUtil();
      }
   }
   // El usuario de la clase obtendrá una instancia con esta función
   public static FileUtil getInstance() {
      if(INSTANCE == null) createInstance();
      return INSTANCE;
   }
   // Las funciones que realmente solucionan nuestras tareas
   public void funcionesUtiles() {
      //...
   }
}
```

Por fortuna, en Kotlin todo es más sencillo, y no tendríamos más que utilizar un objeto en lugar de una clase, de esta forma:

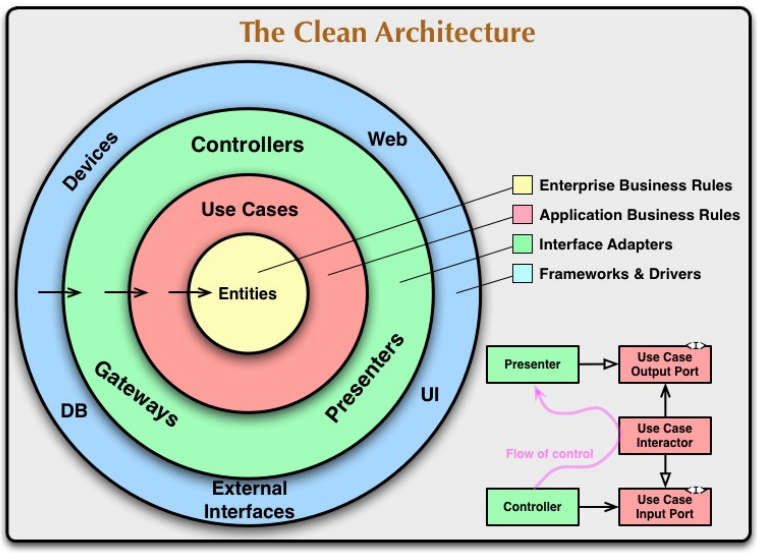
```
object FileUtil {
   // Las funciones que realmente solucionan nuestras tareas
   fun funcionesUtiles() {
      //...
   }
}
```

Son muchos los patrones de diseño, y hay varias formas de agruparlos. Unos se utilizan constantemente, y otros rara vez. En cualquier caso, es aconsejable hacer un repaso de todos ellos, porque pueden ahorrarnos muchos quebraderos de cabeza y evitar que perdamos el tiempo reinventando la rueda. Por fortuna, incluso el SDK de Android nos

ayudará con esto, pues algunas de sus clases implementan ya patrones conocidos, como, por ejemplo, el Modelo Vista VistaModelo, Adapter, Builder.

Clean Code

Robert Martin, también conocido como Uncle Bob, es un gurú de la ingeniería del *software*, autor de varios libros clave en el desarrollo *software*. Clean Code y Clean Architecture son dos de sus libros. En ellos se dan una serie de pautas para la construcción de un código robusto y libre de errores de diseño y programación. El desarrollo Clean trata de mantener unas estructuras y unas normas en nuestro código, con el objetivo de dividir las funcionalidades en tareas cada vez más pequeñas. Así, cada parte del código desarrollará una y solo esa pequeña tarea, y la comunicación entre las partes está controlada para que cada una conozca y utilice solo lo imprescindible de otra. De este modo, los errores de un módulo no se trasladarán al resto y serán fácilmente identificables. El siguiente diagrama de Uncle Bob es ya un clásico en el desarrollo de aplicaciones, veámoslo:



Cuando analizamos los requisitos de nuestra aplicación, encontramos los **casos de uso**, es decir, las funcionalidades, las tareas que el usuario pide a la *app*. Esos casos de uso necesitan estructuras básicas de datos llamadas **entidades**. Por ejemplo, en una *app* de geolocalización, un caso de uso es “dame mi posición actual”, y la entidad sería una clase *Localización* que tuviese como campos la latitud, la longitud, la altura, la precisión, etcétera. Para presentar los datos de la identidad obtenidos por el caso de uso “dame mi posición”, necesitaríamos un código que formatease los datos, un *presenter* que convertirá los datos en coma flotante de latitud y longitud en una cadena de caracteres formateada, por ejemplo, así: “Mi posición: $latitud / $longitud”. Pero ahora necesitamos las clases de la interfaz gráfica para presentar ese texto en un campo

TextView, por ejemplo. Las flechas que se ven hacia dentro del círculo significan la dirección de uso, es decir, las capas exteriores conocen y usan a las capas interiores, pero no al revés. Las capas interiores son abstractas y, por lo tanto, no dependen de las exteriores. Por ejemplo, si decidimos que nuestra *app* utilizará una librería de base de datos en lugar de otra, el cambio no debería afectar a la estructura de nuestros datos, ni a los casos de uso, ni siquiera al código que controla las peticiones a la base de datos.

SOLID

Son las siglas de cinco principios Clean que pueden ayudarnos a mejorar la calidad de nuestro código. En la práctica, quizá no conviene esforzarse en llevarlos todos a cabo, pero algunos son imprescindibles en cualquier proyecto, por simple que sea. Vamos a enumerarlos rápidamente.

Uncle Bob sigue siendo un referente, podemos aprender mucho en su blog:

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

S: *single responsibility*. Cada clase, cada método y cada variable deben tener un único objetivo, no pueden valer para dos cosas distintas, porque de hacerlo estarías complicando el código innecesariamente.

O: *open-closed*. Las entidades deben estar abiertas para ser extendidas con más funcionalidad, pero no debe permitirse su modificación una vez diseñadas, de modo que la funcionalidad primordial siga siendo igual.

L: *Liskov substitution*. Precisamente por el principio anterior, este principio asegura que una clase puede ser sustituida en el código por cualquiera de sus subclases sin modificar el funcionamiento, porque hemos asegurado que llevará a cabo las mismas operaciones básicas, aparte de otras añadidas.

I: *interface segregation*. No debemos mezclar métodos de casos de uso diferentes en la misma interfaz, porque causará problemas. Es mejor crear diferentes interfaces, cada una justo con los métodos que la definan.

D: *dependency inversion*. La funcionalidad de una clase o método debe depender solo de abstracciones, no de código concreto. Es decir, una clase no debería estar compuesta en su interior de objetos de otras clases, porque, de cambiar esas clases, la clase principal fallaría. La clase debe depender de interfaces, y las clases reales que ejecuten las funciones cumplirán esas interfaces.

2.3. Compilación, preverificación, empaquetado y ejecución

Una vez que hemos codificado nuestra aplicación, aún tenemos tareas que realizar. El código Kotlin debe compilarse para convertirse en *bytecodes* dentro de archivos .dex, que serán empaquetados en un APK o Bundle y que luego serán instalados y ejecutados por una máquina Android. Por fortuna, Android Studio hará que estas tareas sean tan sencillas como pulsar un botón. Pero si queremos sacar el máximo partido a nuestras aplicaciones, debemos conocer las opciones que nos ofrece nuestro entorno de desarrollo.

2.3.1. Compilación

Antes de que nuestro código en Kotlin pueda ser ejecutado en el procesador virtual del emulador, o en el real de un terminal físico, debe pasar por varios filtros, conocidos en conjunto como compilador. Cuando seleccionamos la opción *Run* o el icono de flecha verde del menú de Android Studio, el compilador analizará el código y comprobará que no tenga errores, realizará algunas mejoras en él para aumentar su eficiencia y lo transformará en un lenguaje comprensible para el procesador del móvil. Si es la primera vez que ejecutamos nuestra *app*, Android Studio creará algunas nuevas carpetas dentro del directorio de proyecto. Por ejemplo, bajo la carpeta "app", creará "build", "outputs", "apk" y, dentro, "debug". En esa última carpeta, el entorno de desarrollo guardará el archivo app-debug.apk, que es un APK especialmente diseñado para ser depurado en el emulador o terminal que tengamos conectado. Es importante recordar que ese APK no vale para ser distribuido, es solo un APK de pruebas.

Para facilitarnos nuestra tarea como desarrolladores, Android Studio presenta diferentes tipos de compilación para cada momento del desarrollo. Por ejemplo, cuando estamos empezando, no nos interesa que el código compilado sea supereficiente, lo que sí necesitamos es poder depurarlo y ver qué valor toman las variables y los objetos mientras se ejecuta. Sin embargo, cuando nuestra *app* esté a punto, si queremos analizar la eficiencia de la memoria o la rapidez de nuestro código, desearemos que el compilador realice las mejoras típicas de una aplicación lista para producción.

Dentro de Android Studio existe un programa que controla todo el mecanismo de compilación llamado Gradle. Por ello, cuando necesitemos configurar las opciones de compilación, modificaremos los archivos llamados build.gradle. En nuestro proyecto tendremos normalmente dos archivos de configuración de Gradle: uno global para todo el proyecto, o de aplicación, y otro por cada módulo. Pero antes de estudiar el código Gradle, debemos entender los mecanismos que nos permitirán crear los diferentes tipos de APK en cada momento. Veamos qué opciones tenemos para definir los diferentes tipos de compilación:

- ***Build types*** o modos de compilación. Definen ciertas propiedades del compilador, que representan los diferentes estadios del desarrollo de la *app*. Por ejemplo, mientras estamos desarrollando la *app*, usaremos el modo *debug*, con el que nos aseguramos que el código sea depurable y que el APK se firme con la clave por defecto y no con una clave real. Sin embargo, cuando nuestra *app* esté terminada, querremos utilizar el modo *release*, en el que podremos minimizar, ofuscar y firmar la aplicación de forma real para que pueda distribuirse de forma segura en los *stores* y ser ejecutada eficientemente. Normalmente, no se utilizan más que estos dos, pero pueden crearse más.

- ***Product falvors*** o variantes de producto. Representan diferentes versiones de nuestra *app* que necesitamos para distribuir a diferentes usuarios. Imaginemos que tenemos una versión gratuita con funcionalidades reducidas y una versión de pago sin anuncios y totalmente funcional. Ambas compartirán una parte del código, pero tendrán algunas clases diferentes. O quizá necesitamos que la aplicación sea idéntica, pero tenga unos iconos y colores para una empresa y otros diferentes para otra, es decir: una misma *app*, dos *brandings;* solo cambiarían algunos recursos gráficos. Las variantes de producto son opcionales, de hecho, habrá aplicaciones que no hagan uso de ellas.

- ***Build variants*** o variantes de compilación. En realidad, las variantes de compilación no son más que la combinación de las opciones anteriores. A la hora de compilar y empaquetar la *app*, el programador deberá elegir la combinación final de *debug/release* y los diferentes tipos de *product flavors* que el compilador utilizará. Si, por ejemplo, tenemos dos clientes y cada uno tiene un *flavor* diferente, mientras depuramos deberemos comprobar las variantes de compilación *debug/flavor1* y *debug/flavor2*. Y cuando hallamos terminado y queramos distribuir los dos APK, usaremos *release/flavor1* y *release/flavor2*.

Los archivos de configuración de Gradle utilizan un lenguaje llamado Groovy, y aunque no es necesario que lo dominemos, será conveniente que tengamos una idea de cómo funciona. Si abrimos el archivo de módulo build.gradle que ha generado Android Studio, podremos observar un código similar a este:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
android {
  compileSdkVersion 29
  buildToolsVersion "29.0.3"
  defaultConfig {
    applicationId "com.tudominio.tuaplicacion"
    minSdkVersion 23
    targetSdkVersion 29
    versionCode 1
    versionName "1.0"
  }
```

```
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = '1.8'
    }
}
dependencies {
    implementation fileTree(dir: "libs", include: ["*.jar"])
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    implementation 'androidx.core:core-ktx:1.3.0'
    implementation 'androidx.appcompat:appcompat:1.1.0'
}
```

No es necesario que lo entendamos todo. Aquí nos interesa comprender las partes *defaultConfig* y *buildTypes*. En la primera van las configuraciones generales que tendrá por defecto nuestra *app*. ApplicacionId es el nombre completo de nuestra *app*, con nuestro dominio: "com.tudominio" y el nombre de la *app:* "tuaplicacion". Debe ser único en el Play Store, o Google no te dejará subir la *app*. En *defaultConfig* se define también la versión de nuestra *app*, el SDK mínimo con el que es compatible y la versión del SDK para la que está desarrollada. En *buildTypes* está la opción *release*, para definir cómo será compilada la *app* en modo producción. En este caso, está desactivada la opción *minify*. Esta opción permite indicar si queremos reducir, optimizar y ofuscar nuestro código dentro del APK para que sea menos pesado, más rápido y más difícil de decompilar por un *hacker*. Así que, cuando terminemos de depurar, deberíamos activar esta opción usando *true* en lugar de *false* para la variable *minifyEnabled*.

Veamos un ejemplo de cómo modificar el build.grade por defecto de Android Studio para definir nuestras propias variantes de compilación. Primero definamos nuestros tipos de compilación:

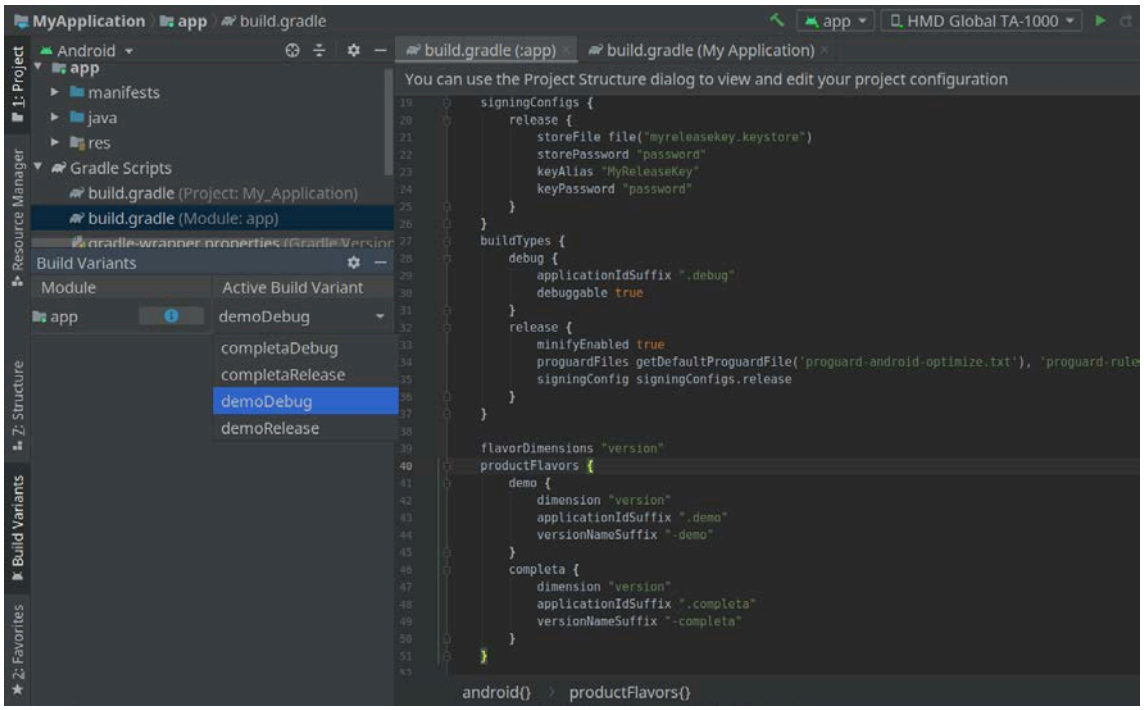
```
signingConfigs {
    release {
        storeFile file("archivo_de_firma.ks")
        storePassword "mipassword"
        keyAlias "mialias"
        keyPassword "mipassword"
    }
}
buildTypes {
    debug {
        applicationIdSuffix ".debug"
        debuggable true
    }
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        signingConfig signingConfigs.release
    }
}
```

Observemos cómo hemos creado la configuración de firma *signingConfigs* válida únicamente para *release*, ya que en *debug* nos basta con la firma de depuración por defecto del compilador. Dentro de *buildTypes*, hemos definido *debug* para añadir al nombre de aplicación el sufijo .debug de modo que podamos distinguirla del APK *release*. Al tener diferente nombre de *app*, podríamos tener la aplicación instalada dos veces en nuestro terminal de pruebas, una en modo *debug* y otra en modo *release*. Hemos especificado que *debug* sea depurable, y que en *release* se lleven a cabo las funciones de *minify*.

Veamos ahora cómo crear un par de variantes de producto, una para la versión demo y otra para la versión completa:

```
flavorDimensions "version"
productFlavors {
  demo {
     dimension "version"
     applicationIdSuffix ".demo"
     versionNameSuffix "-demo"
  }
  completa {
     dimension "version"
     applicationIdSuffix ".completa"
     versionNameSuffix "-completa"
  }
}
```

Creamos la dimensión *version*, que tendrá dos opciones, demo y completa. Cada una añadirá un sufijo al nombre de la aplicación, así como a la versión. Una vez sincronizado el proyecto con los nuevos cambios, podríamos seleccionar la variante de compilación que deseamos ejecutar mediante la pestaña *Build Variants* que podemos encontrar en la barra izquierda de Android Studio. Veamos una imagen de las opciones que tendríamos:



Manejar todas las opciones que permite Gradle es muy difícil, pero a medida que lo utilicemos iremos aprendiendo, en cualquier caso, siempre es buen momento para echarle una ojeada a la documentación oficial.

Para conocer todas las opciones de compilación de Android Studio:

https://developer.android.com/studio/build

2.3.2. Preverificación

Mientras codificamos, podemos desarrollar pruebas unitarias y pruebas instrumentales para asegurarnos de que cada parte del código hace lo que se supone que tiene que hacer. Además, cuando compilamos, cualquier error de sintaxis hará que el compilador se detenga, mostrándonos la línea fallida con un mensaje descriptivo.

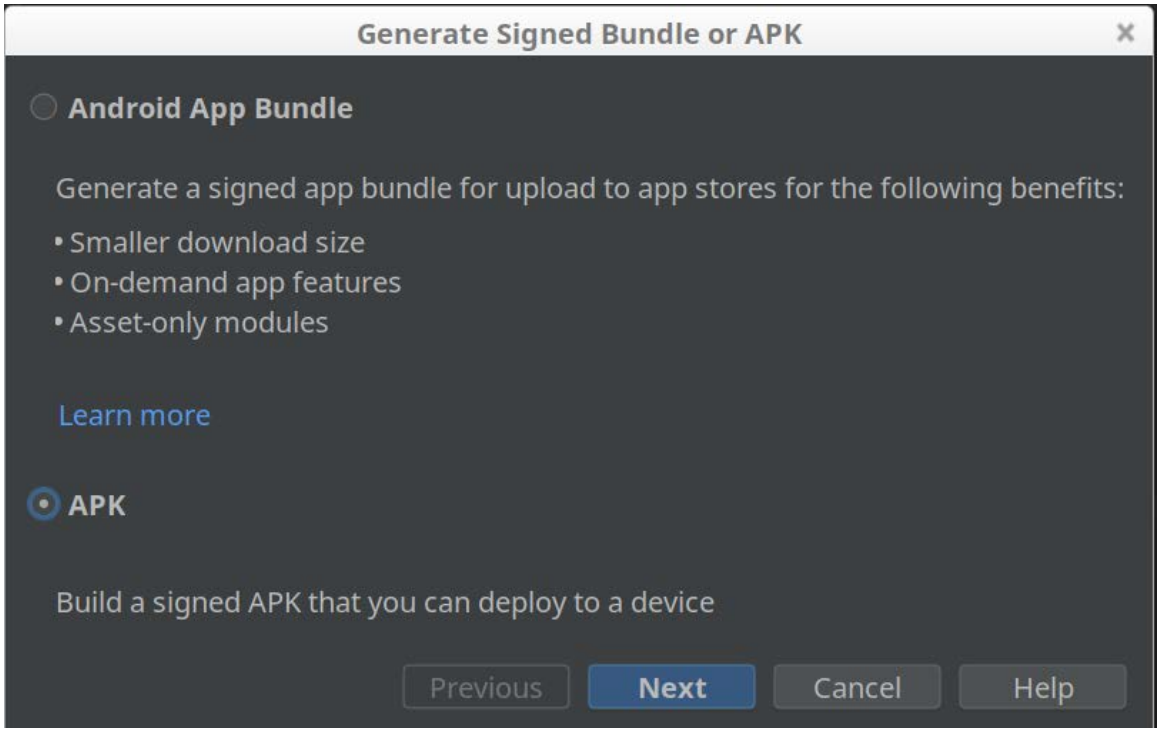
Pero aún podemos ir más allá: Android Studio nos brinda una herramienta de revisión de código llamada Lint, que puede ayudarnos a identificar y corregir problemas estructurales en nuestra *app*. Al pasar el escáner de Lint, la herramienta nos hará un listado de problemas y recomendaciones, cada uno con un mensaje descriptivo y con un nivel de gravedad, para mejorar el código de nuestra aplicación. Podemos configurar Lint para que solo nos avise de errores de cierto calibre, si no nos interesan algunas de las mejoras menos importantes que pueda sugerirnos. La herramienta Lint comprueba nuestro proyecto Android para buscar posibles errores y posibles optimizaciones de

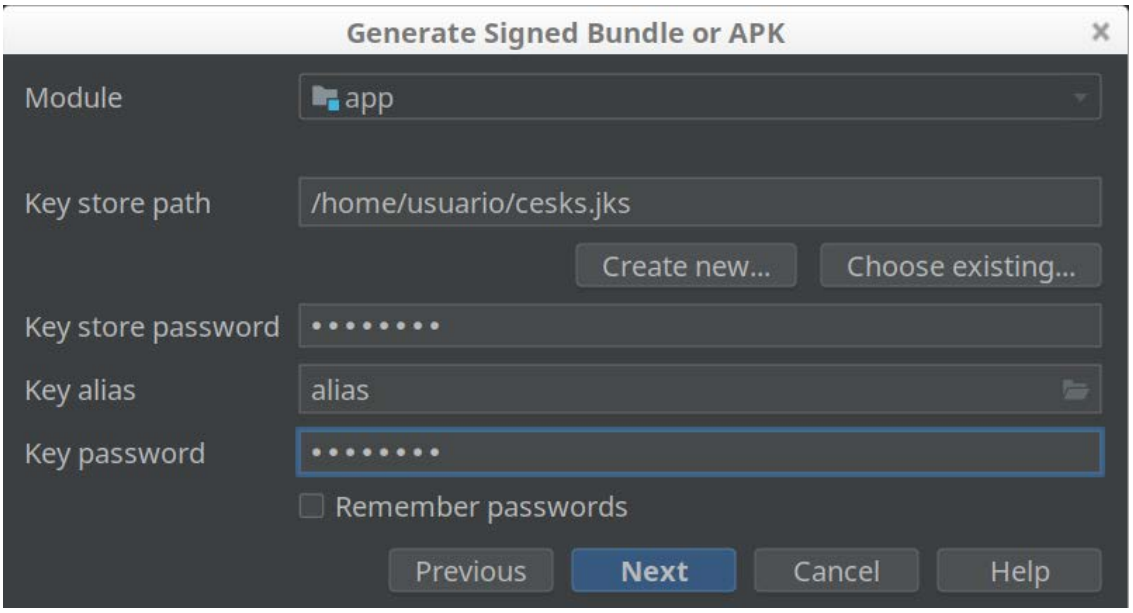
código, de seguridad y rendimiento, de usabilidad y de accesibilidad. Mientras desarrollas en Android Studio, Lint escanea el proyecto para recomendarte cambios en el código. Pero también puedes inspeccionar el código manualmente desde el menú: *Analyze > Inspect Code*, o lanzar Lint desde la línea de comandos: *./gradlew lint*

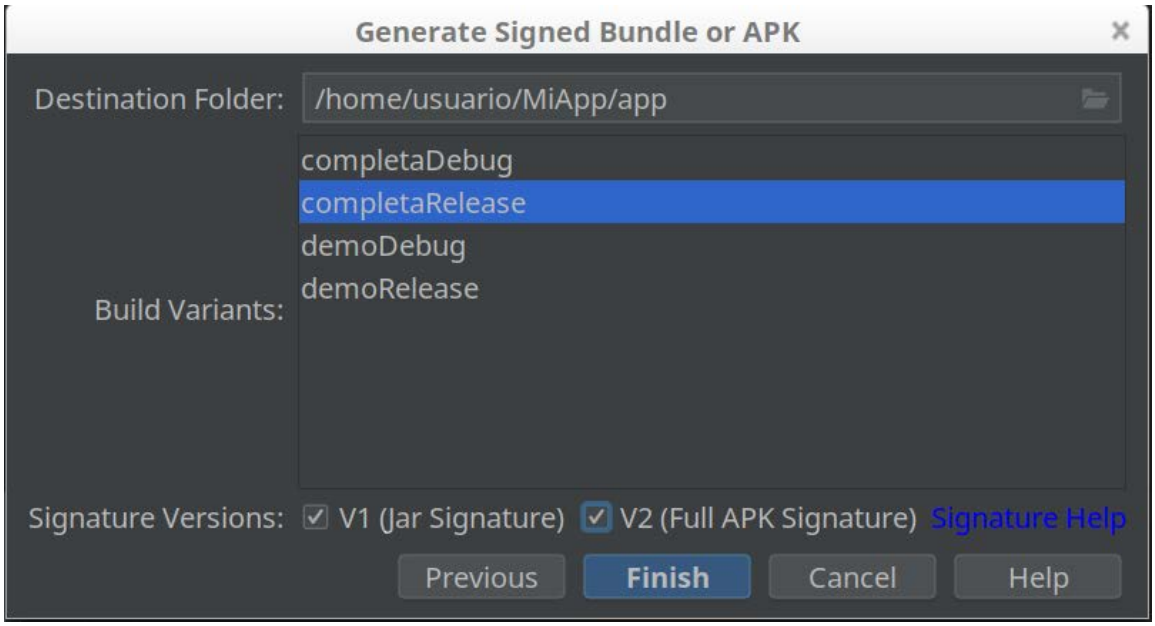
2.3.3. Empaquetado

Ya hemos hablado anteriormente de los paquetes de instalación de Android: del muy conocido APK y del más reciente Bundle. Veamos detalladamente cómo generar estos paquetes de instalación para la distribución de nuestra *app*.

En el menú *Build* de Android Studio disponemos de dos opciones para generar el empaquetado: *Build Bundle(s) / APK(s)*, que generará un Bundle o APK con el *build variant* activo actualmente; y *Generate Signed Bundle / APK*, que nos llevará por una serie de pasos a obtener el paquete que deseamos. Esta última opción requiere un archivo Key Store para firmar el Bundle o APK que se genere. Para que un paquete de instalación Android pueda instalarse, debe estar firmado con un certificado digital, que estará almacenado en el archivo Key Store. Podríamos crear el archivo en el proceso de empaquetado, o tenerlo ya creado y usarlo cada vez que firmamos una de nuestras *apps* para subirla al *store*.



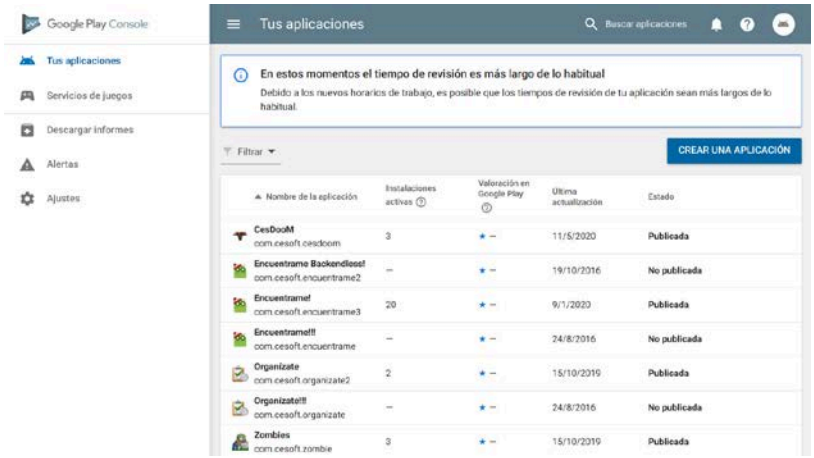




Cuando la compilación y el empaquetado terminen, Android Studio mostrará una notificación en la esquina inferior derecha. En la notificación se nos presenta un enlace al explorador de archivos donde encontraremos el paquete creado. El *path* en el que encontraremos el paquete será algo similar a esto: *file:///home/usuario/MyApplication/app/completa/release*.

Una vez que tenemos nuestro APK o Bundle, podremos publicarlo en el Play Store de Google o en otros repositorios. Aunque el estándar es el Play Store, y a Google le gustaría que fuese el único, existen otros como F-Droid, Aptoide, Getjar, Uptodown o Amazon, pero tengamos en cuenta que, para poder instalar *apps* de estas fuentes, tendremos que activar la opción del móvil: *Ajustes > Seguridad > Orígenes desconocidos*.

Estas plataformas son gratuitas y no piden nada por subir las aplicaciones. Google tampoco pide mucho más, pero debemos registrarnos como desarrolladores antes de que podamos subir nuestra primera *app*. En el proceso de registro se nos pedirá un único pago de 25 dólares. Merece la pena, como programador, para poder acceder a la consola de desarrollador del Play Store y a sus herramientas. Para los desarrolladores de iOS sale bastante más caro, está en 99 dólares anuales, y también son mucho más estrictos con el estilo de las *apps* que permiten subir.



2.3.4. Instalación y ejecución

Sería difícil encontrar a alguien que aún no sepa descargar, instalar y ejecutar una *app* en su móvil desde cualquier *store*. Sin embargo, como desarrolladores debemos aprender también otras formas de instalación y ejecución. Ya conocemos la opción del menú de *Android Studio Run > Run app* o el icono de flecha verde, que lanza en el dispositivo seleccionado la aplicación elegida, pero imaginemos que nos pasan un archivo APK. Para instalarlo lo más rápido sería llamar al ADB desde la línea de comandos. El Android Debug Bridge, o ADB, es una aplicación incluida en el SDK para la depuración de aplicaciones, aunque puede ayudarnos en muchos otros casos. Podemos acceder a la línea de comandos de nuestro ordenador y teclear *adb --version*, con lo que obtendremos la versión instalada. Si el comando falla, deberemos revisar la instalación y la variable *path* del sistema. Al otro lado del puente, tendremos nuestro terminal de pruebas conectado al ordenador mediante un cable USB. Para que el móvil se comunique con el ADB, debemos habilitar la *depuración USB* en *Ajustes > Opciones de desarrollo* del terminal. Si las opciones de desarrollo no aparecen en el menú de ajustes de nuestro teléfono es porque aún no las hemos activado. Para activarlas, debemos acceder a *Ajustes > Información del teléfono > Número de compilación* y pulsar sobre él siete veces seguidas: se nos mostrará un mensaje que informa de que se han activado. En algunos móviles estas rutas pueden ser ligeramente diferentes, como, por ejemplo: *Ajustes > Sistema > Avanzado > Opciones para desarrolladores*. Para asegurarnos de que el ADB ha conectado ordenador y móvil, lancemos el comando *adb devices:*

```
cesar@BITEMYASS:/home/usuario$ adb --version
Android Debug Bridge version 1.0.41
Version 30.0.2-6538114
Installed as /home/cesar/Android/Sdk/platform-tools/adb
cesar@BITEMYASS:/home/usuario$ adb devices
List of devices attached
9db84c5 device
CS22SA1A0000977 device
D1CGAPF710704160        device
```

Como podemos observar, tenemos conectados tres terminales diferentes, a los que podemos acceder mediante comandos. Si solo tenemos un móvil conectado, es muy sencillo, lanzamos el comando y listo. Pero si tenemos varios conectados al mismo tiempo, debemos especificar para qué terminal va dirigida la llamada. Imaginemos que queremos instalar nuestro APK:

```
cesar@BITEMYASS:/home/usuario$ adb shell pm list packages
adb: more than one device/emulator
cesar@BITEMYASS:/home/usuario$ adb -s 9db84c5 shell pm list packages
package:com.funeasylearn.phrasebook.portuguese
package:com.android.cts.priv.ctsshim
package:com.qualcomm.qti.qms.service.telemetry
```

Vemos que es necesario utilizar la opción *-s numero_de_serie*, donde *numero_de_serie* es el número que se lista con el comando *adb devices*. El comando *adb shell pm list packages* lista los paquetes, es decir, las aplicaciones que tenemos instaladas en el dispositivo. Hemos utilizado el comando de *linux grep* para filtrar los paquetes que nos interesan. El comando *adb shell am start* lanza la actividad de la *app* seleccionada. Imaginemos ahora que hemos desconectado los otros dos terminales, para ahorrarnos el comando *-s*. Para instalar el APK, utilizaremos el comando *adb install path/archivo.apk:*

```
cesar@BITEMYASS:/home/usuario$ adb install MiApp/app/completa/release/app-completa-release.apk
Performing Streamed Install
Success
```

Una vez instalada, podríamos lanzar la *app* de la siguiente manera con el comando *adb shell am start -n nombre_app/nombre_activity:*

```
cesar@BITEMYASS:/home/usuario$ adb shell pm list packages | grep ilerna
package:com.ilernaonline.miapp.completa
cesar@BITEMYASS:/home/usuario$ adb shell am start -n com.ilernaonline.miapp.completa/com.ilernaonline.miapp.MainActivity
Starting: Intent { cmp=com.ilernaonline.miapp.completa/com.ilernaonline.miapp.MainActivity }
```

Si no hubiésemos creado los *build variant*, el comando podría haber sido más simple: *adb shell am start -n com.ilernaonline.miapp/.MainActivity*.

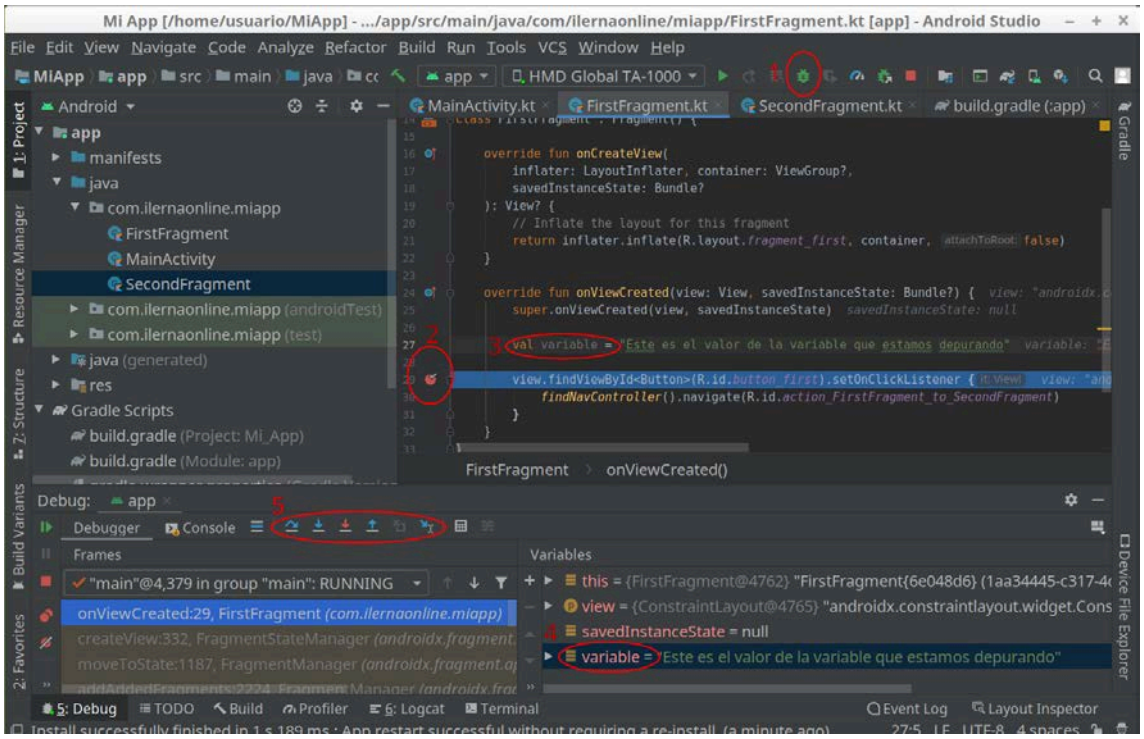
El ADB es imprescindible para el programador Android. Si necesitas más información sobre sus comandos, puedes escribir *adb --help* en la línea de comandos o leer la documentación oficial sobre la herramienta en la web.

Documentación oficial de la utilidad ADB:

https://developer.android.com/studio/command-line/adb

2.3.5. Depuración

En muchas ocasiones, veremos que la aplicación que estamos desarrollando tiene algún comportamiento extraño que no podemos explicar. La mejor manera de conocer la causa del problema y arreglar el error es depurar la app. Tenemos diferentes alternativas. La más evidente es utilizar la opción depuración que tiene Android Studio. Podemos ejecutar la opción del menú: *Run > Debug app* o hacer clic en el icono *debug*, que es un bichito verde. Al seleccionar la opción *debug*, la aplicación será instalada en el dispositivo seleccionado en modo depuración:



1. Es el icono de ejecución modo *debug*. La *app* se lanzará en el terminal seleccionado, en este caso, HMD Global TA-1000. Aparecerá la pestaña *Debug*, en la que vemos la posición actual y las variables de ámbito.

2. Podemos usar lo que se conoce como un *breakpoint* o punto de ruptura. Lo añadimos y eliminamos haciendo clic en la parte izquierda de la línea en la que estamos interesados. Su icono es un círculo rojo, y al pasar por ese punto en modo depuración, la ejecución se detendrá.

3. Cualquier variable del código podrá ser observada en modo depuración. Si se trata de objetos, podrás navegar por todas sus propiedades.

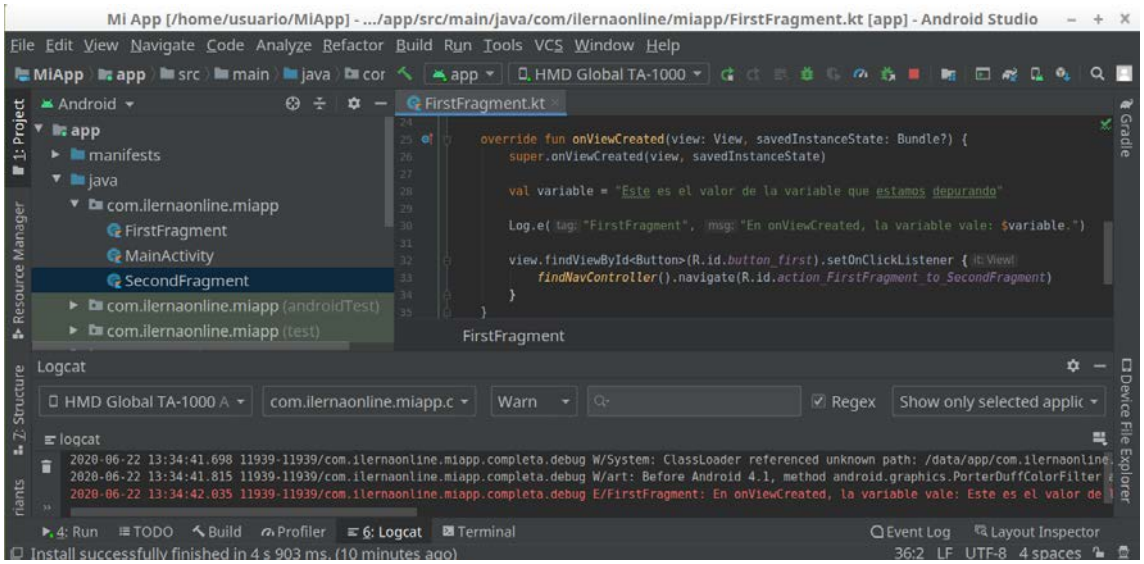
4. Vemos cómo aparecen las variables de la función actual. En este ejemplo, como puedes ver, la función actual es la que aparece en el panel de la izquierda: *onViewCreated*.

5. Una vez que la ejecución se detiene en un *breakpoint*, puedes reanudar la ejecución mediante las opciones: *Step Over*, *Step Into*, *Force Step Into*, *Step Out* y *Run to Cursor*.

Otro método de depuración muy utilizado también sería ejecutar la *app* sin modo de depuración, pero insertando mensajes en puntos clave del código. De esta manera, podríamos comprobar la lista de mensajes para entender por dónde ha pasado la ejecución, qué valores tenía cada variable en ese punto, etcétera. Imaginemos que queremos depurar la misma función: *onViewCreated*. Añadiríamos una llamada a alguna de las función de la clase *Log*, por ejemplo:

```
Log.e("FirstFragment", "En onViewCreated, la variable vale: $variable.")
```

La clase *Log* dispone de diferentes funciones, dependiendo de la gravedad del mensaje que queremos lanzar en la consola Logcat. Tenemos *v()* para *verbose*, los mensajes menos importantes; *d()* para *debug* o mensaje de depuración; *i()* para info o mensaje de información; *w()* para *warning* o alerta; y *e()* para mensajes de error.



El Logcat es una gran herramienta que nos permitirá ver y filtrar los mensajes de depuración de todas las aplicaciones arrancadas en el terminal o emulador. Debemos tener en cuenta que las aplicaciones instaladas en *release*, muy probablemente, no podrán depurarse de este modo, pues se desactiva la opción de depurado. Sí podrán verse, no obstante, los mensajes de las excepciones que generen un *crash* durante la ejecución incluso en *release*.

Otro caso que pudiera darse sería la necesidad de depurar una *app* desde un terminal que no esté conectado a nuestro ordenador mediante un cable USB. Si el dispositivo móvil está conectado a la misma red que nuestro ordenador, por ejemplo, compartiendo la misma red wifi, podremos depurar la *app* con comandos del ADB.

Primero tendremos que apuntar la dirección IP del terminal en la red wifi, por ejemplo, accediendo a *Ajustes > Información del teléfono > Estado > Dirección IP*. En nuestro caso, el móvil está en la 192.168.0.18. Ahora debemos conectar el móvil a nuestro ordenador con el cable USB y lanzar el comando *adb tcpip 5555*, que define el puerto del ADB como el 5555. Ahora podemos desconectar el cable USB y lanzar el comando *adb connect 192.168.0.18*. Si todo va bien, recibiremos la confirmación. Ahora podremos ver el Logcat del terminal como si estuviese conectado por USB.

```
cesar@BITEMYASS:/home/usuario$ adb tcpip 5555
cesar@BITEMYASS:/home/usuario$ adb connect 192.168.0.18
connected to 192.168.0.18:5555
```

2.4. Interfaces de usuario. Clases asociadas

El SDK de Android nos brinda las clases necesarias para acceder a la interfaz de usuario del sistema operativo Android de forma sencilla. Podemos heredar de esas clases para crear nuestra interfaz de usuario a nuestro gusto, para que termine siendo lo más parecida al *mockup* que creamos en la fase de análisis. En este punto, vamos a estudiar las clases más comunes para la construcción de una interfaz de usuario de una app Android.

2.4.1. Actividad

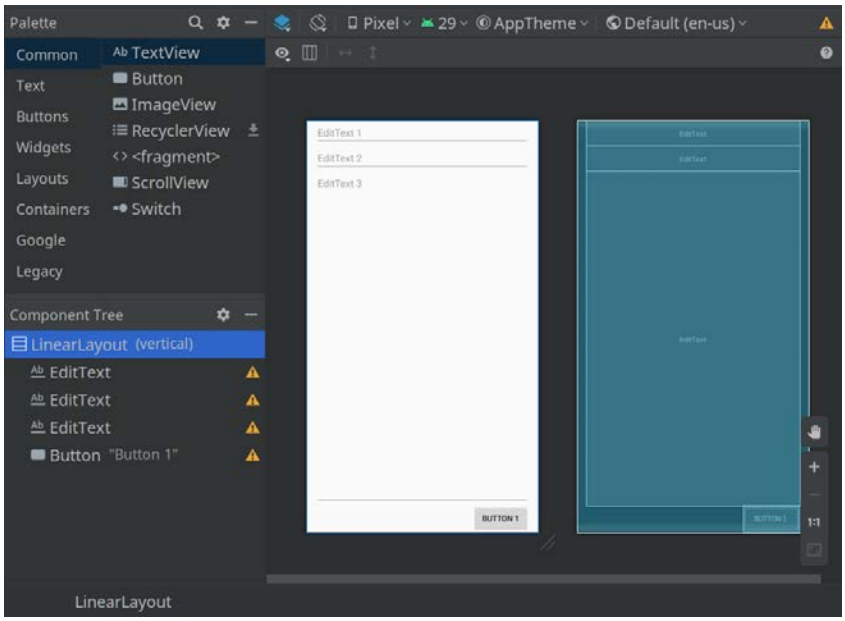
La actividad es la más común de las clases del SDK de Android para crear una pantalla en nuestra *app*. Dentro de cada actividad, colocaremos los diferentes componentes de la interfaz gráfica, como campos de introducción de texto, botones, etcétera. Toda actividad debe declararse en el manifiesto de nuestra app (AndroidManifest.xml), pues, de otro modo, obtendremos un error de compilación. Normalmente, además, crearemos un recurso *layout* que defina los elementos que componen la pantalla, así como su posición dentro de ella. De lo contrario, tendríamos que crear los componentes en tiempo de ejecución, lo que resulta más difícil y tendrá sentido solo en algunos casos particulares.

Como hemos mencionado, el archivo de *layout* definirá, mediante código XML, el aspecto y los componentes de la interfaz gráfica de la actividad. Existen diferentes tipos de *layouts* predefinidos, que nos ayudarán a ordenar los componentes de diferentes formas según nuestras necesidades. Además, estos contenedores pueden agruparse jerárquicamente unos sobre otros, con la intención de crear interfaces gráficas más complejas. Es conveniente, no obstante, utilizar lo menos posible este mecanismo, pues la eficiencia de la interfaz gráfica puede verse reducida. Veamos algunos de los *layouts* más conocidos:

- ***LinearLayout**:* uno de los más sencillos y, a la vez, más usados. Este contenedor agrupa las vistas en su interior de forma lineal, en dirección horizontal o vertical, dependiendo del parámetro *orientation*. Veamos un ejemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:paddingLeft="16dp"
  android:paddingRight="16dp"
  android:orientation="vertical" >
  <EditText
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:hint="EditText 1" />
  <EditText
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:hint="EditText 2" />
  <EditText
     android:layout_width="match_parent"
     android:layout_height="0dp"
     android:layout_weight="1"
     android:gravity="top"
     android:hint="EditText 3" />
  <Button
     android:layout_width="100dp"
     android:layout_height="wrap_content"
     android:layout_gravity="right"
     android:text="Button 1" />
</LinearLayout>
```

El elemento raíz es un *LinearLayout*, y la etiqueta *orientation* indica que los elementos se ordenarán verticalmente. Observemos cómo todos los elementos presentan los atributos *layout_width* y *layout_height*. Estos atributos son obligatorios en todos los tipos de *layout*, e indican el ancho y el alto de cada componente, es decir, cuánto espacio ocupa cada ventana dentro del contenedor. Estos valores pueden ser valores fijos, como 16px (para píxeles), 16dp (siendo dp una medida estándar, un píxel virtual que se adaptará a cada densidad de pantalla), etcétera. Estos valores pueden ser también valores clave, como *match_parent* (utiliza todo el espacio que ofrece el contenedor padre) o *wrap_content* (utiliza solo el espacio que necesite el contenido de la ventana, por ejemplo, un botón ocupará lo que ocupe su título, y no más). Observemos que en el último *EditText* se ha utilizado *layout_height=0dp;* esto no significa que tenga altura cero, sino que pretendemos que la vista se extienda todo el espacio sobrante que han dejado el resto de componentes. En la pestaña *Design* podremos ver el resultado del *layout* que acabamos de codificar:



- ***RelativeLayout**:* este contenedor agrupa a las vistas en una posición relativa al *layout* contenedor o entre las distintas vistas que agrupa.
- ***TableLayout**:* un contenedor tipo tabla que utiliza filas y columnas para alinear los elementos en la vista. Las columnas y filas pueden tener un tamaño particular o ampliarse, y las celdas pueden extenderse más allá de sus límites.
- ***GridLayout**:* muy parecido al *TableLayout*, en el que el diseño se basa en una rejilla de celdas de diferentes tamaños.
- ***ConstraintLayout**:* el *layout* más utilizado actualmente. Antes de la aparición del *ConstraintLayout*, construir una interfaz gráfica compleja requería de la composición jerárquica de varios, a veces demasiados, *layouts* unos dentro de otros para obtener el orden y la colocación precisos de las ventanas en su interior. El problema de ese tipo de anidamiento es que llega un punto en el que la eficiencia del *layout* comienza a degradarse, la presentación de la interfaz gráfica se ralentiza y el consumo de memoria se dispara. Por ello, los desarrolladores de Android crearon el *ConstraintLayout*. Gracias a la versatilidad de este contenedor, raramente será necesario encadenarlo a otro contenedor, pues permite ordenar los componentes de forma eficiente y elástica. Tras su aparición, los demás *layouts* han quedado obsoletos. Además, este *layout* permite el diseño de forma gráfica mediante cajitas con enlaces laterales que pueden asociarse con el contenedor padre o entre las vistas. Ese diseño se traduce en código con los atributos de control de posición:
  - ***layout_constraintLeft_toLeftOf* o *toRightOf**:* la vista se alineará a la izquierda de la parte izquierda o derecha del componente que se indique. El elemento

al que hace referencia se indicará mediante el *id* o con la palabra clave *parent*, que indica que la posición es relativa al contenedor padre.

- ***layout_constraintRight_toLeftOf* o *toRightOf*:** la vista se alineará a la derecha de la parte izquierda o derecha del componente que se indique.
- ***layout_constraintTop_toTopOf* o *toBottomOf*:** la vista se alineará en lo alto de la parte superior o inferior del componente que se indique.
- ***layout_constraintBottom_toTopOf* o *toBottomOf*:** la vista se alineará debajo de la parte superior o inferior del componente que se indique.

Tiene otras opciones para establecer márgenes, centrado, *offsets*, etcétera. Lo mejor para comprobar todas las posibilidades de este *layout* es jugar con la herramienta *Design*, cambiando las configuraciones de las vistas y observando cómo cada parámetro afecta al diseño final de la interfaz gráfica.

Más sobre el eficiente y versátil ConstraintLayout:

https://developer.android.com/reference/androidx/constraintlayout/widget/Constr

Aprendamos más sobre los layouts de Android en la web oficial:

https://developer.android.com/guide/topics/ui/declaring-layout

Dentro de cada contenedor *layout* podremos colocar nuestras vistas, que son los componentes visuales que el usuario puede ver y con los que puede interactuar. Ya hemos visto algunos, pero hagamos una lista de los más usados:

- ***TextView*:** representa un texto estático para el usuario, aunque podemos cambiar su texto desde código. Puede servirnos para presentar información al usuario, como mensajes o etiquetas de otros componentes.
- ***EditText*:** representa un campo de texto que el usuario puede rellenar. Gracias a sus atributos, podemos crear un filtro de entrada, de modo que si, por ejemplo, solo queremos números, el campo solo acepte números, etcétera.
- ***Button*:** un componente que el usuario puede presionar para obtener alguna función. Estableceremos un *listener* o el atributo *onClick* para definir las acciones que efectuar cuando sea pulsado.
- ***ImageButton*:** un botón exactamente igual al anterior, con la diferencia de que podemos añadirle un icono para un mejor aspecto visual de su función.

- ***ImageView**:* representa una imagen estática, aunque podemos cambiarla dinámicamente mediante el código. Puede servirnos para mejorar el aspecto de la interfaz gráfica o para presentar una imagen real relacionada con datos.
- ***RecyclerView**:* una lista de objetos cuya presentación podemos adaptar a nuestras necesidades mediante un *adapter*. Este tipo de listas son muy eficientes, pues solo retienen en memoria los objetos que serán mostrados.
- ***Switch**:* representa un interruptor, puede estar en *on* o en *off*. Puede servirnos para presentar al usuario dos opciones posibles de las que debe escoger una.
- ***Checkbox**:* representa una opción seleccionable. Sería parecido al anterior, mostrando si una opción está activa o no.
- ***RadioButton* + *RadioGroup**:* similar al anterior, con la diferencia de que estas opciones son excluyentes: si eliges una opción, las demás se desactivarán.

Veamos un ejemplo práctico para entender mejor estos conceptos. Crearemos un nuevo proyecto con la opción *File > New > New project > Empty Activity*. Android Studio generará el esqueleto de una *app* lista para ejecutar, con una sola actividad llamada *MainActivity*. Si abrimos el archivo *MainActivity*, veremos algo como:

```
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

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

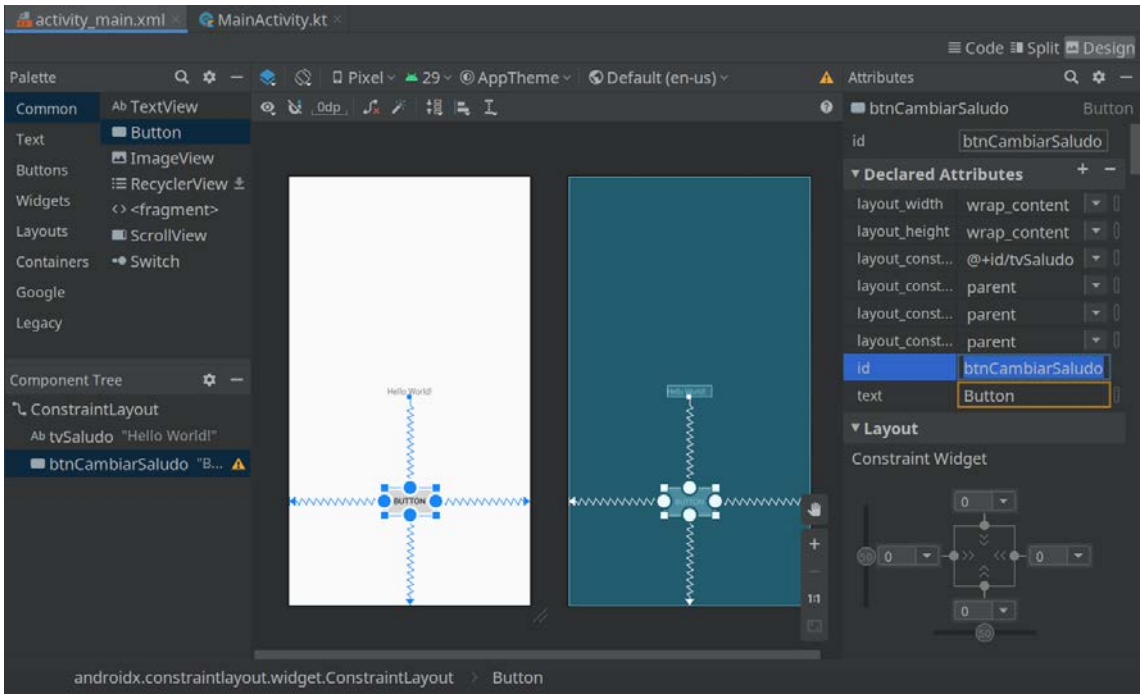
Como observamos en el código autogenerado, nuestra actividad *MainActivity* hereda de *AppCompatActivity* de modo que podamos aprovechar las funcionalidades básicas sin más código. Sí nos veremos obligados a sobrescribir la función *onCreate* para especificar el *layout* que deseamos para nuestra ventana. Si pulsamos la tecla *Ctrl* mientras hacemos clic sobre *activity_main*, navegaremos al editor de *layouts*, que nos mostrará el diseños de la ventana *MainActivity:*

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

Los *layouts* se definen con código XML, pero no debemos alarmarnos antes de tiempo, pues no tendremos que recordar todos y cada uno de los parámetros. Para facilitarnos el trabajo, el editor de *layouts* dispone de capacidades gráficas con las que podremos añadir, eliminar y distribuir los componentes de nuestra interfaz gráfica sin necesidad de escribir código XML. Sí es cierto que, en ocasiones, nos será más sencillo modificar directamente el código, por eso es conveniente ir comprendiendo cada etiqueta XML. Por ejemplo, la segunda línea indica que el *layout* será un *ConstraintLayout*, que es uno de los más utilizados actualmente. Dentro del *layout* encontramos un *TextView*, que no es más que un texto que el usuario verá pero no podrá editar.

Hagamos una prueba: hagamos clic tras *TextView*, pulsemos *Intro* y añadamos la línea: *android:id="@+id/tvSaludo"*. Hemos establecido un indicador para el componente de modo que podamos acceder a él desde nuestro código. Ahora observemos que en la esquina superior derecha del editor existen tres opciones: *Code*, *Split* y *Design*. La opción *Code* muestra el código XML del *layout; Split* muestra tanto el código como el diseño visual; y *Design* muestra únicamente el diseño gráfico de la ventana.

Seleccionemos *Design*. Aparecerá un *mockup* de cómo se verá nuestra actividad. En el lateral izquierdo tenemos un listado de componentes gráficos, que podemos arrastrar hasta la ventana de diseño, para añadirlos a nuestro *layout*. En el lateral derecho podemos ver los atributos del componente actualmente seleccionado. Añadamos un componente *Button* a nuestro *layout:* para disponerlo en un *ConstraintLayout*, tendremos que hacer clic en cada uno de los circulitos y arrastrar hasta los bordes de la ventana para dejarlo anclado. La bolita superior podemos anclarla a la parte baja del *TextView* que ya teníamos:



Recuerda establecer un *id* en el botón para que podamos acceder a él desde el código.

Volvamos al editor de código, tendremos algo como:

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

  <TextView
    android:id="@+id/tvSaludo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <Button
    android:id="@+id/btnCambiarSaludo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Button"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/tvSaludo" />

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

El editor visual ha generado el código para nuestro botón por sí solo. Volvamos al código de la actividad, y añadamos el código:

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
     btnCambiarSaludo.setOnClickListener {
        tvSaludo.text = "Bienvenidos a Android, programadores!!"
     }
  }
}
```

Ahora podemos ejecutar la *app* en un terminal o emulador y pulsar el botón. Veremos que el campo de texto cambia su contenido original por nuestra frase.

Quizá lo más importante de las clases del SDK de Android, y no solo hablamos de las actividades, es el ciclo de vida. Ya vimos un esquema sobre el ciclo de vida de las actividades en el tema anterior, pero merece la pena recalcarlo, pues muchos errores de programación Android son debidos a un descuido en el ciclo de vida tanto de actividades como de fragmentos, servicios y otros componentes del sistema operativo. Estos componentes tienen lo que se denomina un contexto, que es una especie de identificador del recurso. De hecho, una clase *Activity* hereda de la clase *Context* a través de una larga cadena de subclases. Vamos a comprobar cómo un descuido con los contextos puede ser un gran problema. Añadamos un código que se ejecutará siempre que la actividad vuelva a ser visible, sobrescribiendo la función *onResume:*

```
  override fun onResume() {
     super.onResume()
     tvSaludo.text = "Ahora el contexto es: $this"
     Log.e("MainActivity", "Ahora el contexto es: $this")
  }
```

Este código no hace más que cambiar el texto del *TextView* y escribir un mensaje de error en el Logcat. Lancemos la aplicación al terminal. Ahora rotemos el terminal. Observaremos que el identificador ha cambiado. Esto se debe a que, cada vez que rotamos el móvil, el giroscopio o acelerómetro integrado en el terminal avisarán al sistema operativo de que la orientación de la pantalla ha cambiado. El sistema preguntará a la *app* cómo desea adaptarse al cambio de formato de *portrait* a *landscape*, o viceversa. Por defecto, la actividad será destruida y recreada con la nueva configuración de pantalla. Lo mismo ocurriría con otros cambios del sistema, podemos adaptarnos a ellos o dejar que el sistema operativo destruya y vuelva a construir la actividad.

Esto no ocurre cuando llevamos la aplicación a *background* y la volvemos a activar. Por ejemplo, probemos a pulsar el botón de *home* del terminal. Nos aparece el escritorio. Si volvemos a abrir la *app*, el *context* seguirá siendo el mismo, aunque veremos otra línea en el Logcat, porque al activarse se llama a *onResume*. ¿Qué ocurriría si almacenásemos el contexto en una variable e intentásemos utilizarla más tarde sin tener en cuenta el ciclo de vida de la actividad? Pues que estaríamos utilizando un contexto desfasado:

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
     if(contexto == null)
        contexto = this
  }
  override fun onResume() {
     super.onResume()
     tvSaludo.text = "Son iguales los contextos: ${this == contexto}"
     Log.e("MainActivity", "Son iguales los contextos: ${this == contexto}")
  }
  companion object {
     private var contexto: Context? = null
  }
}
```

Si ejecutamos este código y rotamos el terminal, veremos que el contexto ya no es el mismo que era antes de la rotación. Una vez más, debemos tener cuidado y respetar el ciclo de vida de los componentes del sistema Android.

Lo más probable, sobre todo si no utilizamos *fragments*, es que nuestra *app* esté compuesta de más de una *activity*. Para navegar de una a otra, podemos utilizar un objeto *intent*. Veámoslo con un ejemplo: vamos a crear una nueva actividad en nuestra *app*. En el menú de Android Studio, seleccionamos *File > New > Activity > Empty Activity* y aceptamos las opciones por defecto de la nueva actividad. El *wizard* añadirá no solo una clase *MainActivity2*, sino también su *layout* y la línea en el manifiesto. Para navegar de nuestra actividad a la nueva, modificaremos el código de este modo:

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
     btnCambiarSaludo.setOnClickListener {
        val intent = Intent(this, MainActivity2::class.java)
        startActivity(intent)
     }
  }
}
```

Creamos un objeto *intent:* el primer parámetro es el contexto del paquete, pero valdrá con el contexto de nuestra actividad. El segundo parámetro es la clase de la *activity* a la que deseamos navegar. Una vez creado el *intent*, llamaremos a la función de la *startActivity* con él y la nueva actividad se mostrará sobre la actual. Decimos "sobre" porque el sistema Android guardará una pila de vistas una sobre otra según vayamos abriéndolas, lo que se conoce como *back stack*. Cuando cerremos una ventana, presionando *Atrás*, por ejemplo, se mostrará la anterior, la que está debajo en la pila.

Cuando llamamos a otra *activity*, tanto dentro como fuera de nuestra *app*, podemos leer la información que devuelve. Antes de AndroidX, el código de la primera actividad sería algo como:

```
btnLanzarSegundaActividad.setOnClickListener {
  val intent = Intent(this, SecondActivity::class.java)
  startActivityForResult(intent, REQUEST_SEGUNDA_ACTIVIDAD)
}
```

Vemos que, en lugar de utilizar *startActivity*, llamamos a *startActivityForResult*. Utilizamos un identificador *REQUEST_SEGUNDA_ACTIVIDAD*, que será un código entero para identificar el resultado de la actividad que hemos llamado en cada caso. La nueva actividad se mostrará, el usuario utilizará sus funciones y, cuando termine, la actividad se cerrará y devolverá los datos. En la primera *activity*, tendríamos un código para recibir esos datos, algo como:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
  super.onActivityResult(requestCode, resultCode, data)
  if(resultCode == Activity.RESULT_OK && data != null) {
     when(requestCode) {
        REQUEST_SEGUNDA_ACTIVIDAD -> {
           val datos = data.getStringExtra("datos")
        }
     }
  }
}
```

Sobrescribimos el método de la *activity* *onActivityResult* para obtener los datos devueltos por la *activity* a la que llamamos con *startActivityForResult*. Este método nos informa sobre el resultado mediante sus tres parámetros:

- ***requestCode**:* es el identificador que utilizamos al llamar a la actividad, de modo que, si hay más de una llamada a actividades diferentes, podamos identificar cuál de ellas nos está respondiendo.
- ***resultCode**:* normalmente será una de los valores *RESULT_OK* (para indicar que la actividad terminó adecuadamente y pueden utilizarse los valores devueltos) y *RESULT_CANCELED* (si el usuario canceló el proceso y debemos tomar las medidas adecuadas ante el rechazo del proceso).
- ***data**:* un objeto *intent* con los datos requeridos por la *activity* que lanzamos. Podremos obtener los datos mediante diferentes métodos en función al tipo de datos, como *getStringExtra*, *getIntExtra*, *getBooleanExtra*, *getByteExtra*, etcétera.

Imaginemos que el *layout* de la segunda actividad es algo como:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".SecondActivity">
  <Button
     android:id="@+id/btnVolver"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Volver"
     app:layout_constraintBottom_toBottomOf="parent"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toBottomOf="@+id/lblMensaje" />
  <TextView
     android:id="@+id/lblMensaje"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Esta es la segunda actividad"
     app:layout_constraintBottom_toBottomOf="parent"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Simplemente, un botón con *id btnVolver* y un *TextView* con un texto estático. El código de la segunda actividad responsable de devolver los datos sería:

```
import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_second.*

class SecondActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_second)
     btnVolver.setOnClickListener {
        onClick(it)
     }
  }
  fun onClick(view: android.view.View) {
     val intent = Intent()
     val datos = "Estos son datos generados en ${SecondActivity::class.simpleName}"
     intent.putExtra("datos", datos)
     setResult(RESULT_OK, intent)
     finish()
  }
}
```

Hemos creado una actividad *SecondActivity* y sobrescrito el método *onCreate*. Cuando se nombre a la actividad, se creará y se llamará a *onCreate*. Aquí establecemos un *listener* para responder a la pulsación del botón *btnVolver*. Podemos acceder directamente al botón gracias al sistema de *data binding* de AndroidX. Al declarar *import kotlinx.android.synthetic.main.activity_second.** estamos importando todos los campos del *layout* en nuestro código, listos para ser utilizados. Antes de AndroidX, podíamos acceder al control mediante:

```
val btnVolver = findViewById<Button>(R.id.btnVolver)
```

Como hemos indicado, con la función *setOnclickListener* establecemos el código que será ejecutado cuando se presione el botón. Otra opción habría sido configurar directamente en el *layout* el *listener* para el botón:

```
<Button
  android:id="@+id/btnVolver"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:onClick="onClick"
  android:text="Volver"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toBottomOf="@+id/lblMensaje" />
```

Observemos ahora el código de *onClick*. Este método crea un *intent* y, mediante la función *putExtra*, introduce en él los datos que devolver. La llamada a *setResult* devolverá el *intent*, pasando además el *flag RESULT_OK* para indicar que el proceso se llevó a cabo satisfactoriamente. No queda más que terminar el proceso llamando a *finish*, que cerrará la actividad y volverá a la anterior.

Android maneja las pilas de actividades asociadas a lo que se conoce como tareas. Una tarea es un conjunto de actividades con las que interactúa el usuario mientras realiza una función. Normalmente, una aplicación tendrá una tarea con una pila de actividades, aunque no tiene por qué ser siempre así.

El orden de las actividades en la pila no puede modificarse una vez introducidas. Sin embargo, sí puede configurarse de antemano, mediante el uso de parámetros en la etiqueta *<activity>* del *manifest* o mediante parámetros en la función *startActivity*. Los parámetros de la función tendrán prioridad sobre los del *manifest* en caso de que existan ambos y sean incompatibles.

En el *manifest*, tenemos la opción de especificar el atributo *launchMode*, que especifica cómo la actividad encajará en la tarea, por ejemplo:

<activity android:name=".MainActivity" android:launchMode="standard"/>

Hay cuatro modos diferentes, veamos qué hace cada uno:

- ***standard**:* es el modo por defecto. El sistema creará una nueva instancia de la actividad en la tarea. La actividad podrá crearse más de una vez, y cada instancia puede pertenecer a una tarea diferente.
- ***singleTop**:* si ya existe una instancia de la actividad en la tarea actual, el sistema enrutará el *intent* a esa instancia. Como la actividad ya existe, no se llamará a *onCreate*, sino a *onNewIntent*.
- ***singleTask**:* el sistema crea una nueva tarea, e instancia la actividad como la actividad principal de la tarea. Sin embargo, si ya existe una instancia de la

actividad en una tarea separada, el sistema enrutará el *intent* a la actividad existente mediante *onNewIntent*.

- ***singleInstance**:* Sería lo mismo que *singleTask*, excepto que el sistema no lanzará ninguna otra actividad en la tarea que tiene la instancia. La instancia de la actividad será siempre la única que tenga la tarea. Cualquier actividad que se lance de este modo creará una nueva tarea.

Los parámetros que podemos utilizar en la función *onStartActivity* serán:

- ***FLAG_ACTIVITY_NEW_TASK**:* comienza la actividad en una nueva tarea. Si ya existe una tarea con esa actividad, la tarea se traerá al primer plano y la actividad ya existente recibirá la llamada de la función *onNewIntent*. El resultado sería el mismo que utilizar *singleTask* en el *manifest*.
- ***FLAG_ACTIVITY_SINGLE_TOP**:* si la actividad que se quiere lanzar es la actividad actual, la situada en lo alto de la pila, recibirá una llamada a *onNewIntent* en lugar de crearse una nueva. Sería lo mismo que usar *singleTop* en el *manifest*.
- ***FLAG_ACTIVITY_CLEAR_TOP**:* si la actividad que quiere lanzarse está ya en la tarea actual, en lugar de lanzarse una nueva instancia, todas las demás actividades que están sobre ella en la pila se destruirán y la actividad que ya existía, ahora en primer plano, recibirá la llamada a *onNewIntent*. Normalmente, este *flag* se utiliza junto con ***FLAG_ACTIVITY_NEW_TASK***.

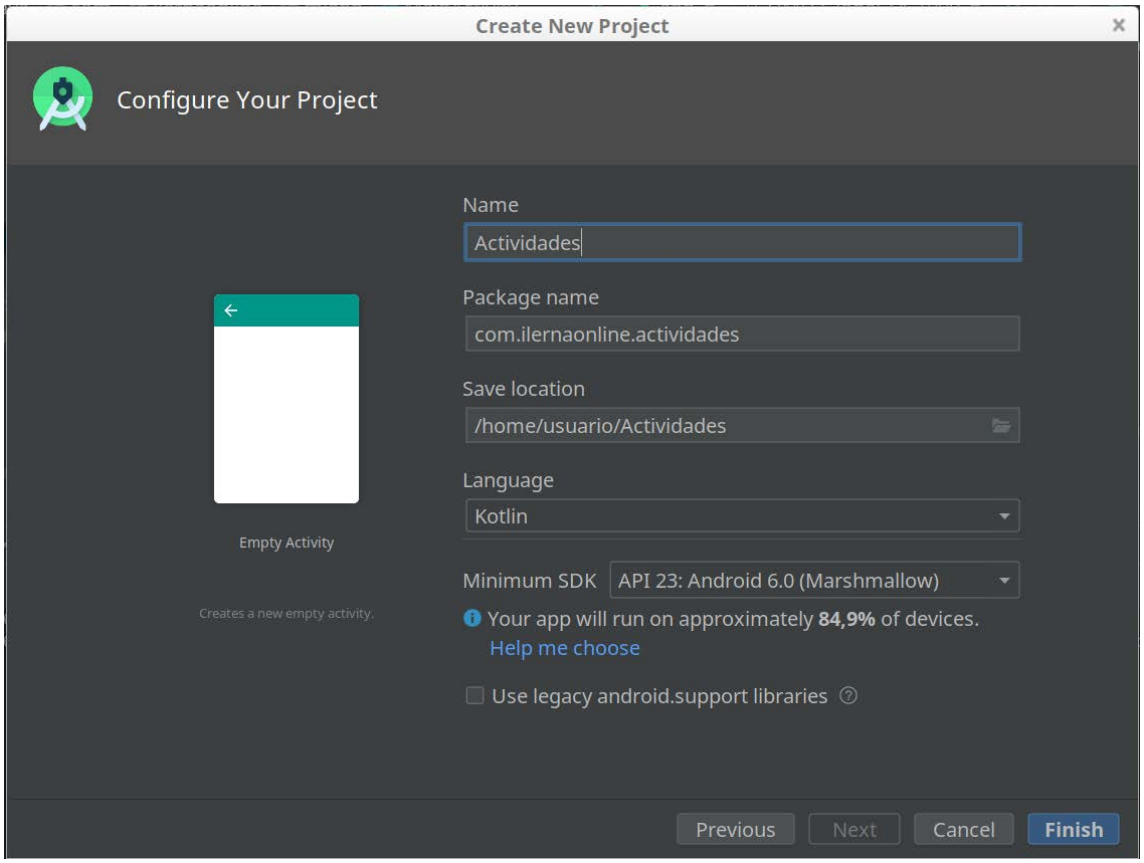
Documentación oficial de la clase Activity:

https://developer.android.com/reference/android/app/Activity

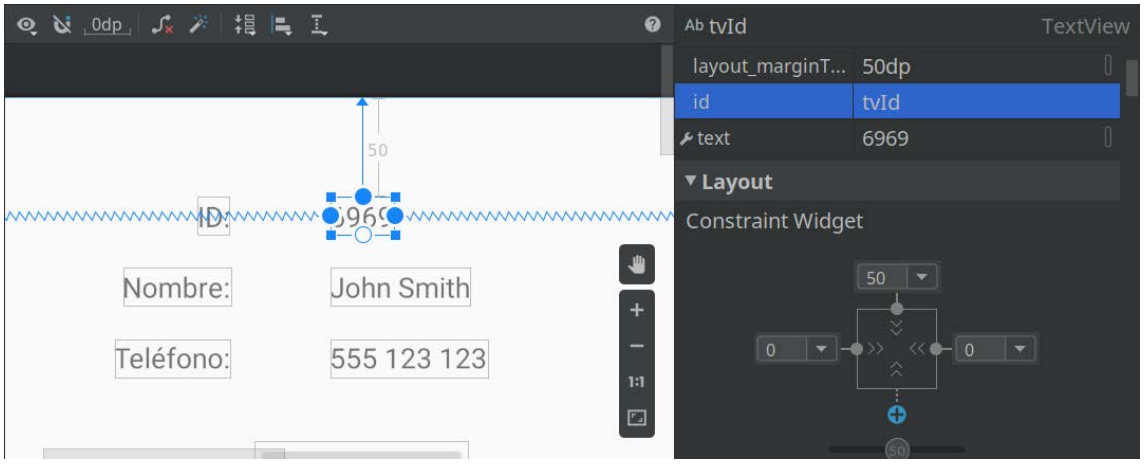
Documentación extendida sobre las tareas y la pila de actividades:

https://developer.android.com/guide/components/activities/tasks-and-back-stack

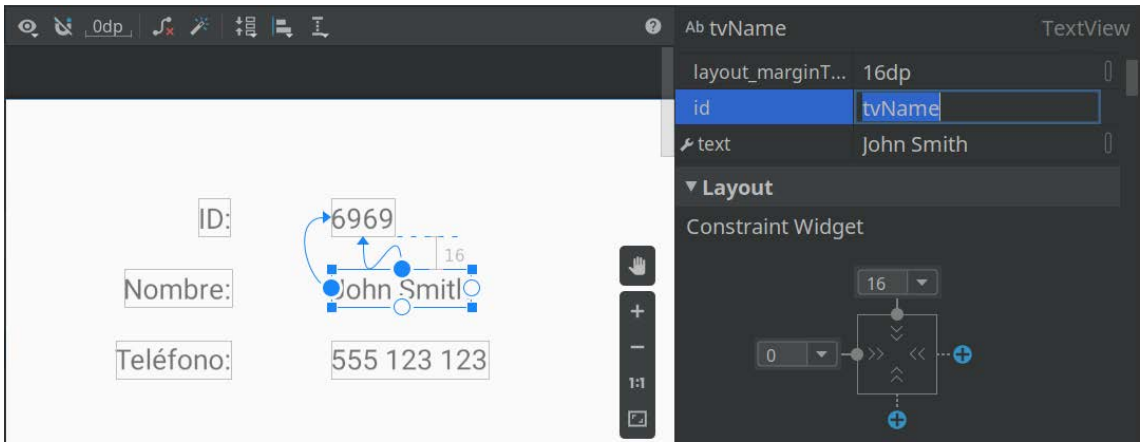
Veamos un ejemplo completo que utiliza actividades propias y externas de la *app*. Crearemos un nuevo proyecto mediante el menú de Android Studio: *File > New > New Project… > Empty Activity > Next > Finish*.



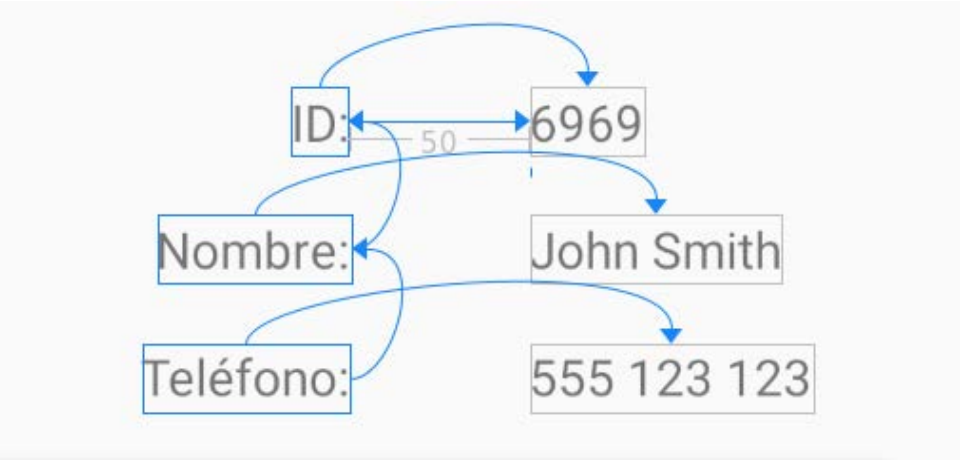
Android Studio creará el proyecto, que tiene una clase *activity* llamada *MainActivity* con un *layout* llamado *activity_main*. Vamos a editar el *layout* para que tenga todos los campos que necesitamos. Primero creamos seis campos *TextView*, unos servirán de etiquetas y otros de valores. Los dispondremos así:



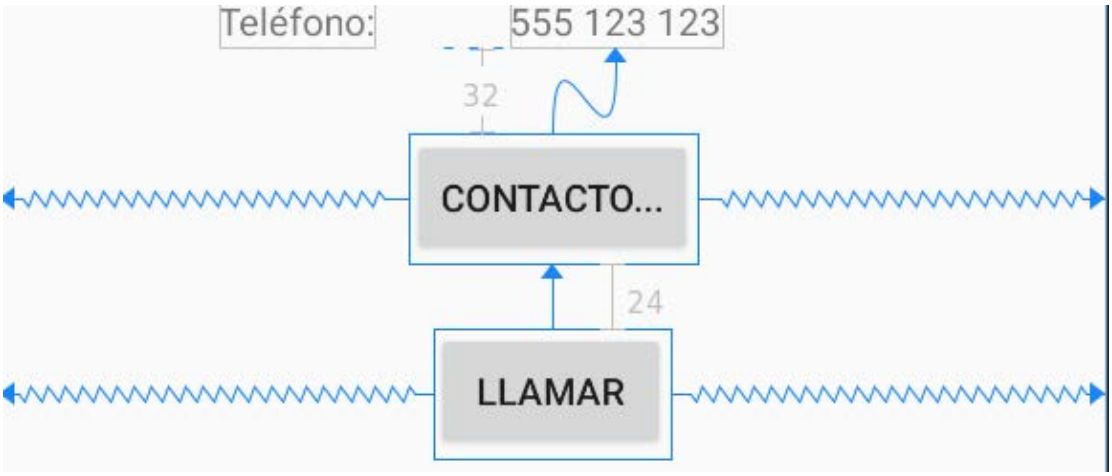
El *layout* más utilizado actualmente es *ConstraintLayout*, que es el que nos ha dejado el *wizard* de Android Studio. En este *layout* podemos colocar los componentes simplemente arrastrando las bolitas a los bordes donde queremos que queden anclados.



Para ordenar los componentes, podemos hacer que empiecen donde acaban otros o que empiecen donde empiezan otros. Es cuestión de jugar con los enlaces de la cajita que aparece en *Constraint Widget*.



Luego añadiremos un par de botones, de esta forma:



En ocasiones, es conveniente empezar en el modo gráfico y luego pasar la código. El código del *layout* debe quedar finalmente como mostramos a continuación:

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

```
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity">

<TextView
    android:id="@+id/tvId"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="50dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="6969" />
<TextView
    android:id="@+id/tvName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    app:layout_constraintStart_toStartOf="@+id/tvId"
    app:layout_constraintTop_toBottomOf="@+id/tvId"
    tools:text="John Smith" />
<TextView
    android:id="@+id/tvPhone"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    app:layout_constraintStart_toStartOf="@+id/tvName"
    app:layout_constraintTop_toBottomOf="@+id/tvName"
    tools:text="555 123 123" />

<TextView
    android:id="@+id/lblId"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="50dp"
    android:text="ID:"
    app:layout_constraintEnd_toStartOf="@+id/tvId"
    app:layout_constraintTop_toTopOf="@+id/tvId" />
<TextView
    android:id="@+id/lblName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Nombre:"
    app:layout_constraintEnd_toEndOf="@+id/lblId"
    app:layout_constraintTop_toTopOf="@+id/tvName" />
<TextView
    android:id="@+id/lblPhone"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Teléfono:"
    app:layout_constraintEnd_toEndOf="@+id/lblName"
    app:layout_constraintTop_toTopOf="@+id/tvPhone" />


<Button
    android:id="@+id/btnContacto"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="Contacto..."
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/tvPhone" />
<Button
    android:id="@+id/btnLlamar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="24dp"
    android:text="Llamar"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.498"
```

```
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/btnContacto" />

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

Editemos ahora el código de *MainActivity*. Vamos a darles una utilidad a los dos botones que tenemos:

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

    btnContacto.setOnClickListener {
      obtenerContacto()
    }
    btnLlamar.setOnClickListener {
      llamar()
    }
  }
```

El botón *Contacto* llamará a la función *obtenerContacto* y el botón *Llamar* a *llamar*. Veamos qué hace la primera función:

```
private fun obtenerContacto() {
  if(tenemosPermiso()) {
    val intent = Intent(Intent.ACTION_PICK, ContactsContract.Contacts.CONTENT_URI)
    startActivityForResult(intent, REQUEST_CONTACTO)
  }
  else {
    pidePermiso()
  }
}
```

Lo realmente importante aquí es que creamos un *intent* de forma diferente a como hemos visto hasta ahora, y es porque estamos accediendo a una *activity* que está fuera de nuestra *app*. Como vemos por los parámetros, queremos acceder a la lista de contactos que guarda el terminal en la *app* de *Contactos*. Luego llamamos a *startActivityForResult* para que nos devuelva el contacto seleccionado por el usuario. ¿Para qué sirven las demás líneas? Para pasar los controles de seguridad de Android. Los contactos de nuestro teléfono son datos personales, y si cualquier *app* pudiese acceder a ellos sin pedirle permiso al usuario, estaríamos violando su privacidad. De modo que, para acceder a cierta información como los contactos o dispositivos como la cámara, el sistema se asegurará primero de que el usuario nos da su consentimiento. Algunos permisos son menos importantes que otros. Los que no son tan importantes basta con pedirlos en el *manifest* para que, al instalar, el usuario sepa que la *app* accederá a esos datos. Los permisos más importantes deberán especificarse tanto en el *manifest* como a la hora de ejecutar la *app*. En nuestro caso, los contactos son considerados muy importantes, por lo que tendremos que pedir permisos tanto en el *manifest:*

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.ilernaonline.actividades">

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

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher" ...
```

Como a la hora de ejecutar la aplicación, mediante las siguientes funciones:

```
private fun tenemosPermiso(): Boolean
      = PermissionChecker.checkSelfPermission(this, PERMISO) ==
                              PermissionChecker.PERMISSION_GRANTED

private fun pidePermiso() {
  if( ! tenemosPermiso()) {
     requestPermissions(arrayOf(PERMISO), REQUEST_PERMISOS)
  }
}

override fun onRequestPermissionsResult(
                                    requestCode: Int,
                                    permissions: Array<out String>,
                                    grantResults: IntArray) {
  super.onRequestPermissionsResult(requestCode, permissions, grantResults)
  if(requestCode == REQUEST_PERMISOS
          && permissions[0] == PERMISO
          && grantResults[0] != PermissionChecker.PERMISSION_GRANTED) {
     mensajeErrorPermiso()
  }
}

companion object {
  private const val TAG = "MainActivity"
  private const val PERMISO = Manifest.permission.READ_CONTACTS
  private const val REQUEST_PERMISOS = 1
  private const val REQUEST_CONTACTO = 1
  private const val REQUEST_SEGUNDA_ACTIVIDAD = 2
}
```

La función *tenemosPermiso* utiliza la función del sistema *checkSelfPermission* para saber si el usuario ha dado ya el permiso a esta *app*. La función *pidePermiso* utiliza la función del sistema *requestPermissions* para requerir al usuario que acepte una lista de permisos, en nuestro caso, solo uno, el de lectura de contactos. Pero el sistema no responde a esto de forma inmediata, sino que presenta al usuario un diálogo por cada permiso para que lo acepte o lo rechace. La respuesta nos llegará en otra función de la *activity* que debemos sobrescribir: *onRequestPermissionsResult*. En esta función comprobamos que es la respuesta que esperábamos y que el usuario finalmente aceptó darnos permiso. Si nos denegó el permiso, mostramos un mensaje de alerta.

Parece mucho trabajo, pero en realidad el sistema nos ayuda bastante a mantener la seguridad del usuario. Si tratásemos de saltarnos este procedimiento, la llamada que hicimos a *startActivityForResult* lanzaría una excepción que haría terminar la *app*.

Bien, ya hemos pedido permiso y hemos lanzado la *activity*. El usuario verá la actividad principal de la *app* de contactos, con una lista de la que podrá seleccionar uno de ellos. Cuando seleccione un contacto, la *app* de contactos se cerrará y volveremos a la nuestra. De hecho, cuando lo haga, se llamará a la función *onActivityResult* de nuestra actividad:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
  super.onActivityResult(requestCode, resultCode, data)
  if(resultCode == Activity.RESULT_OK && data != null) {
     when(requestCode) {
        REQUEST_CONTACTO -> {
           procesarContacto(data)
        }
     }
  }
```

```
}
```

Al sobrescribir esta función, recibiremos la información del contacto que solicitamos, aunque primero debemos comprobar que la actividad acabó de forma correcta si *resultCode == RESULT_OK* y que nos devolvió datos data *!=null*. En caso de que todo esté bien, procesamos los datos recibidos en la función:

```
private fun procesarContacto(data: Intent) {
    val contactData: Uri? = data.data
    val projection = arrayOf(
        ContactsContract.Contacts._ID,
        ContactsContract.Contacts.DISPLAY_NAME
    )
    val cursor: Cursor? = contentResolver.query(contactData!!, projection,"",null,null)
    cursor?.let { c ->
        c.moveToFirst()
        val id = c.getString(c.getColumnIndex(ContactsContract.Contacts._ID))
        val name = c.getString(c.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME))
        tvId.text = id
        tvName.text = name
        val phones: Cursor? = contentResolver.query(
            ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,
            ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = " + id,
            null, null
        )
        phones?.moveToFirst()
        val number = phones?.getString(phones.getColumnIndex("data1"))
        tvPhone.text = number
        phones?.close()
    }
    cursor?.close()
}
```

Este código es algo complejo, pero no hay que asustarse, es la forma que tiene Android de cedernos los datos que tiene en el sistema, y es muy similar en todos los casos. Si un día dudamos, no tenemos más que consultarlo en la documentación o en Google. Básicamente, el *intent* que nos devolvió la actividad de contactos tiene una URL hacia una base de datos del sistema. Mediante esa URL, creamos un cursor para ir recuperando los datos uno por uno. Una vez recuperados los datos, cerramos el cursor y mostramos los datos.

La documentación oficial indica cómo utilizar el editor de layouts:

https://developer.android.com/studio/write/layout-editor

2.4.2. Fragment

Aunque no necesariamente, las actividades suelen entenderse como una pantalla completa en la que distribuir elementos gráficos como imágenes, botones y demás controles. Los fragmentos llegaron para dar mayor flexibilidad a la composición de las pantallas. Pueden ocupar toda o solo una parte de ella. El *fragment* podría entenderse como una unidad de presentación, como una subactividad. En sí mismo, sería como un

componente gráfico más. Imaginemos un *fragment* llamado *DetallesClienteFragment* que muestre los datos de un cliente. Podríamos utilizar ese *fragment*, solo o en composición con otros, en varias actividades de la aplicación, sin ninguna repetición de código. El paso de un *fragment* a otro es más eficiente que el paso entre actividades y, además, no es necesario declarar los fragmentos en el *manifest*. Los *fragments*, como las *activities*, tienen un ciclo de vida que debemos comprender y respetar si queremos que todo funcione como deseamos. Además, un *fragment* siempre irá embebido en una *activity*, que será la responsable de dirigir sus *fragments*.

Veamos un ejemplo de *app* con *fragments*. En realidad, no tendremos que programar una sola línea, gracias a la potencia de Android Studio y sus proyectos predefinidos. Selecciona *File > New > New Project > Basic Activity*. Se creará una *app* con una *activity* que servirá de contenedor para dos *fragments* diferentes. Además, veremos cómo hace uso del relativamente reciente componente *Navigation*, que nos facilita la navegación entre *fragments*.

Quizá lo primero es ver cómo luce, así que lancemos la *app* en un emulador o terminal y juguemos con ella. Estudiando el código, veremos cómo Android Studio ha creado tres clases: *MainActivity*, *FirstFragment* y *SecondFragment*. El código de *MainActivity* no tiene nada de particular, nada que indique cómo se mostrarán los *fragments*. Pero pulsemos *Ctrl* y hagamos clic sobre *activity_main* para ver el *layout* de la actividad. Vemos el *CoordinatorLayout* y la barra de herramientas *Toolbar*. Más abajo veremos un *include* hacia otro *layout*, que será donde se defina el contenido de la ventana. Mientras presionamos *Ctrl*, hacemos clic sobre *@layout/content_main*. Veremos el código XML:

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

  <fragment
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:navGraph="@navigation/nav_graph" />

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

Esto sí es interesante: la etiqueta *fragment* servirá como contenedor de los *fragments* que queremos mostrar. El atributo *name* referencia a la clase *NavHostFragment*, que tendrá la funcionalidad de permitir la navegación entre *fragments*. Cada *NavHostFragment* tiene un *NavController* que define los posibles caminos de navegación y que se define mediante el gráfico de navegación. En el código podemos

ver que el gráfico de navegación se define con el atributo *app:navGraph* y que, en este caso, su valor es *@navigation/nav_graph*. Pulsemos *Ctrl* y clic sobre *nav_graph* para ver el gráfico de navegación:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation 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:id="@+id/nav_graph"
  app:startDestination="@id/FirstFragment">

  <fragment
    android:id="@+id/FirstFragment"
    android:name="com.ilernaonline.miapp.FirstFragment"
    android:label="@string/first_fragment_label"
    tools:layout="@layout/fragment_first">
    <action
      android:id="@+id/action_FirstFragment_to_SecondFragment"
      app:destination="@id/SecondFragment" />
  </fragment>

  <fragment
    android:id="@+id/SecondFragment"
    android:name="com.ilernaonline.miapp.SecondFragment"
    android:label="@string/second_fragment_label"
    tools:layout="@layout/fragment_second">
    <action
      android:id="@+id/action_SecondFragment_to_FirstFragment"
      app:destination="@id/FirstFragment" />
  </fragment>

</navigation>
```

Observamos cómo el grafico tiene un elemento raíz *<navigation>* que define su *id* y establece el *fragment* que será visible al inicio con *app:startDestination*, que en este caso es el *FirstFragment*. Debajo veremos una lista de fragmentos definidos con la etiqueta *<fragment>* con varias propiedades, como la clase definida con *android:name* y el *layout* con *tools:layout*. Una etiqueta hija *<action>* define la acción de navegación de una etiqueta a otra. Igual que con los *layouts*, podemos ver una representación gráfica de la navegación entre *fragments*, pues tenemos las opciones en la esquina superior derecha: *Code*, *Split* y *Design* que ya conocemos de los puntos anteriores. Veamos ahora el código del primer fragmento, haciendo *Ctrl* y clic sobre *FirstFragment*:

```
...
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController

class FirstFragment : Fragment() {
  override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
  ): View? {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_first, container, false)
  }
  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    view.findViewById<Button>(R.id.button_first).setOnClickListener {
      findNavController().navigate(R.id.action_FirstFragment_to_SecondFragment)
    }
  }
}
```

Vemos que *FirstFragment* hereda de *Fragment* y sobrescribe *onCreateView* y *onViewCreated*. En la primera especifica el *layout* que diseña su contenido; en la segunda, establece un *OnClickListener* para el botón *button_first*. Cuando se pulse sobre el botón, se ejecutará el código *findNavController().navigate(action)*. El *NavController* conoce el gráfico de navegación, y sabe que esa acción requiere navegar hacia el *fragment SecondFragment*. El segundo *fragment* es igual de sencillo: simplemente, hace el recorrido contrario.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
  view.findViewById<Button>(R.id.button_second).setOnClickListener {
     findNavController().navigate(R.id.action_SecondFragment_to_FirstFragment)
  }
}
```

2.4.3. Otros tipos de ventanas

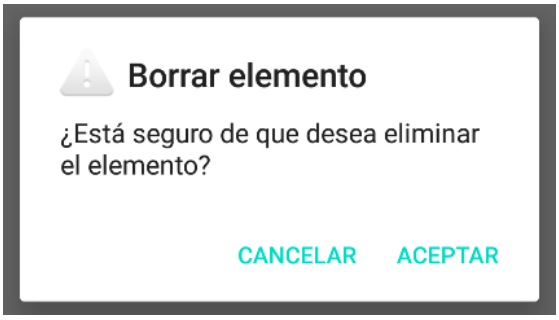
Aparte de *activities* y *fragments*, podemos utilizar los diálogos para ventanas más sencillas. Por norma general, un diálogo no ocupará toda la pantalla, sino que mostrará un mensaje al usuario con un par de botones. Podría mostrar, además, algunos controles que faciliten al usuario la introducción de datos o la elección de alguna preferencia antes de continuar otro proceso. Podríamos crear un diálogo a partir de una *activity:* diseñaríamos el *layout* como hicimos antes, pero utilizando el tema *Theme.Holo.Dialog* en el *manifest: <activity android:theme="@android:style/Theme.Holo.Dialog">*. Entonces, la actividad se mostrará en forma de diálogo en lugar de mostrarse en pantalla completa. Sin embargo, si el diseño del *layout* no es muy complejo, podemos utilizar las subclases de *Dialog* que nos proporciona el SDK, veamos algunas:

- ***AlertDialog**:* esta subclase de *Dialog* muestra un título y varios botones, de forma opcional. Es capaz también de mostrar una lista de elementos de los que el usuario podrá seleccionar. Veamos un ejemplo sencillo: imaginemos que necesitamos asegurarnos de que el usuario quiere borrar un elemento de nuestra *app*. Podemos utilizar un código como:

```
AlertDialog.Builder(context)
    .setTitle("Borrar elemento")
    .setMessage("¿Está seguro de que desea eliminar el elemento?")
    .setPositiveButton(android.R.string.yes) { dialog, which ->
        Log.d("Dialog", "---------------- YES ----------------")
            //TODO: borrar el elemento
    }
        //Si utilizamos null como listener, el diálogo simplemente se cierra
    .setNegativeButton(android.R.string.no, null)
    .setIcon(android.R.drawable.ic_dialog_alert)
    .show()
```

  Como vemos, la clase *AlertDialog* utiliza un patrón de diseño llamado *Builder*. El patrón de diseño *Builder* es un patrón creacional que consigue separar la construcción de un objeto complejo de su representación, de modo que el mismo proceso de construcción pueda ser responsable de crear diferentes

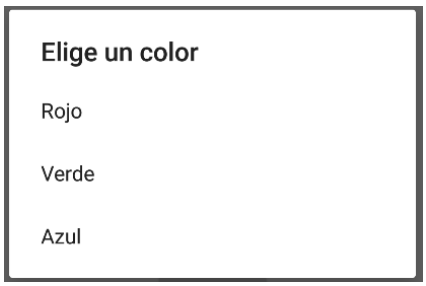
representaciones. Crear el diálogo mediante un *Builder* es sencillo, pues se documenta a sí mismo: *setTitle* y *setMessage* establecen el título y el mensaje que se mostrarán; *setPositiveButton* establece el título del botón positivo y la acción que se llevará a cabo si el usuario acepta; *setNegativeButton* establece el título y la acción si el usuario rechaza la acción; *setIcon* muestra el icono del diálogo; por último, *show* mostrará el diálogo, que paraliza el código hasta que el usuario escoja una opción.



Imaginemos ahora que requerimos al usuario que elija una opción entre varias:

```
val colores = arrayOf("Rojo", "Verde", "Azul")
AlertDialog.Builder(context)
    .setTitle("Elige un color")
    .setItems(colores) { dialog, which ->
        Log.d("Dialog", "Color elegido: ${colores[which]}")
    }
    .create()
    .show()
```

Como vemos, el *Builder* nos facilita la creación de un diálogo diferente, pero siguiendo el mismo patrón que antes. Ahora utilizaremos *setItems* para establecer las opciones disponibles, recibiendo en la variable *which* cuál de las opciones eligió el usuario.



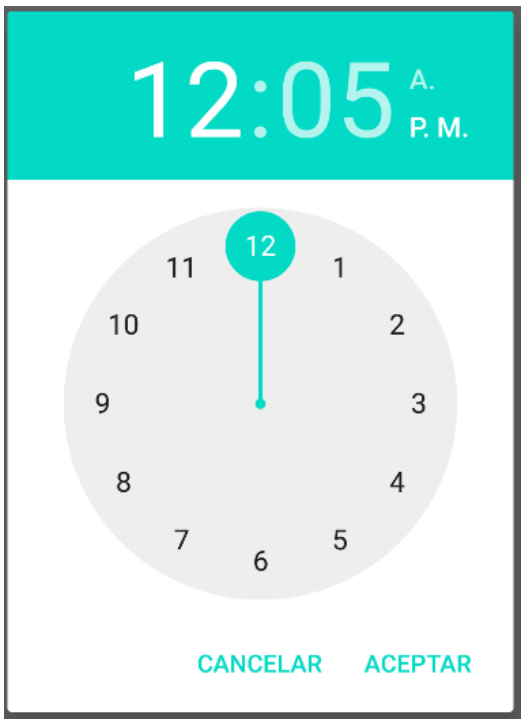
- ***DatePickerDialog* y *TimePickerDialog***: estas dos subclases de *Dialog* han sido especialmente diseñadas para que el usuario seleccione una fecha o una hora, respectivamente. Veamos un ejemplo sencillo:

```
DatePickerDialog(requireContext(),
    OnDateSetListener { view, year, monthOfYear, dayOfMonth ->
        Log.d("Dialog", "Fecha seleccionada: $dayOfMonth/${monthOfYear+1}/$year")
    }, 2020, 7, 16
)
.show()
```

De esta manera, mostramos al usuario un calendario con la fecha 16/08/2020 seleccionada por defecto. Atención a los meses, pues muchas funciones del SDK consideran enero como el mes 0, por lo tanto, el 7 será agosto. Cuando el usuario seleccione la fecha que desee, la obtendremos en las variables *dayOfMonth*, *monthOfYear* y *year*. Para la hora, sería algo similar:

```
TimePickerDialog(requireContext(),
  OnTimeSetListener { view, hourOfDay, minute ->
     Log.d("Dialog", "Hora seleccionada: $hourOfDay:$minute")
  }, 12, 5, false
)
.show()
```

Establecemos la hora que se mostrará por defecto, las 12 horas y 5 minutos; *false* indica que no deseamos que se muestre en formato de 24 horas. Cuando el usuario seleccione la hora que quiere, recibiremos los datos en *hourOfDay* y *minute*. Podemos ver el aspecto que tendrán estos diálogos en un terminal:



Android nos permite muchísimas opciones para diseñar nuestras pantallas y diálogos. Es conveniente consultar la documentación oficial para estar actualizado con todas las opciones.

Para aprender más sobre el uso y diseño de diálogos en Android:

https://developer.android.com/guide/topics/ui/dialogs

2.5. Contexto gráfico. Imágenes

Cualquier *app* puede beneficiarse estéticamente del uso de buenos gráficos que decoren la interfaz gráfica. No solo eso, también puede facilitar al usuario la comprensión de datos complejos mediante la representaciones gráficas interactivas. En gran medida, los usuarios buscan aplicaciones eficientes, al mismo tiempo que fáciles de usar y estéticamente cuidadas. En este punto estudiaremos algunas de las posibilidades gráficas más sencillas y el manejo de imágenes en nuestra *app* Android. En un tema posterior veremos cómo podemos mostrar animaciones y efectos de sonido en nuestra *app*.

2.5.1. Drawables

Ya conocemos la carpeta de recursos **"res"** que los proyectos Android tienen dentro del directorio **"app"**. Dentro de "res" podemos encontrar el directorio **"layout"** que guarda los diseños de nuestras pantallas, ***values***, ***navigation***, ***menu***, etcétera. Encontraremos también la carpeta **"drawable"**. En esa carpeta, el proyecto guardará las imágenes como recursos *drawables* que la *app* utilizará en *layouts*, gráficos, diálogos e iconos. Un recurso *drawable* es una abstracción del SDK para manejar imágenes que puedan ser mostradas en pantalla. Los *drawables* pueden cargarse mediante funciones de la API como *getDrawable* o ser incluidas en otros recursos XML, como *layouts*, con algún atributo como *android:drawable* o *android:icon*. Imaginemos que queremos mostrar una imagen en uno de nuestros *layouts:* podemos utilizar el componente *ImageView*, que es un contenedor de imágenes. El código sería algo como:

```
<ImageView
  android:layout_height="wrap_content"
  android:layout_width="wrap_content"
  android:src="@drawable/imagen" />
```

Si necesitamos modificar la imagen desde el código, primero cargaríamos el recurso *drawable* así:

```
val drawable: Drawable?= ResourcesCompat.getDrawable(resources, R.drawable.imagen, null)
...
imageView.setImageDrawable(drawable)
```

2.5.2. Crear y presentar drawables

Ya hemos visto cómo presentar las imágenes guardadas en el proyecto como recursos *drawables*, pero existen otras opciones, como, por ejemplo, *ShapeDrawable*. La clase *ShapeDrawable* es una subclase de *Drawable*, y es una buena opción cuando pretendemos dibujar un gráfico bidimensional relativamente sencillo. En el código de nuestra *app*, podremos codificar formas básicas, como cuadrados o círculos, utilizando las líneas y colores que deseemos. En los objetos *ShapeDrawable* podemos sobrescribir su método *draw* para personalizar el aspecto de la imagen. Veamos un ejemplo:

```
import android.content.Context
import android.graphics.Canvas
import android.graphics.drawable.ShapeDrawable
import android.graphics.drawable.shapes.OvalShape
import android.util.AttributeSet
import android.view.View

//Constructor que se llamará cuando la vista se llame desde código
class CustomDrawableView(context: Context) : View(context) {

  constructor(context: Context, attributeSet: AttributeSet): this(context) {
     //Constructor que se llamará cuando la vista se utilice en un layout
  }

  private val drawable: ShapeDrawable = run {
     val x = 0
     val y = 0
     val width = 100
     val height = 100
     contentDescription = "Descripción de la imagen!"
     ShapeDrawable(OvalShape()).apply {
        // Si no establecemos el color, será negro por defecto.
        paint.color = 0xff74AC23.toInt()
        // Debemos establecer los límites del gráfico.
        setBounds(x, y, x + width, y + height)
     }
  }
  override fun onDraw(canvas: Canvas) {
     drawable.draw(canvas)
  }
}
```

Hemos creado una subclase de *View* con dos constructores, uno que se llamará cuando se incruste la vista en un *layout* y el otro para ser usado directamente en una *activity* o un *fragment*. Dentro de la clase hemos creado un *ShapeDrawable* con *OvalShape*, que dibujará un óvalo. Como el ancho y el alto son 100, será un círculo. Para personalizar nuestra vista, sobrescribimos el método *onDraw* de *View*. Aquí solo tendremos que llamar a la función *draw* del *drawable*. Veamos cómo podemos utilizar esta vista:

```
class SecondFragment : Fragment() {
  override fun onCreateView(
     inflater: LayoutInflater, container: ViewGroup?,
     savedInstanceState: Bundle?
  ): View? {
        return CustomDrawableView(requireContext())
  }
}
```

En este caso, la vista ocupará todo el *fragment*, pero también podríamos utilizar la vista dentro de un *layout* junto con otros componentes gráficos:

```
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".NuestroFragment">

  <com.ilernaonline.miapp.CustomDrawableView
    android:id="@+id/view"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/button_second" />
...
```

Otra opción para crear un *drawable* sería heredar directamente de *Drawable:*

```
import android.graphics.Canvas
import android.graphics.ColorFilter
import android.graphics.Paint
import android.graphics.PixelFormat
import android.graphics.drawable.Drawable
import kotlin.math.min

class NuestroDrawable : Drawable() {

  private val redPaint: Paint = Paint().apply { setARGB(255, 255, 0, 0) }

  override fun draw(canvas: Canvas) {
    // Obtenemos los márgenes que nos han dejado
    val width: Int = bounds.width()
    val height: Int = bounds.height()
    val radius: Float = min(width, height).toFloat() / 2f
    // Dibujamos un circulo rojo en mitad de la vista
    canvas.drawCircle((width/2).toFloat(), (height/2).toFloat(), radius, redPaint)
  }
  override fun setAlpha(alpha: Int) {
    // Sobrescribir este método es obligatorio aunque no lo uses
  }
  override fun setColorFilter(colorFilter: ColorFilter?) {
    // Sobrescribir este método es obligatorio aunque no lo uses
  }
  override fun getOpacity(): Int =
    // Puede ser PixelFormat.UNKNOWN, TRANSLUCENT, TRANSPARENT, or OPAQUE
    PixelFormat.OPAQUE
}
```

NuestroDrawable hereda de *Drawable*. Crea un objeto *Pain* de color rojo, que utilizará en el método *draw* que ya conocemos. En el método *draw* utilizamos el objeto *Canvas* que nos pasan por parámetro para dibujar un círculo con el método *drawCircle*. La clase *Canvas* es la responsable de dibujar todas las primitivas de imagen, se trata de un lienzo virtual que contiene las llamadas a métodos de dibujo para finalmente presentar una imagen. Los objetos *drawables* pueden utilizar un *bitmap*, es decir, una imagen corriente para definir su apariencia, y/o utilizar la clase *Canvas* para definir formas básicas como óvalos y rectángulos. La clase *ShapeDrawable* que vimos anteriormente no hace más que utilizar las funciones del *Canvas* para definir su aspecto.

Continuemos con nuestro código. En el *layout*, podríamos definir un componente *ImageView:*

```
<ImageView
  android:id="@+id/imageView"
  android:layout_width="100dp"
  android:layout_height="100dp"
  app:layout_constraintBottom_toTopOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent"
  android:contentDescription="Nuestra imagen" />
```

Y en el código lo iniciaríamos con:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
  imageView.setImageDrawable(NuestroDrawable())
...
```

2.5.3. Cargar imágenes fuera de la *app*

Los *drawables* son una buena fuente de gráficos estáticos para decorar la interfaz gráfica. ¿Pero qué ocurre si queremos mostrar una lista de productos, cada uno con una imagen que tenemos en nuestro servidor de internet? Hay muchas formas de cargar imágenes desde el sistema de disco o desde internet, pero una de las más prácticas es mediante la librería Glide. Glide es una utilidad que permite manejar de forma rápida y eficiente imágenes, cargándolas y decodificándolas con gran control sobre la memoria y el cacheo en disco. Un problema clásico en Android relativo a la carga de *bitmaps* desde un archivo era el control de la memoria. Recordemos que la memoria en los terminales móviles está mucho más limitada que en otro tipo de máquinas. Sin embargo, Glide cuenta con los mecanismos necesarios para que la carga, manipulación y presentación de imágenes sea eficiente. Además, si lo que pretendemos es presentar imágenes almacenadas en la nube, su sistema de cacheo puede ayudarnos a ahorrar datos y ganar velocidad.

Para utilizar cualquier librería, lo primero que tenemos que hacer es añadirla a la lista de dependencias de nuestro archivo de compilación build.gradle:

```
dependencies {
  implementation 'com.github.bumptech.glide:glide:4.11.0'
  annotationProcessor 'com.github.bumptech.glide:compiler:4.11.0'
}
```

Utilizar sus funciones es muy sencillo. Imaginemos que en nuestro *layout* tenemos un elemento *ImageView:*

```
<ImageView
  android:id="@+id/imageView"
  android:layout_width="200dp"
  android:layout_height="200dp"
```

Desde el código podríamos cargar una imagen desde internet:

```
Glide
    .with(this)
    .load("https://www.ilerna.es/themes/classic/assets/img/ilerna-online-footer.png")
    .into(imageView)
```

Debemos recordar, en este caso, que para acceder a internet necesitamos declarar el permiso en el *manifest* de nuestro proyecto. De otro modo, el permiso nos será denegado y la *app* fallará:

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

La librería Glide es ampliamente usada en *apps* Android no solo por su sencillez y eficacia, sino por la gran cantidad de opciones que presenta. Por ello, es buena idea leer la documentación oficial y estudiar ejemplos con ella.

Gran cantidad de proyectos están en GitHub. También código de Glide :

https://github.com/bumptech/glide

2.6. Eventos del teclado

Los dispositivos móviles tienen dos tipos de teclas: las físicas, como el botón de encendido y las de volumen, y las del teclado virtual que vemos cuando nos colocamos sobre un campo de texto, por ejemplo. Según la finalidad, podemos interceptar las pulsaciones de unos u otros mediante diferentes métodos. Por ejemplo, tanto la clase *Activity* como *View* implementan la interfaz *KeyEvent.Callback*. Podemos sobrescribir los métodos de la interfaz y de ese modo interceptar los eventos de teclado. Veamos un ejemplo:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
    override fun onKeyDown(keyCode: Int, event: KeyEvent?): Boolean {
        Log.e("Activity", "onKeyDown: keyCode=$keyCode, $event")
        return super.onKeyDown(keyCode, event)
    }
    override fun onKeyUp(keyCode: Int, event: KeyEvent?): Boolean {
        Log.e("Activity", "onKeyDown: keyCode=$keyCode, $event")
        return super.onKeyUp(keyCode, event)
    }
    override fun onKeyLongPress(keyCode: Int, event: KeyEvent?): Boolean {
        Log.e("Activity", "onKeyDown: keyCode=$keyCode, $event")
        return super.onKeyLongPress(keyCode, event)
    }
...
```

Si ejecutamos la *app* y presionamos el botón de volumen, en el Logcat veremos:

Activity: onKeyDown: keyCode=25, KeyEvent { action=ACTION_DOWN, keyCode=KEYCODE_VOLUME_DOWN, scanCode=114, repeatCount=0, eventTime=79300600, downTime=79300600 }

Activity: onKeyDown: keyCode=25, KeyEvent { action=ACTION_UP, keyCode=KEYCODE_VOLUME_DOWN, scanCode=114, repeatCount=0, eventTime=79300800, downTime=79300600 }

Estas funciones *onKeyXXX* nos informan del evento mediante dos parámetros:

- ***keyCode**:* un entero relacionado con una constante que nos indicará la tecla pulsada. Si pulsamos la tecla *Ctrl* y hacemos clic sobre la clase *KeyEvent* en el código anterior, Android Studio abrirá el código fuente de la clase. Aquí podremos ver la definición de todas las funciones y constantes, como, por ejemplo, *KEYCODE_CAMERA*, *KEYCODE_A*, *KEYCODE_Z*, etcétera.
- ***event**:* es una instancia de la clase ***KeyEvent***. Esta clase amplía la información sobre el evento. No solo guarda el *keyCode*, sino otros parámetros como: *action* (*ACTION_DOWN*, *ACTION_UP* o *ACTION_MULTIPLE*), *repeatCount* (número de veces que se presionó la tecla), *isAltPressed* si la tecla *Alt* estaba presionada mientras se pulsó y muchos otros parámetros relativos al evento.

El valor devuelto por las funciones es booleano. Si ya hemos procesado el evento y no deseamos que siga propagándose por las demás vistas, retornaremos *true*. Si, por el contrario, solo queríamos escuchar pero queremos que el evento siga hasta el siguiente *listener*, retornaremos *false*.

También podríamos necesitar manejar los botones de medios. Estos botones son botones físicos que tienen algunos dispositivos periféricos, como los auriculares o *joysticks*, que se conectan al móvil. Al presionar estos botones, la aplicación recibirá un *KeyEvent*, como hemos visto anteriormente. El código de estos botones, sin embargo, comienza con *KEYCODE_MEDIA* en lugar de con *KEYCODE*.

2.7. Técnicas de animación y de sonido

Ya hemos visto anteriormente cómo el uso de imágenes en la interfaz gráfica puede dar un valor añadido a nuestra *app*. En esta ocasión, mostraremos cómo animar los gráficos para conseguir una interacción superior con el usuario. En ocasiones, animaremos las transiciones entre ventanas, en otras, animaremos elementos gráficos para dar una sensación material, como de 3D, de nuestras pantallas. También aprenderemos a lanzar sonidos que alerten al usuario de un mensaje importante o de algún error.

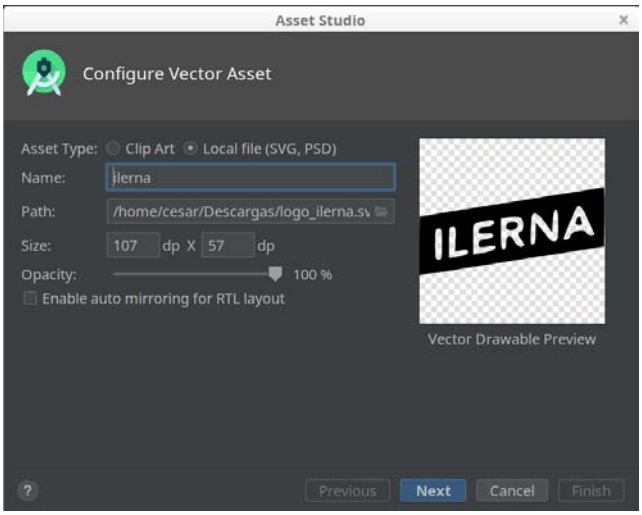
2.7.1. Animaciones

En este punto estudiaremos animaciones sencillas de elementos de nuestras pantallas, lo que dará una ilusión de mayor responsividad al usuario. En el siguiente ejemplo, mostraremos cómo animar un *drawable* mediante la clase *AnimatedVectorDrawableCompat*.

Creemos un nuevo proyecto. En el menú de Android Studio, pulsemos *File > New > New project… > Empty Activity*. Nuestra *app* tendrá una sola clase, llamada *MainActivity*, que hace uso de un *layout activity_main*. En la carpeta "res", tendremos también el directorio "drawable" con los archivos del icono de la *app*. Vamos a añadir un nuevo recurso en esta carpeta. Pulsamos con el botón derecho sobre la carpeta y elegimos *New > Drawable Resource File*, introducimos el nombre "ilerna" y aceptamos. Introducimos el código:

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
  android:width="107dp"
  android:height="57dp"
  android:viewportWidth="107"
  android:viewportHeight="57">
 <path
    android:name="one"
    android:fillAlpha="1"
    android:fillColor="#F00"
    android:pathData="M53,11.6c-28.9,4.6 -52.6,8.3 -52.7,8.4 -0.2,-0 -0.3,7.7 -0.3,17l0,17 3.7,-0.6c2.1,-0.3 26.1,-4 53.3,-
8.3l49.5,-7.7 0.3,-17.2c0.1,-9.5 -0.1,-17.1 -0.5,-17.1 -0.5,0.1 -24.4,3.9 -53.3,8.5zM96.9,21.2c1.8,4 3.5,7.9 3.8,8.5 0.3,0.8 -0.3,1.3 -
1.6,1.3 -1.2,-0 -2.4,-0.7 -2.7,-1.5 -1.1,-2.8 -9.4,-1.1 -9.4,1.9 0,0.8 -0.9,1.6 -2,1.9 -1.1,0.3 -2,0.1 -2,-0.3 0,-0.8 6.5,-16.4 7.6,-18.1 1.4,-
2.3 3.4,-0.3 6.3,6.3zM80.24.9c0,7.5 -0.3,9 -1.7,9.6 -1.3,0.4 -2.5,-0.3 -4.3,-2.6 -1.4,-1.8 -3.5,-4.4 -4.7,-5.8l-2.2,-2.6 -0.1,6.1c0,4.6 -
0.4,6.3 -1.5,6.8 -1.4,0.5 -1.5,-0.7 -1.3,-8.7 0.3,-8.3 0.5,-9.2 2.3,-9.5 1.4,-0.1 3.1,1.4 6,5.4l4,5.6 0.3,-5.3c0.3,-6.3 0.8,-7.9 2.2,-7.9 0.6,-
0 1,3.5 1,8.9zM58.8,21.2c1.7,1.7 1.5,6.7 -0.3,7.4 -2,0.8 -1.9,2 0.5,5.4 1.9,2.6 1.9,2.8 0.3,3.5 -1.7,0.6 -3.3,-0.7 -5.9,-5.1 -1.5,-2.6 -3.4,-
0.9 -3.4,3.2 0,2.4 -0.5,3.4 -1.5,3.4 -1.2,-0 -1.5,-1.7 -1.5,-8.9l0,-9 2.8,-0.4c5.4,-0.8 7.8,-0.7 9,0.5zM43,23.5c0,1.4 -1.4,2 -6.5,2.7 -
1.1,0.2 -2.1,1.1 -2.3,2.2 -0.3,1.5 0.2,1.7 4,1.4 3.3,-0.2 4.3,-0 4.3,1.2 0,1 -1.3,1.7 -4,2 -3.2,0.4 -4.1,0.9 -4.3,2.8 -0.3,2.2 -0.2,2.3 4.9,1.6
4.1,-0.5 5.1,-0.4 4.7,0.6 -0.5,1.5 -2.7,2.3 -9,3.3l-3.8,0.6 0,-8.9c0,-9.8 -0.6,-8.8 6.5,-10.3 4,-0.9 5.5,-0.7 5.5,0.8zM20.8,33.1l-0.3,7.1
3.8,-0.6c2.8,-0.5 3.7,-0.3 3.7,0.8 0,1.5 -1.2,2 -7.2,3.1l-3.8,0.6 0,-9.1c0,-8.5 0.1,-9 2.1,-9 1.9,-0 2,0.5 1.7,7.1zM13,36c0,8.3 -0.1,9 -2,9 -
1.9,-0 -2,-0.7 -2,-9 0,-8.3 0.1,-9 2,-9 1.9,-0 2,0.7 2,9z"
    />
</vector>
```

Este es un *drawable* creado al importar una imagen vectorial. Podemos copiar este código directamente, hay otros en la web. Si queremos ver todo el proceso, no hay más que descargar el logo de Ilerna de la web o cualquier otra imagen que deseemos. Luego utilizaremos un editor de imágenes para exportarlo a una imagen vectorizada SVG. Entonces importamos la imagen en Android Studio con la opción de menú *File > New > Vector Asset:*



Luego editamos el código para añadir algunos campos más. Estos nos servirán para modificar en el animador los valores transparencia y color del *drawable:*

```
android:name="path_ilerna"
android:fillAlpha="1"
android:fillColor="#F00"
```

Una vez que tenemos el *drawable*, crearemos una animación. Pulsamos con el botón derecho sobre la carpeta "res" y elegimos *New > Directory*. Establecemos su nombre como "animator". Ahora pulsamos sobre la carpeta "animator", elegimos *New > Animator Resource File* y usamos el nombre "animacion", con el código:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
<objectAnimator
  android:duration="2500"
  android:propertyName="fillColor"
  android:valueFrom="#F00"
  android:valueTo="#00F" />
<objectAnimator
  android:duration="2500"
  android:propertyName="fillAlpha"
  android:valueFrom="0.2"
  android:valueTo="1" />
</set>
```

El *animator* especifica la duración del efecto, la propiedad sobre la que actuará y de qué a qué valor pasará dicho atributo. Solo nos falta un archivo recurso que una el *drawable* con el *animator*. Pulsamos con el botón derecho sobre la carpeta *drawable* y escogemos *New > Drawable Resource File* con el nombre "ilerna_animacion". Su código será:

```
<?xml version="1.0" encoding="utf-8"?>
<animated-vector
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:drawable="@drawable/ilerna">
  <target
    android:name="path_ilerna"
    android:animation="@animator/animacion" />
</animated-vector>
```

Android nos proporciona muchos mecanismos para animar la interfaz gráfica, tanto que podríamos estar horas viendo todos ellos. En cualquier caso, es conveniente echar un

vistazo a la documentación y buscar exactamente el efecto que hará mucho más interesante la interfaz gráfica de nuestra *app*.

Documentación sobre las animaciones en Android:

https://developer.android.com/training/animation/overview

De nuevo, en GitHub podemos encontrar miles de ejemplos:

https://github.com/android/animation-samples

2.7.2. Sonidos

Los sonidos de nuestra *app* pueden alertar al usuario de un mensaje o un estado especial. Una forma sencilla de generar un sonido es un generador de tonos. Veamos un ejemplo:

```
val tg = ToneGenerator(AudioManager.STREAM_NOTIFICATION, ToneGenerator.MAX_VOLUME)

override fun onResume() {
  super.onResume()
  tg.startTone(ToneGenerator.TONE_CDMA_ABBR_ALERT)
}

override fun onStop() {
  super.onStop()
  tg.release()
}
```

Creamos una instancia de la clase *ToneGenerator*, que sonará en el canal de notificaciones con el volumen máximo. Cuando la ventana se hace activa, llama a *onResume*, donde utilizamos el método *startTone* del generador con uno de los muchos tipos de sonidos de que dispone. Es una buena práctica liberar los recursos del generador una vez que no los necesitemos, por ejemplo, en *onStop*. Es imprescindible elegir adecuadamente en qué canal sonará el aviso, pues el sistema de audio de Android enrutará el tono que generemos en ese canal, mezclándolo con todos los sonidos que lleguen de otras fuentes.

Quizá los sonidos disponibles en el generador de tonos sean demasiado simples para el efecto que buscamos. Si necesitamos sonidos más complejos, utilizaremos archivos que hayamos creado con algún editor de audio o descargado de internet. Para reproducir varios sonidos de forma rápida y eficiente, lo mejor es utilizar un SoundPool. Este objeto guardará en memoria los sonidos y los reproducirá cuando se lo pidamos, mezclándolos en el canal correspondiente.

Veamos cómo funciona con un ejemplo. Hemos creado una *activity*, y en su *layout* tenemos dos botones, *btnSound1* y *btnSound2*. Vamos a definir las variables que usaremos en el código:

```
private var soundPool: SoundPool? = null
private var audioManager: AudioManager? = null
private val streamType = AudioManager.STREAM_MUSIC
private var isLoaded = false
private var sound1 = 0
private var sound2 = 0
private var volume = 0f
```

El primer objeto es el que llevará a cabo todo el trabajo. El *AudioManager* es un servicio del sistema Android que nos permite controlar los parámetros de sonido del dispositivo. La variable *streamType* la usamos para referenciar el canal en el que reproduciremos nuestros archivos de sonido. Otros canales podrían ser: *STREAM_ALARM*, *STREAM_NOTIFICATION*, *STREAM_RING*, etcétera. El booleano *isLoaded* nos servirá para saber si SoundPool tiene ya todos los sonidos en memoria listos para su reproducción. *Sound1* y *sound2* son identificadores de sonido. Por último, *volume* es un *float* con el volumen de audio. Para iniciar estas variables, en *onCreate* llamaremos a la función:

```
private fun prepararSonidos() {

  audioManager = getSystemService(Context.AUDIO_SERVICE) as AudioManager
  audioManager?.let { audioManager →

    val currentVolumeIndex = audioManager.getStreamVolume(streamType).toFloat()
    val maxVolumeIndex = audioManager.getStreamMaxVolume(streamType).toFloat()
    volume = currentVolumeIndex / maxVolumeIndex
    volumeControlStream = streamType

    val audioAttributes = AudioAttributes.Builder()
       .setUsage(AudioAttributes.USAGE_GAME)
       .setContentType(AudioAttributes.CONTENT_TYPE_SONIFICATION)
       .build()
    soundPool = SoundPool.Builder()
       .setAudioAttributes(audioAttributes)
       .setMaxStreams(5)
       .build()

    soundPool!!.setOnLoadCompleteListener { soundPool, sampleId, status ->
       Log.d("Activity", "Se completo la carga: $soundPool, $sampleId, $status")
       isLoaded = true
    }

    sound1 = soundPool?.load(this, R.raw.sonido1, 1) ?: 0
    sound2 = soundPool?.load(this, R.raw.sonido2, 1) ?: 0

    btnSound1.setOnClickListener {
       playSound(sound1)
    }
    btnSound2.setOnClickListener {
       playSound(sound2)
    }
  }
}
```

No es absolutamente necesario, pero lo primero que haremos será decirle al sistema que, mientras nuestra actividad esté activa, los controles de audio afecten al canal con el que estamos trabajando, estableciendo la variable *volumeControlStream*. Lo siguiente

es crear el SoundPool, pero para ello necesitamos un objeto *AudioAttributes* que definirá las características de los sonidos almacenados por SoundPool. Se establece que el uso de los sonidos será como juego, con *USAGE_GAME*, pero podrían especificarse otros, como *USAGE_MEDIA*, *USAGE_VOICE_COMMUNICATION*, *USAGE_ALARM*, o *USAGE_NOTIFICATION_EVENT*.

Ahora creamos el SoundPool con los atributos que acabamos de definir. También establecemos el número máximo de *streams*, es decir, el número máximo de sonidos que podemos reproducir en un momento dado. Si reproducimos un sonido antes de que el anterior haya terminado, se mezclarán unos sobre otros hasta llegar al máximo. Después de crear el SoundPool, establecemos un *listener* para saber cuándo termina de cargar cada archivo de sonido. Cargar en memoria un archivo de sonido puede tardar más o menos, dependiendo del *hardware* y de lo pesado que sea el fichero. Aquí hemos simplificado un poco, porque cada vez que se cargue un sonido se llamará al *listener*, pero quizá la variable *isLoaded* debería establecerse a *true* solo cuando todos los archivos se hayan cargado, y no solo el primero. Pero sigamos adelante.

Ahora encontramos el código de carga de cada sonido. Vemos que los recursos los adquirimos desde *raw*, pues los proyectos de Android no tienen una carpeta específica para recursos de sonido. Lo siguiente es establecer la acción que se llevará a cabo con la pulsación de cada botón. En ambos casos, utilizaremos la siguiente función:

```
private fun playSound(sound: Int) {
  if(isLoaded) {
     val leftVolumn = volume
     val rightVolumn = volume
     val priority = 1
     val loop = 0
     val rate = 1f
     soundPool!!.play(sound, leftVolumn, rightVolumn, priority, loop, rate)
  }
}
```

En ella, utilizamos el SoundPool para reproducir el audio identificado con el valor entero que guardamos al cargar el respectivo archivo de audio. Eso es todo, podemos probar a pulsar repetidamente ambos botones. Escucharemos cómo cada sonido se monta sobre sí mismo y sobre el otro varias veces sin el menor problema. Cuando terminemos con el objeto, deberíamos liberar la memoria, por ejemplo, sobrescribiendo el método *onStop* de la actividad:

```
override fun onStop() {
  super.onStop()
  soundPool?.release()
  soundPool = null
}
```

2.8. Descubrimiento de servicios

Como hemos visto en temas anteriores, los componentes principales de la aplicación, como actividades y fragmentos, tienen un ciclo de vida que depende de cómo el usuario interactúe con ella. La *app* puede detenerse cuando el usuario cambia a otra o cuando la cierra directamente. Imaginemos que nuestra *app* necesita llevar a cabo una tarea que no dependa del ciclo de vida de las ventanas. Necesitaremos un crear un servicio.

Los servicios son componentes que pueden realizar tareas de larga duración en segundo plano. La ejecución en *background* o segundo plano significa que el usuario no percibe la existencia de dichas tareas, ya que no hay interfaz gráfica asociada. No debemos confundir la ejecución en *background*, es decir, sin interfaz gráfica, con la ejecución en paralelo. La ejecución en paralelo significa que dos o más tareas pueden lanzarse al mismo tiempo, ejecutándose de forma independiente la una de la otra. De este modo, un retardo en una tarea no influye a la otra. Para llevar a cabo una ejecución en paralelo o asíncrona, necesitaremos otros mecanismos que veremos más adelante, como hilos o *coroutines*.

Imaginemos una app de reproducción de música. Querremos que la *app* reproduzca nuestras canciones preferidas mientras chateamos en otra *app* de mensajería. ¿Cómo podría ejecutarse una *app* mientras usamos otra? Utilizando un servicio que se ejecuta en segundo plano y, por lo tanto, no necesitamos que la *app* de música aparezca activa en pantalla para hacer sonar la música. De hecho, el servicio podría ejecutarse incluso aunque las actividades se hubiesen cerrado. Igualmente, cuando descargamos un archivo de internet en el navegador y cambiamos a otra *app* antes de que termine, la descarga continúa porque se está ejecutando en segundo plano.

Pero recordemos de nuevo que un servicio no da el poder de ejecución asíncrona o multitarea. El servicio se ejecuta en el mismo proceso que la aplicación. Por ello, si hacemos que el servicio lleve a cabo una tarea pesada, la interfaz gráfica de nuestra *app* dejará de responder. Es por eso que normalmente los servicios utilizan también algún mecanismo de ejecución asíncrona; por ejemplo, creando un hilo de trabajo para que lleve a cabo la tarea en paralelo. Veamos los tipos diferentes de servicios que podremos usar en nuestra *app:*

- ***Foreground* o primer plano**: un servicio de este tipo puede llevar a cabo tareas de larga duración, pero se ve obligado a mostrar un indicador para que el usuario sepa que está activo. Ese indicador no es más que una notificación en la barra de notificaciones. Por ejemplo, en el caso de nuestra *app* de reproducción musical, haríamos que se mostrase un icono en la barra de notificaciones indicando que la *app* está activa. Quizá la notificación tuviese además algunos controles para detener la reproducción, pasar a la siguiente pista o cerrar la aplicación.

- ***Background* o segundo plano**: un servicio de este tipo realiza una tarea sobre la que el usuario no tiene ningún control, como, por ejemplo, un cálculo complejo o la compresión de los mensajes en la *app* de mensajería. Debemos tener en cuenta que, a partir de la versión 26 de la API de Android, los servicios en *background* tienen muchas limitaciones. Android quiere mantener la seguridad y la privacidad, y matará un servicio en *background* que lleve demasiado tiempo corriendo inadvertidamente por el usuario. Por eso, debemos tener cuidado al utilizar este tipo de servicios si no queremos que el sistema nos los cierre antes de que hayan terminado su tarea.

- ***Bound* o enlazado**: un servicio está enlazado cuando un componente de la aplicación llama a *bindService* para establecer una conexión con él. Un servicio enlazado ofrece una interfaz cliente-servidor que permite una comunicación directa con el servicio desde la *app*. El servicio se mantendrá activo solo mientras el componente de la *app* siga enlazado a él.

Veamos un ejemplo de *foreground service*, que es uno de los más usados. Imaginemos que deseamos calcular todos los números primos en un rango definido por el usuario. Primero creamos el proyecto mediante *File > New > New Project… > Empty Activity*. Vamos a añadir un par de *EditText* para que el usuario introduzca el rango de enteros *desde* y *hasta*. Crearemos además un *TextView* para mostrar el resultado y, por último, dos botones que controlarán el cálculo. El *layout* terminaría siendo algo como:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">
  <EditText
    android:id="@+id/txtDesde"
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="8dp"
    android:ems="10"
    android:hint="Desde"
    android:inputType="number"
    android:text="1"
    app:layout_constraintEnd_toStartOf="@+id/txtHasta"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
  <EditText
    android:id="@+id/txtHasta"
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginEnd="8dp"
    android:ems="10"
    android:hint="Hasta"
    android:inputType="number"
    android:text="90000"
```

```
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/txtDesde"
    app:layout_constraintTop_toTopOf="@+id/txtDesde" />
  <Button
    android:id="@+id/btnCalcular"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:text="Calcular"
    android:focusedByDefault="true"
    app:layout_constraintEnd_toStartOf="@id/btnCancelar"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/txtDesde" />
  <Button
    android:id="@+id/btnCancelar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginEnd="8dp"
    android:text="Cancelar"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/btnCalcular"
    app:layout_constraintTop_toTopOf="@+id/btnCalcular" />
  <TextView
    android:id="@+id/txtContenido"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="8dp"
    android:layout_marginTop="32dp"
    android:layout_marginRight="32dp"
    android:scrollbars="vertical"
    android:text=""
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/btnCalcular" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Ahora crearemos un objeto que será el cerebro del cálculo de números primos:

```
object Primos {
  private const val TAG = "Primos"

  var cancelar = false

  fun formateaPrimos(desde: Int, hasta: Int): String {
    val a = calcularPrimos(desde, hasta)
    val sb = StringBuilder()
    for(i in a) {
      sb.append(i)
      sb.append(", ")
    }
    sb.replace(sb.length-2, sb.length-1, ".")
    return sb.toString()
  }

  private fun calcularPrimos(desde: Int, hasta: Int): List<Int> {
    if(desde >= hasta)
      return listOf()
    cancelar = false
    iniTime = System.currentTimeMillis()
    val primos = mutableListOf<Int>()
    for(i in desde..hasta) {
      endTime = System.currentTimeMillis()
      if(cancelar)
        return primos
      if(esPrimo(i)) {
        primos.add(i)
      }
    }
```

```
        endTime = System.currentTimeMillis()
        return primos
    }

    private fun esPrimo(numero: Int): Boolean {
        var esPrimo = true
        for(i in 2..numero/2) {
            if(cancelar)
                return false
            if(numero % i == 0) {
                esPrimo = false
                break
            }
        }
        return esPrimo
    }
}
```

El código no es muy complicado. El método *esPrimo* se encarga de comprobar si un número es primo o no lo es, pura matemática. La función *calcularPrimos* recibe dos parámetros que especifican el rango de enteros que comprobar y devuelve una lista con los enteros que son primos dentro de ese rango. Usando la función *esPrimo*, se trata simplemente de iterar en ese rango, nada más. La función *formateaPrimos* transforma la lista que devuelve *calcularPrimos* y la convierte en una cadena de caracteres para que la *activity* la pueda mostrar. Veamos cómo es nuestra actividad:

```
import android.content.Context
import android.content.Intent
import android.os.Bundle
import android.text.method.ScrollingMovementMethod
import android.util.Log
import android.view.View
import androidx.appcompat.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_main.*
import org.greenrobot.eventbus.EventBus
import org.greenrobot.eventbus.Subscribe
import org.greenrobot.eventbus.ThreadMode

class MainActivity : AppCompatActivity() {

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

        btnCalcular.setOnClickListener {

            val desde = txtDesde.text.toString().toInt()
            val hasta = txtHasta.text.toString().toInt()

            txtContenido.movementMethod = ScrollingMovementMethod()
            txtContenido.text = ""
            txtContenido.scrollY = 0

            Intent(this, PrimosForegroundService::class.java).also { intent ->
                intent.putExtra("desde", desde)
                intent.putExtra("hasta", hasta)
                startService(intent)
            }
        }

        btnCancelar.setOnClickListener {
            Primos.cancelar = true
        }
    }

    override fun onResume() {
```

```
    super.onResume()
    EventBus.getDefault().register(this)
  }

  override fun onPause() {
    super.onPause()
    EventBus.getDefault().unregister(this)
  }

  @Subscribe(threadMode = ThreadMode.MAIN)
  fun onEventBusEvent(res: String) {
    txtContenido.text = res
  }
}
```

En *onCreate* establecemos el *listener* para el botón *btnCalcular*. Cuando se presione, obtendremos los valores *desde* y *hasta* que haya introducido el usuario y los convertiremos en enteros. Luego borraremos el campo *resultado* para que empiece de cero. Ahora crearemos un *intent*, con el que arrancaremos el servicio. Como vemos, los *intents* no solo nos sirven para lanzar actividades, sino también servicios, y, al igual que las actividades, los servicios deben declararse en el *manifest*. Aprovechando que hay que editar el *manifest*, añadiremos un permiso necesario para ejecutar servicios en *foreground:*

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

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

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

    <service android:name=".PrimosForegroundService" />

  </application>
</manifest>
```

Pero sigamos con la *activity*. Creamos el *intent* para llamar al servicio *PrimosForegroundService* y le pasamos dos argumentos: los enteros *desde* y *hasta*. Después arrancamos el servicio llamando a *startService*. Veamos cómo crear el servicio en primer plano:

```
import android.app.*
import android.content.Intent
import android.os.Build
import android.os.IBinder
import androidx.core.app.NotificationCompat
import org.greenrobot.eventbus.EventBus

class PrimosForegroundService : Service() {
  override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
```

```
        val desde = intent.getIntExtra("desde", 0)
        val hasta = intent.getIntExtra("hasta", 0)

        createNotificationChannel()

        val notificationIntent = Intent(this, MainActivity::class.java)
        val pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0)
        val notification: Notification = NotificationCompat.Builder(this, CHANNEL_ID)
                .setContentTitle("Primos Foreground Service")
                .setContentText("Calculando primos desde $desde hasta $hasta")
                .setSmallIcon(R.drawable.ic_launcher_foreground)
                .setContentIntent(pendingIntent)
                .build()

        startForeground(1, notification)

        Thread {
            run {
                val res = Primos.formateaPrimos(desde, hasta)
                EventBus.getDefault().post("PrimosService: $res")
                stopSelf()
            }
        }.start()

        return START_NOT_STICKY
    }

    private fun createNotificationChannel() {
        if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
            val serviceChannel = NotificationChannel(
                CHANNEL_ID,
                "Foreground Service Channel",
                NotificationManager.IMPORTANCE_DEFAULT
            )
            val manager = getSystemService(NotificationManager::class.java)
            manager.createNotificationChannel(serviceChannel)
        }
    }

    override fun onBind(intent: Intent?): IBinder? {
        return null
    }

    companion object {
        const val CHANNEL_ID = "PrimosChannel"
    }
}
```

Nuestro servicio hereda de la clase *Service*. Sobrescribimos el método *onStartCommand*, que se llamará cuando el sistema cree nuestro servicio después de que la *activity* llame a *startService*. En esta función, primero rescatamos los parámetros que nos pasaron a través del *intent*. Luego llamamos a una función en la que crearemos el canal de notificaciones. Desde Android 8 (la versión 26 de la API), las notificaciones deben estar asociadas a un canal. Cada canal permite agrupar las notificaciones de una aplicación, establecer un icono determinado, etcétera. De este modo, el usuario puede deshabilitar un canal o establecer el volumen de las notificaciones correspondientes a ese canal.

Ya creado el canal, podemos crear la notificación. Para ello, utilizamos un *intent*. También creamos un objeto de la clase *PendingIntent*, que servirá para abrir la *activity* cada vez que el usuario presione el icono de la notificación. Simplificando, un *pending intent* es como la capacidad futura de lanzar un *intent*. Una vez creada la notificación, podemos llamar a *startForeground*. Esta llamada es imprescindible, y si tardamos

demasiado en llamarla después de la creación del servicio, el sistema Android matará el servicio por motivos de seguridad. El sistema quiere asegurarse de que el usuario tiene constancia de la existencia del servicio, y mediante esta función se crea la notificación necesaria. A partir de ahora podemos relajarnos y ejecutar el código realmente práctico del servicio, lo que se conoce como *payload*.

Como comentamos antes, sin embargo, si ejecutamos un procesamiento intensivo en el servicio, podemos paralizar la interfaz gráfica de la aplicación, porque ambos se ejecutan en el mismo hilo de proceso. Por ello, en este ejemplo sencillo utilizaremos la clase *Thread* para abrir un nuevo hilo de proceso en el que ejecutar el cálculo de números primos. En un tema posterior estudiaremos mejor el multiproceso. Dentro de este nuevo hilo de ejecución, por lo tanto, hacemos el cálculo mediante la llamada a *Primos.formateaPrimos*. El resultado lo enviamos de vuelta a la *activity* mediante un mensaje de *EventBus*, aunque podríamos haber utilizado también *Broadcasts*. La comunicación entre un servicio y el resto de componentes puede realizarse de varias maneras, pero en este caso hemos utilizado la librería *EventBus*, que es un método sencillo a la vez que eficaz de comunicación entre varios componentes de la *app*. Como todas las librerías, debemos insertarla en nuestro archivo de configuración build.gradle:

```
dependencies {
    ...
    implementation 'org.greenrobot:eventbus:3.2.0'
}
```

La clase que quiera recibir mensajes debe registrarse llamando a la función *EventBus.getDefault().register(this)*. Cuando decida dejar de recibir mensajes, llamará a *EventBus.getDefault().unregister(this)*, y los mensajes se recibirán en la función que utilice la anotación *@Subscribe*.

Hay mucho que aprender sobre servicios en Android:

https://developer.android.com/guide/components/services

Sobre la forma de comunicación entre clases Broadcast:

https://developer.android.com/guide/components/broadcasts

2.9. Bases de datos y almacenamiento

Los programas, ya sean de escritorio, web o móvil, necesitan guardar datos de una ejecución a otra para su correcto funcionamiento. Quizá necesiten almacenar opciones de usuario, quizá guarden *logs* de depuración con errores y alertas, quizá estemos hablando de un juego en el que debemos guardar los jugadores y los puntos que han conseguido en cada partida. Veamos los métodos más usuales para guardar información en el dispositivo móvil.

2.9.1. Persistencia

Si lo que necesitamos guardar no es más que un pequeño conjunto de pares clave-valor, deberíamos utilizar la clase *SharedPreferences*. Un objeto de este tipo utilizará un archivo para escribir y leer pares clave-valor. Pero nosotros solo veremos una simple interfaz con algunos pocos métodos con los que podremos hacer persistentes nuestros datos. Si necesitamos el objeto desde un objeto de nuestra *app* que no sea una activity, tendríamos que crearlo con:

val sharedPref = getSharedPreferences("configuracion", Context.MODE_PRIVATE)

Lo que hace esta función es crear, o abrir si ya existiese, un archivo de pares clave-valor con el nombre pasado como primer parámetro. El segundo parámetro es un *flag* que controla la creación del archivo. *MODE_PRIVATE* hará que este archivo sea privado para otras *apps*. Este archivo sería accesible desde cualquier parte de la aplicación. Si, por el contrario, quisiéramos almacenar solo valores para una actividad en concreto, podríamos crear el objeto con:

val sharedPref = getPreferences(Context.MODE_PRIVATE)

Aquí se ahorra el nombre del archivo, porque el archivo estará asociado unívocamente con la *activity* que lo llame. De este modo, podríamos tener un archivo por actividad sin riesgo de confundirnos con los nombres. Veamos ahora cómo podemos utilizar estos objetos. Cuando quisiéramos guardar algunos valores, podríamos hacer:

```
val sharedPref = getPreferences(Context.MODE_PRIVATE)
with(sharedPref.edit()) {
  putInt("claveInt", 69)
  putString("claveString", "valor")
  putBoolean("claveBoolean", true)
  //...
  //commit()//llamada síncrona, ten cuidado con el main thread
  apply()//llamada asíncrona, puedes llamarla en el main thread
}
```

Vemos que lo primero es acceder a una instancia de *SharedPreferences* cono uno de los dos métodos que hemos visto. Después, llamaríamos a *edit* para ponernos en modo escritura. Ahora podríamos llamar a los métodos *putX* necesarios para guardar todas

nuestras variables. Por cada tipo de datos, tendremos una llamada a la clase. Cuando hallamos terminado, llamaremos a *apply*, que dará por finalizada la edición, aunque el proceso de escritura definitivo lo ejecutará en un hilo distinto para no paralizar la interfaz gráfica de la *activity*. Si fuesen pocos campos, podríamos llamar a *comit*, pero salvo que sea imprescindible, preferiremos utilizar el método *apply*. Veamos ahora cómo podríamos recuperar los datos desde el archivo:

```
val sharedPref = getPreferences(Context.MODE_PRIVATE)
val valorInt = sp1.getInt("claveInt", 0)
val valorString = sp1.getString("claveString", null)
val valorBoolean = sp1.getBoolean("claveBoolean", false)
//...
```

Y eso es todo, es sencillo y rápido. Si no son demasiados valores y son básicamente pares de clave-valor, este será el método que utilizaremos.

Si no nos bastase con eso, siempre podríamos escribir y leer de un archivo de disco. Será algo más tedioso, pero tendremos mayor libertad. Veamos un ejemplo: crearemos una función para escribir y otra para leer del archivo:

```
private fun writeToFile(data: String) {
    try {
        val file = openFileOutput("configuracion.txt", Context.MODE_PRIVATE)
        val outputStreamWriter = OutputStreamWriter(file)
        outputStreamWriter.write(data)
        outputStreamWriter.close()
    }
    catch(e: IOException) {
        Log.e("Error", "File write failed: ", e)
    }
}

private fun readFromFile(): String? {
    var ret = ""
    try {
        val inputStream: InputStream? = openFileInput("configuracion.txt")
        if (inputStream != null) {
            val inputStreamReader = InputStreamReader(inputStream)
            val bufferedReader = BufferedReader(inputStreamReader)
            var receiveString: String? = ""
            val stringBuilder = StringBuilder()
            while (bufferedReader.readLine().also { receiveString = it } != null) {
                stringBuilder.append("\n").append(receiveString)
            }
            inputStream.close()
            ret = stringBuilder.toString()
        }
    }
    catch (e: FileNotFoundException) {
        Log.e("login activity", "File not found: ", e)
    }
    catch (e: IOException) {
        Log.e("login activity", "Can not read file: ", e)
    }
    return ret
}
```

En la función de escritura, vemos cómo mediante el método *openFileOutput* abrimos el archivo para escritura. Luego utilizamos ese *stream* para crear otro que nos permita escribir de forma cómoda nuestros datos, el *OutputStreamWriter*. Basta llamar a *write* con nuestros datos y luego a *close* para dar por finalizada la sesión de escritura. El

archivo contendrá nuestros datos. Ahora, para leerlos, utilizaremos el método *openInputStream* para abrir el archivo en modo lectura. El *stream* resultante lo usaremos para crear un *InputStreamReader*, y luego un *BufferedReader*, que nos facilita mucho las operaciones de lectura. Llamaremos al método *readLine* una y otra vez hasta que no haya más líneas en el archivo. Cerraremos el archivo y devolvemos los datos. En este caso escribimos cadenas de caracteres, pero podríamos haber escrito cualquier otro dato. Vemos cómo Android nos facilita bastante cualquier tarea, solo es necesario conocer las clases adecuadas para cada necesidad.

2.9.2 Bases de datos en Android

Como ya hemos visto, si necesitamos guardar únicamente algunos pares clave-valor, podemos utilizar las *SharedPreferences*. Para almacenar un conjunto de datos o líneas de texto que serán escritas o leídas en bloque, podríamos utilizar un archivo en el almacenamiento interno (solo visible a nuestra *app*) o externo. Pero si necesitamos almacenar información con una estructura más compleja, de rápido y fácil acceso, necesitamos una base de datos. Una base de datos o BBDD es un *software* que permite almacenar y recuperar datos mediante consultas inteligentes, manteniendo la coherencia entre las estructuras de datos. Existen principalmente dos tipos de BBDD: relacionales y no relacionales. Para la mayoría de los casos de uso, una BBDD relacional será idónea. Por ello, los sistemas operativos móviles tienen integrado *software* de gestión de bases de datos para que las aplicaciones puedan utilizarlas.

Cuando creamos una base de datos desde nuestra aplicación, básicamente estamos creando un archivo que el sistema guardará en la carpeta privada de la *app*, de la misma forma que con los archivos del almacenamiento interno. De este modo se mantiene la privacidad de los datos, pues el directorio de la aplicación no es accesible para otros usuarios o aplicaciones. Desde las primeras versiones de iOS y Android, la mejor forma de mantener una base de datos local ha sido mediante SQLite. Aunque existen muchos otros, el sistema de gestión de base de datos SQLite está escrito en C, por lo que es muy eficiente, es simple pero completo y ocupa muy poco.

Hoy en día sigue siendo el preferido y está preinstalado en el sistema. Sin embargo, debemos tener algunas consideraciones antes de usar SQLite de forma directa, porque los sistemas relacionales como este tienen algunos inconvenientes. Su lenguaje de consulta, llamado SQL (Structured Query Language), es potente, pero nada tiene que ver con los lenguajes de programación orientados a objetos actuales, de modo que existe un desacople entre el código de la aplicación y el que entiende la base de datos. Para solucionar ese problema, podemos usar una librería ORM (Object Relational Mapping) que traducirá de un lenguaje al otro dentro de la aplicación sin esfuerzo por nuestra parte. Existen muchas librerías que nos ayudan a usar SQLite, pero actualmente la preferida es Room.

Antes de continuar, estudiemos cómo es el lenguaje SQL que utilizan todas las bases de datos relacionales. Sus comandos se clasifican en dos grupos: DDL (Data Definition Language) y DML (Data Manipulation Language). El DDL se utiliza, como su nombre indica, para definir la estructura de los datos. Sus comandos son: *CREATE*, *ALTER*, *DROP* y *TRUNCATE* para crear, modificar y eliminar estructuras como las tablas. El DML sirve para modificar los datos dentro de las estructuras ya existentes, y tiene los comandos: *SELECT*, *INSERT*, *UPDATE* y *DELETE* para recuperar, insertar, actualizar y borrar los datos dentro de las tablas.

Veamos un ejemplo de código SQL. Imaginemos que nuestra aplicación es un videojuego y necesitamos guardar los datos de los usuarios y los puntos de cada partida. Con el DDL de SQL crearíamos dos tablas:

```
CREATE TABLE jugadores (
      jugador_id INTEGER PRIMARY KEY,
      nombre TEXT NOT NULL UNIQUE,
      avatar TEXT,
      cinturon INTEGER,
);

CREATE TABLE partidas (
      partida_id INTEGER PRIMARY KEY,
      jugador_id INTEGER,
      fecha INTEGER,
      puntos INTEGER,
      nivel INTEGER
);
```

La tabla ***jugadores*** almacenará a los jugadores que han echado una partida hasta ahora. Sus campos serán: ***jugador_id***, un identificador único para identificar al jugador; ***nombre***, que será el alias del jugador en la interfaz gráfica del juego; ***avatar***, que será una URL o ruta del sistema de archivos hacia una imagen que identifique gráficamente al jugador en el juego; y un ***cinturon***, que no es más que el nivel que ha alcanzado el jugador en el juego por puntos o niveles superados. Por otro lado, la tabla ***partidas*** guardará los datos de todas las partidas jugadas por los diferentes jugadores, y sus campos serán: ***partida_id***, que es el identificador único de cada partida; ***jugador_id***, que

identifica qué jugador en la partida; ***fecha*** de la partida; ***puntos*** conseguidos por el jugador en la partida; y ***nivel*** máximo alcanzado en la partida.

Imaginemos que un nuevo jugador echa una partida: mediante el DML de SQL guardaríamos esa nueva información:

```
INSERT INTO jugadores (jugador_id, nombre, avatar, cinturon)
      VALUES (69, “Ninja Rookie”, “http://img.com/ninja”, 0)

INSERT INTO partidas (partida_id, jugador_id, fecha, puntos, nivel)
      VALUES (3, 69, strftime('%s','now'), 30, 2)
```

Con el comando *INSERT INTO* introducimos los datos del jugador Ninja Rookie con el código identificador 69 en la tabla ***jugadores***, y la partida que ha jugado con el identificador 3 y los 30 puntos obtenidos hasta el nivel 2 del juego se guardan en ***partidas***. Como SQLite no tiene un tipo de datos para fechas, usaremos un número entero, como el número de segundos desde 1970, una forma clásica de contar el tiempo en computación. De ahí la función ***strftime***, que convierte la fecha ***now*** a segundos ***%s*** y lo pasa a entero. Cuando después quisiéramos recuperar los datos del jugador 69, haríamos la consulta:

```
SELECT * FROM jugadores WHERE jugador_id = 69
```

Y si quisiéramos recuperar todas las partidas del jugador 69 haríamos:

```
SELECT * FROM partidas WHERE jugador_id = 69
```

Como puede observarse, SQL es sencillo y potente, solo requiere algo de estudio y práctica. Si tuviésemos todos los datos desperdigados en un archivo del disco sería muy costoso acceder a cualquiera de ellos sin tener que leerlo todo y decodificarlo antes. Con SQL pedimos lo que queremos y el sistema gestor de BBDD nos lo devuelve. Sin embargo, como programadores, no seremos nosotros los que hablemos con la BBDD, será nuestra aplicación. Veamos entonces cómo podríamos crear y usar una base de datos SQLite desde nuestra aplicación Android con las funciones integradas de bajo nivel, es decir, sin utilizar ningún ORM como Room que nos facilite las cosas.

Lo primero es definir las estructuras que almacenarán nuestros datos, las tablas. Para ello heredamos de la clase *SQLiteOpenHelper*, que será la responsable de mantener esas estructuras en el archivo de base de datos:

```
class JuegoDbHelper(context: Context)
  : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {
```

Como vemos, necesita un contexto, el nombre del archivo donde se almacenará la BBDD y la versión actual, que como en nuestro caso será la primera, podríamos definir como 1. Después sobrescribimos la función *onCreate* para definir las tablas que darán forma a nuestra base de datos:

```
override fun onCreate(db: SQLiteDatabase) {
  db.execSQL(SQL_CREATE_JUGADORES)
  db.execSQL(SQL_CREATE_PARTIDAS)
}
```

Se nos pasa un objeto *db* de base de datos y llamamos a su *execSQL*, una función que ejecutará el código SQL que se le pase. Las constantes *string SQL_CREATE_JUGADORES* y *SQL_CREATE_PARTIDAS* que más tarde mostraremos son el código SQL para crear las tablas que deseamos. Si mientras creamos la BBDD desde nuestra aplicación resulta que ya existía otra con una versión anterior, se llamará a *onUpdate*, que será la encargada de actualizar los datos antiguos a la nueva estructura:

```
override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
  db.execSQL(SQL_DELETE_JUGADORES)
  db.execSQL(SQL_DELETE_PARTIDAS)
  onCreate(db)
}
```

Para no complicar el código, y puesto que solo tenemos una versión de nuestras tablas, hacemos lo más sencillo, que es eliminar las tablas antiguas y volverlas a crear con la nueva estructura, sin tener en cuenta que perderemos los datos de la versión previa en caso de que los hubiera. El código completo sería:

```
import android.content.Context
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper

class JuegoDbHelper(context: Context)
  : SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {
  // Se llama cuando la base de datos aun no existe y debe crearse
  override fun onCreate(db: SQLiteDatabase) {
    // Ejecutamos el SQL que crea las tablas
    db.execSQL(SQL_CREATE_JUGADORES)
    db.execSQL(SQL_CREATE_PARTIDAS)
  }
  // Se llama cuando hemos modificado la estructura de las tablas
  // y hemos incrementado el número de la versión pero aún tenemos
  // una BBDD antigua con una versión anterior, aquí se actualizará
  override fun onUpgrade(db: SQLiteDatabase,
                oldVersion: Int, newVersion: Int) {
    // Ejecutamos el SQL que borra las tablas,
    // después el que las crea con la nueva versión
    db.execSQL(SQL_DELETE_JUGADORES)
    db.execSQL(SQL_DELETE_PARTIDAS)
    onCreate(db)
  }
  companion object {
    // Si necesitamos cambiar las tablas una vez que la
```

```
    // app está en producción, debemos incrementar este numero
    const val DATABASE_VERSION = 1
    const val DATABASE_NAME = "juego.db"

    const val TABLE_JUGADORES = "jugadores"
    private const val SQL_CREATE_JUGADORES =
        "CREATE TABLE $TABLE_JUGADORES (" +
            "jugador_id INTEGER PRIMARY KEY," +
           "nombre TEXT NOT NULL UNIQUE," +
            "avatar TEXT," +
            "cinturon INTEGER"
    private const val SQL_DELETE_JUGADORES =
            "DROP TABLE IF EXISTS $TABLE_JUGADORES"

    const val TABLE_PARTIDAS = "partidas"
    private const val SQL_CREATE_PARTIDAS =
        "CREATE TABLE $TABLE_PARTIDAS (" +
            "partida_id INTEGER PRIMARY KEY," +
            "jugador_id INTEGER NOT NULL," +
            "fecha INTEGER NOT NULL," +
            "puntos INTEGER NOT NULL," +
            "nivel INTEGER NOT NULL"
    private const val SQL_DELETE_PARTIDAS =
            "DROP TABLE IF EXISTS $TABLE_PARTIDAS"
  }
}
```

Hemos definido los comandos SQL como constantes *string* para tenerlos bien definidos solo en un lugar y no desperdigados por el código. Ahora podríamos introducir datos en nuestra nueva base de datos. Como programamos con Kotlin, nuestros datos serán objetos de alguna clase. Definamos nuestra clase *Jugador*, asociada a la tabla ***jugadores***:

```
data class Jugador(
        val id: Int,
        val nombre: String,
        val avatar: String?,
        val cinturon: Int?)
```

Nada más que un *data class* de Kotlin con los campos de nuestro jugador. Añadamos una función a la clase para insertar el propio objeto en la BBDD:

```
  fun introducirEnBBDD(db: SQLiteDatabase) {
    // Juntamos todos los campos en una colección ContentValues
    val values = ContentValues().apply {
        put("nombre", jugador.nombre)
        put("avatar", jugador.avatar)
        put("cinturon", jugador.cinturon)
    }
    // Insertamos la fila de datos, y el sistema devuelve el id
    id = db?.insert(JuegoDbHelper.TABLE_JUGADORES, null, values)
  }
```

Esta nueva función *introducirEnBBDD* recibe por parámetro un objeto *SQLiteDatabase* que crearemos mediante nuestro *JuegoDbHelper*. Este objeto de base de datos nos permite insertar, seleccionar, actualizar y borrar. Introducimos los datos en un objeto *ContentValues*, que no es más que un objeto contenedor de pares clave-valor, y

llamamos a la función *insert*, que devolverá el identificador de la fila insertada en la tabla. Para introducir un nuevo jugador, el código sería:

```
val dbHelper = JuegoDbHelper(context)
val db = dbHelper.writableDatabase //Creamos base de datos en modo escritura

val jugador = Jugador(69, "Ninja Rookie", "http://img.com/ninja.png", 0)

jugador.insertarEnBBDD(db)
```

Primero creamos nuestra clase de ayuda para BBDD. Con ese objeto creamos nuestra base de datos *db*. Luego creamos el *Jugador*, y llamamos a su método *insertarEnBBDD* con el objeto *db*.

Como podemos observar, utilizar las funciones de bajo nivel para manejar nuestra base de datos es posible, pero requiere mucho código para cada tabla. Pasar un objeto de Kotlin a una fila de base de datos SQL es incómodo y repetitivo, entre otras cosas porque para cada acción de BBDD necesitamos desgranar los campos del objeto en pares clave-valor. Además, necesitamos conocer el lenguaje SQL para definir las tablas y sus relaciones, y para codificar las consultas de selección, actualización y borrado.

Por fortuna, existe Room, una librería con la que crear y acceder a los datos de nuestra BBDD SQLite de forma fácil, eficiente y orientada a objetos. Veamos cómo utilizar este ORM. Lo primero es añadir la librería en Gradle para que esté disponible en nuestro proyecto:

```
implementation "androidx.room:room-ktx:$room_version"
kapt "androidx.room:room-compiler:$room_version"
```

A continuación definiríamos nuestras clases de Kotlin, al mismo tiempo que definimos las tablas de la base de datos:

```
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "jugadores")
data class JugadorEntity(
  @PrimaryKey val id: Int,
  val nombre: String,
  val avatar: String?,
  val cinturon: Int?
)


@Entity(tableName = "partidas")
data class PartidaEntity(
  @PrimaryKey val id: Int,
  val jugador_id: Int,
  val fecha: Long,
  val puntos: Int?,
  val nivel: Int?
)
```

Cuando el compilador ve las anotaciones de Room *@Entity* sobre nuestras clases, llamará a funciones internas de la librería Room que generarán código SQL para crear la base de datos con sus tablas ***jugadores*** y ***partidas***. De este modo, cuando hagamos operaciones sobre la BBDD, podremos utilizar directamente nuestros objetos Kotlin y no tendremos que traducir parejas de clave-valor. Así hemos definido las estructuras de

datos; ahora definiremos las operaciones sobre los datos. Para ello utilizamos un objeto DAO (Data Access Object) mediante la anotación de Room *@Dao* en nuestra interfaz *JuegoDao*:

```
import androidx.room.*

@Dao
interface JuegoDao {
  //SELECT
  @Query("SELECT * FROM jugadores ORDER BY nombre")
  suspend fun getJugadores(): List<JugadorEntity>
  @Query("SELECT * FROM partidas ORDER BY fecha DESC")
  suspend fun getPartidas(): List<PartidaEntity>
  @Query("SELECT * FROM jugadores WHERE id = :idJugador")
  suspend fun getJugadorById(idJugador: Int): JugadirEntity
  //INSERT
  @Insert(onConflict = OnConflictStrategy.REPLACE)
  suspend fun addPartida(partida: PartidaEntity)
  //DELETE
  @Query("DELETE FROM partidas")
  suspend fun deletePartidas()
}
```

Este es un ejemplo de DAO en el que hemos definido tres funciones de selección, una de inserción y otra de borrado sobre nuestras dos tablas, pero podemos crear fácilmente todas las que sean necesarias. Vemos que sobre cada función existe una anotación de Room que indica el tipo de comando y la orden SQL que ejecutar. También podemos preguntarnos qué significa la palabra clave ***suspend*** que hay delante de la definición de cada función. Esa palabra reservada indica que la función puede suspender la ejecución del código. Entenderemos mejor qué significa en un capítulo posterior, cuando estudiemos el multiproceso. Tengamos en cuenta, simplemente, que el acceso a la BBDD es muy lento, igual que el acceso a un archivo, a la red o a la cámara, por lo que tenemos que usar algún mecanismo por el que la interfaz gráfica de la aplicación siga respondiendo al usuario mientras otro código realiza alguna operación pesada como una consulta a base de datos o una llamada a un servicio web.

Ahora que hemos definido la estructura y la funcionalidad de la base de datos mediante nuestras entidades *JugadorEntity* y *PartidaEntity* y la interfaz DAO, solo nos queda crear la base de datos mediante la anotación *@Database:*

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(
  entities = [JugadorEntity::class, PartidaEntity::class],
  version = 1,
  exportSchema = false)
abstract class JuegoDb : RoomDatabase() {
  abstract val dao: JuegoDao
  companion object {
     private const val NAME = "juego.db"
     fun buildDefault(context: Context) =
        Room.databaseBuilder(context, JuegoDb::class.java, NAME)
           .fallbackToDestructiveMigration()
           .build()
  }
}
```

Para crear nuestra clase de BBDD heredamos de *RoomDatabase*, que declara un objeto interno DAO y una función factoría que devuelve una instancia de la clase. Al crear la base de datos mediante las funciones de Room, podemos utilizar diferentes opciones. Por ejemplo, en lugar de tener una BBDD permanente mediante un archivo, podemos crear una BBDD en memoria que existirá solo mientras la *app* esté activa. También podemos definir la forma en la que migrar los datos cuando cambie la versión, etcétera.

Para conocer todas las opciones de la clase Room, investiga un poco más en:

https://developer.android.com/reference/androidx/room/Room

Y ese es todo el código que necesitamos. Una vez definidas las entidades y la base de datos, es muy sencillo insertar y consultar nuestros datos. Ya podemos olvidarnos de tablas y consultas, todo será programación orientada a objetos:

```
val dao = JuegoDb.buildDefault(application).dao
val jugador = dao.getJugadorById(1)
dao.addPartida(PartidaEntity(0, 1, System.currentTimeMillis(), 10, 1))
```

En la primera línea, creamos una base de datos con *buildDefault* con una referencia a la aplicación como contexto. De la base de datos obtenemos el objeto DAO, que nos servirá para manipular los datos. En la segunda línea obtenemos un objeto *JugadorEntity* desde la base de datos mediante la función del DAO *getJugadorById*. En la tercera, añadimos una línea a la tabla de ***partidas*** mediante la función del DAO *addPartida*.

Utilizando el ORM Room, hemos creado una interfaz que nos aísla de toda la complejidad de usar una BBDD relacional en Android. Para pasar a producción, nuestro ejemplo aún necesitaría añadir algunos mecanismos, como la actualización de los datos a una nueva versión, el control de excepciones, las llamadas asíncronas, etcétera, pero gracias a la funcionalidad y a la abstracción de Room todo será sencillo y ordenado. Para una mejor comprensión y dominio, la documentación de Android es muy interesante, echa un vistazo tanto al núcleo de SQLite en Android como al ORM Room:

Cómo guardar datos en tu aplicación con las funciones de SQLite:

https://developer.android.com/training/data-storage/sqlite

Cómo guardar datos en tu aplicación con la librería Room:

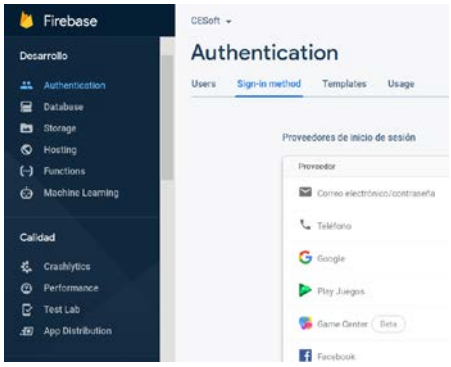
https://developer.android.com/training/data-storage/room

2.9.3. Bases de datos en la nube

En ocasiones, una base de datos local puede no ser suficiente para nuestros propósitos. Imaginemos una aplicación de chat que guarde las conversaciones y los contactos. Si almacenásemos estos datos en una BBDD local, un mismo usuario no podría acceder a ellos desde dos terminales distintos. Unas conversaciones estarían en el móvil y las otras en la *tablet*, y sería un verdadero engorro. Para que el usuario se conecte desde cualquier dispositivo y acceda a todos sus datos indistintamente, tendríamos que realizar un complejo sistema de sincronización entre las bases de datos locales de todos los dispositivos, lo que implicaría un servidor en internet con la base de datos central y un *software* de control muy complejo.

Hoy en día, hay sistemas como los descritos bien diseñados, eficientes y disponibles de forma gratuita y/o de pago. Sus API de desarrollo son bastante sencillas para los programadores, y el usuario solo necesitará una conexión a internet para acceder a sus datos desde cualquier terminal. En ocasiones, estos sistemas permiten el cacheo y el almacenamiento local cuando el dispositivo ha perdido acceso a la red, para luego sincronizarse cuando obtenga conexión de nuevo. Hablamos por ejemplo de servicios en la nube como Firebase o Backendless. Firebase es una potente herramienta diseñada por Google, por lo que parece el candidato ideal para el uso en Android. Las bases de datos Firebase tienen opciones gratuitas y también de pago, como muchos otros servicios en la nube, dependiendo del volumen de tráfico y otros servicios añadidos. Tanto Firebase como Backendless y otros permiten un modo básico gratuito que nos permitirá hacer pruebas o *apps* sencillas.

Las bases de datos de Firebase son no relacionales (o NoSQL), lo que significa que no tienen una estructura de tablas y filas ni utilizan SQL como lenguaje de definición o acceso a datos. Sus tablas son más bien colecciones de objetos con campos anidados de forma jerárquica en estructura de árbol de directorios que tendremos la posibilidad definir como creamos oportuno. Para acceder al servicio de base de datos de Firebase, necesitamos crear una cuenta en la consola de desarrollo disponible en https://console.firebase.google.com. Desde la consola podremos gestionar usuarios, diferentes modos de autenticación de usuario, reglas de seguridad, bases de datos, almacenamiento de archivos, hosting de webs estáticas, funciones, etcétera.



Ya creada la cuenta, podremos hacer *login* en la consola y crear un nuevo proyecto y una base de datos. Firebase ofrece varias alternativas, varios tipos de bases de datos, dependiendo de nuestras necesidades. Una vez definida la base de datos, la consola nos facilitará la descarga de un archivo de configuración llamado google-services.json, que debemos colocar en la carpeta "app" de nuestro proyecto Android. Además, tendremos que añadir las librerías de Firebase necesarias en nuestro build.gradle. Siempre necesitaremos el núcleo de la API definido en *firebase-core*. Si utilizamos la autenticación de usuarios, añadiremos *firebase-auth*. Para la base de datos, *firebase-database*. Si nuestra *app* almacenará archivos de datos como imágenes o archivos de texto, tendremos que importar *firebase-storage*. El build.gradle quedaría:

```
// Firebase
implementation "com.google.firebase:firebase-core:$core_version"
implementation "com.google.firebase:firebase-auth:$auth_version"
implementation "com.google.firebase:firebase-database:$database_version"
implementation "com.google.firebase:firebase-storage:$storage_version"
```

Veamos un pequeño extracto de código de acceso a Firebase. No pretendemos explicar el funcionamiento de la API de Firebase, lo que nos llevaría más tiempo del que disponemos, pero nos ayudará a tener una idea de cómo podemos utilizar este tipo de bases de datos en la nube:

```
import com.google.firebase.auth.FirebaseAuth
import com.google.firebase.database.DatabaseReference
import com.google.firebase.database.FirebaseDatabase

class Fire {
  private val CHATS: String = "ListaChats"
  private val CONTACTOS: String = "ListaContactos"
  data class Chat(val id: String, val otrosDatos: String)
  data class Contacto(val id: String, val otrosDatos: String)
  private var fbdb: FirebaseDatabase? = null

  private fun getDBInstance(): FirebaseDatabase {
    if (fbdb == null) {
      fbdb = FirebaseDatabase.getInstance()
      fbdb!!.setPersistenceEnabled(true)
    }
    return fbdb!!
  }
  private fun getCurrentUserID(): String? {
    val a = FirebaseAuth.getInstance()
    return if(a.currentUser == null) "" else a.currentUser!!.uid
  }
  fun getFirebase() = getDBInstance().reference.child(getCurrentUserID()!!)
  fun newFirebaseChats(): DatabaseReference? = newFirebase().child(CHATS)
  fun newFirebaseContactos():DatabaseReference? = newFirebase().child(CONTACTOS)
  fun guardarChat(chat: Chat) {
    val chats: DatabaseReference = newFirebaseChats()?.child(chat.id)
    chats?.setValue(chat)
  }
  fun guardarChat(chat: Chat) {
    val chats: DatabaseReference = newFirebaseChats()?.child(chat.id)
    chats?.setValue(chat)
  }
}
```

Importamos la clase *FirebaseAuth* del paquete *firebase.auth* y las clases *FirebaseDatabase* y *DatabaseReference* del paquete *firebase.database*. *FirebaseAuth* nos permite acceder a los datos de autenticación de usuarios. Como mencionamos

antes, desde la consola de Firebase podemos establecer diferentes sistemas de autenticación de los usuarios para el acceso a nuestras bases de datos. Uno de las más prácticos es utilizar cuentas de Google, ya que todo usuario de Android tendrá una; pero también admite *email* y contraseña, Facebook y otras. En cualquier caso, una vez que el usuario se haya identificado por cualquiera de los sistemas disponibles, la clase *FirebaseAuth* nos permitirá acceder a los datos de autenticación del usuario.

En este ejemplo, se utiliza el ID único de cada usuario para crear una rama bajo la que se almacenarán sus datos. Para eso utilizamos la función *getCurrentUserID*, que obtiene una instancia de *FirebaseAuth* y accede al campo *uid*. Cuando obtengamos una instancia de *FirebaseDatabase*, podremos acceder al elemento *uid* bajo el que almacenaremos todos los datos de ese usuario. Podríamos organizar los datos de otro modo, pero de esta forma separamos completamente los datos entre usuarios, evitando posibles filtraciones.

Para obtener esa instancia de la base de datos, llamamos a la función *getDBInstance*, cuya principal tarea es llamar al método getInstance de la clase *FirebaseDatabase*. En esta función, además, llamamos a *setPersistenceEnabled*, de este modo le decimos a Firebase que almacene los datos localmente mientras no exista conexión a internet para subirlos a la nube cuando se obtenga cobertura. La función *getFirebase* hace lo que comentamos antes: obtiene una referencia a la base de datos y accede a la rama del usuario. *getFirebaseChat* utiliza la referencia anterior para acceder, dentro de la rama de usuario, a la lista de elementos *Chat*. Imaginamos que nuestra *app* es un servicio de mensajería y necesita almacenar los mensajes y los contactos de chat. Por último, tenemos la función *guardarChat*, que utilizaría la referencia obtenida del método anterior para insertar o sobrescribir un objeto *Chat*.

En resumen, la API de Firebase nos permite acceder a los elementos de una colección agrupados en forma de árbol. En este caso particular, decidimos que las ramas principales fuesen los identificadores de usuario, y debajo de cada una de las ramas de usuario guardaremos las colecciones de chats y contactos de cada usuario. Es una simplificación, y Firebase nos permitirá funcionalidades más completas y complejas, pero es suficiente para hacernos una idea.

La documentación oficial es extensa y se mantiene actualizada. Deberíamos estudiarla detenidamente, pues las herramientas de Firebase se actualizan con frecuencia. El uso de bases de datos en la nube puede llevar nuestra *app* a otro nivel sin necesidad de contratar, configurar y programar un servidor de base de datos en internet, lo que sería costoso y mucho más complejo.

Información sobre el uso de Firebase en una app Android:

https://firebase.google.com/docs/database/android/start

2.10. Modelo de hilos

Ya hemos visto en puntos anteriores la importancia de no bloquear el hilo principal de la aplicación, también conocido como *main thread* o *UI thread*, por ejemplo, en operaciones de I/O, entrada y salida, como cuando accedemos a una base de datos, leemos o escribimos en un archivo de disco, accedemos a la red o a un dispositivo Bluetooth. Todos esos tipos de operaciones, así como cálculos complejos o manejo de grandes cantidades de datos, deberíamos ejecutarlos en paralelo al proceso normal de la aplicación si queremos que funcione adecuadamente. De lo contrario, el sistema Android percibiría que nuestro código no responde adecuadamente ante el usuario y le daría la oportunidad de cerrar nuestra *app*. Ante un retardo en la respuesta de la *app*, mostrará un diálogo llamado ANR o Application Not Responding, y el usuario podrá seguir esperando o cerrar nuestra *app*. Nunca deberíamos crear una *app* tan poco responsiva, pues sería casi tan malo como una *app* que fallase constantemente. Pero no debemos preocuparnos, la solución es sencilla: la programación asíncrona.

Incluso antes de que existiesen los móviles, incluso antes de que los ordenadores tuviesen varios *cores* o multiprocesadores, los desarrolladores de *software* desarrollaron la programación asíncrona. Actualmente, todos nuestros *smartphones* disponen de un procesador con varios núcleos; sería una pena no sacarles mayor partido que ejecutar programas en un solo proceso.

En programación llamamos hilos de proceso a caminos por los que el código puede fluir y ser ejecutado. Cuando creamos un nuevo hilo, nuestro flujo de proceso se bifurca como un río: por un lado, seguirá el hilo principal, y por otro, el código de acceso a la base de datos, por ejemplo. En un punto determinado, tendrán que sincronizarse para pasarse la información necesaria uno al otro. Así será como mantendremos responsivo el hilo principal, y, consecuentemente, la interfaz gráfica de nuestra app, para que el usuario no quede paralizado y pueda seguir interactuando con nuestro código. En Android tenemos bastantes formas diferentes de programación asíncrona, veamos algunas de las más comunes:

- ***Java Thread API**:* la más básica y antigua en Java, que, por supuesto, también podremos utilizar en Kotlin. La vimos en un ejemplo anterior, con la creación de un *Thread* y su ejecución. Es potente y bien conocida, sin embargo, no es la mejor que podremos utilizar en Android. Debido a ser la más básica, debemos conocer muy bien su programación o cualquier pequeño fallo terminará en un problema grave, como un *deadlock* y el cuelgue de nuestra *app*. Es difícil de depurar, y existen en Android opciones mucho más sencillas y potentes.

```
Thread {
  run {
     val res = Primos.formateaPrimos(desde, hasta)
     EventBus.getDefault().post("PrimosService: $res")
     stopSelf()
  }
}.start()
```

- ***AsyncTask**:* es una clase sencilla de utilizar, de ahí su gran uso durante años. Sin embargo, su funcionalidad es algo limitada, y se corre el peligro de crear una laguna de memoria si creas un *AsyncTask* dentro de la clase de una *activity*. Veamos un ejemplo, crearemos una clase *PrimosAsyncTask:*

```
import android.os.AsyncTask
import org.greenrobot.eventbus.EventBus

class PrimosAsyncTask : AsyncTask<Int?, Int?, String>() {
  override fun doInBackground(vararg rango: Int?): String {
     if(rango.size >= 2 && rango[0]!=null && rango[1]!=null) {
        return Primos.formateaPrimos(rango[0]!!, rango[1]!!)
     }
     return ""
  }

  override fun onProgressUpdate(vararg progress: Int?) { }

  override fun onPostExecute(res: String) {
     EventBus.getDefault().post("PrimosAsyncTask: $res")
  }
}
```

  Vemos cómo nuestra clase hereda de *AsyncTask*. La clase está parametrizada, lo que significa que podemos establecer los tipos de datos que usará en sus funciones sobrescritas. El primer parámetro es un entero, que será el tipo de los datos recibidos por *doInBackground*, que es la función que ejecuta la tarea. El segundo, en este caso, es también un entero, que será el parámetro que reciba la función *onProgressUpdate*. Esta función sirve para comunicar el progreso de la tarea, pero en este caso no lo hemos utilizado. Por último, tendríamos un valor *string*, que será lo que devuelva la función *doInBackground* y lo que reciba la función *onPostExecute*, que se ejecutará en el *main thread* cuando haya terminado el proceso de *doInBackground*. Podríamos llamar a esta clase desde cualquier parte, de este modo:

  PrimosAsyncTask().execute(desde, hasta)

  Creamos una instancia de la clase y llamamos a su método *execute*. Interiormente, la clase crearía un nuevo hilo de trabajo en el que ejecutaría la llamada a *Primos.formateaPrimos*, que podría tardar más o menos sin impedir que la interfaz gráfica siguiese ejecutándose en el *main thread*. Como vemos, para tareas sencillas *AsyncTask* puede ser una opción.

- **Programación reactiva**: hace unos años, el concepto de programación reactiva adquirió mucha fuerza en diversos ámbitos del *software*, por lo que es probable que en el futuro aún veamos *apps* que utilizan esta tecnología. En Android se utilizaba la librería RxJava, con o sin el uso de otra librería RxAndroid, que se adaptaba mejor a las necesidades de Android. El concepto Rx se basa en patrón de diseño *Observer*. Básicamente, se trata de un objeto capaz de recibir peticiones de suscripción por parte de otros objetos, a los que avisará cuando cambie de estado. A estos objetos, además, se los dotó de funciones complejas para encadenar estos flujos de eventos. Con pocas líneas en Rx se pueden

completar complejas tareas asíncronas. Sin embargo, la curva de aprendizaje de Rx es muy pronunciada, y en algunos aspectos puede parecerse a programación funcional, de modo que los conceptos que utiliza son muy diferentes a los que estamos acostumbrados en programación orientada a objetos.

- ***Coroutines**:* son la solución recomendada para la programación asíncrona en Android, aunque en sí mismo el concepto es bastante antiguo, utilizado desde hace años en otros lenguajes de programación. Las *coroutines* se fundamentan en las funciones suspendidas. Mediante la suspensión, el código asíncrono se asemeja mucho al flujo de un programa síncrono, facilitando mucho la comprensión del código y su depuración. Las funciones suspendidas indican que su ejecución no bloqueará el flujo actual porque serán ejecutadas en otro hilo, pero son tratadas como si fuesen código síncrono. Podríamos destacar algunas de las características que las hacen el método recomendado para la programación asíncrona:
  - Eficiencia: podríamos ejecutar muchísimas *coroutines* en un solo hilo gracias al sistema de suspensión de la ejecución, que ahorra memoria en comparación con otros mecanismos asíncronos que necesitan bloquear la ejecución.
  - Cancelación: las *coroutines* disponen de sistemas de cancelación de las funciones suspendidas, por lo que no hay que implementar mecanismos extra.
  - Menos fallos, menos lagunas de memoria debido a su fácil estructura.
  - Jetpack: las *coroutines* están integradas en muchas librerías del Jetpack.

Veamos un ejemplo para aclarar cómo funcionan las *coroutines*. Imaginemos que necesitamos realizar un cálculo complejo que podría paralizar la ejecución del *main thread*. Como solo necesitamos realizar el cálculo si nuestra *app* está activa, no utilizaremos un servicio, simplemente un sistema de programación asíncrona. Imaginemos que tenemos un módulo que calcula los números primos en un rango:

```
object Primos {
  var cancelar = false

  fun formateaPrimos(desde: Int, hasta: Int): String {
     val a = calcularPrimos(desde, hasta)
     val sb = StringBuffer()
     for(i in a) {
        sb.append(i)
        sb.append(", ")
     }
     sb.replace(sb.length-2, sb.length-1, ".")
     return sb.toString()
  }
```

```
    private fun calcularPrimos(desde: Int, hasta: Int): List<Int> {
        if(desde >= hasta)
            return listOf()
        cancelar = false
        val primos = mutableListOf<Int>()
        for(i in desde..hasta) {
            if(cancelar)
                return primos
            if(esPrimo(i)) {
                primos.add(i)
            }
        }
        return primos
    }

    private fun esPrimo(numero: Int): Boolean {
        var esPrimo = true
        for(i in 2..numero/2) {
            if(cancelar)
                return false
            if(numero % i == 0) {
                esPrimo = false
                break
            }
        }
        return esPrimo
    }
}
```

Nada de particular, ya lo hemos visto antes. Simplemente, le ofrecemos un par de enteros y nos devolverá una cadena de caracteres con todos los números primos en ese rango. Vamos a crear la interfaz gráfica mediante la definición del *layout:*

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

    <EditText
        android:id="@+id/txtDesde"
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="32dp"
        android:ems="10"
        android:inputType="number"
        android:text="1"
        app:layout_constraintEnd_toStartOf="@+id/txtHasta"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <EditText
        android:id="@+id/txtHasta"
        android:layout_width="100dp"
        android:layout_height="wrap_content"
        android:layout_marginEnd="16dp"
        android:ems="10"
        android:inputType="number"
        android:text="90000"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/txtDesde"
        app:layout_constraintTop_toTopOf="@+id/txtDesde" />

    <Button
        android:id="@+id/btnCalcular"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="24dp"
    android:text="Calcular"
    app:layout_constraintEnd_toStartOf="@id/btnCancelar"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/txtDesde" />

  <Button
    android:id="@+id/btnCancelar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="24dp"
    android:text="Cancelar"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/btnCalcular"
    app:layout_constraintTop_toBottomOf="@+id/txtDesde" />

  <ProgressBar
    android:id="@+id/progressBar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <TextView
    android:id="@+id/txtResultado"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginLeft="16dp"
    android:layout_marginTop="16dp"
    android:layout_marginRight="16dp"
    android:layout_marginBottom="16dp"
    android:scrollbars="vertical"
    android:scrollbarSize="50dp"
    android:text=""
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/btnCalcular" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Veamos cómo llamar desde nuestra *activity* a la función *formateaPrimos* sin congelar el hilo principal. El código de la actividad será:

```
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.lifecycleScope
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.*
...

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

    progressBar.visibility = View.GONE

    btnCalcular.setOnClickListener {
      //GlobalScope.launch {
      lifecycleScope.launch {
        progressBar.visibility = View.VISIBLE

        val desde = txtDesde.text.toString().toInt()
        val hasta = txtHasta.text.toString().toInt()
        txtResultado.text = calcular(desde, hasta)

        progressBar.visibility = View.GONE
```

```
        }
    }

    btnCancelar.setOnClickListener {
        Primos.cancelar = true
        progressBar.visibility = View.GONE
    }
  }

  private suspend fun calcular(desde: Int, hasta: Int): String {
    return withContext(Dispatchers.Default) {
        Primos.formateaPrimos(desde, hasta)
    }
  }
}
```

Como vemos, salvo por unos detalles, el código parece síncrono, como si no utilizase hilos. Simplemente, llamamos a la función *calcular* con el rango establecido por el usuario y el resultado lo asignamos al campo *txtResultado*. Sin embargo, si nos fijamos, veremos que la función *calcular* es una función suspendida. Vemos cómo la función hace uso de *withContext*, que crea un contexto de ejecución diferente: en este caso, utilizando *Dispatchers.Default*, utilizará un hilo de trabajo para llevar a cabo la llamada a *Primos.formateaPrimos*.

Para llamar a una función suspendida, debemos hacerlo desde un contexto de *coroutine*. En este ejemplo hemos utilizado el *lifecycleScope* de la actividad, de modo que, si la actividad termina, también lo hace la *coroutine*, liberándose los recursos necesarios. Podríamos haber utilizado un contexto global con *GlobalScope*, pero tendríamos que estar pendientes de cancelar la *coroutine* cuando fuese necesario, de otro modo, seguiría ejecutándose sin que nos diésemos cuenta, malgastando recursos. Como vemos, las *coroutines* son mucho más fáciles de utilizar que otros tipos de programación asíncrona. Debemos recordar, no obstante, añadir las librerías necesarias en Gradle:

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutines_version"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:$coroutines_version"
implementation "androidx.lifecycle:lifecycle-runtime-ktx:$lifecycle_version"
```

Hay muchos detalles que aprender para usar correctamente las *coroutines:*

https://developer.android.com/kotlin/coroutines

2.11. Comunicaciones: clases asociadas. Tipos de conexiones

Los terminales móviles tienen una gran habilidad para conectarse con otros dispositivos, ya sea mediante llamadas de voz, de VoIP, Bluetooth, USB, internet mediante datos del operador o wifi; algunos incluso disponen de comunicación infrarroja IR, o NFC. Comentemos brevemente algunos de estos tipos de conexiones:

- **Telecom**: el *framework* Telecom de Android administra las llamadas y videollamadas. Eso incluye las llamadas clásicas mediante el *framework* Telephony y las llamadas VoIP mediante la API ConnectionService.

- **Internet**: Ya sea mediante datos del operador de telecomunicaciones o mediante conexión a una red wifi a la que tenga acceso, el dispositivo móvil podrá acceder a internet. Podríamos utilizar la siguiente función para comprobar si disponemos de conexión; de no tener, podríamos pedir al usuario que la activase:

```
private fun hayConexionInternet(context: Context): Boolean {
    var result = false
    val connectivityManager =
context.getSystemService(Context.CONNECTIVITY_SERVICE) as ConnectivityManager
    if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
        val networkCapabilities = connectivityManager.activeNetwork ?: return false
        val capabilities =
connectivityManager.getNetworkCapabilities(networkCapabilities) ?: return false
        result = when {
            capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) -> true
            capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) -> true
            capabilities.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET) -> true
            else -> false
        }
    }
    else {
        connectivityManager.run {
            connectivityManager.activeNetworkInfo?.run {
                result = when (type) {
                    ConnectivityManager.TYPE_WIFI -> true
                    ConnectivityManager.TYPE_MOBILE -> true
                    ConnectivityManager.TYPE_ETHERNET -> true
                    else -> false
                }
            }
        }
    }
    return result
}
```

  Utilizamos los métodos del servicio *ConnectivityManager* para comprobar qué tipos de conexiones están disponibles. Más adelante veremos cómo conectarnos a servicios web y ampliar así la funcionalidad de nuestra *app*.

- **Bluetooth**: las conexiones BT se utilizan en miles de aplicaciones. Podemos usar Bluetooth para enviar sonido desde una *app* hasta nuestros auriculares, o para obtener los datos de nuestra pulsera de ejercicio. Aquí debemos distinguir dos tipos de conexión Bluetooth: la Classic y la Low Energy o LE. En los últimos años, las comunicaciones BT Low Energy han tenido un gran auge debido a la cantidad de pequeños dispositivos conectables con nuestro *smartphone*, que solo

necesitan una cantidad limitada de datos para comunicarse. El Low Energy va montado sobre las funcionalidades del Classic, pero, una vez establecida la conexión, la transmisión de datos es más eficiente energéticamente. Android dispone de las clases necesarias para escanear el entorno en busca de dispositivos BT, parearse con ellos, conectarse y comenzar el intercambio de datos. Para ello, dispondremos de clases como *BluetoothAdapter* y *BluetoothDevice*, localizadas en el paquete android.bluetooth. Más adelante veremos un ejemplo de comunicación Bluetooth.

- **NFC**: Near Field Communication es un conjunto de tecnologías *wireless* de rango muy corto: para iniciar la comunicación, normalmente debe haber un máximo de cuatro centímetros. Sin embargo, a diferencia del Bluetooth, es una comunicación más sencilla y rápida al no necesitar descubrimiento de dispositivos ni emparejamiento alguno. NFC permite compartir datos entre una etiqueta NFC y un móvil con antena NFC, o entre dos móviles de este tipo, o entre el móvil y un sistema de pago, etcétera.

Más información sobre conexiones NFC en dispositivos Android:

https://developer.android.com/guide/topics/connectivity/nfc/nfc

2.11.1. Gestión de la comunicación inalámbrica

En cualquier tipo de comunicación inalámbrica que elijamos, debemos tener presentes ciertos problemas comunes a todos ellos:

- **Estabilidad**: las comunicaciones inalámbricas no utilizan ningún medio para transmitirse, pueden hacerlo en el aire, el agua o el vacío. Por ello, cualquier otra señal ajena a nuestro canal puede interponerse fácilmente entre el emisor y el receptor, degradando la comunicación. Normalmente, todos los sistemas de comunicaciones tienen un sistema corrector de errores en su interior que intentará corregir cualquier dato que haya sido corrompido por el canal, pero, llegado a un límite, la conexión puede perderse y, como programadores, debemos tener estos factores en cuenta. Por ejemplo, podríamos tener un *listener* que reaccionase ante una desconexión, intentando la reconexión.
- **Seguridad**: como hemos dicho, el medio en el que se construye el canal está abierto a cualquiera, no es un cable que haya que conectar. Existe el problema de que un agente malicioso se adhiera al canal, escuchando o enviando datos, haciéndose pasar por otro y comprometiendo la seguridad de la información.

Como programadores, debemos adherirnos a los protocolos y mecanismos que permiten la comunicación segura entre dispositivos: TLS, certificados, etcétera.

- **Eficiencia**: debemos tener en cuenta la velocidad de cada sistema de transmisión, el gasto de batería que supone cada uno, el espacio que necesitarán los datos en memoria y los procedimientos manuales que serán obligatorios para el usuario, entre otras cuestiones. Cada mecanismo se adaptará a cada necesidad mejor que otro, y es nuestra obligación encontrar el mejor en cada caso.

2.12. Búsqueda de dispositivos

En este punto vamos a ver detenidamente cómo conectarnos a otros dispositivos mediante Bluetooth. Antes de empezar, debemos entender el protocolo de búsqueda y emparejamiento entre dispositivos BT. Para que dos dispositivos puedan conectarse, uno de ellos tiene que establecerse en modo de visible y conectable; el otro estará visible y en búsqueda hasta que encuentre al primero. Entonces se llevará a cabo el emparejamiento de los dos dispositivos. El emparejamiento puede ser directo o requerir una contraseña. Una vez emparejados, los dispositivos pueden establecer una conexión y comenzar el intercambio de datos de uno a otro.

Veamos cómo programar una aplicación que busque los dispositivos Bluetooth que tenga a su alcance. Abrimos Android Studio y ejecutamos *File > New > New Project… > Empty Activity > Finish*. Tendremos la actividad *MainActivity* y su *layout activity_main*. Primero vamos a añadir en el build.gradle del módulo de aplicación las siguientes librerías:

```
def coroutines_version = '1.3.7'

// Coroutines
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutines_version"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:$coroutines_version"

// EventBus
implementation 'org.greenrobot:eventbus:3.2.0'

// RecyclerView
implementation 'androidx.recyclerview:recyclerview:1.1.0'
```

Las primeras dos instrucciones *implementation* son para poder utilizar *coroutines* en nuestra *app*, de modo que podamos llamar a funciones de BT sin que el hilo principal se bloquee. Después incluimos *EventBus* para enviar mensajes desde una clase a otra. Por último, incluimos la librería de Android para utilizar las listas reciclables. El *RecyclerView* es un componente de interfaz gráfica que nos permite presentar una lista de elementos de forma eficiente.

Para manipular el dispositivo Bluetooth del terminal, nuestra *app* tendrá que registrar algunos registros en el *manifest*. Además, definiremos una clase *App:*

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

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

  <application
     android:name=".App"
     android:allowBackup="true"
     android:icon="@mipmap/ic_launcher"
     android:label="@string/app_name"
     android:roundIcon="@mipmap/ic_launcher_round"
     android:supportsRtl="true"
     android:theme="@style/AppTheme">
     <activity android:name=".MainActivity">
        <intent-filter>
           <action android:name="android.intent.action.MAIN" />
           <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
     </activity>
  </application>
</manifest>
```

Como la hemos definido en el *manifest*, tendremos que crear una clase *Application* para nuestra aplicación. Pulsemos con el botón derecho sobre el nombre del paquete de la *app* y elijamos *File > New > Kotlin File/Class*. El código es:

```
import android.app.Application

class App : Application() {
  override fun onCreate() {
     super.onCreate()
     _instance = this
  }

  companion object {
     private var _instance: App? = null
     val instance: App
        get() = _instance!!
  }
}
```

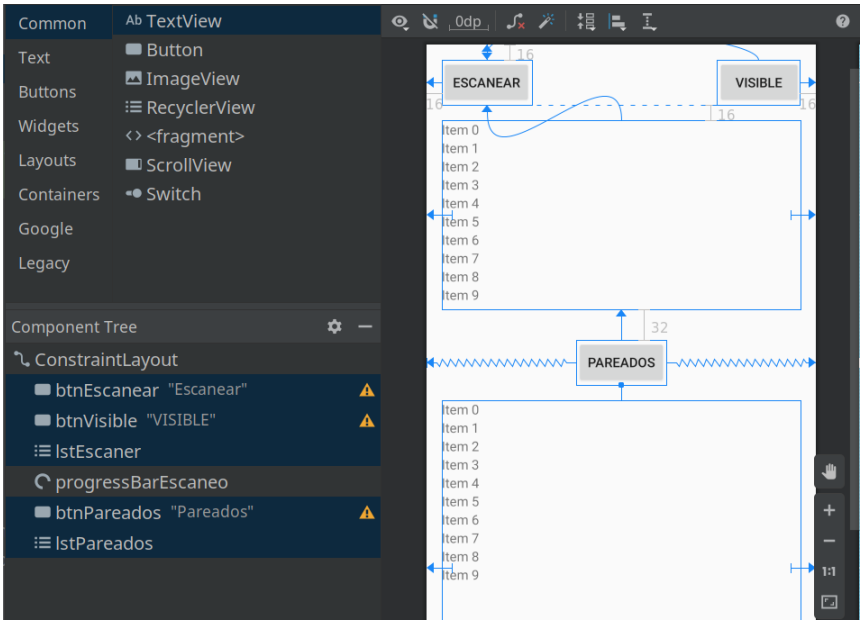
En realidad, solo necesitamos la clase *App* para obtener el contexto global de la aplicación. Hay otros modos de conseguirlo, pero para el ejemplo nos vale.

Ahora podemos empezar a diseñar el *layout*. Añadiremos algunos botones y un par de *RecyclerView* que mostrarán listas de dispositivos Bluetooth:

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

  <Button
     android:id="@+id/btnEscanear"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
```

```
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:text="Escanear"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
  <Button
    android:id="@+id/btnVisible"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="16dp"
    android:text="VISIBLE"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="@+id/btnEscanear" />

  <androidx.recyclerview.widget.RecyclerView
    android:id="@+id/lstEscaner"
    android:layout_width="0dp"
    android:layout_height="200dp"
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="16dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/btnEscanear" />
  <ProgressBar
    android:id="@+id/progressBarEscaneo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="@+id/lstEscaner"
    app:layout_constraintEnd_toEndOf="@+id/lstEscaner"
    app:layout_constraintStart_toStartOf="@+id/lstEscaner"
    app:layout_constraintTop_toTopOf="@+id/lstEscaner" />

  <Button
    android:id="@+id/btnPareados"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="Pareados"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/lstEscaner" />
  <androidx.recyclerview.widget.RecyclerView
    android:id="@+id/lstPareados"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginStart="16dp"
    android:layout_marginTop="16dp"
    android:layout_marginEnd="16dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/btnPareados" />

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

El resultado sería algo como esto:



Antes de ponernos con la actividad, vamos a crear un objeto que llevará a cabo todo el trabajo de activar y escanear dispositivos Bluetooth. Pulsamos con el botón derecho sobre el nombre del paquete de nuestra aplicación y escogemos *New > Kotlin File/Class*. Lo llamaremos *MiBluetooth*, y lo codificaremos así:

```
import android.app.Activity
import android.bluetooth.*
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.content.IntentFilter
import android.util.Log
import androidx.core.content.ContextCompat.startActivity
import org.greenrobot.eventbus.EventBus

object MiBluetooth {

    /// EventBus event class
    class EscaneoTerminado

    private val appContext: Context? by lazy(LazyThreadSafetyMode.NONE) {
        App.instance
    }

    private val _adapter: BluetoothAdapter? by lazy(LazyThreadSafetyMode.NONE) {
        val bluetoothManager =
        appContext!!.getSystemService(Context.BLUETOOTH_SERVICE) as BluetoothManager
        bluetoothManager.adapter
    }
    private val adapter: BluetoothAdapter
        get() = _adapter!!

    val escaneados = ArrayList<BluetoothDevice>()

    val estaDesactivado: Boolean
        get() = !adapter.isEnabled

    fun activar() {
        adapter.enable()
```

```
    }

    fun visibilizar(context: Context, segundos: Int) {
        val discoverableIntent
        = Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE).apply {
            putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, segundos)
        }
        startActivity(context, discoverableIntent, null)
    }

    private val escanearBroadcastReceiver = object: BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent) {
            when(intent.action) {
                BluetoothDevice.ACTION_FOUND, BluetoothDevice.ACTION_NAME_CHANGED -> {
                    val device: BluetoothDevice
                = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)!!
                    escaneados.add(device)
                    EventBus.getDefault().post(device)
                }
                BluetoothAdapter.ACTION_DISCOVERY_FINISHED -> {
                    EventBus.getDefault().post(EscaneoTerminado())
                }
            }
        }
    }
    fun comenzarEscaner(context: Context) {
        val intentFilter = IntentFilter()
        intentFilter.addAction(BluetoothAdapter.ACTION_DISCOVERY_FINISHED)
        intentFilter.addAction(BluetoothDevice.ACTION_NAME_CHANGED)
        context.registerReceiver(escanearBroadcastReceiver, intentFilter)
        adapter.startDiscovery()
    }
    fun finalizarEscaner(context: Context) {
        try {
            context.unregisterReceiver(escanearBroadcastReceiver)
        } catch (e: Exception) { }
    }
    fun listarPareados(): List<BluetoothDevice> {
        return adapter.bondedDevices.toList()
    }
}
```

Podemos observar que *BluetoothAdapter* es la clase más importante de este código, pues está presente en todas las llamadas a Bluetooth. Para obtener una instancia de la clase, llamaremos a:

adapter = appContext.getSystemService(Context.BLUETOOTH_SERVICE).adapter

No debemos preocuparnos por la manera tan sofisticada del código, hace más o menos lo mismo, simplemente utilizamos *by lazy* para no obtener el objeto hasta que no sea necesario. El resto de funciones utilizarán este objeto: por ejemplo, la función *activar* no hace más que *adapter.enable*.

Para escanear los dispositivos Bluetooth cercanos, el proceso es algo más complicado. Primero debemos crear un *BroadcastReceiver*, que es un objeto que recibe llamadas del sistema o de otras clases. Es un mecanismo del sistema Android bastante parecido a *EventBus*, de hecho, podríamos utilizar *BroadcastReceiver* en lugar de *EventBus*, pero este último es más moderno, cómodo y eficiente. Una vez creado el *BroadcastReceiver*, lo programamos para que solo reciba ciertos mensajes mediante el *IntentFilter*. En nuestro caso, nos interesan los mensajes: *ACTION_NAME_CHANGED*, que recibiremos

cuando se detecte un nuevo dispositivo, y *ACTION_DISCOVERY_FINISHED*, que nos llegará cuando el sistema decida que el escáner ha terminado. Una vez preparado nuestro receptor de mensajes, llamaremos a *adapter.startDiscovery*.

Cada vez que recibamos un mensaje *ACTION_NAME_CHANGED*, lo guardaremos en una lista y enviaremos un mensaje *EventBus* para que cualquier otra clase pueda enterarse del evento.

Veamos ahora cómo podemos utilizar esta clase en nuestra actividad:

```
import android.app.Activity
import android.bluetooth.BluetoothDevice
import android.content.Intent
import android.os.Bundle
import android.util.Log
import android.view.View
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.LinearLayoutManager
import kotlinx.android.synthetic.main.activity_main.*
import org.greenrobot.eventbus.EventBus
import org.greenrobot.eventbus.Subscribe
import org.greenrobot.eventbus.ThreadMode

class MainActivity : AppCompatActivity() {

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

    btnVisible.setOnClickListener {
      MiBluetooth.visibilizar(this, CINCO_MINUTOS)
    }

    btnEscanear.setOnClickListener {
      btnEscanear.isEnabled = false
      progressBarEscaneo.visibility = View.VISIBLE
      MiBluetooth.comenzarEscaner(applicationContext)
    }

    btnPareados.setOnClickListener {
      btnPareados.isEnabled = false
      lstPareados.adapter = BluetoothListAdapter(MiBluetooth.listarPareados())
      btnPareados.isEnabled = true
    }

    progressBarEscaneo.visibility = View.GONE
    val layoutManager1 = LinearLayoutManager(this)
    lstEscaner.layoutManager = layoutManager1
    val layoutManager2 = LinearLayoutManager(this)
    lstPareados.layoutManager = layoutManager2
  }

  override fun onResume() {
    super.onResume()
    MiBluetooth.activar()
    EventBus.getDefault().register(this)
  }
  override fun onPause() {
    super.onPause()
    MiBluetooth.finalizarEscaner(this)
    EventBus.getDefault().unregister(this)
  }

  @Subscribe(threadMode = ThreadMode.MAIN)
  fun onBluetoothDevice(device: BluetoothDevice) {
```

```
        lstEscaner.adapter = BluetoothListAdapter(MiBluetooth.escaneados)
    }
    @Subscribe(threadMode = ThreadMode.MAIN)
    fun onBluetoothEscaneoTerminado(escaneo: MiBluetooth.EscaneoTerminado) {
        btnEscanear.isEnabled = true
        progressBarEscaneo.visibility = View.GONE
        lstEscaner.adapter =
            BluetoothListAdapter(MiBluetooth.escaneados)
    }

    companion object {
        private const val CINCO_MINUTOS = 300
    }
}
```

En *onCreate* establecemos las acciones para los botones. Para el botón *Visible*, llamamos a *MiBluetooth.visibilizar*, que pedirá al sistema Android que haga visible el dispositivo actual durante el tiempo especificado. Para el botón *Escanear*, llamamos a la función *MiBluetooth.comenzarEscaner*, que explicamos antes. Para obtener los dispositivos ya pareados, llamamos a *MiBluetooth.listarPareados*. Observemos cómo el resultado se utiliza como parámetro de un objeto que aún no hemos visto, *BluetoothListAdapter*, pero luego hablaremos de él y de los *RecyclerView*.

Tras establecer las acciones para los botones, vemos cómo creamos un *layout* que estableceremos en sendos *RecyclerView*. Esto es así debido a la gran capacidad de adaptación que tiene este control. En nuestro caso, nos basta con el *layout* más sencillo, pues solo le pedimos a cada elemento de la lista que muestre un texto plano.

Después de *onCreate*, tenemos *onResume* y *onPause*. En estas funciones aprovechamos para registrarnos como observadores de mensajes *EventBus*, de modo que podamos escuchar las respuestas de *MiBluetooth*. Además, en *onResume* activamos el Bluetooth de nuestro dispositivo, por si no estuviese activo aún.

Ahora vemos dos funciones de suscripción a mensajes de *EventBus*. Sus nombres no importan realmente, ya que *EventBus* los distingue por los parámetros, de modo que el primero recibe un *BluetoothDevice*, que es el mensaje que obtendremos cuando el sistema nos comunique que ha encontrado un nuevo dispositivo BT, y el segundo mensaje lo recibiremos cuando el escáner haya terminado. Si hubiésemos metido todo el código de *MiBluetooth* en *MainActivity*, nos hubiésemos ahorrado los mensajes *EventBus*, pues nos llegarían mensajes directamente al *BroadcastReceiver*, pero de este modo demostramos que es posible liberar a la *activity* de cualquier tarea, incluso de las que están asociadas a su contexto.

Por último, solo nos queda entender cómo funciona el *BluetoothListAdapter*. Esta clase será la responsable de administrar los elementos de los *RecyclerView*. Veamos su código:

```
import android.bluetooth.BluetoothDevice
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
```

```
class BluetoothListAdapter(
  private val dataSet: List<BluetoothDevice>)

:  RecyclerView.Adapter<BluetoothListAdapter.ViewHolder>() {

  class ViewHolder(v: View) : RecyclerView.ViewHolder(v) {
     val textView: TextView = v.findViewById(R.id.txt)
  }

  override fun onCreateViewHolder(viewGroup: ViewGroup, viewType: Int): ViewHolder {
     val v = LayoutInflater
        .from(viewGroup.context)
        .inflate(R.layout.bluetooth_item, viewGroup, false)
     return ViewHolder(v)
  }

  override fun onBindViewHolder(viewHolder: ViewHolder, position: Int) {
     viewHolder.textView.text = dataSet[position].name
     viewHolder.textView.setOnClickListener {
        val device: BluetoothDevice = dataSet[position]
     }
  }

  override fun getItemCount() = dataSet.size
}
```

Esta clase recibe una lista de objetos, en nuestro caso, una lista de *BluetoothDevices*. La eficiencia de los *RecyclerView* reside en que, para mostrar una lista de cientos de elementos, el componente solo tiene en memoria la representación gráfica de los objetos que pueden verse, y no más. Para ello, define la clase *ViewHolder*, que es la vista de un elemento. Cuando el usuario haga *scroll* sobre la lista, el *RecyclerView* irá reutilizando (reciclando) los *ViewHolder* con los datos de los nuevos elementos ahora visibles. Vemos cómo en *onCreateViewHolder* utilizamos otro *layout* para definir qué aspecto tendrán los elementos de la lista. En nuestro caso, el *layout* del elemento es:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_marginLeft="8dp"
  android:layout_marginRight="8dp"
  android:gravity="center_vertical">
  <TextView
     android:id="@+id/txt"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:textSize="16sp"
     tools:text="Bluetooth Fake Name"
     />
</FrameLayout>
```

Simplemente, un campo de texto que recibirá el nombre del Bluetooth. Como vemos, las funciones relativas al Bluetooth son bastante sencillas gracias a las clases del SDK de Android.

2.13. Búsqueda de servicios

Hemos visto cómo las clases de Android nos facilitan la búsqueda de dispositivos Bluetooth. Una vez conectados al dispositivo, podemos transmitir audio u otros tipos de información sobre el canal Bluetooth. Sin embargo, mantener ese canal continuamente conectado requiere un alto consumo de energía. Para hacer posible la existencia de pequeños dispositivos conectados a nuestro *smartphone*, tuvo que diseñarse una forma de comunicación alternativa que consumiese menos datos y, por lo tanto, menos batería. Hablamos del protocolo Low Energy.

En un dispositivo Bluetooth LE se definen servicios a los que nuestro *smartphone* podrá conectarse y obtener información. Por ejemplo, en una pulsera *fit* podremos conectarnos a sus servicios para obtener las pulsaciones cardiacas y el número de pasos dados en el día, o en un dispositivo *headset* puede haber un servicio de control de volumen, de cambio de canal, etcétera. Cada fabricante puede utilizar sus propios servicios, pero tendrá que registrarlos en bluetooth.org. En la siguiente URL podemos comprobar los actuales servicios de BLE: https://www.bluetooth.com/specifications/assigned-numbers/16-bit-uuids-for-members. Cada registro tiene un identificador único, un UUID, que es un entero de 16 bits.

Veamos un ejemplo de cómo podemos escanear dispositivos BLE y listar sus servicios. Para ello, modificaremos el proyecto del punto anterior. Primero, tendremos que añadir un par de elementos en nuestro *manifest:*

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

<uses-feature
  android:name="android.hardware.bluetooth_le"
  android:required="false" />
```

Es extraño, pero por algún motivo Android asocia el Low Energy con la localización, así que, si necesitamos las funciones de BLE en nuestra *app*, tendremos que requerir el permiso. Además, declaramos el uso de la característica BLE, aunque no lo hacemos obligatorio, por lo que podrá seguir instalándose en móviles sin BLE. Ahora añadiremos el objeto de utilidad:

```
import android.bluetooth.BluetoothAdapter
import android.bluetooth.BluetoothDevice
import android.bluetooth.BluetoothManager
import android.bluetooth.le.*
...

object MiBluetoothLE {
  private const val tag = "MiBluetoothLE"
  private const val REPORT_DELAY = 10000L

  private val appContext: Context? by lazy(LazyThreadSafetyMode.NONE) {
    App.instance
  }

  private val _adapter: BluetoothAdapter? by lazy(LazyThreadSafetyMode.NONE) {
    val bluetoothManager =
```

```
        appContext!!.getSystemService(Context.BLUETOOTH_SERVICE) as BluetoothManager
    bluetoothManager.adapter
}

private val adapter: BluetoothAdapter
    get() = _adapter!!

private val estaDesactivado: Boolean
    get() = !adapter.isEnabled

val tieneBLE: Boolean
    get() = appContext!!
            .packageManager.hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)

fun startScan(onOk: (results: List<BluetoothDevice>)->Unit, onError: (Int)->Unit) {
    stopScan()
    val ssb = ScanSettings.Builder()
        .setScanMode(ScanSettings.SCAN_MODE_LOW_POWER)
        .setReportDelay(REPORT_DELAY)
    val settings = ssb.build()
    val filter = ScanFilter.Builder().setDeviceName(null).build()
    val filters = ArrayList<ScanFilter>()
    filters.add(filter)
    scanCallback.onOk = onOk
    scanCallback.onError = onError
    adapter.bluetoothLeScanner?.startScan(filters, settings, scanCallback)
}

fun stopScan() {
    adapter.bluetoothLeScanner?.flushPendingScanResults(scanCallback)
    adapter.bluetoothLeScanner?.stopScan(scanCallback)
}

private val scanCallback = object: ScanCallback() {
    var onOk: ((results: List<BluetoothDevice>)->Unit)?= null
    var onError: ((Int)->Unit)? = null

    override fun onScanResult(callbackType: Int, result: ScanResult) {
        result.scanRecord?.deviceName?.let {
            onOk?.invoke(listOf(result.device))
        }
    }

    override fun onBatchScanResults(results: List<ScanResult>) {
        val lista = ArrayList<BluetoothDevice>()
        for(item in results) {
            lista.add(item.device)
        }
        onOk?.invoke(lista)
    }

    override fun onScanFailed(errorCode: Int) {
        onError?.invoke(errorCode)
    }
}
}
```

Vemos que las funciones de Bluetooth LE no son muy diferentes a las que vimos en el Bluetooth Classic. De hecho, utilizamos el mismo servicio del sistema *BluetoothAdapter*. Esta vez, para escanear los dispositivos utilizaremos la función *startScan* del objeto *bluetoothLeScanner* que nos proporciona el *adapter*. En realidad, también podríamos realizar la búsqueda con el Classic y mirar uno por uno si el dispositivo encontrado es Classic, LE o Dual. Se dice que un dispositivo Bluetooth es Dual cuando soporta ambos protocolos. Veamos cómo sería el *layout* de la nueva *activity:*

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

  <Button
     android:id="@+id/btnEscanear"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_marginStart="16dp"
     android:layout_marginTop="16dp"
     android:text="@string/escanear"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="parent" />

  <androidx.recyclerview.widget.RecyclerView
     android:id="@+id/lstEscaner"
     android:layout_width="0dp"
     android:layout_height="200dp"
     android:layout_marginStart="16dp"
     android:layout_marginTop="16dp"
     android:layout_marginEnd="16dp"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toBottomOf="@+id/btnEscanear" />

  <ProgressBar
     android:id="@+id/progressBarEscaneo"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     app:layout_constraintBottom_toBottomOf="@+id/lstEscaner"
     app:layout_constraintEnd_toEndOf="@+id/lstEscaner"
     app:layout_constraintStart_toStartOf="@+id/lstEscaner"
     app:layout_constraintTop_toTopOf="@+id/lstEscaner" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Simplemente, un botón y un *RecyclerView*. Veamos cómo los utiliza la actividad:

```
import android.Manifest
import android.app.AlertDialog
import android.bluetooth.BluetoothDevice
import android.widget.Toast
import kotlinx.android.synthetic.main.activity_ble.btnEscanear
import kotlinx.android.synthetic.main.activity_ble.lstEscaner
import kotlinx.android.synthetic.main.activity_ble.progressBarEscaneo
...

class BLEActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_ble)
     btnEscanear.setOnClickListener {
        progressBarEscaneo.visibility = View.VISIBLE
        MiBluetoothLE.startScan(::onOk, ::onError)
     }
     val layoutManager = LinearLayoutManager(this)
     lstEscaner.layoutManager = layoutManager
     lstEscaner.adapter = BluetoothListAdapter(listOf(), {})
     progressBarEscaneo.visibility = View.GONE
  }
  private fun onOk(results: List<BluetoothDevice>) {
     progressBarEscaneo.visibility = View.VISIBLE
     lstEscaner.adapter = BluetoothListAdapter(results, {})
     progressBarEscaneo.visibility = View.GONE
```

```
  }
  private fun onError(error: Int) {
     progressBarEscaneo.visibility = View.GONE
     Toast.makeText(this, R.string.error_scan_ble, Toast.LENGTH_LONG).show()
  }
  override fun onResume() {
     super.onResume()
     btnEscanear.isEnabled = MiBluetoothLE.tieneBLE
     checkPermissionsForBluetoothLowEnergy()
  }
  private fun checkPermissionsForBluetoothLowEnergy() {
     if(checkSelfPermission(Manifest.permission.ACCESS_FINE_LOCATION)
           != PackageManager.PERMISSION_GRANTED)
        requestPermissions(arrayOf(Manifest.permission.ACCESS_FINE_LOCATION), 1)
  }
}
```

Establecemos la acción para el botón, en la que llamamos a la función *startScan* de *MiBluetoothLE;* iniciamos la *layoutManager* y el *adapter* para el *RecyclerView;* y ocultamos la *progressBar*. Cuando la operación de escaneo retorne con la lista de dispositivos, creamos un nuevo *adapter* y lo asignamos a la lista *lstEscaner*. Vemos cómo en *onResume* activamos el botón de escáner si el dispositivo soporta BLE. Además, comprobaremos que nuestra *app* tenga el permiso de localización, pues, si no, las funciones BLE fallarán.

Si quisiéramos hacer un listado de los servicios, necesitaríamos conectarnos con un código como:

```
bluetoothGatt = device.connectGatt(
            appContext,
            true,
            gattCallback,
            BluetoothDevice.TRANSPORT_LE)

private var bluetoothGatt: BluetoothGatt? = null
private val gattCallback = object : BluetoothGattCallback() {
  override fun onCharacteristicChanged(
        gatt: BluetoothGatt,
        characteristic: BluetoothGattCharacteristic) {
     Log.e(tag,"gattCallback:onCharacteristicChanged:${characteristic.uuid}")
  }
  override fun onConnectionStateChange(gatt:BluetoothGatt, status:Int, newState:Int) {
     super.onConnectionStateChange(gatt, status, newState)
     Log.e(tag,"gattCallback:onConnectionStateChange:device=${gatt.device}")
     when (newState) {
        BluetoothProfile.STATE_CONNECTED -> {
        }
        BluetoothProfile.STATE_DISCONNECTED -> {
        }
     }
  }
  // New services discovered
  override fun onServicesDiscovered(gatt: BluetoothGatt, status: Int) {
     when (status) {
        BluetoothGatt.GATT_SUCCESS -> {
           Log.e(tag, "onServicesDiscovere : GATT_SUCCESS")
           for (service in gatt.services) {
              Log.e(tag,"${service.uuid},${service.type},${service.instanceId}")
           }
        }
     }
  }
}
```

2.14. Establecimiento de la conexión. Cliente-servidor

En la actualidad, el *smartphone* es un dispositivo muy potente, tiene una capacidad de procesamiento que hace años se hubiese pensado imposible en un dispositivo tan pequeño. Sin embargo, un móvil, de la misma manera que un ordenador de sobremesa, no puede guardar toda la información del mundo, pero puede acceder a ella a través de internet sobre todo o de cualquier otra forma, y, normalmente, en todas esas comunicaciones encontraremos un patrón cliente-servidor.

El cliente es nuestro navegador web, que pide su contenido a los distintos servidores por los que navega. Cliente es nuestra *app* de mensajería cuando se comunica con el servidor para enviar o recibir mensajes. Incluso en los juegos en red tenemos la *app* instalada como cliente y el servidor que sincroniza el mundo de los jugadores entre todos ellos.

En realidad, el servidor no tendría por qué ser tampoco una máquina potente escondida en algún lugar de internet. El patrón cliente-servidor también sirve para la comunicación entre dos iguales, por ejemplo, dos móviles. Podría ser sobre internet, o sobre Bluetooth, o sobre cualquier otro medio. Simplemente, es un marco que facilita el establecimiento y control de la conexión. Uno de los pares actuaría de cliente, que iniciará la conexión, mientras que el otro esperará pacientemente peticiones de conexión desde el cliente.

El establecimiento de la conexión del cliente con cada servidor dependerá de los protocolos del canal que se utilicen. Por ejemplo, para los niveles más bajos de internet podemos utilizar TCP o UDP. El servidor abriría lo que se conoce como *socket*, escuchando peticiones desde una dirección IP y un puerto preestablecido, y el cliente abriría otro *socket* que se conectaría con el del servidor. Estemos seguros de que Android dispondrá de las clases y *frameworks* necesarios para ayudarnos en su desarrollo. También dispondremos de librerías que nos facilitarán el trabajo. En los puntos posteriores veremos diferentes tipos de comunicación.

2.14.1. Envío y recepción de mensajes de texto. Seguridad y permisos

Cuando todavía los dispositivos móviles no tenían la capacidad de conectarse a internet, los mensajes cortos de texto o SMS eran de gran utilidad. Aún hoy se utilizan tanto para comunicación como para verificación de identidad, etcétera. Vamos a ver cómo podríamos enviar y recibir SMS desde nuestra *app* Android. Lo primero sería pedir los permisos necesarios en el *manifest:*

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

y en la *activity:*

```
private fun compruebaPermiso() {
  if(ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS) !=
PackageManager.PERMISSION_GRANTED) {
     if (ActivityCompat.shouldShowRequestPermissionRationale(this, Manifest.permission.SEND_SMS)) {
        Toast.makeText(this, "Esta app necesita el permiso para enviar SMS...",
Toast.LENGTH_LONG).show()
     }
     else {
        ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.SEND_SMS),
REQUEST_ENVIO_SMS)
     }
  }
...
```

Veamos cómo sería la función para enviar:

```
fun enviaSMS(telefono: String, texto: String) {
  val smsManager = SmsManager.getDefault()
  smsManager.sendTextMessage(telefono, null, texto, null, null)
}
```

Tan simple como esto. Obtenemos una instancia de la clase *SmsManager* y llamamos a *sendTextMessage* con el teléfono y el mensaje como parámetros. El mensaje se enviaría de inmediato. El único problema es que no recibiríamos aviso de que se haya enviado ni recibido. Para obtener esta confirmación, necesitaríamos un *BroadcastReceiver* para saber si se ha enviado y otro para saber si se ha recibido por el destinatario. Veamos cómo sería:

```
private val brEnviado = object : BroadcastReceiver() {
  override fun onReceive(arg0: Context?, arg1: Intent?) {
     when(resultCode) {
        Activity.RESULT_OK ->
           Toast.makeText(baseContext, "SMS enviado", Toast.LENGTH_SHORT).show()
        SmsManager.RESULT_ERROR_GENERIC_FAILURE ->
        Toast.makeText(baseContext, "Fallo desconocido", Toast.LENGTH_SHORT).show()
        SmsManager.RESULT_ERROR_NO_SERVICE ->
           Toast.makeText(baseContext, "Sin servicio", Toast.LENGTH_SHORT).show()
        SmsManager.RESULT_ERROR_NULL_PDU ->
           Toast.makeText(baseContext, "Null PDU", Toast.LENGTH_SHORT).show()
        SmsManager.RESULT_ERROR_RADIO_OFF ->
           Toast.makeText(baseContext, "Radio off", Toast.LENGTH_SHORT).show()
     }
  }
}
private val brEntregado = object : BroadcastReceiver() {
  override fun onReceive(arg0: Context?, arg1: Intent?) {
     when (resultCode) {
        Activity.RESULT_OK ->
          Toast.makeText(baseContext,"SMS delivered", Toast.LENGTH_SHORT).show()
        Activity.RESULT_CANCELED ->
          Toast.makeText(baseContext,"SMS not delivered", Toast.LENGTH_SHORT).show()
     }
  }
}
```

Y ahora, utilizaríamos esos *BroadcastReceiver* en la llamada *sendTextMessage:*

```
fun enviaSMS(telefono: String, texto: String) {
  registerReceiver(brEnviado, IntentFilter(ACCION_ENVIADO))
  registerReceiver(brEntregado, IntentFilter(ACCION_ENTREGADO))
  //
  val sms = SmsManager.getDefault()
  val piEnviado = PendingIntent.getBroadcast(this, 0, Intent(ACCION_ENVIADO), 0)
  val piEntregado = PendingIntent.getBroadcast(this, 0, Intent(ACCION_ENTREGADO), 0)
  try {
     sms.sendTextMessage(telefono, null, texto, piEnviado, piEntregado)
  }
  catch (e: Exception) {
     Log.e(tag, "Error:e:", e)
     Toast.makeText(applicationContext, e.toString(), Toast.LENGTH_LONG).show()
  }
}
```

Nada complicado. Veamos ahora cómo recibir SMS. Lo primero, los permisos en *manifest:*

```
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="android.permission.READ_SMS" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
```

Además, necesitamos definir un *BroadcastReceiver* para que el sistema nos avise de la llegada de un mensaje SMS:

```
<receiver
  android:name=".ReceptorSMS"
  android:enabled="true"
  android:exported="true">
  <intent-filter android:priority="1000">
     <action android:name="android.provider.Telephony.SMS_RECEIVED" />
  </intent-filter>
</receiver>
```

La clase *ReceptorSMS* hereda de *BroadcastReceiver*, y sería algo como:

```
class ReceptorSMS : BroadcastReceiver() {
  @TargetApi(Build.VERSION_CODES.M)
  override fun onReceive(context: Context?, intent: Intent) {
     val bundle = intent.extras
     val msgs: Array<SmsMessage?>
     var strMessage = ""
     val format = bundle!!.getString("format")
     val pdus = bundle["pdus"] as Array<Any>?

     if(pdus != null) {
        val isVersionM = Build.VERSION.SDK_INT >= Build.VERSION_CODES.M
        msgs = arrayOfNulls(pdus.size)
        for(i in msgs.indices) {
           msgs[i] = SmsMessage.createFromPdu(pdus[i] as ByteArray, format)
           strMessage += "SMS desde " + msgs[i]?.originatingAddress
           strMessage += " :${msgs[i]?.messageBody}"
           EventBus.getDefault().post(strMessage)
        }
     }
  }
```

Como hemos visto, normalmente Android nos comunicará eventos del sistema mediante los *BroadcastReceiver*. No hay mucho más, tras recibir el SMS, podríamos analizar el mensaje para el uso que deseemos o, simplemente, mostrarlo al usuario.

2.14.2. Envío y recepción de mensajes multimedia. Sincronización de contenidos. Seguridad y permisos

Como hemos visto en el punto anterior, el envío y recepción de mensajes cortos de texto es bastante sencillo, y puede utilizarse para la identificación de usuario o sencillamente para comunicar un mensaje. El envío y recepción de mensajes multimedia no es mucho más complicado, si bien su uso es escaso. Los MMS, como sabemos, pueden incluir imágenes además de texto. Pero veamos por encima qué clases y funciones necesitaríamos en el caso de necesitar manejar este tipo de mensajes. La clase *Telephony* es la responsable de manejar todas las actividades relativas al teléfono, como la lista de APN, SMS y MMS, llamadas, etcétera. La *app* deberá crear y registrar un *BroadcastReceiver* para obtener notificaciones del sistema de tipo *Telephony.Sms.Intents.WAP_PUSH_DELIVER_ACTION*, que indicará que se ha recibido un MMS. También debemos tener en cuenta que tanto SMS como MMS son almacenados en las bases de datos del sistema. De este modo, tampoco sería necesario registrar un *BroadcastReceiver*, sino que podríamos leer directamente esas tablas con los mensajes recibidos.

La documentación completa de la clase *Telephony:*

https://developer.android.com/reference/android/provider/Telephony

2.14.3. Tratamiento y control de conexiones HTTP y HTTPS

Las conexiones a internet que utilizan la mayoría de las *apps* hoy en día utilizan los protocolos HTTP y HTTPS. En principio, el protocolo HTTP se ideó como una capa superior al TCP/IP que serviría para compartir documentos con enlaces de hipertexto por los que fuese fácil navegar. Fue el surgimiento de la web, el sistema de comunicación que más difundió el uso de internet en todo el mundo. El protocolo HTTP permitía que el navegador, una aplicación cliente que entendía y formateaba documentos HTML, hiciese peticiones a servidores en internet, y estos devolvían los documentos que guardaban en sus discos duros. Pero, con el tiempo, los servidores web empezaron a ser más complejos. Ya no solo devolvían documentos estáticos, sino que podían componerlos en el momento de la petición, compilando información de sus bases de datos o de otros servidores conectados. Los servidores web pasaron de ser archivadores

de documentos a aplicaciones web que servían información dinámicamente en respuesta a consultas dinámicas del cliente. De una petición de documento como http://servidor.com/doc9.html pasaron a consultas web como http://servidor.com/mis_docs.php?userId=1969&doc=9. Fue el surgimiento de las aplicaciones web, y muchas empresas empezaron a dar servicio a sus clientes devolviendo estos documentos dinámicos.

No obstante, los servidores continuaron su avance y pronto pasaron de servir documentos a usuarios humanos a servir datos a clientes automáticos. Pasaron de albergar aplicaciones web a servicios web. Técnicamente, no es un gran paso; simplemente, se trata de formatear los datos de una forma más estructurada para que los clientes pudiesen parsearla, es decir, traducirla. Pasaron de devolver documentos HTML a devolver información formateada en JSON, XML o cualquier otro formato que pudiera ser entendido por una máquina.

La razón por la que el protocolo HTTP es tan usado en las *apps* móviles es debido a la potencia de los actuales servicios web. Mediante servicios web, por ejemplo, una *app* puede informar al usuario sobre la temperatura que hace en su zona: la *app* pide al sistema que le informe sobre la posición GPS del terminal; entonces la *app* se conecta a internet y hace una petición a un servidor web meteorológico; este consulta sus datos y transforma la latitud y longitud en una temperatura, la formatea y la devuelve a la *app;* finalmente, la *app* formatea y muestra al usuario la información. Existen millones de posibilidades, y millones de *apps* que utilizan millones de servicios web. Incluso las entidades financieras se unieron a esta tecnología. Con el surgimiento del HTTP seguro o HTTPS, los dispositivos pueden validar la autenticidad de los mensajes y encriptar las comunicaciones con los servidores web.

Android nos permite comunicarnos a través de internet con los protocolos más básicos: *sockets* de TCP/IP o UDP. Pero como no queremos reinventar la rueda, utilizaremos alguna de las muchas librerías disponibles que construyen por nosotros la pila de protocolos necesarios para llegar a utilizar servicios web. En la práctica, al menos de momento, las librerías más utilizadas son OkHttp como capa base y, sobre ella, Retrofit o Volley, que nos ayudarán aún más a convertir peticiones HTTP en meras funciones que podremos llamar cómodamente desde nuestro código. Para transformar la información, podremos utilizar librerías como GSON, que traduce JSON a un objeto en Kotlin.

Pero mejor veamos un ejemplo. Crearemos un nuevo proyecto de Android *Studio File > New > New project… > Empty Activity*. Primero, vamos a añadir las librerías que necesitamos en build.gradle:

```
def coroutines_version = '1.3.7'
def okhttp_version = '4.7.2'
def retrofit_version = '2.9.0'
def gson_version = '2.8.1'

// Coroutines
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutines_version"
```

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:$coroutines_version"
implementation "androidx.lifecycle:lifecycle-runtime-ktx:2.3.0-alpha05"

// Network
implementation "com.squareup.okhttp3:okhttp:$okhttp_version"
implementation("com.squareup.okhttp3:logging-interceptor:$okhttp_version")
implementation "com.squareup.retrofit2:retrofit:$retrofit_version"
implementation "com.squareup.retrofit2:converter-gson:$gson_version"
```

Hemos añadido las librerías para las *coroutines*, porque las llamadas a internet pueden tardar y no queremos paralizar el *main thread*. Después añadimos las librerías para acceso a internet OkHttp y Retrofit. Como haremos uso de internet, debemos declarar el permiso en el *manifest:*

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

En esta aplicación vamos a utilizar el servicio web que nos ofrece GitHub. Vamos a definir solo una función de las muchas que sirve. Lo que vemos a continuación es una especie de descripción que le pasaremos a Retrofit para que sepa cómo hablar con el servicio de GitHub:

```
import retrofit2.Response
import retrofit2.http.*

interface GithubApi {
  @GET("/repositories")
  suspend fun getRepoList(): Response<List<RepoEntity>>
}
```

Básicamente, le estamos diciendo a Retrofit que utilizaremos un servicio web con una sola función, y que se accede a ella mediante una petición *GET /repositories*. Podemos ver los datos que devuelve el servicio en nuestro navegador con la URL https://api.github.com/repositories. El protocolo HTTP dispone de varias acciones, y la más utilizada es *GET*, que solicita el documento que se le pide. Existen otras como *PUT* y *POST* para subir datos, *DELETE* para borrar, etcétera. Ahora veamos cómo utilizar esta interfaz, una vez que hemos informado a Retrofit sobre el servicio que utilizar:

```
import kotlinx.coroutines.sync.Mutex
import kotlinx.coroutines.sync.withLock
import okhttp3.Cache
import okhttp3.OkHttpClient
import okhttp3.logging.HttpLoggingInterceptor
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object GithubApiImpl {
  private const val GITHUB_API_URL = "https://api.github.com/"

  private val mutex = Mutex()
  var lastErrorCode: Int = 0

  private val api: GithubApi by lazy {
    return@lazy Retrofit.Builder()
      .baseUrl(GITHUB_API_URL)
      .client(createHttpClient())
      .addConverterFactory(GsonConverterFactory.create())
      .build()
      .create(GithubApi::class.java)
  }

  private fun createHttpClient(): OkHttpClient {
```

```
    val interceptor= HttpLoggingInterceptor(object : HttpLoggingInterceptor.Logger {
        override fun log(message: String) {
            Log.e(tag, "HttpLoggingInterceptor: $message")
        }
    })
    return OkHttpClient.Builder().build()
  }

  suspend fun getRepoLista(): List<RepoEntity>? {
    mutex.withLock {
        val res = api.getRepoList()
        return if(res.isSuccessful) {
            lastErrorCode = 0
            res.body()
        }
        else {
            lastErrorCode = res.code()
            null
        }
    }
  }
}
```

El objeto más importante es la instancia de *GithubApi* que creamos a partir de la interfaz de Retrofit. Establecemos la URL del servicio, el cliente OkHttp, que controlará el flujo del protocolo HTTP, y el parseador de datos, en este caso, GSON. Por último, desarrollamos la función que realmente nos interesa, *getRepoLista*. Esta función utiliza el objeto generado por Retrofit para llamar al servicio. Aquí hemos utilizado un objeto *Mutex*, que no es imprescindible pero ayuda a que no nos aburramos demasiado. Cuando programamos código asíncrono, es imprescindible tener en cuenta si una función puede llamarse varias veces al mismo tiempo. Si una función puede llamarse una vez más antes de que haya terminado, debemos tener cuidado de que el código que hay dentro sea coherente. En este caso, nos aseguramos de que no pueda llamarse varias veces, porque el objeto *Mutex* bloquea las llamadas entrantes hasta que la llamada actual no haya terminado. Salvo por eso, la función es muy simple: comprueba que la respuesta del servicio sea correcta y, si lo es, devuelve el objeto parseado automáticamente por Retrofit.

Vemos cómo Retrofit traduce automáticamente esos datos devueltos por el servidor web de GitHub a un objeto que aún no hemos definido. Es tan fácil porque utiliza la librería GSON para hacer el trabajo duro. Nosotros no tendremos más que definir el objeto con algunas anotaciones para que GSON sepa cómo iniciar, con los datos que le llegan, cada campo del objeto:

```
import androidx.annotation.Keep
import com.google.gson.annotations.SerializedName

@Keep
data class RepoEntity(
  @SerializedName("id")
  val id: String?,
  @SerializedName("name")
  val name: String?,
  @SerializedName("description")
  val description: String?
)
```

No tenemos por qué utilizar todos los campos que nos devuelve el servicio, solo los que nos interesen. El parseador GSON olvidará el resto. Veamos cómo utilizar esta clase desde nuestra *activity:*

```
import androidx.lifecycle.lifecycleScope
import androidx.recyclerview.widget.LinearLayoutManager
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.launch

class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
     val layoutManager = LinearLayoutManager(this)
     lista.layoutManager = layoutManager
  }

  override fun onResume() {
     super.onResume()
     lifecycleScope.launch {
        val res = getRepoListSame()
        if(res != null)
           lista.adapter = RepoListAdapter(res)
     }
  }

  private suspend fun getRepoListSame(): MutableList<RepoEntity>? {
     val res = GithubApiImpl.getRepoLista()
     return if (res != null) {
        MutableList(res.size) { i -> res[i] }
     }
     else null
  }
}
```

Simplemente llamamos al método *getRepoLista* de *GithubApiImpl*. La lista de objetos la utilizamos para crear un *list adapter* que asociar con el *RecyclerView*. El *layout* será únicamente ese *RecyclerView* ocupando toda la ventana.

Como hemos visto, con la ayuda de Retrofit hemos diseñado un mecanismo para consultar al servicio web y traducir su respuesta en un objeto con el que trabajar en nuestro código.

2.15. Complementos de los navegadores para visualizar el aspecto de un sitio web en un dispositivo móvil

Muchas empresas comprendieron en su día la importancia vital de tener su lugar en la web. Muchas aún no se han dado cuenta de la necesidad comercial que supone tener una *app* para *smartphones*, pero, gracias a las analíticas de acceso a la web, han comprendido que gran parte de los usuarios de sus webs acceden mediante *tablets* o móviles. Si no se hace nada para evitarlo, normalmente una web diseñada para ordenadores de sobremesa no se verá adecuadamente en la pequeña pantalla de un dispositivo móvil; en ocasiones, la web será inutilizable desde el móvil. Sin embargo, hay

soluciones: con un buen diseño, cualquier página o aplicación web puede adaptarse a pantallas pequeñas de diversos tamaños. Es lo que se conoce como *responsive design* o diseño responsivo.

Existen webs que permiten comprobar si nuestra página es responsiva:

https://search.google.com/test/mobile-friendly

Para desarrollar una web responsiva, debemos tener en cuenta algunos puntos:

- Utilizar unidades relativas de tamaño, nunca píxeles: debemos utilizar tamaños porcentuales (%) o estándar (em) para el tamaño de los componentes, fuentes e imágenes. De este modo, todos los componentes tendrán un tamaño relativo al de la pantalla.
- Utilizar la etiqueta *viewport* en la cabecera: esta etiqueta controla el aspecto del contenido HTML en el navegado del móvil. Esto hará que la web admita el ancho máximo de la pantalla. Suele utilizarse algo como:

  <meta name="viewport" content="width=device-width, initial-scale=1">
- Podemos utilizar la clase *@media* de CSS3: esta clase nos permite utilizar las propiedades de la pantalla del terminal desde el código CSS, de modo que podamos establecer un diseño u otro en función del tamaño. Algunas de las propiedades más importantes son: *orientation*, que indica la orientación actual, ya sea *portrait* o *landscape; height* y *width*, el alto y ancho del navegador; *device-height* y *device-width*, el alto y ancho de la pantalla, etcétera.

Si no somos unos expertos en CSS, siempre podemos utilizar la ayuda de algún *framework* que nos permita avanzar rápidamente en el desarrollo de una web bien diseñada y responsiva. Por ejemplo, tenemos Skeleton, aplicación bastante sencilla de utilizar y disponible de forma gratuita en www.getskeleton.com.

2.16. Pruebas y documentación

Las pruebas son necesarias durante y al finalizar cualquier desarrollo *software*. Pueden ser más o menos sofisticadas, puede haber un departamento de QA que las realice o quizá solo el mismo programador, pueden estar estructuradas en diferentes tipos o simplemente probar la *app* como un usuario más. En cualquier caso, Android Studio nos ayudará mucho en la realización de todo tipo de pruebas. En Android tenemos

básicamente dos tipos de pruebas: las pruebas unitarias *(unit tests)*, que prueban módulos puros de lenguaje sobre la máquina virtual, es decir, módulos que no tengan llamadas al sistema de Android, y las pruebas instrumentales *(instrumental tests)*, que comprueban módulos más complejos con llamadas al sistema operativo y que deben ser ejecutadas sobre un terminal, físico o emulador.

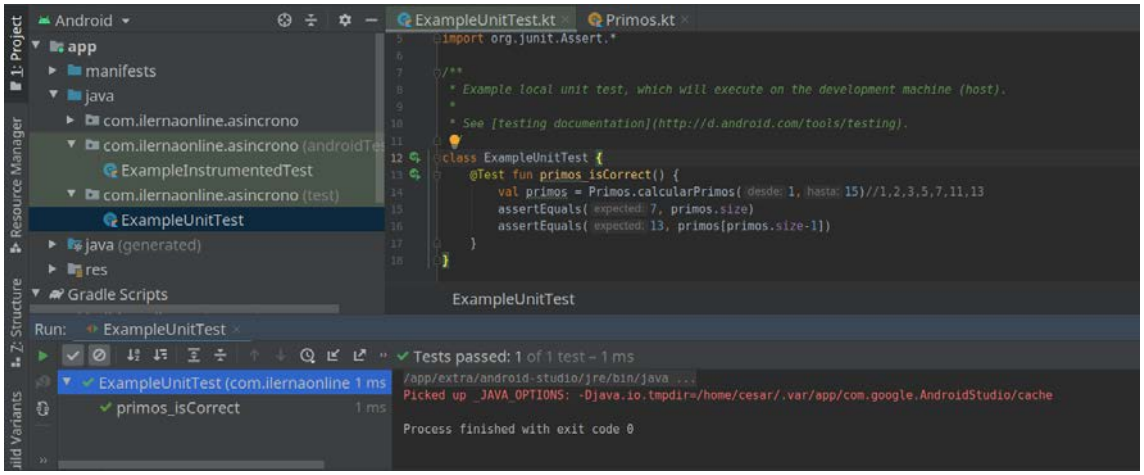
En ambos tipos pueden utilizarse, además, otras herramientas que pueden facilitarnos las tareas. Por ejemplo, dado que las pruebas unitarias son más rápidas y efectivas de ejecutar, quizá querríamos utilizar siempre este tipo. ¿Pero qué ocurre si nuestro código tiene algunos objetos del sistema, como una *activity*? Podríamos utilizar *mocks* o clases que imitan ser otras más complejas, de modo que al final el código de prueba no tenga ninguna llamada real al sistema. Existen *frameworks* como Mockito que nos ayudan en esas tareas. Para las pruebas instrumentales, también disponemos de ayudas, como las herramientas Espresso y UI Automator.

2.16.1. Pruebas unitarias

En Android Studio, las pruebas unitarias se guardan en el directorio *modulo/src/test/java*. Este tipo de pruebas se ejecutan en la máquina virtual de Java (JVM) instalada en nuestro ordenador, por lo que se ejecutan rápidamente. Su única desventaja es que no pueden hacer llamadas al sistema Android, de modo que, si nuestro código las tiene, deberemos emularlas, por ejemplo, con Mockito. Veamos un ejemplo:

```
class ExampleUnitTest {
  @Test fun primos_isCorrect() {
     val primos = Primos.calcularPrimos(1,15)//1,2,3,5,7,11,13
     assertEquals(7, primos.size)
     assertEquals(13, primos[primos.size-1])
  }
}
```

En este caso, pasaría el test de esta forma:

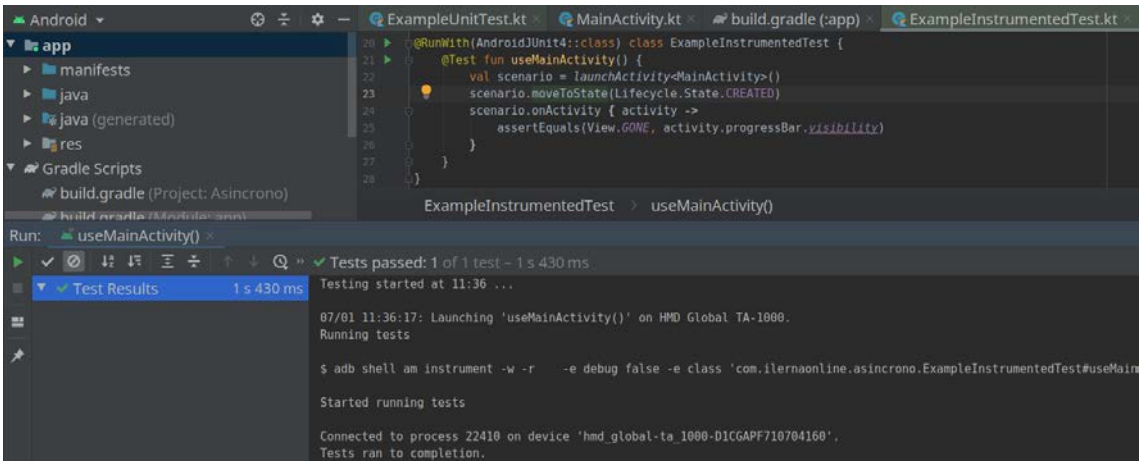


2.16.2. Pruebas instrumentales

Cuando no nos basta con comprobar el código puro, sino que deseamos comprobar cómo funciona junto con el sistema Android, entonces tendremos que utilizar pruebas instrumentales. Un ejemplo sería cuando deseemos probar las funciones de la interfaz gráfica, o de una parte del *hardware*. Estas pruebas se ejecutan en un dispositivo móvil o emulador con su sistema operativo Android. Al tener que instalarse en el dispositivo cada vez que se ejecutan las pruebas, son menos eficientes que las pruebas unitarias, pero más reales. Veamos un ejemplo:

```
@RunWith(AndroidJUnit4::class) class ExampleInstrumentedTest {
  @Test fun useMainActivity() {
     val scenario = launchActivity<MainActivity>()
     scenario.moveToState(Lifecycle.State.CREATED)
     scenario.onActivity { activity ->
        assertEquals(View.GONE, activity.progressBar.visibility)
     }
  }
}
```

En este caso, pasaría el test de esta forma:



Al utilizar algunas funciones de testeo, quizá tengamos que añadir librerías a nuestro build.gradle. En este ejemplo, para usar el *launchActivity* necesitamos:

```
// Tests
androidTestImplementation 'androidx.test:core-ktx:1.2.0'
```

2.16.3. Documentación

Documentar el código es una tarea importante. Aunque una estructura y un código Clean es la mejor documentación, no está de más añadir toda la información que podamos a nuestro proyecto. En programación Java, existieron múltiples herramientas para la generación automática de documentación. Kotlin lleva menos tiempo, pero ya disponemos de algunas buenas herramientas también, por ejemplo, Dokka, que es el generador de documentación para Kotlin desarrollado por JetBrains. La documentación

automática suele depender de comentarios formateados adecuadamente dentro de nuestro código. La herramienta parseará el código en busca de estos comentarios y generará la documentación. Veamos cómo podría ser nuestro código para que la herramienta de documentación hiciese su trabajo:

```
/**
* Esta sería la documentación de la clase NuestraActivity,
* podríamos explicar para que vale, etc.
*
* @param layout: Esta sería la explicación del parámetro layout.
*/
class NuestraActivity(layout: Int) : Activity {
  /**
   * Este sería un mensaje explicando la utilidad de una propiedad callback.
   */
  private val callback: Callback()
  /**
   *  Este sería un mensaje explicando cómo y para qué tenemos este método.
   *
   * @return Application: Mensaje explicando el objeto devuelto por el método.
   */
  fun app() = getApplication<Application>()

  ...
}
```

La documentación completa de cómo utilizar este documentador:

https://github.com/Kotlin/dokka

3. Utilización de librerías multimedia integradas

Las aplicaciones multimedia son unas de las más utilizadas y descargadas de los *stores*. Gracias a su reducido tamaño y su capacidad multimedia, los terminales móviles han hecho que radios y dispositivos reproductores de MP3 portátiles queden obsoletos cuando salimos fuera de casa. Y también dentro utilizamos sistemas operativos móviles, ya sean *tablets* o *smart TV*, para consumir películas y series desde alguna plataforma de *streaming*. Las capacidades multimedia de los sistemas móviles no solo nos permiten desarrollar aplicaciones de audio y vídeo, sino también extender el alcance de otro tipo de aplicaciones, como *apps* de redes sociales o de mensajería, que proporcionan videollamadas, *broadcasts* de vídeo, reproducción de mensajes de audio, etcétera.

Los desarrolladores de sistemas operativos móviles comprenden la importancia de estas funcionalidades y proporcionan a los programadores potentes librerías multimedia integradas en el SDK. En este tema estudiaremos las capacidades de las librerías multimedia del sistema Android. Son estas suficientemente capaces como para no necesitar librerías de terceros y, al estar programadas por los propietarios del sistema, estarán siempre actualizadas y bien ajustadas a los dispositivos en los que las encontraremos.

Veremos algunos ejemplos de cómo podemos emplear el multimedia para mejorar las capacidades de nuestras *apps*. Estudiaremos algunas de las clases más interesantes, como MediaPlayer, SoundPool o MediaRecorder y comprobaremos qué características las hacen más convenientes en según que casos de uso. Comprender sus funcionalidades nos permitirá en el futuro escoger los mecanismos más adecuados para llevar a cabo la tarea que necesitemos en nuestras propias *apps*.

3.1. Conceptos sobre aplicaciones multimedia

En el desarrollo de *apps* multimedia para dispositivos móviles debemos tener en cuenta varios factores, como puedan ser las diversas capacidades de cada terminal. Existe una amplia variedad de dispositivos diferentes, con diferentes resoluciones y tamaños de pantalla, distintos tipos de procesadores, diversas capacidades de memoria, diferentes códecs soportados, etcétera. Además, los usuarios pueden utilizar diferentes periféricos, como teclados, *joysticks*, auriculares Bluetooth o con cable, micros y cámaras. Por ejemplo, los auriculares Bluetooth pueden llevar botones de *play*, *pause*, canal, *push to talk* o control de volumen, y tendremos que tratarlos adecuadamente. También deberemos tener en cuenta que estos dispositivos pueden ser desconectados en cualquier momento durante la reproducción o grabación.

Los sistemas operativos móviles tratan de estandarizar, mediante las librerías multimedia, sus clases e interfaces, todas las diferentes funcionalidades del *hardware*, ocultando además las posibles diferencias entre terminales. De este modo, el programador podrá centrarse en su tarea en lugar de codificar mecanismos que mantengan la estabilidad de la *app* ante diferencias de *hardware*. Pero debemos tener en cuenta también que la evolución del *hardware* y el *software* de los dispositivos móviles ha sido tan violenta que incluso en las librerías encontraremos diferentes clases para llevar a cabo las mismas funciones. Es decir, que, mientras programamos, nos encontraremos con diferentes clases o interfaces que han ido quedando obsoletas, así como interfaces que funcionarán para un versión moderna del sistema operativo pero no para una más antigua. Este es el caso de los paquetes Camera, Camera2 y CameraX, por ejemplo.

Manejar el *hardware* de la cámara a bajo nivel, o la reproducción de vídeo, no es una tarea sencilla en ningún caso, y aunque el SDK nos facilite el trabajo, tendremos la obligación de entender bien la arquitectura interna del sistema para no cometer fallos que impidan a nuestra *app* dar lo mejor de sí misma. Algo que ya sabíamos también es que el manejo de imágenes requiere mucha memoria. Imaginemos entonces la memoria y el tiempo de CPU que requieren el tratamiento de imágenes en movimiento, y más aún el vídeo en *streaming*, que debe reproducirse en tiempo real. Tendremos que tener en cuenta las limitaciones de memoria de los terminales, así como los protocolos de compresión que soportan.

Por último, debemos considerar las fuentes de los datos multimedia, cómo acceder a esas fuentes de la manera más eficiente. Los datos están en el almacenamiento interno, en el externo, en internet o llegarán desde los sensores del terminal, la cámara o el micrófono.

En los temas siguientes estudiaremos la arquitectura del *software* necesario para desarrollar aplicaciones multimedia eficientes. Más adelante estudiaremos las clases, interfaces y métodos más utilizados para cada tarea. Finalmente, veremos algunos ejemplos prácticos de cómo utilizar las librerías multimedia integradas en el SDK de Android.

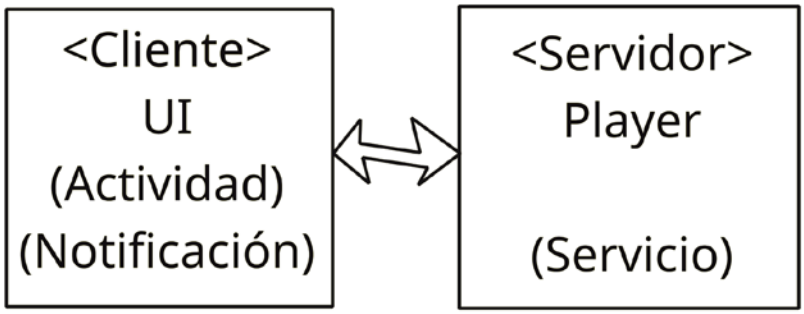
3.2. Arquitectura del API utilizado

Antes de comenzar el desarrollo de cualquier sistema de *software*, el análisis y diseño de la arquitectura es uno de los factores más importantes. En el caso de las *apps* multimedia, es vital tener los conceptos claros sobre cómo se estructuran los módulos que las componen. Tengamos en cuenta, por ejemplo, la diferencia entre una aplicación reproductora de audio y otra de vídeo. Mientras el usuario escucha música desde

nuestra *app*, puede estar trabajando con un editor de textos o chateando por una *app* de mensajería. Si en un momento dado necesita detener la reproducción o cambiar la canción, puede lanzar comandos a nuestra aplicación, ya sea mediante la activación de nuestra *activity* o mediante botones en una notificación que habremos lanzado en la barra de notificaciones. Entendemos por ello que nuestra *app* de audio se ejecutará como un servicio en segundo plano. Sin embargo, si el usuario está viendo un videoclip en otra de nuestras *apps*, es lógico suponer que nuestra *app* tendrá simplemente una vista enlazada al reproductor y se ejecutará solo en primer plano. En el caso de una *app* captadora de vídeo o audio, podrían plantearse diferentes alternativas. Normalmente, si grabamos vídeo, necesitaremos una *app* en primer plano que muestre en pantalla lo que captamos a través de la cámara. Si adquirimos audio, podríamos plantearnos que el servicio corra en *background* o no, dependiendo del análisis de necesidades que queramos cubrir. Veamos cada caso en los siguientes puntos.

3.2.1. Arquitectura de una *app* de audio

El sistema Android promueve la división de las aplicaciones de reproducción de audio en dos módulos bien diferenciados. Por una parte, tendríamos el módulo de control, que mediante algún tipo de interfaz gráfica permite al usuario lanzar comandos como *play*, *pause*, siguiente pista, pista anterior, etcétera. Por otro lado, tendríamos un servicio corriendo en *background*, responsable de reproducir cada pista de audio según las acciones que puedan llegarle desde el módulo control. De este modo, tendríamos una arquitectura clásica cliente-servidor. En este caso, la parte de control e interfaz gráfica sería el cliente, el servidor sería puramente el reproductor de audio.



Además, con el uso de algunas clases de Android podríamos conseguir que el sistema funcionase con más de un cliente. Imaginemos que no solo deseamos controlar la pista actual que se está reproduciendo en el servidor mediante los controles de nuestra *app*, sino que, además, deseamos permitir que otra *app* instalada en un *smartwatch* pueda controlar el servidor. El sistema Android nos facilita el trabajo mediante el uso de las clases *MediaBrowser* y *MediaBrowserService*. Si no necesitamos que otras aplicaciones o módulos de terceros accedan a nuestro sistema para reproducir audio, podríamos utilizar un servicio mucho más sencillo, heredando directamente de la clase *Service*.

El equipo de Android nos muestra un ejemplo de reproductor de audio:

https://github.com/android/uamp

3.2.2. Arquitectura de una *app* de vídeo

Una *app* que muestre vídeos será en principio más sencilla, ya que no tendrá sentido separar los controles del propio reproductor. Normalmente, el reproductor estará enlazado a una ventana en la que volcará las imágenes del vídeo. No obstante, también deberíamos diferenciar en el código qué parte es la interfaz gráfica y qué otra parte se dedica al control del media, como la carga, la codificación, etcétera. En el caso de una *app* reproductora de vídeo, además, la parte gráfica tiene diferentes posibilidades para enlazar la salida del reproductor hacia la interfaz gráfica. Podríamos, por ejemplo, utilizar un elemento *VideoView* en nuestro *layout* y cargar en él los vídeos directamente, utilizando un *MediaController* para permitir que el usuario pueda controlar la reproducción a su gusto.

```
<VideoView
  android:id="@+id/videoView"
  android:layout_width="match_parent"
  android:layout_height="match_parent" />
```

El código para cargar los vídeos en *VideoView* sería:

```
val videoView = findViewById<VideoView>(R.id.videoView)

// Opción 1: Archivo en disco  (Nota: android.permission.WRITE_EXTERNAL_STORAGE)
val clip = File(Environment.getExternalStorageDirectory(), "test.mp4")
videoView.setVideoPath(clip.path);

// Opción 2: Archivo en directorio res/raw (Nota: no debemos escribir la extensión)
videoView.setVideoURI(Uri.parse("android.resource://$packageName/raw/test"))

// Opción 3: Archivo en Internet (Nota: android.permission.INTERNET)
videoView.setVideoPath("http://videocdn.bodybuilding.com/video/mp4/62000/62792m.mp4")

val mediaController = MediaController(this)
mediaController.setMediaPlayer(videoView)
videoView.setMediaController(mediaController)
videoView.requestFocus()
videoView.start()
```

En el caso de acceder a archivos de vídeo en el disco o a vídeos en internet, no debemos olvidarnos de añadir los permisos correspondientes en el *manifest:*

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

Esta sería la opción más sencilla, pero no permitiría tanto control como utilizando directamente los objetos MediaPlayer o ExoPlayer. Más adelante veremos arquitecturas más complejas que nos permitirán un mayor control sobre la reproducción de vídeo.

3.2.3. *App* captadora de audio o vídeo. Ejemplos

Si nuestra *app* no tiene como objetivo la reproducción sino la captación de media, su arquitectura no tendrá por qué guardar relación directa con las arquitecturas vistas hasta ahora. Deberemos tener en cuenta, sin embargo, otros factores, por ejemplo, si nuestra *app* utilizará otras aplicaciones como interfaz hacia el *hardware* o si, por el contrario, accederá directamente a la cámara o al micro. Si nuestra *app* no pretende ser lo último en grabación, quizá sea conveniente llamar directamente a la grabadora instalada por defecto en nuestro móvil, con lo que ahorraremos esfuerzo y tiempo de desarrollo. Este podría ser el caso de una *app* que gestione tareas y permita además grabar notas de voz o vídeo. Quizá resultaría innecesario para una tarea tan sencilla meternos a programar el *hardware* de la cámara y el sonido. No obstante, si realmente nuestra *app* se basa en la captación de media, querremos tener total control sobre el proceso y tendremos que estar al tanto de las interfaces y arquitectura de la cámara o el micro.

Más adelante veremos cómo utilizar el *hardware* con Camara2, estudiaremos su arquitectura y las clases necesarias, pero por ahora veamos la forma más sencilla de capturar vídeo. Utilizaremos una *app* preinstalada en el sistema que se anuncie como grabadora de vídeo y le solicitaremos que grabe uno para nosotros. Lo primero que necesitaremos es un *FileProvider*. Por motivos de seguridad, Android no nos permite pasar rutas de una *app* a otra de forma directa; de ser así, la otra *app* podría escribir o borrar lo que quisiera dentro de nuestro directorio de aplicación. Mediante un *FileProvider* podremos compartir archivos de forma segura entre diferentes *apps*. En el *manifest*, dentro de la etiqueta *application*, definiremos nuestro *provider* de la siguiente manera:

```
<provider
  android:name="androidx.core.content.FileProvider"
  android:authorities="${applicationId}.provider"
  android:exported="false"
  android:grantUriPermissions="true">
  <meta-data
    android:name="android.support.FILE_PROVIDER_PATHS"
    android:resource="@xml/provider_paths"/>
</provider>
```

El *FileProvider* hace referencia a un documento XML que definirá nuestras rutas. Crearemos una carpeta "xml" dentro de la carpeta "res" y añadiremos el archivo provider_paths.xml con el siguiente código:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
 <files-path name="v" path="videos" />
</paths>
```

Ahora podremos llamar a una aplicación que permita la grabación de vídeo, es decir, a una aplicación preinstalada que responda a una acción *MediaStore.ACTION_VIDEO_CAPTURE*. Nuestro código sería:

```
import android.content.Intent
import android.net.Uri
import android.provider.MediaStore
import androidx.core.content.FileProvider
import java.io.File

class Actividad: Activity() {
  private val AUTHORITY = BuildConfig.APPLICATION_ID + ".provider"
  private val VIDEOS = "videos"
  private val FILENAME = "grabacion.mp4"
  private val REQUEST_ID = 6969
  private var outputUri: Uri? = null

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

  private fun grabar() {
    val output = File(File(filesDir, VIDEOS), FILENAME)
    if(output.exists())
      output.delete()
    else
      output.parentFile?.mkdirs()

    outputUri = FileProvider.getUriForFile(this, AUTHORITY, output)
    val intent = Intent(MediaStore.ACTION_VIDEO_CAPTURE)
    intent.putExtra(MediaStore.EXTRA_OUTPUT, outputUri)
    intent.putExtra(MediaStore.EXTRA_VIDEO_QUALITY, 1)
    intent.addFlags(Intent.FLAG_GRANT_WRITE_URI_PERMISSION
        or Intent.FLAG_GRANT_READ_URI_PERMISSION)
    startActivityForResult(intent, REQUEST_ID)
  }

  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    if (requestCode == REQUEST_ID && resultCode == RESULT_OK) {
      val view = Intent(Intent.ACTION_VIEW)
        .setDataAndType(outputUri, "video/mp4")
        .addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
      startActivity(view)
      finish()
    }
  }
}
```

Vemos cómo arrancamos la *app* con *startActivityForResult*, de modo que, cuando la *activity* de la *app* grabadora termine, nos devolverá el archivo de vídeo. En *onActivityResult* llamaremos a la *app* por defecto para mostrar vídeo. También podríamos mostrar el vídeo en nuestra propia *app*, como hicimos en el punto anterior. Utilizando actividades externas, no hay más que hacer. Salvo por la complicación de configurar el *FileProvider* para pasar el archivo de una *activity* a la otra, nuestra *app* será así de sencilla. Sin embargo, al utilizar una *app* externa para la captura, no tendremos la oportunidad de controlar las opciones de vídeo, el formato, el tamaño máximo, la calidad de imagen, etcétera.

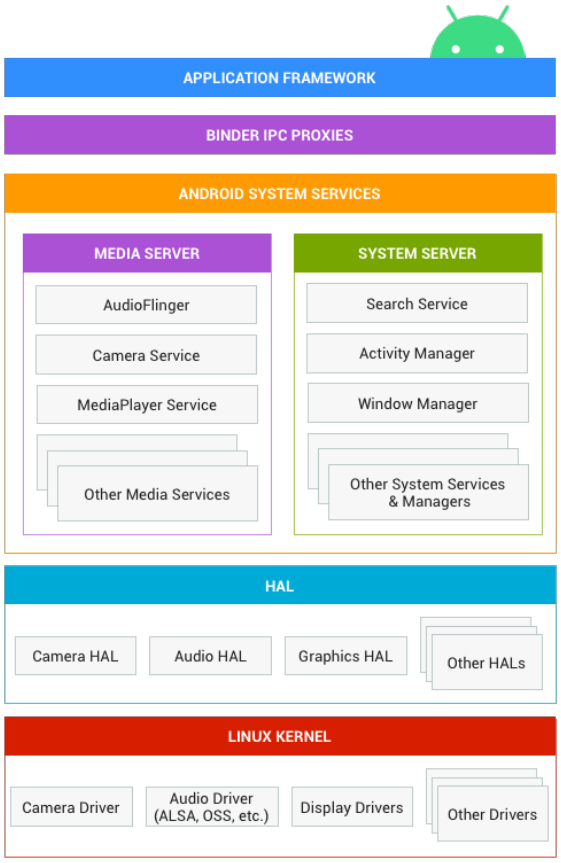
En el caso de una *app* captadora de audio, utilizar una *app* externa sería igual de sencillo que en el caso anterior. Veamos un ejemplo:

```
import android.content.Intent
import android.provider.MediaStore

class Actividad : Activity() {

  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     val intent = Intent(MediaStore.Audio.Media.RECORD_SOUND_ACTION)
     startActivityForResult(intent, REQUEST_ID)
  }

  override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
     if (requestCode == REQUEST_ID && resultCode == RESULT_OK) {
        Toast.makeText(this, "Grabación terminada", Toast.LENGTH_LONG).show()
     }
     finish()
  }

  companion object {
     private const val REQUEST_ID = 1337
  }
}
```

Pero esto no son más que soluciones rápidas a un problema complejo. Tendremos muchos más recursos si comprendemos las librerías disponibles que nos ofrece el sistema, antes que utilizar las pocas funcionalidades que anuncian otras aplicaciones ya instaladas.

3.2.4. Arquitectura del sistema

Vamos a aprovechar esta sección para comentar brevemente la arquitectura del propio sistema Android, porque, aunque no sea imprescindible, nos será de ayuda al hablar de algunos conceptos de bajo nivel, como los códecs. Como vemos en la imagen, el *framework* de aplicación es el nivel más alto: es el SDK, con el que los programadores de *apps* nos comunicaremos. El IPC sirve para que podamos acceder a los servicios de Android con mayor facilidad.



Tenemos dos tipos de servicios: los servicios del sistema y los servicios de los media. Aquí podemos encontrar la cámara, el MediaPlayer y otros. Estos servicios, a su vez, se comunican con el HAL. El *hardware abstraction layer* o HAL es una definición estándar de las llamadas al *hardware* que los fabricantes de móviles deben cumplir para que el sistema Android pueda funcionar en ellos. Más abajo tenemos el núcleo de Linux, que es el código central de todo el sistema operativo. Ahí encontraremos los *drivers* que controlarán la cámara, la pantalla, el micrófono y el sonido. Cuando desde nuestra *app* preguntemos al sistema sobre un códec, es posible que haga todo este recorrido hacia abajo al interior del sistema antes de respondernos.

3.3. Descripción e instalación de las librerías multimedia

Android es un sistema operativo potente y eficaz. En las primeras versiones, el SDK era muy básico, y los programadores necesitaban reinventar la rueda o utilizar las pocas librerías de terceros disponibles. Pero el desarrollo Android ha cambiado mucho desde sus inicios. Hoy en día, el SDK ha evolucionado mucho, y sus capacidades son enormes. Para el manejo del multimedia, Android dispone de librerías tan potentes que en la mayoría de los casos no necesitamos nada más para implementar nuestros proyectos. La mayoría de estas clases las podremos encontrar en el paquete android.media.*.

En este punto es imprescindible hablar sobre la clase MediaPlayer. Este objeto nos permitirá la reproducción de audio y vídeo de forma sencilla a la vez que eficaz. Admite gran cantidad de formatos distintos y es eficiente en relación con la memoria y el consumo de batería. Como es una librería del sistema, no necesitamos ningún tipo de instalación para comenzar a utilizarla en nuestro código. La versatilidad de MediaPlayer, unida a la funcionalidad que pueden facilitar otras clases del SDK adheridas a ella, la hacen idónea para cualquier tarea multimedia de nuestra *app*. Sin embargo, aunque funcione bien en cualquier tipo de proyecto, en algunos casos tendremos disponibles clases más adecuadas a una funcionalidad concreta.

Con respecto a la reproducción de audio, aparte de MediaPlayer tendremos otras clases más sencillas a la vez que eficientes en casos concretos. Algunos ejemplos serían: SoundPool, AudioTrack y ToneGenerator. Veamos cuál es el fuerte de cada una de ellas, para utilizarlas en lugar del comodín que puede resultar en ocasiones MediaPlayer:

- **SoundPool**: tiene la capacidad de solapar múltiples fuentes de sonido, dando prioridad a cada fuente. De este modo, pueden lanzarse diferentes fuentes de sonido al mismo tiempo sobre SoundPool. La clase será responsable de decidir qué sonido ser reproduce o no, y mezclando los que resulten escogidos, de forma que el resultado sea una mezcla natural. Esta clase será imprescindible, por ejemplo, en el caso de que programemos un juego. En un juego pueden lanzarse muchos sonidos a la vez: el ruido de pisadas, el ruido de armas disparando, el chillido de alienígenas a punto de mordernos, etcétera. SoundPool controlará la mezcla de todos ellos para que el resultado sea eficiente y coherente con el juego. Si detectamos que el móvil en el que está instalada la *app* tiene pocos recursos, podemos indicar a SoundPool que utilice dos canales solamente, de modo que, en todo momento, solo dos de los múltiples sonidos que le llegan serán mezclados, liberando tiempo de CPU.

- **AudioTrack**: es una clase de bajo nivel. Su poder reside en la eficiencia a la hora de reproducir rápidamente fuentes de audio que ya han sido decodificadas. Si tenemos un archivo de música en MP3, será más sencillo para nosotros utilizar MediaPlayer, que se hará cargo de la decodificación sin mayor trabajo por nuestra parte, pero si la velocidad al reproducir unos pocos sonidos que guardaremos en memoria, ya decodificados en PCM *(pulse code modulation)* es nuestra prioridad, entonces utilizaremos AudioTrack.

- **ToneGenerator**: en el supuesto de que los sonidos que necesitamos sean tan sencillos como un par de frecuencias en forma de pitido, utilizaremos esta clase. Es la forma de reproducir sonido más sencilla y eficaz, ya que no hace falta almacenar ningún dato en memoria. El generador simplemente hará utilizar su circuitería para generar impulsos de frecuencias preprogramadas. El único problema es su simplicidad.

```
val toneGenerator = ToneGenerator(AudioManager.STREAM_MUSIC, 100)
toneGenerator.startTone(android.media.ToneGenerator.TONE_SUP_RINGTONE)
//...
toneGenerator.startTone(android.media.ToneGenerator.TONE_PROP_BEEP)
```

Con respecto a la reproducción de vídeo, apenas tendremos alternativas a MediaPlayer dentro del SDK. Por ello, vamos a comentar aquí la poderosa herramienta ExoPlayer, que, si bien no pertenece al SDK de Android, está bien integrada dentro del sistema y es muy sencilla de incluir en nuestros proyectos. Podemos utilizar las funciones de alto nivel de ExoPlayer, que llevará a cabo todo el trabajo duro por nosotros, pero ExoPlayer también nos permite configurar y ajustar todos los parámetros de la reproducción mediante sus interfaces de más bajo nivel. Además, dispone de algunas funcionalidades de las que no dispone MediaPlayer, sobre todo las relacionadas con el *streaming* de vídeo y audio. Por ejemplo, esta librería soporta el Dynamic Adaptative Streaming sobre HTTP (DASH), el Smooth, el SmoothStreaming o el Common Encryption. Hoy en día, este tipo de tecnologías son imprescindibles para el funcionamiento de canales de vídeo por *streaming* tan populares como Netflix, Amazon y otros. Veamos cómo podemos usar ExoPlayer en nuestro proyecto, añadiendo unas líneas a nuestro archivo de configuración build.gradle:

```
// Incluir todas las funcionalidades de ExoPlayer
implementation 'com.google.android.exoplayer:exoplayer:2.10.4'
// O solo algunas de ellas
implementation 'com.google.android.exoplayer:exoplayer-core:2.11.5'
implementation 'com.google.android.exoplayer:exoplayer-ui:2.11.5'
//...
```

Como vemos, podemos utilizar simplemente la primera línea e incluirlo todo. También podremos reducir el tamaño de nuestra *app* si solo añadimos los componentes que vayamos a utilizar en nuestro proyecto en el resto de líneas. Más adelante veremos cómo utilizar algunas de las funcionalidades de ExoPlayer en un ejemplo práctico.

3.4. Fuentes de datos multimedia. Clases

3.4.1. Formatos de audio y vídeo

El sistema Android proporciona de forma estándar soporte para algunos formatos de vídeo y audio, es decir, viene con algunos códecs preinstalados por defecto. Como programadores, estos son los que nos interesa utilizar si nuestra intención es que la *app* funcione en todos los dispositivos disponibles. Si nuestro interés se centra únicamente en una marca y modelo, o en una versión específica de Android, entonces podríamos utilizar algunos códecs más que quizá solo estén disponibles en ese terminal. Desde la versión 10 de Android (nivel de API 29) y superiores, disponemos de algunos métodos interesantes en la clase *MediaCodecInfo* que pueden ayudarnos:

- ***isSoftwareOnly**:* devuelve *true* si el códec es puramente virtual, es decir, no utiliza ninguna ayuda del *hardware* para la conversión y, por lo tanto, puede que su velocidad y rendimiento sean inferiores.
- ***isHardwareAccelerated**:* lo contrario, el códec utiliza *hardware* y será más veloz.
- ***isVendor**:* nos dice si el códec es del fabricante o si es estándar de Android.

Veamos los formatos disponibles en Android. Primero los de audio:

Códec	***Coder***	***Decoder***	**Detalles**	**Tipo de archivo**
AAC LC	Sí	Sí	Soporta contenido mono/estéreo/5.0/5.1 con *sampling rates* de 8 a 48 kHz	3GPP (.3gp) MPEG-4 (.mp4, .m4a) ADTS MPEG-TS
HE-AACv1	Sí	Sí		
HE-AACv2	No	Sí	Soporta contenido estéreo/5.0/5.1 con *sampling rates* de 8 a 48 kHz	
AAC ELD	Sí	Sí	Soporta contenido mono/estéreo con *sampling rates* de 16 a 48 kHz	
AMR-NB	Sí	Sí	4,75 a 12,2 kbps sampleadas a 8 kHz	3GPP (.3gp)
AMR-WB	Sí	Sí	6,60 a 23,85 kbps sampleadas a 16kHz	3GPP (.3gp)
FLAC	Sí	Sí	Mono/estéreo sin multicanal hasta 48 kHz. 16 bit recomendados	FLAC (.flac)
GSM	No	Sí	Android soporta decodificación GSM en dispositivos de teléfono	GSM (.gsm)
MIDI	No	Sí	MIDI tipo 0 y 1. DLS versiones 1 y 2. XMF y Mobile XMF. Formatos RTTTL/RTX, OTA, y iMelody	Tipo 0 y 1 (.mid, .xmf, .mxmf) RTTTL/RTX (.rtttl, .rtx) OTA (.ota) iMelody (.imy)
MP3	No	Sí	Mono/Stereo 8 a 320Kbps CBR o VBR	MP3 (.mp3)
Opus	No	Si		Matroska (.mkv)
PCM	Sí	Sí		WAVE (.wav)
Vorbis	No	Sí		Ogg (.ogg) Matroska (.mkv)

Formatos y códecs de vídeo:

Códec	***Coder***	***Decoder***	**Detalles**	**Tipo de archivo**
H.263	Sí	Sí	Soporte para H.263 es opcional en Android 7.0+	3GPP (.3gp) MPEG-4 (.mp4)
H.264 AVC Baseline Profile	Sí	Sí		3GPP (.3gp) MPEG-4 (.mp4) MPEG-TS (.ts, AAC audio)
H.264 AVC Main Profile	Sí	Sí	El *decoder* es obligatorio, el *coder* es opcional	
H.265 HEVC	No	Sí	Main Profile Level 3 para móviles y Main Profile Level 4.1 para Android TV	MPEG-4 (.mp4)
MPEG-4 SP	No	Sí		3GPP (.3gp)
VP8	Sí	Sí	*Streamable* solo en Android 4.0+	WebM (.webm) Matroska (.mkv)
VP9	No	Sí		

3.4.2. Hardware

El paquete android.hardware proporciona al programador las clases necesarias para el manejo del *hardware* del dispositivo. Una de las más utilizada en las *app* multimedia de toma de imágenes será la clase Camera2, si no se han actualizado a CameraX. Debemos tener en cuenta que no todos los dispositivos dispondrán de las características de *hardware* que soporta el sistema operativo Android, por ello es necesario que declaremos en el *manifest* de nuestra *app* cuáles de ellas usaremos y cuáles son indispensables para su funcionamiento. De esta manera, nos aseguraremos de que las aplicaciones se ejecuten en el entorno adecuado y no se rompa la ejecución por falta de *hardware*. Por ejemplo, si en nuestra *app* usamos la cámara pero no es fundamental, deberemos declararlo en el *manifest* de este modo:

<uses-feature android:name="android.hardware.camera.any" android:required="false" />

Pero si es necesaria para la *app*, declararemos:

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

Otras características de *hardware* que podemos declarar son:

- *android.hardware.camera:* cámara trasera del terminal.
- *android.hardware.camera.any:* cámara trasera o cámara frontal indistintamente.
- *android.hardware.camera.autofocus:* característica de autoenfoque de cámara.
- *android.hardware.camera.external:* el terminal puede tener cámara externa.

- *android.hardware.camera.flash:* el *flash* de la cámara.
- *android.hardware.camera.front:* cámara frontal del dispositivo.
- *android.hardware.audio.low_latency:* aceleración del procesamiento de audio.
- *android.hardware.audio.output:* función de salida de audio por altavoces, *jack* de audio, Bluetooth o cualquier otro mecanismo que permita la función.
- *android.hardware.audio.pro:* funcionalidades de audio de alto nivel.
- *android.hardware.audio.microphone:* funcionalidad de grabación de audio.
- *android.hardware.gamepad:* el dispositivo puede tener un *gamepad*.

Todas las características de *hardware* disponibles en Android:

https://developer.android.com/guide/topics/manifest/uses-feature-element

En ocasiones, cuando la eficiencia sea fundamental para el rendimiento de nuestra *app*, tendremos que comunicarnos con el *hardware* de forma más directa. En tales casos, podremos decidir utilizar el NDK o Native Development Kit. El NDK es un conjunto de herramientas y librerías que nos permiten acceder a niveles más bajos del sistema. Debemos tener en cuenta que, para utilizar el NDK, tendremos que programar en C y C++, por lo que el desarrollo será más complicado y nos llevará más tiempo que con un lenguaje de alto nivel como Kotlin.

3.4.3. Fuentes de datos

Antes de empezar a codificar, debemos pensar desde dónde nos llegarán los datos multimedia, pues, dependiendo de cada caso, la forma de acceso cambiará. Por ejemplo, para capturar vídeo, podremos acceder directamente al *hardware* de la cámara o utilizar una *app* intermedia. Si queremos reproducir un vídeo, quizá es estático y lo incrustaremos en el APK. Podríamos hacerlo guardándolo en la carpeta *app/src/main/res/raw* o en la carpeta *app/src/main/assets*. La forma de acceso sería diferente en cada caso. Debemos tener en cuenta también otras diferencias entre los dos directorios. Al estar "raw" dentro de "res", significa que todos los archivos que dejemos allí se tratarán como un recurso y se les asignará un identificador que algunas funciones aceptarán fácilmente sin tener que abrir y leer el archivo nosotros mismos. Al ser un recurso, podremos también guardar varios distintos según el lenguaje del terminal o el tamaño de pantalla. Nada de eso es aplicable a la carpeta "assets", pero esta tiene otras ventajas. Por ejemplo, en la carpeta "assets" podemos crear nuestro

árbol de directorios libremente. Además, los recursos de "raw" están limitados en tamaño si no van ya comprimidos.

Veamos un ejemplo: queremos reproducir un vídeo que tenemos guardado en diferentes lugares del sistema. Imaginemos que en nuestro *layout* tenemos un objeto *VideoView* en el que reproduciremos el vídeo:

```
val videoView = findViewById<VideoView>(R.id.videoView)

// Opción 1: Archivo en disco  (Nota: android.permission.WRITE_EXTERNAL_STORAGE)
val clip = File(Environment.getExternalStorageDirectory(), "test.mp4")
videoView.setVideoPath(clip.path)

// Opción 2: Archivo en directorio res/raw (Nota: no debemos escribir la extensión)
videoView.setVideoURI(Uri.parse("android.resource://$packageName/raw/test"))

// Opción 3: Archivo en Internet (Nota: android.permission.INTERNET)
videoView.setVideoPath("http://videocdn.bodybuilding.com/video/mp4/62000/62792m.mp4")

val mediaController = MediaController(this)
mediaController.setMediaPlayer(videoView)
videoView.setMediaController(mediaController)

videoView.requestFocus()
videoView.start()
```

Primero obtenemos el objeto *VideoView* desde el *layout*. Después mostramos tres maneras diferentes de acceder al vídeo que hemos guardado en un archivo en disco, en "res/raw", y, por último, en el directorio "assets". Una vez que establecemos la fuente en nuestro *VideoView*, utilizamos un *MediaController* para que el usuario vea los controles de *play*, *stop*, *fast forward*, etcétera. Por último, no queda más que comenzar la reproducción mediante la función *start*.

3.5. Datos basados en el tiempo

En cualquier aplicación *software*, el tiempo de procesamiento de datos es importante, pero existen ciertos casos en los que el control del tiempo es fundamental, por ejemplo, en aplicaciones de control en tiempo real como las de reproducción multimedia, el tiempo de procesamiento tendrá un límite máximo que no podrá superarse. Al fin y al cabo, un archivo de audio no es más que valores de voltaje relativos al tiempo que el sistema de audio traducirá a presión de aire en los altavoces, y un fichero de vídeo no es más que un conjunto de imágenes que deben proyectarse en pantalla de forma secuencial a una determinada frecuencia mínima para generar la ilusión de movimiento.

A diferencia de la toma de imágenes fijas, que puede ser más o menos veloz, la captación de audio o vídeo requiere que el proceso sea suficientemente rápido o se perderán datos y la toma carecerá de la calidad suficiente. Por fortuna, los procesos que llevan a cabo las librerías multimedia integradas tienen en cuenta dichos aspectos por nosotros. Por ejemplo, cuando utilizamos MediaPlayer para reproducir un audio, será este el que

genere los hilos de proceso adecuados para llevar a cabo sus tareas de decodificación, cacheado, *buffering* y reproducción, sin que todo eso afecte al hilo principal de la *app*. En algún caso excepcional, la velocidad de estas clases no será suficiente y quizá nos veamos obligados a utilizar el NDK y saltarnos así algunos pasos innecesarios en los procedimientos habituales del reproductor. Más probablemente, necesitaremos utilizar alguna otra clase de las librerías multimedia más adaptada a esa necesidad particular.

Por otra parte, el tiempo también significa tamaño. Si comenzamos a grabar un audio a una velocidad de muestreo de 8 kbps, significa que, cada segundo, nuestro archivo de audio aumentará 8 kbits, por lo que debemos tener en cuenta el tamaño de disco disponible en el dispositivo antes de empezar una grabación. Normalmente, las librerías multimedia del sistema nos permitirán establecer límites de tiempo y/o tamaño del archivo resultante de la captación de vídeo o audio. En el siguiente punto comprobaremos lo explicado con un ejemplo.

3.5.1. Ejemplo: *app* grabadora de audio

Veamos un ejemplo de una *app* grabadora de audio. Crearemos un nuevo proyecto en Android Studio con *File > New > New Project > Empty Activity*. Primero añadiremos permisos y uso de *hardware* en el *manifest:*

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-feature android:name="android.hardware.microphone" android:required="true"/>
```

Necesitamos los permisos para grabar audio y para guardar el audio en un archivo de disco externo. Además, necesitamos un dispositivo con micrófono. En nuestro *layout*, añadiremos dos botones, uno para comenzar y detener la grabación y otro para reproducir el archivo de audio resultado:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">
  <TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
  <Button
    android:id="@+id/btnGrabar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="Grabar"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
```

```
    app:layout_constraintTop_toTopOf="parent" />
  <Button
    android:id="@+id/btnReproducir"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="24dp"
    android:text="reproducir"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/btnGrabar" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Veamos el núcleo de la aplicación, la clase *MainActivity:*

```
import android.Manifest
import android.media.MediaPlayer
import android.media.MediaRecorder
import kotlinx.android.synthetic.main.activity_main.*
import java.io.*

class MainActivity : AppCompatActivity() {
  private var isGrabando = false
  private lateinit var grabadora: MediaRecorder

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

    btnGrabar.setOnClickListener {
      onGrabar()
    }
    btnReproducir.isEnabled = false
    btnReproducir.setOnClickListener {
      val file = getArchivoSalida()
      val reproductor = MediaPlayer()
      reproductor.setDataSource(file?.path)
      reproductor.prepare()
      reproductor.start()
    }
  }

  private fun onGrabar() {
    if(isGrabando) {
      isGrabando = false
      btnGrabar.text = "Grabar"
      stopGrabacion()
    }
    else {
      isGrabando = true
      btnGrabar.text = "Detener"
      startGrabacion()
    }
  }

  override fun onStart() {
    super.onStart()

    if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {

      requestPermissions(arrayOf(
        Manifest.permission.RECORD_AUDIO,
        Manifest.permission.WRITE_EXTERNAL_STORAGE),
        6969)
    }
    grabadora = MediaRecorder()
    grabadora.setMaxDuration(15000)//15 segundos max
    grabadora.setMaxFileSize(1024*1024)//1 Mb max
    grabadora.setOnErrorListener { mr: MediaRecorder, what: Int, extra: Int ->
      showMensaje("Error: Error en $mr, ocurrió $what con $extra")
      onGrabar()
    }
```

```
        grabadora.setOnInfoListener { mr: MediaRecorder, what: Int, extra: Int ->
            when(what) {
                MediaRecorder.MEDIA_RECORDER_INFO_MAX_DURATION_REACHED -> {
                    showMensaje("Se alcanzó la duración máxima!")
                    onGrabar()
                }
                MediaRecorder.MEDIA_RECORDER_INFO_MAX_FILESIZE_REACHED -> {
                    showMensaje("Se alcanzó el tamaño máximo!")
                    onGrabar()
                }
            }
        }
    }

    override fun onStop() {
        grabadora.release()
        super.onStop()
    }

    private fun startGrabacion() {
        val recording = getArchivoSalida()
        if(recording?.exists() == true) {
            recording.delete()
        }
        grabadora.setOutputFile(getArchivoSalida()?.path)
        grabadora.setAudioSource(MediaRecorder.AudioSource.MIC)
        grabadora.setOutputFormat(MediaRecorder.OutputFormat.AMR_NB)
        grabadora.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB)
        grabadora.setAudioChannels(2)
        try {
            grabadora.prepare()
            grabadora.start()
        }
        catch(e: Exception) {
            showMensaje("Error preparando la grabadora "+e.message)
        }
    }

    private fun stopGrabacion() {
        try {
            grabadora.stop()
        }
        catch(e: java.lang.Exception) {
            showMensaje("Error al detener la grabación "+e.message)
        }
        grabadora.reset()

        val recording = getArchivoSalida()
        if(recording?.exists()==true && recording.length() > 0) {
            btnReproducir.isEnabled = true
            showMensaje("Grabación finalizada")
        }
        else {
            showMensaje("Error en la grabación")
        }
    }

    private fun getArchivoSalida(): File? {
        return File(getExternalFilesDir(null), archivo)
    }

    private fun showMensaje(msg: String, e: Exception? = null) {
        Log.e(tag, msg, e)
        Toast.makeText(this, msg, Toast.LENGTH_LONG).show()
        textView.text = msg
    }

    companion object {
        private const val tag = "Activity"
        private const val archivo = "grabacion.3gp"
    }
}
```

Empezamos en *onCreate*, donde establecemos las acciones de ambos botones. Al ser pulsado *btnGrabar*, se llamará a la función *onGrabar* que veremos luego. Cuando pulsamos *btnReproducir*, crearemos un objeto MediaPlayer, le asignaremos el archivo de audio que habremos grabado antes mediante *setDataSource*, prepararemos los datos mediante *prepare* y comenzaremos la reproducción mediante *start*. El método *onGrabar* es simplemente un interruptor: según el valor de *isGrabando*, comenzaremos la grabación o la detendremos, para ello, llamaremos a las funciones *startGrabacion* y *stopGrabacion*.

Sobrescribimos *onStart*, donde pediremos los permisos necesarios al usuario, ya que ambos permisos requieren extra seguridad y no basta con declararlos en el *manifest*. Además, en *onStart* creamos el *MediaRecorder*. Este objeto nos permitirá grabar audio desde el micrófono del dispositivo. Utilizaremos *setMaxFileSize* y *setMaxDuration* para establecer un límite al tamaño del archivo de audio resultante.

Con *setOnErrorListener* establecemos el código que reaccionará ante un error en la grabación; en este caso, simplemente volvemos a establecer el estado del botón de grabación. Con *setOnInfoListener* podremos determinar las acciones que tomar si la grabación se detuvo por exceso de tiempo o tamaño, si la duración se extendió más allá de los milisegundos de *setMaxDuration* o si ocupó más bytes de lo especificado con *setMaxFileSize*. En *onStop* tendremos la buena costumbre de llamar a *release* para liberar los recursos que puede haber reservado el *MediaRecorder*.

Creamos la función *startGrabacion*, donde creamos un objeto *File* con un nombre predefinido. Después comprobamos si el archivo existe, y de ser así lo borramos, porque el *MediaRecorder* lo creará por nosotros más tarde. Mediante *setOutputFile* le decimos al *MediaRecorder* en qué archivo guardar la grabación. Mediante *setAudioSource* le decimos que utilice el micro del dispositivo.

Los métodos *setOutputFormat* y *setAudioEncoder* especifican el formato y la codificación de archivo de audio. Podemos estudiar detenidamente las características de cada formato para así poder elegir mejor cuál será el más apropiado para nuestro proyecto. En este caso, elegimos *AMR_NB*, que es adecuado para almacenar voz. Con *setAudioChannels* establecemos el número de canales de audio. Establecemos dos para hacerlo estéreo, aunque muy probablemente el dispositivo solo tenga un micro y ambos canales terminen siendo iguales. Podríamos haber seleccionado 1 y habríamos obtenido un audio en mono.

Antes de empezar la grabación, debemos llamar a *prepare*, que compilará todas las opciones que le indicamos previamente y preparará la grabación. Ahora sí, podemos llamar a *start* y comenzar a captar audio del micro.

En *stopGrabacion* llamamos al método *stop* del *MediaRecorder* para detener la grabación y cerrar el archivo de audio. Con la opción *reset* dejamos el *MediaRecorder*

con la configuración por defecto, listo para ser configurado de nuevo más tarde. Después comprobamos que el archivo de audio exista y tenga datos. De ser así, activaremos el botón *btnReproducir*, que utilizará el *MediaPlayer* para reproducir el audio, como vimos antes.

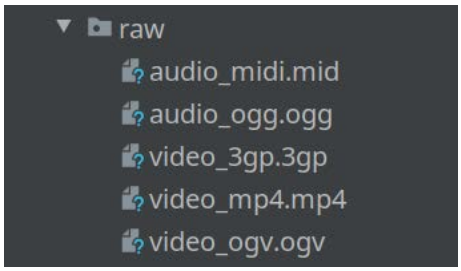
Como hemos visto en el ejemplo, las clases de las librerías multimedia son tan potentes que no tenemos por qué preocuparnos por el multiproceso ni por la configuración y el acceso al *hardware*. Las clases nos presentarán mecanismos para informarnos del estado del proceso mientras ellas hacen todo el trabajo duro en *background*.

3.6. Clips de audio, secuencias MIDI, clips de vídeo, entre otros

En Android, la potencia de la clase MediaPlayer hace que reproducir clips de vídeo o audio en diferentes formatos sea prácticamente lo mismo. Siempre que el terminal disponga de los códecs para los formatos utilizados, el procedimiento de cargar el MediaPlayer con ellos y reproducirlos será siempre igual. Simplemente, deberemos configurar el reproductor a nuestro gusto.

3.6.1. Clips de audio y vídeo

Vamos a crear una *app* que reproduzca tanto videoclips como audioclips. En el código podremos elegir entre varios formatos, y veremos que no es necesario cambiar la configuración del MediaPlayer para que siga funcionando. En el directorio "res/raw" insertaremos varios clips de audio y vídeo para reproducirlos desde nuestro código indistintamente:



En Android Studio, elegimos la opción *File > New > New Project… > Empty Activity*. Diseñaremos una interfaz gráfica sencilla del siguiente modo:

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

    <TextView
```

```
    android:id="@+id/lblAudio"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginTop="8dp"
    android:text="@string/audio_clip"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="@+id/btnPlayAudio" />
  <Button
    android:id="@+id/btnPlayAudio"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginTop="32dp"
    android:text="@string/play"
    app:layout_constraintStart_toEndOf="@+id/textView2"
    app:layout_constraintTop_toTopOf="parent" />
  <Button
    android:id="@+id/btnStopAudio"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:text="@string/stop"
    app:layout_constraintStart_toEndOf="@+id/btnPlayAudio"
    app:layout_constraintTop_toTopOf="@+id/btnPlayAudio" />

  <TextView
    android:id="@+id/lblVideo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginTop="8dp"
    android:text="@string/video_clip"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="@+id/btnPlayVideo" />
  <Button
    android:id="@+id/btnPlayVideo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginTop="24dp"
    android:text="@string/play"
    app:layout_constraintStart_toEndOf="@+id/lblVideo"
    app:layout_constraintTop_toBottomOf="@+id/btnPlayAudio" />
  <Button
    android:id="@+id/btnStopVideo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:text="@string/stop"
    app:layout_constraintStart_toEndOf="@+id/btnPlayVideo"
    app:layout_constraintTop_toTopOf="@+id/btnPlayVideo" />
  <VideoView
    android:id="@+id/videoView"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginTop="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/btnPlayVideo" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Simplemente, hemos añadido al *ConstraintLayout* botones para controlar el audio y el vídeo, y un *VideoView* para mostrar la secuencia de vídeo. El archivo de cadenas para internacionalización *res/values/strings* sería algo como:

```
<resources>
  <string name="app_name">AudioVideoClips</string>
  <string name="play">Play</string>
```

```
  <string name="pause">Pause</string>
  <string name="stop">Stop</string>
  <string name="audio_finish">Audio clip terminado</string>
  <string name="video_finish">Video clip terminado</string>
  <string name="video_clip">Vídeo Clip:</string>
  <string name="audio_clip">Audio Clip:</string>
</resources>
```

El código más importante lo tendremos en la actividad principal:

```
import android.media.MediaPlayer
import android.net.Uri
import android.os.Bundle
import android.widget.MediaController
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
  private lateinit var mp: MediaPlayer

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    /// Audio
    initAudio()
    /// Video
    initVideo()
  }

  ////////////////////////////////////////////////////////////////////////////////
  /// AUDIO
  private fun initAudio() {
    mp = MediaPlayer.create(this, R.raw.audio_midi)
    //mp = MediaPlayer.create(this, R.raw.audio_ogg)
    mp.setOnCompletionListener { mp ->
      stopAudio()
      Toast
        .makeText(this, getString(R.string.audio_finish), Toast.LENGTH_LONG)
        .show()
    }
    btnPlayAudio.setOnClickListener {
      if(mp.isPlaying)
        pauseAudio()
      else
        playAudio()
    }
    btnStopAudio.setOnClickListener {
      stopAudio()
    }
    btnPlayAudio.text = getString(R.string.play)
    btnStopAudio.isEnabled = false
  }
  private fun playAudio() {
    mp.start()
    //
    btnPlayAudio.text = getString(R.string.pause)
    btnStopAudio.isEnabled = true
  }
  private fun stopAudio() {
    mp.stop()
    mp.prepare()
    mp.seekTo(0)
    //
    btnPlayAudio.text = getString(R.string.play)
    btnStopAudio.isEnabled = false
  }
  private fun pauseAudio() {
    mp.pause()
    //
    btnPlayAudio.text = getString(R.string.play)
```

```
    }

    ////////////////////////////////////////////////////////////////////////////
    /// VIDEO
    private fun initVideo() {
        videoView.setOnCompletionListener {
            stopVideo()
            Toast
                .makeText(this, getString(R.string.video_finish), Toast.LENGTH_LONG)
                .show()
        }
        btnPlayVideo.setOnClickListener {
            when {
                videoView.isPlaying -> pauseVideo()
                videoView.currentPosition != 0 -> resumeVideo()
                else -> playVideo()
            }
        }
        btnStopVideo.setOnClickListener {
            stopVideo()
        }
        btnPlayVideo.text = getString(R.string.play)
        btnStopVideo.isEnabled = false
    }
    private fun playVideo() {
        // Archivo en directorio res/raw (Nota: no debemos escribir la extensión)
     //videoView.setVideoURI(Uri.parse("android.resource://$packageName/raw/video_mp4"))
     //videoView.setVideoURI(Uri.parse("android.resource://$packageName/raw/video_ogv"))
     videoView.setVideoURI(Uri.parse("android.resource://$packageName/raw/video_3gp"))
        val mediaController = MediaController(this)
        mediaController.setMediaPlayer(videoView)
        videoView.setMediaController(mediaController)
        videoView.requestFocus()
        videoView.start()
        //
        btnPlayVideo.text = getString(R.string.pause)
        btnStopVideo.isEnabled = true
    }
    private fun resumeVideo() {
        videoView.start()
        //
        btnPlayVideo.text = getString(R.string.pause)
    }
    private fun pauseVideo() {
        videoView.pause()
        //
        btnPlayVideo.text = getString(R.string.play)
    }
    private fun stopVideo() {
        videoView.stopPlayback()
        //
        btnPlayVideo.text = getString(R.string.play)
        btnStopVideo.isEnabled = false
    }
}
```

Primero definimos una variable de tipo MediaPlayer que nos servirá para reproducir el audio. Para el vídeo, utilizaremos el componente *VideoView* que ya está compuesto de una instancia de MediaPlayer en su interior. En *onCreate* llamamos a dos funciones, una para iniciar el audio y otra para el vídeo.

El método *initAudio* inicia la variable MediaPlayer mediante un contexto y el identificador del archivo de audio que deseamos reproducir. Como vemos, tenemos dos líneas iguales, una para un clip de audio en formato MIDI y otra en formato OGG.

Podemos comentar una de ellas, y será lo único que necesitemos reproducir un tipo u otro: la clase MediaPlayer utilizará el códec correspondiente a cada uno cuando lo reproduzca.

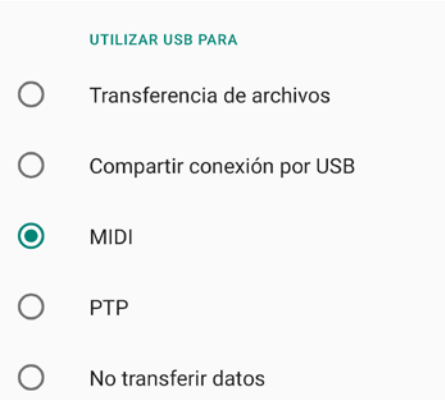
Hemos establecido el código para el evento *onCompletion*, que se ejecutará cuando el MediaPlayer termine de reproducir el archivo. En nuestro caso, simplemente mostraremos un mensaje y cambiaremos el estado de los botones. Al pulsar el botón *btnPlayAudio*, comprobaremos si el MediaPlayer está reproduciendo; si es así, pausaremos el audio, de lo contrario, arrancaremos la reproducción. Esto será tan sencillo como llamar a los métodos *start* y *pause* de la clase MediaPlayer. El botón *btnStopAudio* utilizará la función *stop* para detener el audio. Además, llamaremos a *prepare* para restaurar el estado del reproductor, y *seekTo* con el parámetro 0 para que el MediaPlayer comience desde el principio del clip la próxima vez que se llame a *start*.

La función *initVideo* establece el código para el *onCompletionListener*, en el que mostraremos un mensaje al usuario mediante la función *Toast* y actualizaremos el estado de los botones. El botón *btnPlayVideo* comprueba si se está reproduciendo un vídeo y, de ser así, se pausa mediante la función *pause;* si no se está reproduciendo pero el vídeo está avanzado, significa que lo pausamos en algún momento, de modo que lo volvemos a arrancar mediante la función *start;* de otro modo, significa que la reproducción estaba totalmente detenida, por lo que volveremos a configurar el MediaPlayer con el videoclip y lo arrancaremos con la función *start*. El botón *btnStopVideo* detendrá la reproducción mediante la llamada *stopPlayback*.

Habríamos conseguido el mismo resultado de haber usado ExoPlayer en lugar de MediaPlayer. Para los clips de audio, también podríamos usar un SoundPool o un AudioTrack. No habría gran diferencia en lo que respecta al formato, ya que estas clases asumirán la responsabilidad de utilizar los códecs adecuados en cada caso, librando al programador del trabajo duro.

3.6.2. MIDI

Ya hemos visto cómo podemos reproducir clips de audio, incluidos clips MIDI, pero merece especial atención, además, la capacidad del sistema Android para controlar dispositivos MIDI conectados a él mediante USB o Bluetooth LE. Observemos cómo, cuando conectamos nuestro terminal Android a un ordenador de sobremesa mediante un cable USB, en las opciones de conexión aparecerá MIDI:



Nuestro terminal Android puede conectarse a un dispositivo *hardware* o *software* MIDI para controlar o recibir comandos MIDI. Por ejemplo, podríamos recibir datos de un teclado MIDI conectados a nuestra *app* para generar los sonidos, o al contrario, podríamos desarrollar una *app* que mostrase un teclado en pantalla que enviase comandos MIDI a un dispositivo *hardware* para que genere el sonido o a un PC con una aplicación instalada que emule el *hardware* MIDI.

El SDK de Android dispone de la librería android.media.midi, que nos permite enumerar los dispositivos MIDI y conectarnos a ellos, enviar y recibir comandos, etcétera. A partir de Android Q (API 29), tenemos la oportunidad también de utilizar la API nativa de MIDI, que tiene características mejoradas con respecto a la anterior, pero necesitaremos tener conocimientos de lenguaje C. Una descripción completa de cualquiera de estas dos API necesitaría más páginas de las que disponemos, pero si nos vemos en la necesidad de desarrollar una *app* de control de dispositivos MIDI, podemos consultar la amplia documentación de estas dos librerías.

Documentación de la API MIDI android.media.midi:

https://developer.android.com/reference/android/media/midi/package-summary

Native MIDI API:

https://developer.android.com/ndk/guides/audio/midi

3.7. Procesamiento de objetos multimedia. Clases. Estados, métodos y eventos

En puntos anteriores, hemos estudiado ya algunos ejemplos de grabación y reproducción de audio y vídeo. En este punto vamos a enumerar las clases más importantes para el proceso de multimedia, y mostraremos algunos ejemplos más. Demostraremos así la versatilidad de las librerías multimedia integradas en Android, y mencionaremos en qué casos es mejor utilizar unas clases u otras. Comprobaremos cómo las librerías de Android abarcan todos los puntos necesarios para el desarrollo de *apps* multimedia de gran capacidad y alto rendimiento. Obviaremos aquí el uso de aplicaciones externas para la captura de audio y vídeo, así como para la reproducción multimedia. Recordemos las principales clases y sus usos:

- **SoundPool**: esta clase permite la reproducción de clips de audio. SoundPool nos permitirá cargar en memoria clips de audio desde la carpeta de recursos o desde un archivo del disco. Esta clase utiliza los servicios de decodificación de MediaPlayer para convertir los clips de audio en el formato básico PCM de 16 bits mientras se cargan en la memoria. De este modo, la reproducción de los sonidos será más veloz, pues solo serán decodificados una vez. SoundPool es capaz, además, de mezclar todos los clips de audio de forma eficiente: establece prioridades para cada uno y controla el número máximo de ellos que pueden ser reproducidos al mismo tiempo. SoundPool será adecuado si el número de clips de audio y su longitud no es excesiva, de lo contrario, la memoria utilizada para almacenarlos podría ser más de la que disponga el terminal. Veremos un ejemplo de SoundPool en el siguiente punto.

- **MediaRecorder**: esta clase permite la captación de audio y vídeo, según las fuentes que se le configuren. Permitirá configurar el formato y la longitud de los archivos de salida. Cuando se utilice con la clase Camera2, por ejemplo, nos permitirá grabar clips de vídeo y almacenarlos en disco en el formato deseado.

- **MediaPlayer**: el reproductor de audio y vídeo por excelencia en Android. Ya hemos visto la potencia y facilidad de su uso en algunos ejemplos. La única desventaja sería a la hora de reproducir audio o vídeo en *streaming*, en cuyo caso podríamos pasar a ExoPlayer.

- **ExoPlayer**: es un paquete fuera de la API de Android, pero bien integrado en ella. Su potencia y versatilidad le permiten su uso tanto para aplicaciones sencillas como para las más complejas. En algunos casos, sin embargo, su potencia nos jugará una mala pasada en lo que respecta al consumo de batería, de modo que la utilizaremos principalmente cuando la tarea sea compleja o requiera de la potencia extra de ExoPlayer, donde MediaPlayer se quede corto, por ejemplo, en *videostreaming*.

- **Camera2**: la segunda versión de una clase para el control de la cámara del dispositivo. Nos permite el control de todos los parámetros de la toma de imágenes y vídeo desde el *hardware*.
- **CameraX**: aún en desarrollo mientras se escriben estas líneas, pretende ser el sustituto de Camera2, o la forma fácil de acceder a Camera2. Es una biblioteca de compatibilidad que utiliza las funcionalidades de Camera2, pero con una programación más sencilla basada en casos de uso. Además, evita que el programador tenga que enfrentarse a problemas de compatibilidad entre diferentes dispositivos y *hardware*.

Presentamos aquí otras clases menos conocidas, quizá por ser más complejas, de bajo nivel o específicas para un determinado propósito:

- **AudioTrack**: esta clase controla la carga y reproducción de un solo clip de audio PCM cada vez. Un objeto AudioTrack puede funcionar en modo estático o en *streaming*. En modo *streaming*, AudioTrack acepta escrituras continuas de bytes que lanza a la capa nativa para su reproducción. Este modo es útil cuando los clips de audio son demasiado grandes para ser almacenados en memoria, o porque los datos llegan de alguna otra fuente de *streaming*, como algún servidor en internet. El modo estático será útil cuando trabajemos con sonidos cortos y de poco tamaño que puedan ser cargados en memoria sin problema. Este modo es rápido y eficiente, y será el preferido para juegos y pequeños sonidos relacionados con la interacción de la interfaz gráfica.
- **MediaCodec**: clase que nos permite acceder a los códecs del sistema.
- **Surface**: usado como un *buffer* de vídeo para clases como MediaRecorder o SurfaceTexture, y controlado por un productor de vídeo como MediaPlayer.
- **MediaMuxer**: una clase que nos permite la mezcla de *streams* de vídeo y audio. Soporta codificación de salida MP4, Webm y 3GP.

Existen muchas otras clases. Es recomendable que echemos un vistazo a la documentación oficial para tener en cuenta todas sus posibilidades.

Documentación oficial de API multimedia de Android:

https://developer.android.com/reference/android/media/package-summary

3.7.1. Reproducción de objetos multimedia. Clases. Estados, métodos y eventos

Plantearemos aquí algunos ejemplos de clases multimedia que aún no hemos probado anteriormente:

Ejemplo 1: **SoundPool**

Esta *app* cargará en memoria algunos clips de audio y los reproducirá a elección del usuario. Cada sonido tendrá asociado un botón que el usuario podrá presionar para reproducirlo. Primero, veamos el aspecto del *layout:*

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">
  <Button
     android:id="@+id/button1"
     android:layout_width="0dp"
     android:layout_height="wrap_content"
     android:layout_marginStart="32dp"
     android:layout_marginTop="8dp"
     android:text="Sonido1"
     android:tag="1"
     android:onClick="onClick"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toTopOf="parent" />
  <Button
     android:id="@+id/button2"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Sonido2"
     android:tag="2"
     android:onClick="onClick"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toEndOf="@+id/button1"
     app:layout_constraintTop_toTopOf="@+id/button1" />
  <Button
     android:id="@+id/button3"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_marginTop="32dp"
     android:text="Sonido3"
     android:tag="3"
     android:onClick="onClick"
     app:layout_constraintStart_toStartOf="@+id/button1"
     app:layout_constraintTop_toBottomOf="@+id/button1" />
  <Button
     android:id="@+id/button4"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Sonido4"
     android:tag="4"
     android:onClick="onClick"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toEndOf="@+id/button3"
     app:layout_constraintTop_toTopOf="@+id/button3" />
  <Button
     android:id="@+id/button5"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_marginTop="32dp"
```

```
    android:text="Sonido5"
    android:tag="5"
    android:onClick="onClick"
    app:layout_constraintStart_toStartOf="@+id/button3"
    app:layout_constraintTop_toBottomOf="@+id/button3" />
  <Button
    android:id="@+id/button6"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Sonido6"
    android:tag="6"
    android:onClick="onClick"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/button5"
    app:layout_constraintTop_toTopOf="@+id/button5" />
  <Button
    android:id="@+id/button7"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="Sonido7"
    android:tag="7"
    android:onClick="onClick"
    app:layout_constraintStart_toStartOf="@+id/button5"
    app:layout_constraintTop_toBottomOf="@+id/button5" />
  <Button
    android:id="@+id/button8"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Sonido8"
    android:tag="8"
    android:onClick="onClick"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/button7"
    app:layout_constraintTop_toTopOf="@+id/button7" />
  <Switch
    android:id="@+id/swtBucle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="En bucle"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/button7" />
  <EditText
    android:id="@+id/txtCanales"
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:ems="10"
    android:hint="Núm. Canales"
    android:text="8"
    android:inputType="number"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/swtBucle" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

No son más que ocho botones, un interruptor y una entrada de texto. Todos los botones llaman a la misma acción cuando son presionados: *onClick*. Hemos añadido el atributo *tag* para diferenciar los distintos botones en el código. El interruptor hará que los sonidos se reproduzcan una sola vez o en bucle. La caja de texto permitirá al usuario crear un SoundPool con un número de canales diferente: por ejemplo, si introduce 3, solo se escucharán tres sonidos al mismo tiempo, aunque se pulsen cuatro botones

consecutivamente. Estudiemos el código de la *activity,* donde se lleva a cabo toda la lógica:

```
import android.media.AudioAttributes
import android.media.AudioManager
import android.media.SoundPool
import android.widget.Button
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
  private lateinit var audioManager: AudioManager
  private lateinit var soundPool: SoundPool

  data class Sonido(val id: Int, var boton: Button?=null, var cargado: Boolean=false)
  private val sonidos = HashMap<Int, Sonido>()
  private val sonidosRes = arrayListOf(
     R.raw.animal_bark_and_growl,
     R.raw.animal_hiss_and_rattle,
     R.raw.crow_call,
     R.raw.distant_dog_barking,
     R.raw.dog_barking,
     R.raw.dog_growling,
     R.raw.dog_snarling,
     R.raw.mouse_squeaking
  )
  private lateinit var botonesRes: ArrayList<Button>
  private var actVolume = 0f
  private var maxVolume = 0f
  private var volume = 0f

  override fun onCreate(savedInstanceState: Bundle?) {
     super.onCreate(savedInstanceState)
     setContentView(R.layout.activity_main)
     botonesRes = arrayListOf(
        button1, button2, button3, button4,
        button5, button6, button7, button8)
     txtCanales.doOnTextChanged { text, start, before, count ->
        val canales = getCanalesFromUI()
        if(canales != null) {
           soundPool.release()
           initAudio(canales)
        }
     }
  }
  override fun onResume() {
     super.onResume()
     initAudio(getCanalesFromUI() ?: 8)
  }
  override fun onPause() {
     soundPool.release()
     super.onPause()
  }

  private fun getCanalesFromUI(): Int? = txtCanales.text.toString().toIntOrNull()
  private fun deshabilitarBotones() {
     button1.isEnabled = false
     button2.isEnabled = false
     button3.isEnabled = false
     button4.isEnabled = false
     button5.isEnabled = false
     button6.isEnabled = false
     button7.isEnabled = false
     button8.isEnabled = false
  }
  private fun initAudio(canales: Int) {
     deshabilitarBotones()
     audioManager = getSystemService(Context.AUDIO_SERVICE) as AudioManager
     actVolume = audioManager.getStreamVolume(AudioManager.STREAM_MUSIC).toFloat()
     maxVolume = audioManager.getStreamMaxVolume(AudioManager.STREAM_MUSIC).toFloat()
     volume = actVolume / maxVolume
```

```
        volumeControlStream = AudioManager.STREAM_MUSIC

        val audioAttr = AudioAttributes.Builder()
            .setLegacyStreamType(AudioManager.STREAM_MUSIC)
            .build()
        soundPool = SoundPool.Builder()
            .setMaxStreams(canales)
            .setAudioAttributes(audioAttr)
            .build()
        soundPool.setOnLoadCompleteListener { soundPool, sampleId, status ->
            sonidos[sampleId]?.cargado = status == 0
            sonidos[sampleId]?.boton?.isEnabled = status == 0
        }

        for((boton, idRes) in sonidosRes.withIndex()) {
            val id = soundPool.load(this, idRes, 1)
            sonidos[id] = Sonido(id, botonesRes[boton])
        }
    }
    fun onClick(view: View?) {
        val i = view?.tag as String
        val sonido = sonidos[sonidos.keys.elementAt(i.toInt() -1)]
        if(sonido != null) {
            if(swtBucle.isChecked)
                playLoop(sonido)
            else
                play(sonido)
        }
    }
    private fun play(sonido: Sonido) {
        if(sonido.cargado) {
            soundPool.play(sonido.id, volume, volume, 1, 0, 1f)
        }
    }
    private fun playLoop(sonido: Sonido) {
        if(sonido.cargado) {
            soundPool.play(sonido.id, volume, volume, 1, -1, 1f)
        }
    }
    fun pause(sonido: Sonido) {
        soundPool.pause(sonido.id)
    }
    fun stopSound(sonido: Sonido) {
        soundPool.stop(sonido.id)
    }
}
```

Empezamos definiendo una variable *AudioManager* para controlar el volumen de la *app*. También una variable SoundPool, que será la que lleve a cabo todo el trabajo. Creamos una clase de datos para definir los datos de los ocho sonidos, asociarles un botón y controlar si están cargados o no. En la función *onCreate*, cuando el *layout* ya se ha desplegado en la vista, podemos acceder a los componentes de la interfaz gráfica. Creamos entonces la lista de botones y definimos el código que se ejecutará cuando cambie el valor de la caja de texto. Básicamente, cuando el usuario cambie el número de canales en este *EditText*, nos veremos obligados a recrear el SoundPool con la nueva configuración, ya que no puede cambiarse el número de canales dinámicamente. En *onResume* iniciamos el SoundPool, y en *onPause* liberamos los recursos que este objeto mantiene en memoria.

El método *initAudio* configurará el sistema de audio. Lo primero que hace es obtener el volumen relativo mediante el *AudioManager*, que nos servirá para establecer el volumen de audio en los clips de SoundPool. Mediante el atributo de la *activity volumeControlStream*, establecemos que el canal de *stream* de la actividad sea *STREAM_MUSIC*. Este será también el tipo de *stream* que utilizaremos para el SoundPool, del que hemos obtenido el volumen. De este modo, cuando el usuario cambie el volumen, el SoundPool utilizará el nuevo volumen seleccionado para reproducir los audios. Podríamos haber utilizado otro tipo de *stream*, siempre y cuando hubiésemos utilizado ese mismo tipo en las demás funciones.

Android dispone de varios canales para que el usuario pueda establecer diferentes valores de volumen en cada uno de ellos. Es posible que quiera escuchar música al máximo pero el canal de notificaciones dejarlo en silencio o a otro volumen diferente. Después crearemos una instancia de la clase SoundPool con el número de canales obtenido del usuario. El número de canales es el número máximo de audios que la clase mezclará al mismo tiempo. También establecemos un *listener* para detectar cuándo ha terminado de cargarse cada clip de audio. Cuando SoundPool termine de decodificar y cargar en memoria uno de los audios, se llamará a esta función, donde habilitaremos el botón correspondiente al sonido. Con el bucle *for*, iteramos sobre la lista de sonidos que hemos guardado en la carpeta de recursos "res/raw", para cargarlos uno a uno en el SoundPool. El *id* que nos devuelve lo utilizaremos para nuestra lista de control.

Todos los botones llaman a la misma función *onClick* cuando son pulsados. Esta función obtiene el valor del *tag* del botón, lo convierte a entero y le resta uno. De esta manera, cada botón estará asociado a un índice de la lista de sonidos. Una vez obtenido el sonido que queremos reproducir, llamamos a la función *play*, o *playLoop* si el conmutador de bucle está activo. Ambas funciones son prácticamente iguales: comprueban que el sonido se haya cargado en memoria y llaman al método *play* del SoundPool. Este método acepta primero el identificador de carga de sonido que se requiere reproducir; en segundo y tercer lugar, el volumen del canal izquierdo y derecho respectivamente de un sistema estéreo, con valores de 0 como mínimo a 1 como máximo, en nuestro caso, queremos que ambos canales suenen con el mismo volumen; el cuarto parámetro es la prioridad del sonido que reproducir, con 0 como valor mínimo de prioridad. En el supuesto de que el número de canales sea inferior o igual a los sonidos que se están reproduciendo, un nuevo sonido no podrá mezclarse sin más: SoundPool calculará qué sonido debe dejar de reproducir para incluir el nuevo, y ahí la prioridad de cada sonido entra en el cálculo.

El quinto parámetro indica si queremos que el sonido se reproduzca una vez o en bucle de forma constante. Aquí es donde nuestras dos funciones difieren, ya que *play* utiliza un valor de 0, que indica tocar solo una vez, mientras que *playLoop* utilizará un valor de -1 para indicar que queremos que se toque de forma continua. El sexto parámetro es la velocidad de reproducción, con un mínimo de 0,5 (reproducción lenta o grave) y un

máximo de 2 (reproducción rápida o aguda), siendo 1 una reproducción normal (sin modificación en la velocidad del audio). Por último, para detener temporalmente un sonido utilizaremos el método *pause*, y para detenerlo definitivamente usaremos el método *stop*.

3.7.2. Protocolo de transmisión en tiempo real RTP

Hoy más que nunca, este protocolo es importante. El Realtime Transport Protocol nos permite enviar y recibir audio y vídeo a través de internet. Puede utilizarse para llamadas telefónicas de VoIP, para escuchar música de radios digitales en internet o para ver contenidos de vídeo desde alguna plataforma como Netflix o similar. Muchos servicios de emergencia han abandonado ya el uso de sus radios y *walkie-talkies* para dar paso a las nuevas tecnologías. El uso de *apps* de PTT *(push to talk)* sobre *smartphones* (PTT over Cellular o PoC) permite el envío en tiempo real de audio y vídeo de uno a uno o en grupos, permite asignar prioridades, etcétera.

En cualquier caso, el protocolo que utilizarán todas esas herramientas para el envió de *streams* multimedia será uno basado en RTP. Por otra parte, existirá el RTP Control Protocol (RTPC), que se encarga del control del flujo y de calidad del servicio (QoS), ayudando a la sincronización de los diferentes *streams*. En cada caso concreto existirán otros protocolos para el correcto funcionamiento del servicio. Por ejemplo, en el caso del PTT o llamadas de VoIP, se utilizará además el protocolo SIP para realizar el inicio de sesión de los agentes involucrados en la comunicación.

El protocolo RTP lleva en su interior el *payload*, es decir, los datos de audio o vídeo que tendrán que llegar en tiempo, de modo que el cliente no perciba retardos o saltos en la reproducción. Para ello, los paquetes RTP llevan *timestamps* y números de secuencia y cuentan con mecanismos que detectan la pérdida de paquetes de datos y otros problemas clásicos de la comunicación en tiempo real, sobre todo cuando se utiliza UDP como protocolo de internet.

El protocolo puede utilizarse tanto de uno a uno, por ejemplo, en una llamada telefónica de VoIP, como de uno a varios, en el caso de un proveedor de servicios multimedia a través de IP *multicasting*. Como comentamos antes, para el establecimiento de la sesión entre dos o más agentes que deseen compartir datos multimedia en tiempo real, puede utilizarse un protocolo como SIP, pero también existen otros, como H.232, RTSP o Jingle. Estos mecanismos podrán utilizar el Session Description Protocol (SDP) para especificar los parámetros de la sesión, el formato del audio o vídeo que transmitir, etcétera. Cada *stream* multimedia creará una sesión RTP, incluso puede separarse el vídeo del audio en diferentes sesiones, permitiendo así que el cliente seleccione cada componente por separado, por ejemplo, para el idioma de una película.

Ya hemos visto los protocolos básicos para la transmisión en tiempo real de contenido multimedia, pero la realidad puede complicarse bastante más. Por ejemplo, los proveedores de contenidos multimedia requieren la seguridad de que nadie podrá consumir sus productos sin haber pasado antes por caja. También aseguran, de este modo, los *copyright* de las obras transmitidas por internet, al impedir que los *streams* sean copiados y reproducidos libremente. Herramientas de *digital right management* (DMR) como Widevine se aplican a los *streams* para encriptar e impedir la copia de los datos involucrados. La clase MediaPlayer es compatible con la reproducción de contenido protegido por DRM a partir de Android 8 (API 26), aunque también disponemos de otras clases como MediaDrm en caso de que necesitemos mayor control sobre los procesos DRM. Veamos un ejemplo de cómo sería el código Android:

```
mediaPlayer?.apply {
  setDataSource()
  setOnDrmConfigHelper() // opcional, para otras configuraciones
  prepare()
  drmInfo?.also {
     prepareDrm()                  //Prepara el DRM para la fuente actual
     getKeyRequest()       //Pide una prueba de clave al servidor de licencias
     provideKeyResponse()          //Procesa la prueba de clave para iniciar sesión
  }
  // MediaPlayer listo para reproducir
  start()
  // ...play/pause/resume...
  stop()
  releaseDrm()
}
```

Como vemos, las librerías de Android nos permitirán codificar las más avanzadas técnicas multimedia. Sí es cierto que puede llegar a ser complejo según el caso, por ello, también tenemos la opción de utilizar ExoPlayer, un potente reproductor que podemos incrustar en nuestra *app*. ExoPlayer nos permite diferentes niveles de complejidad. Ya tenemos una idea de lo compleja que resulta la reproducción en *streaming*, veamos cómo ExoPlayer nos puede facilitar en gran medida nuestro trabajo, con el siguiente ejemplo:

```
import android.net.Uri
import com.google.android.exoplayer2.ExoPlayerFactory
import com.google.android.exoplayer2.SimpleExoPlayer
import com.google.android.exoplayer2.source.MediaSource
import com.google.android.exoplayer2.source.ProgressiveMediaSource
import com.google.android.exoplayer2.upstream.DefaultDataSourceFactory
import com.google.android.exoplayer2.util.Util


class RadioPlayerActivity : Activity() {
  private lateinit var exoPlayer: SimpleExoPlayer
  private lateinit var ms: MediaSource
  private lateinit var dsf: DefaultDataSourceFactory

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

     exoPlayer = ExoPlayerFactory.newSimpleInstance(this)
     dsf = DefaultDataSourceFactory(this, Util.getUserAgent(this, "ExoPlayer"))

     val radio1 = "http://playerservices.streamtheworld.com"
           + "/api/livestream-redirect/LOS40.mp3"
```

```
    val radio2 = "http://us3.internet-radio.com:8342/listen.pls&t=.pls"
    //...
    ms = ProgressiveMediaSource.Factory(dsf).createMediaSource(Uri.parse(radio1))
    with(exoPlayer) {
        prepare(ms)
        btnPlay.setOnClickListener {
            playWhenReady = true
        }
        btnStop.setOnClickListener {
            playWhenReady = false
        }
    }
  }

  override fun onDestroy() {
    exoPlayer.playWhenReady = false
    super.onDestroy()
  }
}
```

No olvidemos añadir la librería ExoPlayer a nuestro build.gradle:

```
dependencies {
  implementation 'com.google.android.exoplayer:exoplayer:2.10.4'
  ...
```

Puesto que los datos llegan de servidores en internet, debemos incluir en nuestro *manifest* el permiso *android.permission.INTERNET*.

3.7.3. Control y mono

Son muchas las clases de las librerías multimedia que nos permiten controlar la reproducción de audio y vídeo. Por ejemplo, tenemos la clase *VolumeShaper* para insertar atenuaciones de volumen al comienzo, al final o de transición entre clips de audio. Podemos crear instancias de *VolumeShaper* desde objetos MediaPlayer o AudioTrack y configurar con él las atenuaciones que deseemos mediante los parámetros que definen la curva de sonido.

Documentación e instrucciones de uso de *VolumeShaper:*

https://developer.android.com/guide/topics/media/volumeshaper

Otra forma de control del media, muy importante en la actualidad, consiste en el enrutado de audio o vídeo hacia dispositivos externos. Imaginemos que queremos programar una *app* que no solo sea capaz de reproducir media en el *smartphone*, sino también en otros dispositivos como *home cinema* o *smart TV* conectados a la misma wifi. Las librerías *media router* de Android están diseñadas para poder reproducir media en dispositivos remotos utilizando una interfaz común. Como programadores de una *app* que permita estas funcionalidades, utilizaremos la interfaz *MediaRouter* cuando nos

conectemos a dispositivos externos. Los fabricantes de dispositivos que permiten este tipo de conexiones responderán con clases de tipo *MediaRouterProvider*, que nos permitirán conectarnos a los dispositivos y enviar comandos de control de la reproducción. Un caso concreto son los dispositivos Google Cast. En este caso, Google nos presenta una librería más potente y concreta, el Cast SDK. Con este *framework*, será relativamente sencillo conectarnos a dispositivos Google Cast y controlar la reproducción remota de vídeo y audioclips.

Documentación del SDK de Cast:

https://developers.google.com/cast/docs/developers

Una característica esencial que nos permite manejar Android es el control del volumen de audio. Ya vimos en un ejemplo anterior cómo la llamada al método *setVolumeControlStream* nos permite especificar el canal en el que reproduciremos nuestros clips. Estudiamos los diferentes canales que permite Android para controlar por separado el volumen de las notificaciones, la música, los tonos de llamada, etcétera. De este modo, cuando el usuario modifique el volumen mientras nuestra *activity* está activa en pantalla, el volumen afectará al canal que hayamos elegido y a la reproducción de nuestros clips, estén actualmente en *play* o en pausa.

Por otra parte, podemos evitar cambios drásticos de sonido al desconectar el *jack* de audio o unos cascos inalámbricos. Al desconectarse estos periféricos, la salida de audio se redirecciona a los altavoces del dispositivo móvil, pudiendo causar inconvenientes para el usuario. Dependiendo de la aplicación, podremos elegir una estrategia como pausar el audio, o dejar que siga sonando, como en el caso de un videojuego. De todos modos, el sistema nos avisará mediante un *intent* con una acción de tipo *ACTION_AUDIO_BECOMING_NOISY*. En nuestra *app*, tendremos que crear un *BroadcastReceiver* que admita esta acción y actúe en consecuencia. Veamos un ejemplo:

```
private val intentFilter = IntentFilter(AudioManager.ACTION_AUDIO_BECOMING_NOISY)
private val myNoisyAudioStreamReceiver = BecomingNoisyReceiver()
private val callback = object : MediaSession.Callback() {
  override fun onPlay() {
     registerReceiver(myNoisyAudioStreamReceiver, intentFilter)
  }
  override fun onStop() {
     unregisterReceiver(myNoisyAudioStreamReceiver)
  }
}
private class BecomingNoisyReceiver : BroadcastReceiver() {
  override fun onReceive(context: Context, intent: Intent) {
     if (intent.action == AudioManager.ACTION_AUDIO_BECOMING_NOISY) {
        // Aquí pausamos el playback, o la acción que creamos oportuna
        Log.e(tag, "BecomingNoisyReceiver:onReceive:")
     }
  }
}
```

A la hora de reproducir audio, también tenemos la posibilidad de elegir una reproducción mono o estéreo, según los requerimientos de la aplicación. Para ello, es tan sencillo como utilizar las constantes de *AudioFormat: CHANNEL_OUT_MONO* o *CHANNEL_OUT_STEREO* en *setChannelMask* al crear un AudioTrack, como en el ejemplo siguiente:

```
val minBufferSize = AudioTrack.getMinBufferSize(16000,
    AudioFormat.CHANNEL_OUT_STEREO, AudioFormat.ENCODING_PCM_16BIT)
val track = AudioTrack.Builder().setAudioFormat(
    AudioFormat.Builder()
        .setChannelMask(AudioFormat.CHANNEL_OUT_STEREO)
        .setEncoding(AudioFormat.ENCODING_PCM_16BIT)
        .setSampleRate(44100)
        .build())
    .setTransferMode(AudioTrack.MODE_STREAM)
    .setBufferSizeInBytes(minBufferSize)
    .build()
track.play()
```

Dependiendo del nivel de control que deseemos, es posible que tengamos que movernos hacia las clases de más bajo nivel para poder configurar más detalles de la reproducción. En cualquier caso, Android tendrá una librería adecuada para cada problema, solo tendremos que evaluar qué nivel de control deseamos.

4. Motores de juegos: tipos y utilización

Un **motor gráfico o motor de videojuego** es la representación gráfica de unas determinadas rutinas de programación que ofrecen al usuario un diseño en un escenario gráfico, ya sea 2D o 3D.

La **principal tarea de un motor** es proveer al juego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, *scripts* de programación, animaciones, inteligencia artificial, gestión de memoria y un escenario gráfico.

Actualmente, hay una gran variedad de motores gráficos, *como* **Ogre (que es *open source*), Doom Engine, Quake Engine, Unity, CryEngine, Source engine, Unreal engine, Game Maker**, etc.

Estos motores suelen **proporcionar**:

- API y SDK para el desarrollo.
- Algunos motores permiten crear juegos sin necesidad de escribir código. Tan solo es necesario hacer uso de los diferentes mecanismos implementados y documentados en su API. Algunos requieren del uso de su propio lenguaje de programación.
- Conjunto de herramientas de edición.
- A pesar de no proporcionar conjuntos ya creados de elementos visuales, permiten su creación a través de las herramientas de edición del software proporcionado.

Es posible **clasificar** los motores en función de:

- **Según las facilidades ofrecidas:**
  - **Librerías gráficas**: según sus facilidades para el desarrollo y uso. SDL, XNA, DirectX, OpenGL.
  - **Motores**: si el motor ya tiene un desarrollo visual completo o requiere *scripts* de programación para la utilización de elementos visuales. Por ejemplo, OGRE, Unreal o id Tech son algunos de los que requieren del uso de *scripts* de apoyo para la funcionalidad.
  - **Herramientas de creación especializadas:** algunos de los motores se han desarrollado con un carácter exclusivo, orientados en su finalidad, como pueden ser videojuegos u otros tipos. Por ejemplo, GameMaker o ShiVa son para el desarrollo exclusivo de aplicaciones de juegos. En el caso de Unity, es posible utilizarlo para diferentes géneros.

- **Según la licencia:**
  - **Motores privados** (algunos con licencia gratuita)
  - **Motores *open source***

La elección de un determinado motor gráfico es muy subjetiva. En la mayor parte de las ocasiones esta elección estará condicionada por el tipo de aplicación del juego a desarrollar y los recursos disponibles de los que se dispone para su desarrollo.

Creación de videojuegos. Motores

Un motor gráfico es un software integrado generalmente en programas de diseño para dibujar y crear gráficos en la pantalla, es decir, digitales.

Concretamente, un motor gráfico se refiere al software que permite la creación y el desarrollo de videojuegos. Los motores gráficos tienen herramientas comunes que permiten al desarrollador aplicar a su diseño ciertas características, como puede ser el render.

El objetivo de los motores gráficos es trasladar las ideas creativas de los diseñadores a la pantalla, facilitando así la plasmación gráfica del juego.

Generación y manipulación de modelado 3D

Unity puede importar archivos desde Maya: solamente hay que colocar un archivo *.mb* o *.ma* en la carpeta correspondiente del proyecto. Para ello, tendremos que asegurarnos de tener Maya instalado en el ordenador. Puede importar:

- Nodos con posición, rotación y escala
- *Meshes* con colores de vértices, normales y hasta dos conjuntos de UV
- Materiales con *Texture* y color *diffuse*
- Animaciones Fk & KI
- Animaciones basadas en huesos
- *BlendShapes*

Cuando importamos formatos en 3D, solemos recurrir a dos tipos de archivos: los formatos de archivo 3D importado (*.OBJ*) o archivos propios de la aplicación 3D (por ejemplo, un archivo de blender como *.Blend*). Esto ofrece muchas ventajas, ya que solo se exportan los datos que se necesitan. También podemos importar mediante conversión de archivos como, por ejemplo, blender, cinema 4D, *.ma*, etcétera.

Además, los archivos que importemos se almacenarán como *assets* de Unity y, generalmente, se dividirán en otros varios *assets*.

4.1. Conceptos de animación. Animación 2D y 3D

En la actualidad, los dispositivos móviles forman parte del día a día de las personas, agrupando dentro de un mismo dispositivo todas las necesidades de los usuarios. Este hecho hace que los dispositivos móviles sean usados tanto para motivos laborales o de comunicación, como también para el ocio. Aquí es donde surge el desarrollo de aplicaciones animadas para los dispositivos.

Una **animación** es el cambio de una o varias propiedades de un objeto que hacen percibirlo con un aspecto distinto a lo largo del tiempo.

La base de creación de estas animaciones reside en la **programación**. Android ofrece una serie de mecanismos dedicados a la creación de animaciones para objetos tanto 2D como 3D.

- **Canvas**: se trata de la plantilla que permite definir un control a nivel de interfaz de usuario en las aplicaciones. Canvas puede suponer la representación de cualquier objeto como óvalos, líneas, rectángulos, triángulos, etc.
- **Animadores (*animators*):** es la propiedad que permite añadir a cualquier objeto una determinada animación mediante el uso de propiedades o estilos de programación.
- **Dibujables animados (*drawable animation*):** permite cargar una serie de recursos *drawable* para crear una animación. Utiliza animaciones tradicionales como, por ejemplo, poner una imagen detrás de otra en orden como si fuera una película.
- **OpenGL**: se trata de una librería para gráficos de alto rendimiento 2D y 3D. Es una de las más importantes. Android incluye soporte para su uso.

4.2. Arquitectura del juego. Componentes

Antes de comenzar con el desarrollo de un juego para plataformas móviles, es necesario previamente definir cuál va a ser su **arquitectura**. Esta arquitectura permitirá detallar cómo será la estructura de desarrollo de la aplicación.

Dicha estructura estará formada por una serie de **bloques** con una función específica dentro del juego:

- **Interfaz de usuario**: se encargará de recoger todos los eventos que el usuario ha creado.
- **Lógica del juego**: es la parte central del videojuego. Se encargará de procesar todos los eventos del usuario y dibujar continuamente la escena del juego. Esto es lo que se conoce como ***game loop***. Se comprueba continuamente el estado del juego y se dibuja para cada estado una nueva interfaz, repitiendo este proceso de forma casi infinita.

  En este bloque se controlarán también las colisiones y los *sprites*. Mediante el uso de librerías como OpenGL se modelan y diseñan los diferentes personajes dentro del juego. A través de esta, es posible generar animaciones en personajes haciendo uso de *sprites* (imágenes con transparencia). Esto se conoce como **renderizado de canvas.**

  Cada renderizado quedará encapsulado en un ***frame* de animación**.

- **Recursos utilizados**: dentro de la lógica del juego también es necesario controlar todos los efectos de sonido e imágenes.

  Este bloque es la parte fundamental para el desarrollador. Todas las funciones tienen que ser programadas y controladas a través de código.

  Los juegos, a diferencia de las aplicaciones, necesitan de un mayor consumo de los recursos del dispositivo, pudiendo, en muchos casos, utilizar casi la totalidad de los mismos. Es, por tanto, tarea del programador optimizar el uso de estos recursos.

- ***Framework* Android**: Android proporciona un *framework* potente que permite animar una gran cantidad de objetos y representar dichos objetos de una multitud de formas. Para ello, cuenta con diferentes **mecanismos** que ofrecen al desarrollador estas funcionalidades:
  - **Animación de propiedades** (*property animations*): permite definir algunas propiedades de un objeto para ser animado, definiendo opciones como: duración de una animación, tiempo de interpolación, repetición de comportamientos o animaciones, agrupación de series de animaciones de forma secuencial, *frames* de actualización de una animación, etc.
  - **Animación de vistas** *(view animations):* permite hacer uso de los diferentes

mecanismos de animación de vistas, como translaciones, rotaciones, escalados, etc. Con ellos se da la sensación de transformación de una imagen en otra en un determinado momento.

- **Animación de dibujables** (*drawable animations):* haciendo uso de los recursos *drawable,* se pueden crear una serie de animaciones como si de una película se tratara.

- **Salida**: son la sucesión de escenas que se van actualizando en la interfaz de usuario.

Estos *frames* son procesados uno detrás de otro ofreciendo al usuario una sensación de continuo movimiento y animación. Todos estos bloques de *frames* dan como resultado la escena del juego en los diferentes momentos del tiempo.

4.3. Motores de juegos: tipos y utilización

Un **motor gráfico o motor de videojuego** es la representación gráfica de una serie de rutinas de programación que ofrecen al usuario un diseño en un escenario gráfico 2D o 3D.

La principal tarea de un motor es proveer al juego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, *scripts* de programación, animaciones, inteligencia artificial, gestión de memoria y un escenario gráfico.

En la actualidad existen una gran variedad de motores gráficos, como: **Ogre (que es *open source*), Doom Engine, Quake Engine, Unity, CryEngine, Source engine, Unreal engine, Game Maker**, etc.

Estos motores suelen **proporcionar**:

- API y SDK para el desarrollo.
- Algunos motores permiten crear juegos sin necesidad de escribir código. Tan solo es necesario hacer uso de los diferentes mecanismos implementados y documentados en su API. Algunos requieren del uso de su propio lenguaje de programación.
- Conjunto de herramientas de edición.

- A pesar de no proporcionar conjuntos ya creados de elementos visuales, permiten su creación a través de las herramientas de edición del software proporcionado.

Es posible **clasificar** los motores en función de:

- **Según las facilidades ofrecidas**:
  - **Librerías gráficas**: según sus facilidades para el desarrollo y uso (SDL, XNA, DirectX, OpenGL).
  - **Motores**: si el motor ya tiene un desarrollo visual completo o requiere *scripts* de programación para la utilización de elementos visuales. Por ejemplo, OGRE, Unreal o id Tech son algunos de los que requieren del uso de *scripts* de apoyo para la funcionalidad.
  - **Herramientas de creación especializadas:** algunos de los motores se han desarrollado con un carácter exclusivo, orientados en su finalidad, como pueden ser videojuegos u otros tipos. Por ejemplo*:* GameMaker o ShiVa son para el desarrollo exclusivo de aplicaciones de juegos. Por su parte, Unity es posible utilizarlo para diferentes géneros.

- **Según la licencia**:

- **Motores privados** (algunos con licencia gratuita)
- **Motores *open source***

La elección de un determinado motor gráfico es muy subjetiva. En la mayor parte de las ocasiones esta elección estará condicionada por el tipo de aplicación del juego a desarrollar, y los recursos disponibles de los que se dispone para su desarrollo.

4.4. Ventajas de la utilización de un motor de juegos

En general, los videojuegos son aplicaciones software, y dependiendo de las características del juego, pueden llegar a ser aplicaciones de gran complejidad tecnológica. Al fin y al cabo, la mayoría de ellos tratan de emular algún tipo de realidad: reglas internas, leyes físicas, múltiples personajes moviéndose y actuando de forma independiente y relativamente inteligente, música y efectos de audio y vídeo, gráficos tridimensionales, compartiendo su estado a través de Internet con otros jugadores, etc.

Por lo tanto, podemos comprender que el desarrollo de un videojuego no es tarea sencilla; es complejo y requiere mucho tiempo y recursos. Por ello, necesitamos los **motores de juegos**.

Como otros tipos de *frameworks*, los motores de juegos se crearon con la intención de ofrecer un marco de trabajo que permitiese a los desarrolladores evolucionar en la creación del producto final, sin depender de los detalles tecnológicos más frecuentes en su desarrollo. Con un motor de juegos podremos centrarnos en la creación artística de nuestro juego, sin perder el tiempo en complejas técnicas de renderizado, de optimización de velocidad, de algoritmos físicos (saltos, caída de objetos, palancas, colisiones, etc.), de algoritmos de IA o del problema de compatibilidad entre dispositivos.

Imaginemos que nuestro objetivo es desarrollar un videojuego de tiros en primera persona (FPS), una especie de Doom. Sin un motor de juegos, nos veríamos obligados a programar desde cero un motor de renderizado 3D, como hicieron en id Software. Además, necesitaríamos una serie de recursos: una librería matemática, para el cálculo de colisiones de objetos en 3D, y otra para las emulaciones físicas; desarrollar un constructor, el cual nos permitiese diseñar nuestras escenas y laberintos como hace un arquitecto con sus proyectos; diseños gráficos para las texturas de los muros y de los personajes; un conjunto de algoritmos de inteligencia artificial, para que los enemigos sean suficientemente inteligentes y divertidos de combatir o para crear un plano lógico del laberinto que hemos diseñado; e implementar un algoritmo *path finder* como A*. Este algoritmo se centra en encontrar el camino de menor coste entre un origen y una meta, tratando de evadir aquellos caminos con más coste. El coste se verá determinado por el programador, pudiendo ser camino más largo, camino con más enemigos, camino con más trampas, etc. Así los agentes sabrán moverse por él y nos perseguirán con eficiencia.

Además, también necesitaríamos: una librería para administrar y reproducir los sonidos; un módulo para acceder a internet y sincronizar los datos entre jugadores, si queremos un juego en red; doble o triple codificación, en caso de que queramos que sea compatible con diferentes dispositivos.

Como vemos, el desarrollo es arduo, complejo y extenso, con un gran trabajo detrás. Un motor de juegos nos ahorrará la mayoría de estos problemas que pueden surgir durante el desarrollo de cualquier tipo de juego. Todos los mecanismos que solucionan esos problemas ya han sido implementados para que no tengamos que preocuparnos una y otra vez por ellos.

Desde los primeros videojuegos, los programadores empezaron a tomar nota de los problemas y necesidades que se repetían una y otra vez. ¿Por qué reinventar la rueda? Cuando un nuevo motor de juegos aparece y consigue un hueco en el mercado, es muy probable que los demás se hayan quedado anticuados o este disponga de mejores capacidades o sea más sencillo o más potente. Así, podremos escoger el que mejor se adapte a nuestras necesidades, pero empezar desde cero nuestro proyecto sin un motor de juegos sería tremendamente arriesgado.

Como programadores en proceso de aprendizaje, podríamos apostar por un proyecto sencillo y aprender de la experiencia. Sin embargo, si el juego es algo más complejo que las tres en raya y deseamos terminarlo con éxito, será mejor utilizar un motor de juegos y aprender a utilizarlo adecuadamente. En el caso de querer programar nuestro juego en Java o Kotlin, quizá la mejor alternativa sería LibGDX, ya que es gratuito y *open source*, tiene módulo de colisiones, renderizado, IA y es compatible con múltiples plataformas, entre otras ventajas. Otros usuarios preferirán Unity por su gran difusión y relativa facilidad de uso, y que existen miles de tutoriales y recursos dirigidos a este *framework*. A pesar de ello, es una plataforma comercial y no permite programar en Java, sino en C#. Otros valorarán la potencia de Unreal Engine, pero deberás sufrir una empinada curva de aprendizaje y programar en lenguaje C.

En cualquier caso, todos ellos tendrán sus pros y sus contras, pero definitivamente utilizar cualquiera de ellos será mejor que el arduo e incierto camino del desarrollo de un videojuego desde cero, sin un motor que nos facilite el fascinante pero duro camino de la creación de juegos.

4.5. Áreas de especialización, librerías utilizadas y lenguajes de programación

La elección y utilización de librerías gráficas durante el desarrollo de un juego siempre es necesaria. Ello permite realizar determinadas animaciones sobre objetos de una manera más sencilla. Estas librerías se pueden clasificar por **áreas de especialización** en función de su funcionalidad:

- **Renderizados y efectos:** OpenGL, Direct3D, GKS, PHIGS, PEX, GKS, etc.
- **Basados en gráficos de escena:** OpenGL Performer, Open Inventor, OpenGL Optimizer, PHIGS+, etc.
- **Librerías de herramientas gráficas:** World Toolkit, AVANGO, Game Engines, etc.

Es posible programar un videojuego en una multitud de lenguajes. Los más utilizados en el desarrollo de videojuegos son **C**, **C++**, **C#** y **Java**.

El uso de un lenguaje u otro viene definido por el tipo de juego que se quiere desarrollar. Los juegos 2D o plataformas, que trabajan con *sprites* sencillos, normalmente están basados en el lenguaje C. En el caso de juegos cuya complejidad gráfica es mayor, sobre todo cuando se va a trabajar con objetos tridimensionales y sus propiedades, la mayor parte de programadores usan C++, C# y Java. Al requerir del uso de una máquina virtual, en ocasiones son menos elegidos, aunque su potencia en desarrollo de juegos también es alta.

Existen un gran número de librerías que se pueden utilizar durante el desarrollo de un juego. Algunas de las **librerías** son las siguientes:

- **Allegro**: librería libre y de código abierto basada en lenguaje C. Permite el uso de elementos gráficos, sonidos, dispositivos como teclado y ratón, imágenes, etc.
- **Gosu**: librería que permite el desarrollo de juegos en 2D y basada en lenguaje C++ y Ruby. Es de software libre bajo licencia del MIT. Provee de una ventana de juego que permite el uso de teclado, ratón y otros dispositivos. Se caracteriza por su uso para sonidos y música dentro de un juego.
- **SDL**: conjunto de librerías para el diseño de elementos 2D, también de software libre. Permite la gestión de recursos multimedia como sonidos y música, así como el tratamiento de imágenes. Está basada en lenguaje C, aunque permite el uso de otros lenguajes, como: C++, C#, Basic, Ada, Java, etc.
- **libGDX**: basada en Java. Librería orientada a su uso en aplicaciones multiplataforma que permite escribir el código en un solo lenguaje y posteriormente exportarlo a otros. Permite una integración fácil con otras herramientas Java.
- **LWJGL**: librería destinada al desarrollo de juegos en lenguaje Java. Proporciona acceso al uso de librerías como OpenGL.
- **OpenGL**: librería de gráficos para el desarrollo de juegos 2D y 3D. Es una de las librerías más utilizadas hoy en día. Es de software libre y código abierto. Permite el uso de elementos básicos como las líneas, puntos, polígonos, etc., así como otros elementos de mayor complejidad, como son: texturas, transformaciones, iluminación, etc.
- **Direct3D**: conjunto de librerías multimedia. Es el gran competidor de OpenGL en el mundo de los juegos. Permite el uso de elementos como líneas, puntos o polígonos, así como gestión de texturas o transformaciones geométricas. Es propiedad de Microsoft.

4.6. Componentes de un motor de juego

Un **motor de juegos** es una parte fundamental del código de programación de un juego. Este motor gráfico se encarga de la mayor parte de los aspectos gráficos de un juego.

Una de sus tareas es establecer la comunicación y aprovechar todos los recursos que una tarjeta gráfica ofrece.

Los **componentes** principales de un motor de juegos son:

- **Librerías**: todas aquellas librerías de las que se hace uso para el desarrollo de figuras, polígonos, luces y sombras, etc.
- **Motor físico**: es el encargado de gestionar las colisiones, animaciones, *scripts* de programación, sonidos, físicos, etc.
- **Motor de renderizado**: es el encargado de renderizar todas las texturas de un mapa, todos los relieves, suavizado de objetos, gravedad, trazado de rayas, etc.

Estos componentes recogen de manera global todos los elementos que aparecen dentro de un juego. Cada uno de estos elementos forma parte de un conjunto de **recursos** que en todo motor gráfico se pueden encontrar:

- ***Assets***: todos los modelos 3D, texturas, materiales, animaciones, sonidos, etc. Este grupo representa todos los elementos que formarán parte del juego.
- **Renderizado**: todas las texturas y materiales en esta parte hacen uso de los recursos diseñados para el motor gráfico. Esto mostrará el aspecto visual y potencial de un motor gráfico.
- **Sonidos**: es necesario configurar dentro del motor cómo serán las pistas de audio. El sonido del videojuego dependerá de la capacidad de procesamiento de estos sonidos. Algunas de las opciones configurables son: modificación del tono, repetición en bucle de sonidos (*looping*), etc.
- **Inteligencia artificial**: es una de las características más importantes que puede desarrollar un motor gráfico. Esto añade estímulos al juego, permitiendo que el desarrollo del mismo suceda en función de una toma de decisiones definida por una serie de reglas. Además, define el comportamiento de todos los elementos que no son controlados por el jugador usuario, sino que forman parte de los elementos del juego.
- ***Scripts* visuales**: no solo es posible ejecutar porciones de código definidas en el juego, sino que además se pueden ejecutar en tiempo real dentro del aspecto gráfico del juego.

- **Sombreados y luces**: el motor gráfico dota de colores y sombras a cada uno de los vértices que forman parte de la escena.

Como se ha podido comprobar, las tareas que componen un motor gráfico exigen la utilización de un gran número de recursos dentro del equipo. De ahí que, cuanto mayor sea la capacidad de procesado y velocidad de una tarjeta gráfica, mejor será el resultado de la escena de un juego. Para reducir el coste de esto, algunos motores emplean una serie de técnicas que permiten renderizar los terrenos o los materiales y que no consumen recursos, sino que aparecen dentro del espacio visual, lo cual se conoce como ***culling*.**

Los motores gráficos son un aspecto clave dentro de un juego. Han sido creados exclusivamente para el desarrollo de los mismos, y hoy en día son la herramienta fundamental de creación de videojuegos. La evolución de los juegos y el entretenimiento está ligada a la evolución de los motores de juegos.

4.6.1. Motor gráfico o de renderizado (2D/3D)

Técnicas de animación 2D y 3D mediante motores

Animación 3D: MECANIM

MECANIM es el sistema de Unity que permite la animación a objetos 3D. Este sistema posee *animation clips* estructurados en el *Animator controller*, los cuales nos permitirán crear animaciones de modelos 3D. Además, Unity también contiene el sistema Avatar, que permite otorgar características especiales a los personajes humanoides. Estas tres funcionalidades se integran en el *Animator component*.

El funcionamiento sería el siguiente:

- Los *animation clips* son importados o creados en Unity.
- Estos clips se agregan en un *Animator controller* que se visualiza como un *asset* en la ventana del proyecto.
- El personaje se integra con Avatar a través de un mapeo.
- El personaje tendrá un componente *animator* adjunto para darle animación.

Animación 2D

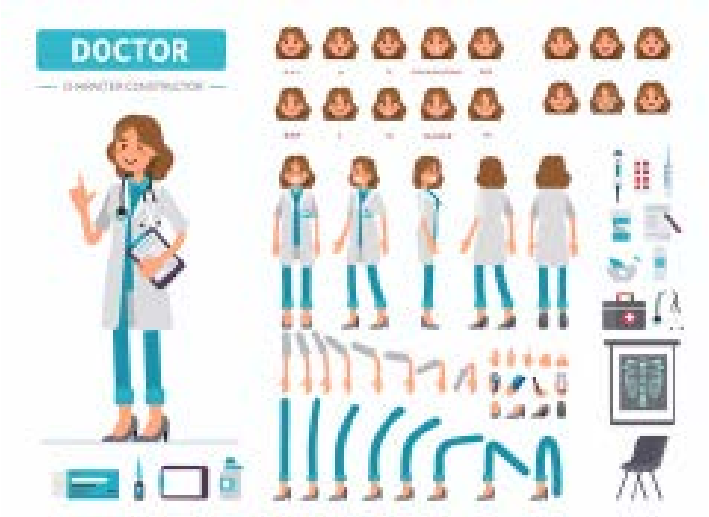
Para esto nos hará falta nuestro personaje 2D con las imágenes de la animación separadas en poses clave.

Por cada pose se diseña una imagen del personaje: una imagen para cuando salte, otra para cuando duerma, otra para cuando hable… y así tendremos una o varias imágenes para cada acción que realice. No obstante, habrá acciones que requieran más de una imagen, ya que en cada movimiento de un salto, por ejemplo, el personaje mantendrá una postura distinta.

Para crear la animación donde el personaje corra, por ejemplo, deberemos seleccionar todas las imágenes referidas a esta acción y arrastrarlas a la escena. Entonces se creará una ventana que posibilitará la creación de esa animación a la que tendremos que nombrar. En este caso, le daremos el nombre de *Correr*.

El personaje aparecerá en la escena y podremos ver cómo ha quedado la animación pulsando *Play*. Si queremos cambiar el tamaño del personaje, lo haremos desde *Transform*, en los parámetros X e Y. Así, una vez tengamos el tamaño que queremos, arrastraremos el elemento *Correr* (que será nuestro personaje) al área de imágenes.

De todas formas, existe otra manera de trabajar con animaciones, y es haciéndolo con un único *sprite* dividido en partes (es decir, la cabeza por un lado, el brazo por otro, etcétera).



Creando un *sprite* múltiple en Unity, podemos crear cada parte como un *sprite* independiente y juntarlo para formar nuestro personaje, del cual podríamos ir animando pieza a pieza creando fotogramas clave en una línea de tiempo.

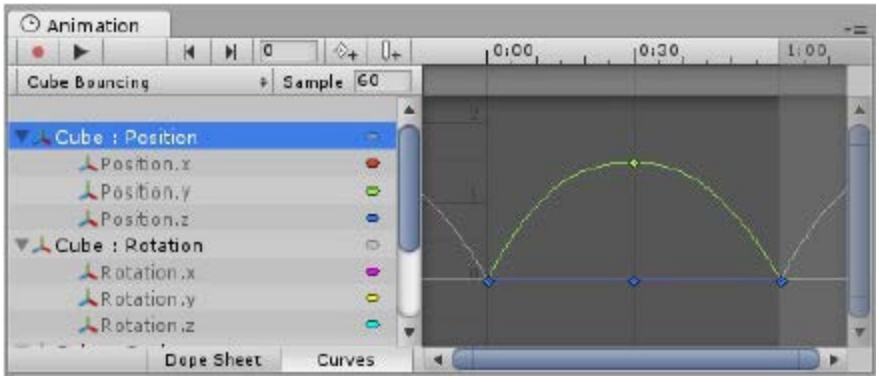
En cuanto a estos dos métodos, no hay uno mejor ni otro peor, simplemente son diferentes y podremos usarlos según el estilo que busquemos para nuestro proyecto.

Vamos a hablar de los procesos más importantes:

- **División de animaciones:** los modelos con los que trabajaremos pueden ser de varios tipos:
  - **Animaciones predivididas:** exportadas una a una, divididas con su nombre adecuado.
  - **Animaciones sin dividir:** en estos casos, tenemos un clip de duración extensa donde podemos definir el rango de *frames* que corresponde a cada secuencia de animación.

  Una vez creados los clips de animación, estos pueden servir para animar objetos o propiedades a través de la ventana de animación de Unity.

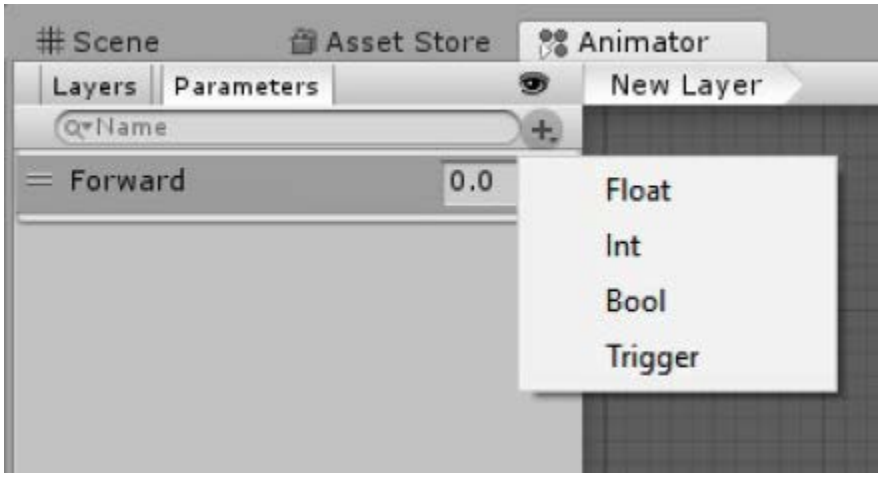
- **Curvas de animación:** permiten añadir datos adicionales a nuestros fotogramas claves y se pueden conectar con clips importados desde las *Animation import settings*.
  Estas curvas nos permitirán hacer más suaves las transiciones de las animaciones o agregar nuevos *timing* utilizando con más facilidad las leyes de la animación.



- **Curvas de animación**: permiten añadir datos adicionales a nuestros fotogramas claves y se pueden conectar con clips importados desde las *Animation import settings*.
  Estas curvas nos permitirán hacer más suaves las transiciones de las animaciones o agregar nuevos *timing* utilizando con más facilidad las leyes de la animación.
- **Estados de máquina**: las acciones típicas como andar, saltar y demás están referenciadas como *states*, ya que el personaje se encuentra en determinado estado cuando realiza determinada acción. Para que el personaje pueda pasar de un estado a otro, deberá tener *State transitions*. Todo ello forma la *State machine*.

  Para representar los estados, a menudo se utiliza un diagrama en el que los nodos representan los estados y las flechas las transiciones. Así, cada estado tendrá un *motion* que se reproducirá cuando la máquina esté en ese estado.

  - **Parámetros de animación en los estados de máquina:** los parámetros de animación son variables definidas dentro de un *Animator controller* desde el que se pueden asignar valores de *scripts*. Se pueden configurar los valores del parámetro en la sección *Parámetros de la ventana Animator*. Pueden ser los siguientes:
    - *Int*: un número entero.
    - *Float*: un número con decimales.
    - *Bool*: un valor de *true* o *false* (está representado por una casilla de verificación).
    - *Trigger*: un parámetro *booleano* que se reinicia desde el controlador cuando se realiza una transición (está representado por un botón).



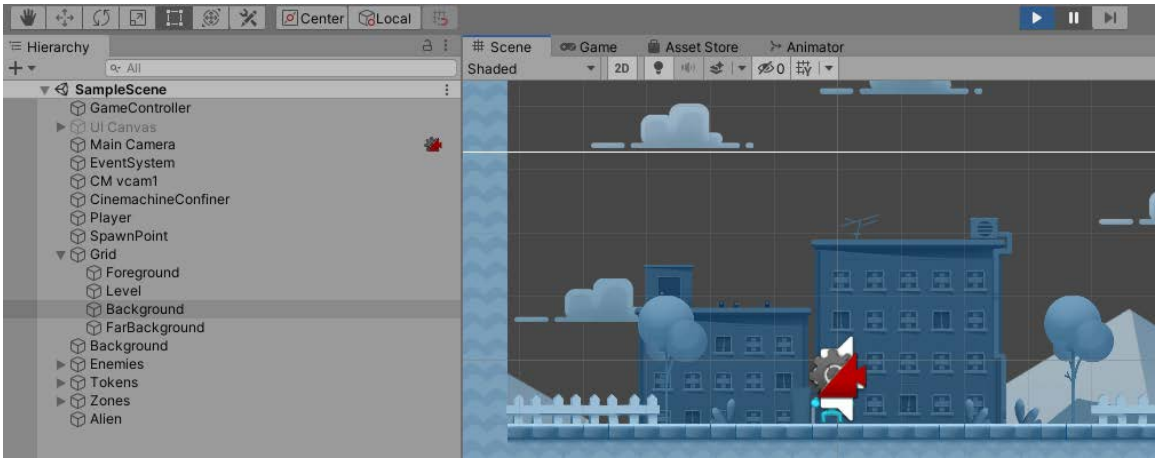
Estas variables ya os sonarán de la UF anterior. Las podemos usar para que los estados de máquina cambien cuando lleguen a cierto requisito (como, por ejemplo, cuando nuestra variable *float* llegue a 8.5; podemos hacer que nuestro *state maquine* pase de andar a correr).

Al final de esta unidad formativa veremos ejemplos de cómo realizar cada una de las animaciones anteriormente descritas.

4.6.2. Grafo de escena

En Unity, el grafo de escena nos muestra una visión global de los elementos que compondrán la escena, una de las múltiples de las que puede componerse el juego. Entendemos la escena como un nivel de nuestro juego, de modo que podremos crear y diseñar diferentes escenas, así como pasar de una a otra según el jugador vaya avanzando en su aventura. Dentro del grafo de escena crearemos los elementos gráficos que la componen. Algunos serán meramente decorativos, como fondos, nubes, etc.; otros tendrán una función activa en el juego, como ascensores, enemigos, etc.

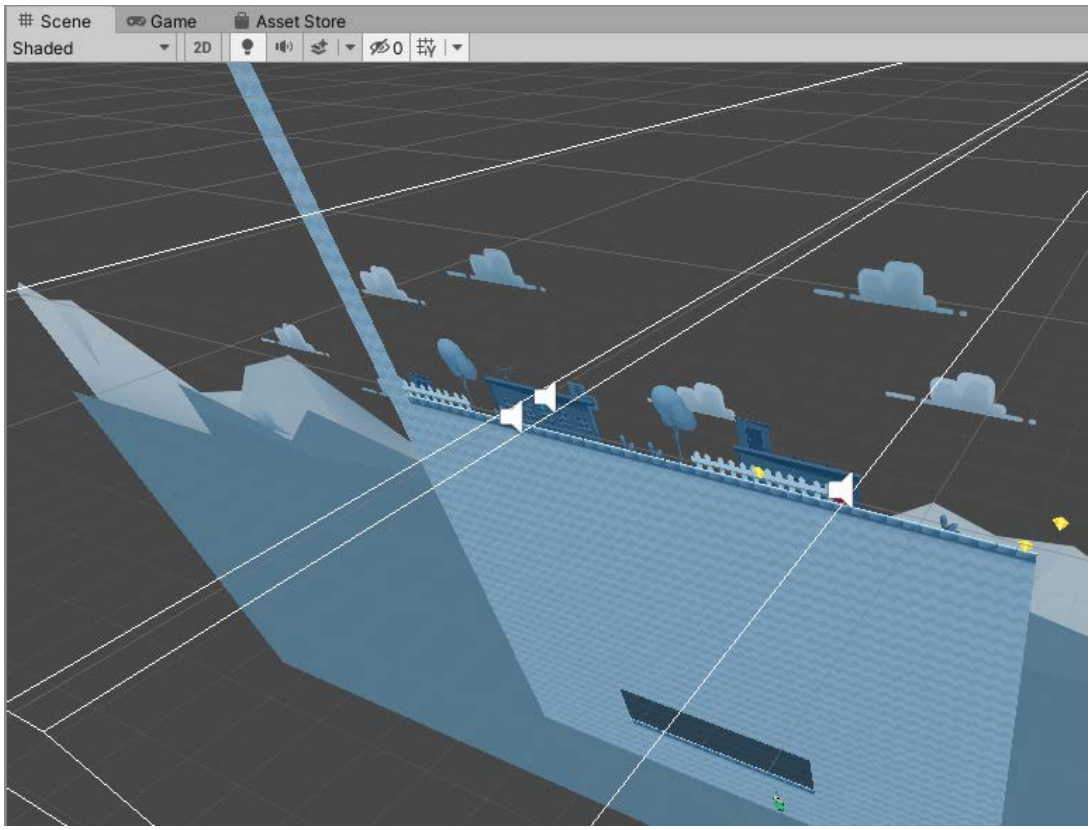
Como vemos en la siguiente imagen, desde la ventana de escena podemos probar cómo se vería el juego en funcionamiento pulsando la tecla *Play*:



Para modificar los atributos de cada elemento de la escena, podremos seleccionarlo desde el árbol de elementos, o directamente haciendo clic con el botón izquierdo sobre el objeto visible en el grafo de escena. Haciendo clic con el botón derecho del ratón sobre la ventana, podremos arrastrar completamente la escena en las cuatro direcciones, y observar así más fácilmente los elementos situados en las zonas más alejadas cuando la escena excede el tamaño de la pantalla.

Dependiendo de si el juego es 2D o 3D, tendrá más sentido utilizar elementos 2D o 3D en la escena. No obstante, no es obligatorio, ya que en un juego 3D podemos modelar como *sprites* planos a los enemigos si consideramos que ello tiene mayores ventajas que utilizar modelos 3D. Incluso en un videojuego 2D, no solo serán importantes las coordenadas X e Y. La coordenada Z de profundidad ayudará a añadir elementos que se solapen en la distancia, como el *skyline*, las nubes, una verja, una farola… Así, la diversa profundidad de los elementos logrará un aspecto más realista y divertido.

El **grafo de escena** presenta una barra de herramientas con algunas opciones que nos ayudarán en el diseño. Por ejemplo, la opción *2D* nos permite observar la escena como se verá en la pantalla del juego. Si la desactivamos, veremos una representación tridimensional de las capas que componen la escena, la profundidad:

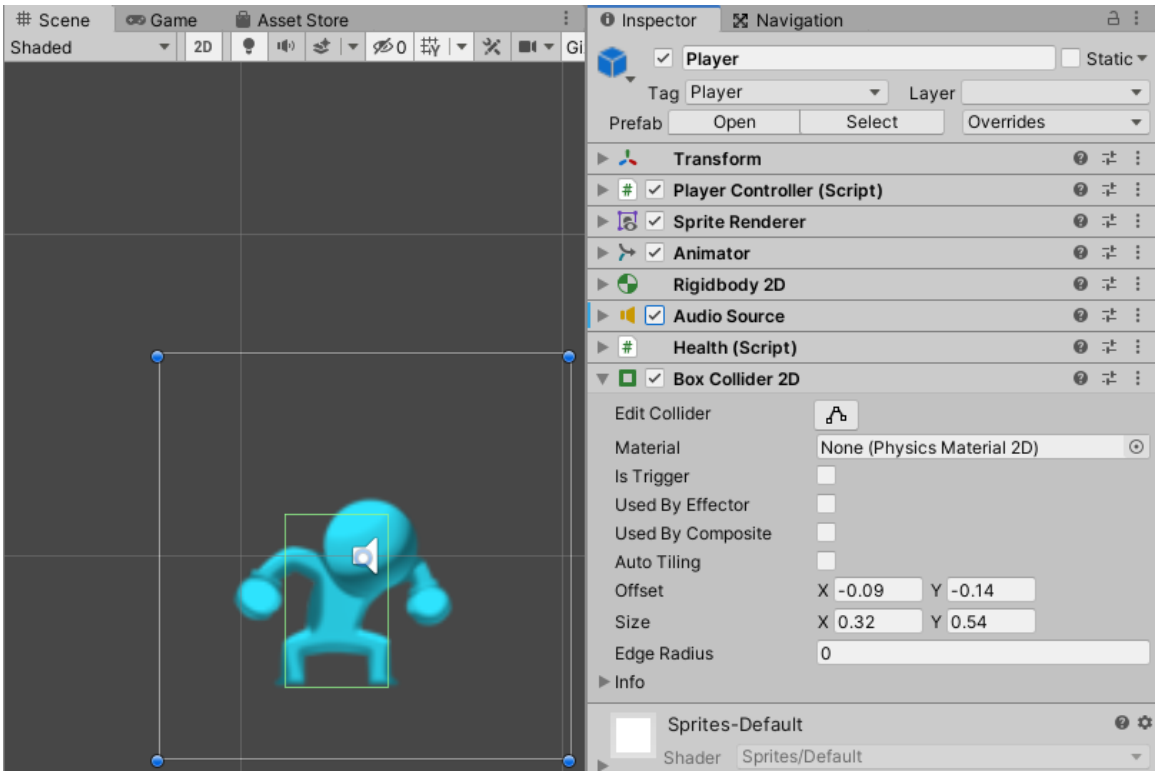


Podremos navegar por la escena 3D mediante el botón derecho y la rueda central del ratón, entre otras opciones. De este modo, podremos ver qué capas están delante de otras y ordenar la escena en la coordenada Z de profundidad. Otras opciones del menú de herramientas se refieren a la activación de los puntos de luz, del sonido, el número de elementos ocultos de la escena o el tipo de vista (*shaded, wireframe*, etc.), entre otras.

4.6.3. Detector de colisiones

La detección de colisiones es fundamental en la mayoría de videojuegos. Cuando en un First Person Shooter corremos por los pasillos del laberinto con nuestro bazuca en la mano, no podemos traspasar los muros, nos chocamos contra ellos, como es de esperar, gracias al detector de colisiones 3D. Cuando el caza espacial intergaláctico de nuestro videojuego tipo arcade dispara a las naves enemigas, el detector de colisiones 2D será el que calcule dónde penetró el láser, y si la nave enemiga debe reventar en el espacio con un fulgurante resplandor, o tan solo ser dañada y escupir humo.

En Unity, de nuevo, esto es una tarea sencilla. A cada elemento que creamos en la escena podemos asociarle un elemento *collider*. Este elemento tendrá la forma que queramos darle y se moverá junto al elemento asociado de forma invisible mientras hace su trabajo; comunicándonos con un evento de colisión siempre que toque a otro *collider*. Dependiendo de nuestro juego, utilizaremos *colliders* 2D o 3D. Tengamos en cuenta que las matemáticas de cálculo de colisiones son complejas, de modo que cuantos más elementos se encuentren en escena, el cálculo completo requerirá más recursos y tardará un mayor tiempo en completarse. En terminales móviles antiguos, muchos elementos en escena podrían saturar el sistema.



En la imagen anterior, vemos como nuestro elemento *player*, está rodeado de un rectángulo verde, conocido como *box collider 2D*. Para la mayoría de juegos sencillos, un rectángulo será suficiente. Si el elemento tiene forma de pelota, podremos utilizar un *circle collider 2D*.

Unity nos ofrece múltiples opciones para la forma de los *colliders*. Lo mismo sucederá en una escena 3D, en la que tendremos diferentes formas básicas de objetos de colisión, como cilindro, esfera, cubo, etc. Para asociar un *collider* al elemento actual seleccionado, pulsaremos el botón *Add component* al final del *Inspector*, escogeremos la opción *Physics 2D* y elegiremos el *collider* que creamos que se adapta mejor al juego.

4.6.4. Motor de físicas

En un motor de juegos, el motor físico es el módulo responsable del cálculo del movimiento y de la interacción natural de los elementos dentro de la escena con respecto a algún tipo de emulación de leyes físicas. En realidad, el módulo de colisiones pertenece a este motor físico, pero dispone además de otros mecanismos.

En Unity, el principal elemento del módulo físico es el *rigidbody*. El *rigidbody* podrá tener asociado un *collider*, y tendrá un objeto *transform* que permitirá establecer la posición y el ángulo del elemento dentro de la escena. Desde el código, podremos manipular nuestros objetos *rigidbody* para mover los personajes dentro de la escena, y sus *colliders* nos indicarán cuando se tocan. Los *rigidbody* tienen un atributo *bodyType* que definirá su movimiento físico en la escena. Existen tres tipos:

- ***Dynamic*:** un *rigidbody* de este tipo se moverá bajo estímulo y no cambiando directamente los valores de su *transform*. Será sensible a la gravedad que simulemos, y a las fuerzas de empuje que le apliquemos. Si tiene un *collider* asociado, este tipo de objetos detectará las colisiones con todos los demás objetos de la escena. Es el tipo más común para los elementos de nuestra escena que pensemos que deben moverse, como el jugador, los enemigos, cajas, etc.
- ***Kinematic*:** estos objetos no se verán afectados por las leyes básicas de la física, sino que tendrá que ser el programador el que indique su posición. Por ello no tiene atributos como masa o fuerza de rozamiento. Además, estos objetos solo colisionan con otros *dynamic*. Al ser más sencillos, serán menos exigentes con los recursos del sistema. Podrían ser puentes levadizos, puertas corredizas, rayos láser, etc.
- ***Static*:** los elementos *static* no se moverán bajo ningún tipo de estímulo. Si reciben una colisión, se comportarán como objetos inamovibles de masa infinita. Será el caso de muros de un laberinto, que no deberán ceder cuando nos estrellemos

contra ellos. En cualquier caso, solo colisionarás con objetos *dynamic*. Son los elementos más sencillos y menos exigentes con el sistema.

Además de los elementos *rigidbody* y de los *colliders*, el motor físico de Unity dispone de otros componentes menores para la mejora de las simulaciones físicas:

- ***Physics material*:** permite ajustar el índice de rozamiento y elasticidad de los componentes, para que cuando se vean arrastrados o colisionen, muestren un comportamiento físico más realista.
- ***Joints*:** permiten unir diferentes objetos mediante una articulación, para poder crear objetos articulados complejos.
- ***Constant force:*** permite aplicar una fuerza constante a un objeto, de modo que se vea empujado como lo haría una bola de cañón, acelerándose con el tiempo después de haber sido disparada.
- ***Effectors:*** permiten, unidos a los *colliders*, dirigir las fuerzas físicas de un modo personalizados cuando existe una colisión entre objetos. Por ejemplo, podemos lograr efectos como las plataformas, que nos permiten saltar hacia ellas atravesándolas, pero luego nos quedaremos sobre ellas sin volver a caer. También elementos de atracción o repulsión sobre otros elementos, hacer que los elementos floten, o cambiar la fuerza y el ángulo de la fuerza.

4.6.5. Motor de Inteligencia Artificial

Los ordenadores siguen siendo máquinas, poco más que mecanismos de relojería fabricados con semiconductores. Su funcionamiento es programable, pero la mayoría de los programas que desarrollamos para ellas son lineales, predecibles, limitados. Son muy potentes y la velocidad de sus cálculos supera con creces la de nuestro cerebro. Sin embargo, aún no son capaces de resolver problemas que a nosotros nos parecen triviales, no muestran creatividad ni personalidad. La IA (inteligencia artificial) es un área del desarrollo informático que pretende traspasar estas capacidades humanas, a las máquinas.

El desarrollo de videojuegos ha sido siempre un gran promotor de la evolución en IA y no solo en el mundo del ajedrez. Hoy cada vez más, gracias al gran desarrollo del hardware y el software, la IA está presente en casi todas partes: móviles, centralitas, cámaras, vehículos, etc. Así, podemos beneficiamos de su poder para aumentar la calidad y cantidad de servicios que las máquinas nos ofrecen en todos los campos. En el mundo de los videojuegos, hoy más que nunca la IA tiene un papel imprescindible.

Cada vez pueden desarrollarse aventuras gráficas más potentes, jugadores automáticos (*non-player characters*) que actúan con su propia personalidad, grupos de agentes inteligentes que se comunican y actúan como manadas de animales, o escuadrones de guerreros que avanzan por terrenos complejos, sin que su comportamiento y su movimiento esté programado de antemano. Todo esto se lo debemos a los miles de algoritmos de IA que matemáticos y programadores han ido construyendo durante décadas. Como desarrolladores, es bueno poseer un conocimiento básico de algunos de esos algoritmos. Pero como hemos mencionado en varias ocasiones, no tiene sentido reinventar la rueda. Además, debemos tener en cuenta que los cálculos necesarios para estos algoritmos requieren muchos recursos del sistema, y tendremos que hacerlos compatibles con el proceso de renderizado. Por ello, los algoritmos tienen que ser lo más eficientes posibles.

La mayoría de los motores de juegos actuales disponen de módulos de IA. Mediante su uso, los enemigos que programemos tendrán la inteligencia necesaria y podrán resultar una verdadera amenaza para los jugadores más quemados. Además el juego sabrá adaptarse al nivel del jugador, de modo que no sea ni demasiado fácil ni imposible de batir.

Algunos motores como Unity incluso se apuntan a las últimas tecnologías en *machine learning*, que permitirán que nuestros agentes pasen por fases de aprendizaje que les hagan más inteligentes o, al menos, más eficientes en sus tareas. Sus librerías son tan potentes que incluso empresas no relacionadas con los videojuegos las utilizan para desarrollar y probar nuevos algoritmos de IA. Veamos algunas características del módulo de IA de Unity que podemos aprovechar en nuestros proyectos:

- ***Finite state machines* y *Behavior trees*:** nos permitirán controlar el estado de nuestros agentes inteligentes y sus acciones. Por ejemplo, los enemigos podrían estar en uno de los siguientes estados: haciendo la ronda, persiguiendo al jugador, disparando al jugador, etc. De igual modo, podrían pasar de un estado a otro según diferentes mecanismos. Por ejemplo, al tener contacto con el jugador, un enemigo podría comenzar a perseguirlo, y si la distancia es suficientemente corta, podría disparar. En caso de perder de vista al jugador, podría volver a su ronda de vigilancia.
- ***Flocking*:** sistemas que nos permite imitar la inteligencia de colmena o de enjambre en nuestros agentes automáticos. La inteligencia de colmena es una rama de la IA que estudia el complejo movimiento conjunto pero descentralizado de agentes autoorganizados, el cual puede apreciarse en la naturaleza, por ejemplo, de abejas, hormigas, aves migratorias, rebaños, etc. Este podría servir también para describir el movimiento de ejércitos medievales en una batalla. Programar uno a uno el

movimiento de los componentes del grupo sería imposible, pero se han desarrollado algoritmos que se asemejan al comportamiento natural y son más eficientes.

- ***Steering*:** serían los algoritmos necesarios para que el agente se mueva dentro de una trayectoria preestablecida. Por ejemplo, podríamos utilizar *steering* para que los agentes conduzcan dentro de los límites del circuito de carreras, esquivándose mutuamente pero sin abandonar el circuito.
- ***Path finding* y *Navigation mesh*:** uno de los algoritmos más utilizados para calcular el movimiento de los *non-player characters* (NPC) dentro del juego. Los algoritmos de búsqueda de camino, como el A*, nos darán la ruta más eficiente desde un punto a otro de un plano preestablecido. Por ejemplo, para calcular el camino más rápido que debe tomar nuestro *alien* digital a través de los retorcidos pasillos de la nave Nostromo hasta alcanzar al desprevenido jugador, pasaríamos las posiciones de ambos, y junto con el plano, el algoritmo nos indicaría el camino que nuestro *alien* debe tomar. Lo malo de usar un algoritmo como el A* es que debemos crear manualmente nuestro plano virtual para que el algoritmo realice sus cálculos. Unity nos lo vuelve a poner fácil con el *Navigation mesh*. Con esta herramienta, una vez diseñada la escena con sus obstáculos, no necesitaremos crear ningún mapa en base a ellos, sino que Unity lo creará de forma automática y calculará las rutas necesarias de un punto a otro del plano.
- ***Machine learning*:** si creemos que necesitamos aún más inteligencia de nuestros NPC, podríamos usar técnicas de ML que nos ofrece Unity mediante el Unity Machine Learning Agents Toolkit. Esta tecnología, relativamente reciente, trata de usar las herramientas más potentes en IA (como las redes neuronales) para conseguir que los agentes aprendan de forma automática las tareas que necesitamos que realicen, en lugar de utilizar algoritmos específicos para cada problema. Debemos advertir que la complejidad de desarrollar estas funcionalidades en nuestro juego requerirán bastantes conocimientos sobre IA. No obstante, es interesante investigar cuanto se pueda sobre estas tecnologías, que no solo cambiarán la capacidad de nuestros juegos, sino las aplicaciones software del futuro.

4.6.6. Motor de sonidos

Cualquier videojuego estaría incompleto sin un tema musical de fondo y algunos efectos de sonido asociados a los principales eventos de la aventura. Los eventos de sonido conseguirán que el jugador se involucre aún más con el entorno del juego y que sea

consciente incluso cuando su vista se aparte de la pantalla. Por lo tanto, no debemos infravalorar el poder del sonido.

Unity pone a nuestra disposición potentes mecanismos para el control de audio, como el sonido espacial 3D, mezcladores de sonido en tiempo real, efectos predefinidos, etc. El sistema de sonido de Unity soporta la mayoría de formatos, y dispone de mecanismos para que los efectos de sonido sean más realistas. Por ejemplo, para simular que el sonido proviene de cada elemento en la escena, asociaremos el sonido al elemento, y Unity calculará su distancia y su velocidad hacia el jugador, de modo que pueda modificar el volumen y la frecuencia relativa al efecto Doppler. También podemos añadir otros efectos, como eco o reverberación, de forma manual a lugares predefinidos como túneles o pozos.

En Unity, podemos importar archivos de audio con formatos AIFF, Wav, MP3 y Ogg. No tendremos más que arrastrar los archivos desde el explorador hacia el panel de proyecto, como haríamos con cualquier otro recurso. Unity convertirá el archivo en un *audio clip*, que podrá ser arrastrado a un objeto *audio source* o utilizado desde un *script*. Como música de ambiente, Unity permite también importar archivos de tipo *.xm, .mod, .it* y *.s3m* como *tracker modules*, que darán lugar a objetos *audio clip* que usaremos de la misma forma que los efectos de sonido.

Veamos un resumen de las clases más útiles relativas al sonido:

- ***Audio clip*:** contiene los datos de sonido, ya sean mono, estéreo o multicanal, usado directamente por *audio source*.
- ***Audio source**:* reproduce un *audio clip* en la escena, ya sea hacia un *audio listener* o a través de un *audio mixer*.
- ***Audio listener*:** actúa como un micrófono, obteniendo el sonido de cualquier *audio source* de la escena y emitiéndolo hacia los altavoces.
- ***Audio mixer*:** puede llamarse desde un *audio source* y proporciona un procesamiento más complejo del sonido generado desde un *audio source*.

4.6.7. Gestión de redes

No es ninguna novedad que los videojuegos pueden disfrutarse en grupo, estén los jugadores en la misma sala o en la punta opuesta del planeta. Gracias al avance de las redes de comunicaciones, el ancho de banda permite compartir en tiempo real todo tipo de datos. En el caso de los videojuegos, los jugadores pueden compartir el estado absoluto del juego entre todos ellos, de modo que navegan por un entorno virtual sincronizado.

Normalmente se utilizará un servidor central que controlará las sesiones de juego y dispondrá el mundo virtual que todos los jugadores podrán compartir. El estado de cada elemento del juego será sincronizado entre todos, y así veremos como el campo de batalla, o cualquier otro tipo de escena, cambia a la vez en las pantallas de los jugadores. Todo este proceso (las comunicaciones, la sincronización del estado del juego, el control de sesión, etc.) conforma un sistema bastante complejo. Por ello, tendremos suerte de utilizar un potente motor de juegos que nos facilite el desarrollo de un sistema como este.

Unity pone a nuestra disposición dos tipos de API para el desarrollo de videojuegos multijugador en red: una de alto nivel, con las funcionalidades más comunes cubiertas de modo sencillo, y otra de bajo nivel, que nos permitirá mayor control, pero cuya complejidad requerirá de mayores conocimientos. Veamos algunas de las funcionalidades de la API de alto nivel:

- Controlar el estado del juego utilizando un *network manager*.
- El desarrollo de juegos albergados en el propio cliente, en lugar de utilizar un servidor central en Internet.
- Serializadores de datos, para facilitar la comunicación de nuestros objetos de juego.
- Envío y recepción de mensajes a través de la red entre clientes.
- Envío de comandos desde los clientes al servidor del juego.
- Llamadas a procedimientos remotos (RPC) de servidor a clientes.
- Envío de eventos desde el servidor a los clientes.

Unity tiene sus capacidades de red integradas en el entorno de desarrollo visual, de modo que es mucho más sencillo manejar los diferentes componentes de red y adaptarlos al juego. Además, la librería dispone de mecanismos de tiempo real, para realizar optimizaciones en las comunicaciones y las arquitecturas de nuestra red de jugadores. Por otro lado, Unity nos ayudará con la autenticación, mediante algunos mecanismos básicos. En el caso de que necesitemos aumentar la seguridad, nos permitirá añadir librerías de terceros más sofisticadas que las incluidas en el paquete.

4.7. Librerías que proporcionan las funciones básicas de un motor 2D / 3D

Como se ha comentado ya en algunos de los apartados anteriores, las **librerías** son un apartado clave en el proceso de desarrollo de un juego. Para dotar a un objeto o elemento de un aspecto más realista, es necesario que los motores gráficos procesen una serie de funciones que dibujan en 2D o 3D dichos objetos. Ya que el diseño de un objeto requiere de una gran labor de programación y este será representado en muchas ocasiones en un juego, la creación de estos objetos queda recogida en una serie de funciones que son proporcionadas por las librerías.

Estas librerías permiten abstraer al programador de los aspectos más complejos de representación de elementos visuales. Tan solo es necesario llamar a la función de la librería encargada de ello y recoger el objeto devuelto para su representación en la escena.

Las **funciones básicas** utilizadas por los motores gráficos son aquellas que permiten trabajar con elementos visuales, como son puntos, rectas, planos o polígonos. Proveen de los recursos fundamentales en un juego como sonidos y música. El apartado de modelado de personajes deberá recoger el uso de *sprites* para 2D y el uso de modelos (*assets*) para el desarrollo de plataformas de juego 3D.

4.7.1. API gráficos 3D

OpenGL es una API de dibujo 3D que permite realizar aplicaciones que producen gráficos. Esta API se compone de un gran número de funciones para la creación de elementos y objetos tridimensionales.

El **objetivo** de esta API es proveer al desarrollador de un documento donde poder encontrar todos los recursos y, de esta forma, disminuir la complejidad en la comunicación con las tarjetas gráficas.

El funcionamiento de este tipo de librerías consiste en tratar de aceptar como entrada una serie de primitivas, las cuales son: líneas, puntos y polígonos, y convertirlas en píxeles.

OpenGL actualmente tiene la versión 4. Cada versión ha ido desarrollando una evolución en cuanto a texturas, formas y transformaciones de los objetos. Es posible

hacer uso de cualquiera de ellas, empleando la documentación proporcionada en su página oficial. Esta documentación ofrece numerosos códigos de ejemplo, libros o videotutoriales para hacer uso de la librería deseada.

A partir de la versión 3, OpenGL desarrolló su **propio lenguaje** de renderizado, llamado **GLSL.** Este permite llevar a cabo, mediante programación, el desarrollo de una escena.

Otra API que ofrece este mismo tipo de gráficos 3D es **Direct 3D**. Ofrece una API 3D de bajo nivel, en la que se pueden encontrar elementos básicos como: sistemas de coordenadas, transformaciones, polígonos, puntos y rectas.

Es una librería con recursos gráficos que exigen de un nivel de programación experimentado en este tipo de recursos. Uno de los puntos fuertes de este API es que es independiente del tipo de dispositivo que se utiliza, lo que permite un desarrollo más versátil.

4.8. Estudio de juegos existentes

En la actualidad, el **mercado de juegos** para dispositivos es muy grande. Ya existen una gran cantidad de juegos para todos los tipos de géneros posibles. Esto, en muchas ocasiones, dificulta el éxito de algunos de ellos. Por esto, es recomendable hacer un **estudio de mercado** antes de su desarrollo, centrando la atención en aquellos juegos de carácter similar al juego que se va a crear.

Si el juego va a ser publicado en Internet, es importante **conocer a qué tipo de público** está destinado. De igual forma, es bueno conocer **cuáles son las limitaciones** de desarrollo, así como medir la **cantidad de recursos necesarios** para su creación, desarrollo y publicación.

En muchas ocasiones el desarrollo de un tipo de género de videojuego está relacionado con el éxito de alguno de ellos. Cuando las descargas de un juego determinado aumentan considerablemente, ese tipo de juego es un buen reclamo para los usuarios. Este hecho puede ser aprovechado por los desarrolladores de juegos con menos recursos con el objetivo de introducirse dentro del mercado.

En la industria de los juegos para dispositivos móviles tienen aceptación tanto los juegos 2D como los 3D, por lo que el abanico de posibilidades es ilimitado.

4.9. Aplicación de modificaciones sobre juegos existentes

Los **juegos Android** ocupan prácticamente el mayor porcentaje de descarga de aplicaciones relacionadas con el ocio, por lo que es posible encontrar ejemplos de juegos ya probados y cuyo éxito ha sido medido. El proceso de creación de un nuevo juego no es sencillo, y muchas veces ello obliga a que este desarrollo sea llevado a cabo por un gran número de personas de especialidades diferentes, las cuales realizan su trabajo dentro del proyecto de forma conjunta. Sin embargo, el ritmo de vida en la sociedad **obliga a estos equipos a renovarse continuamente**, por lo que es muy necesario conseguir que ese juego se adapte a los nuevos tiempos, añadiendo nuevas funcionalidades y optimizando su rendimiento en los dispositivos.

A diferencia de las aplicaciones, los juegos, por lo general, no suelen ser de código abierto, por lo que no es posible añadir de forma legal nuevas modificaciones al juego si no se forma parte del equipo de desarrollo o se es el autor de uno de ellos.

Cuando se publica un juego en Internet en plataformas como Google Play, se adquiere un compromiso de mantenimiento con esa aplicación por el que los desarrolladores deben corregir en la medida de lo posible todos aquellos errores que se detecten, tanto por parte de los usuarios como por parte de los propios desarrolladores. Este control de modificaciones es muy importante para el posible éxito de un juego.

5. Desarrollo de juegos 2D y 3D

En este tema se estudiará el desarrollo de los juegos 2D y 3D, sus principales entornos de desarrollo, la integración del motor de juegos en dicho entorno, conceptos de programación 3D, sus fases de desarrollo y las propiedades de los objetos. Asimismo, se mostrarán las diferentes aplicaciones, tanto de las funciones del motor gráfico como del grafo de escena, así como el análisis de ejecución y optimización del código.

5.1. Entornos de desarrollo para juegos

Diseñar un juego no es una tarea sencilla. Requiere conocimientos de diferentes especialidades como **programación, diseño y animación.** Para hacer un poco más fácil esta tarea están los **entornos de desarrollo**. Se trata de unas plataformas software que ofrecen una interfaz gráfica para la creación de juegos mediante el uso de una serie de herramientas.

Existen diferentes **tipos de entornos** que están orientados a un tipo de juegos, ya sea para 2D o para 3D. También es posible encontrar diferentes entornos dependiendo de la complejidad del juego a realizar.

Si el objetivo es realizar **juegos sencillos** cuya interfaz no sea muy exigente para plataformas en 2D, es posible hacer uso de **entornos** como:

- **Stencyl**: es una plataforma que permite la creación de juegos en 2D mediante el uso de bloques de código, los cuales ayudan a comprender las estructuras básicas de programación, por lo que no es necesario desarrollar líneas de código. Permite añadir imágenes para los personajes, que se añaden a una escena simplemente arrastrándolos. Se trata de una plataforma sencilla y fácil de utilizar.
- **Pygame**: entorno de desarrollo de juegos que utilizan el lenguaje *Python*. Permite la creación de juegos en 2D. Se basa en el uso de *sprites* para los personajes y bibliotecas de recursos de sonido y multimedia. La programación es algo más compleja, ya que es necesario realizar las estructuras de control y las variables a través de código.

Cuando el juego a desarrollar requiere de una **potencia gráfica mayor**, como es el caso del 3D, es necesario que los entornos de desarrollo sean, a su vez, más completos. Algunos de los más importantes son:

- **Unity 3D**: hoy en día, Unity es una de las herramientas más utilizadas en el mundo de los juegos, así como una de las mejor valoradas. Unity permite exportar un juego creado en cualquiera de los distintos dispositivos. Unity está basado en el lenguaje **C#**. Tiene un motor propio para el desarrollo de la parte gráfica, lo que permite llevar a cabo un desarrollo muy completo de todas las escenas de un juego. Es posible configurar todos los elementos necesarios, como pueden ser: la iluminación, las texturas, los materiales, los personajes, los terrenos, los sonidos, las físicas, etc.
- **Unreal Engine**: junto con Unity 3D, es uno de los entornos más conocidos y valorados dentro del mundo del desarrollo de juegos. Permite la configuración y diseño de recursos gráficos avanzados de la misma forma que Unity.

Ambos entornos requieren de un **nivel significativo de programación**.

5.2. Integración del motor de juegos en entornos de desarrollo

Una vez se ha configurado Android en el equipo, es necesario configurar la integración de Android dentro del entorno de desarrollo. En este caso el entorno escogido es **Unity**. Este debe conocer dónde está el SDK para poder compilar y, posteriormente, enviar la aplicación al dispositivo.

Para ello, el primer paso es, una vez seleccionado el proyecto, ir al menú de edición y seleccionar *Preferencias*. Esto mostrará la ventana de preferencias de Unity y, dentro de ella, en el apartado de *Herramientas externas*, se podrán visualizar los distintos parámetros de configuración del compilador. Es posible elegir el editor para la programación de las líneas de código, así como la ruta en la que se encuentra en el equipo el SDK de Android instalado.

Una vez que se ha seleccionado esta ruta, Unity será capaz de compilar un proyecto para Android. Para realizar la compilación, debemos seleccionar el *Menú configuración de la compilación*. En esta ventana se podrá elegir cuál será la plataforma de compilación, en este caso Android. Ello realizará todo el proceso de renderizado de gráficos, así como de programación para dicha plataforma. En este punto se añadirán las escenas deseadas para compilar. Una vez escogida la plataforma, se tiene que seleccionar la opción de *Compilar*.

Android no permite compilar sin un **identificador de paquete**, por lo que será necesario definir dicho identificador que, posteriormente, será usado por Google Play para su

publicación. Dentro del apartado de *Ajustes del proyecto* se especificarán todos los apartados del paquete.

Estos **apartados** son los siguientes:

- **Resolución** y presentación de la aplicación.
- **Icono** de la aplicación.
- ***Splash image***: será la imagen previa al comienzo del juego una vez se inicia la aplicación.
- **Renderizado**: ajuste de parámetros de renderizado para Android.
- **Identificación**: en este apartado se especificará el identificador del paquete, que, por lo general, suele ser el nombre de la estructura del proyecto.
- **Versión** del código.
- **Nivel mínimo** del API de Android.
- **Versión de gráficos utilizados**: lugar de instalación de la aplicación por defecto en el dispositivo.

Por último, una vez completados todos estos apartados, se generará en el equipo un ***.apk*** **(extensión de las aplicaciones en Android),** que será el archivo ejecutable que se instalará en el dispositivo deseado.

5.3. Conceptos avanzados de programación 3D

El desarrollo y programación de un juego tridimensional conlleva aplicar algunos **conceptos** que son de carácter avanzado, como **los movimientos, las físicas y las colisiones.** Estos permiten que el juego sea lo más realista posible. **Unity** ofrece una serie de clases que permiten definir y configurar estas propiedades sobre modelos de personajes y objetos.

Esta clase se denomina **controlador de personajes** (*character controller*), y permite aplicar físicas y colisiones en forma de cápsula a los personajes. Para ello proporciona un simple colisionador (*simple collider*). Esto hace que el personaje camine por el suelo y no suba por las paredes.

Los **tipos de colisionador** que existen son:

- ***Box collider***: se trata de una colisión en forma de cubo. Estos generalmente son empleados en objetos de forma cúbica, como, por ejemplo, una caja o un cofre.
- ***Capsule collider***: se trata de una cápsula de forma ovalada formada por dos semiesferas.
- ***Mesh collider***: son colisionadores más precisos que van asociados a objetos 3D ya diseñados. Ello permite crear un colisionador ajustado completamente a la forma del objeto.
- ***Sphere collider:*** es un colisionador básico de forma esférica. Suele ser aplicado en objetos esféricos, como pelotas, piedras, etc. Este efecto tiene un gran impacto en objetos, los cuales aparecen rodando en la escena o se están cayendo, por ejemplo.

Otro concepto frecuentemente usado en el desarrollo de juegos es el **sistema de partículas de Unity**. No siempre los objetos que se van a representar en una escena son sólidos o son elementos con formas bien definidas. Por ello, cuando se quiere realizar la representación de fluidos o líquidos que están en movimiento (humo, nubes, llamas, etc.), es necesario hacer uso de los efectos que proporciona el sistema de partículas. Este sistema de partículas está formado por imágenes simples y generalmente pequeñas, las cuales aparecen en la escena repitiéndose continuamente en una dirección. Esto hace que todas ellas representen, en conjunto, un elemento único para el usuario. Es necesario definir cuál será la forma de estas pequeñas imágenes, durante cuánto tiempo se mostrarán estas imágenes y con qué frecuencia y cantidad aparecerán en la escena.

Por último, otro de los conceptos, probablemente el más complejo, es la **inteligencia artificial (IA).** Esta permite en Unity crear personajes que son capaces de interaccionar en la escena e incluso evitar colisiones entre los elementos de la misma. Esta herramienta en Unity recibe el nombre de ***NavMesh***.

A través del inspector de creación de un agente se definen las **propiedades** que van a caracterizar al mismo, como son:

- **Radio**: radio que el personaje tendrá a la hora de moverse para evitar colisiones.
- **Altura**: define la altura máxima de los obstáculos por los que el personaje podrá acceder pasando por debajo de ellos.
- **Velocidad**: velocidad máxima en unidades por segundo que tendrá el personaje.
- **Aceleración**: aceleración del movimiento y acciones del personaje.
- **Área**: definirá el camino que tomará el personaje y cuáles no podrá escoger.

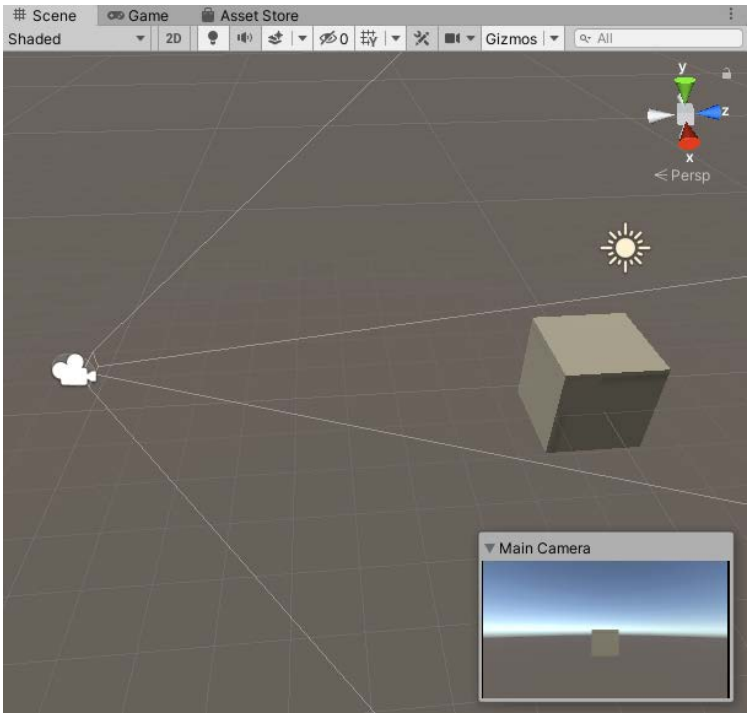
5.3.1. Sistemas de coordenadas

A la hora de diseñar una escena tridimensional, es importante conocer el sistema de coordenadas del motor de renderizado que estamos utilizando. El sistema de coordenadas nos ayudará a comprender la dirección y la orientación de los objetos, a efectuar transformaciones en los modelos o a diseñar con exactitud nuestra escena. Además, debemos tener en cuenta que, según la necesidad, puede tener sentido utilizar un sistema diferente de coordenadas, dependiendo de qué parte de la escena vayamos a estudiar. Por ejemplo, podemos tener las coordenadas de pantalla, que son bidimensionales y medidas en píxeles, comenzando en la esquina izquierda inferior en (0,0).

Sin embargo, las coordenadas de pantalla pueden cambiar con la resolución del dispositivo y con la orientación de la pantalla. Por ejemplo, las coordenadas de la interfaz de usuario son iguales a las de pantalla, con la peculiaridad de que el origen se encuentra en la esquina superior izquierda. Además, incluso en un videojuego 3D podemos necesitar textos, etiquetas o botones bidimensionales sobre la escena, los cuales nos indican la vida que nos queda, los puntos acumulados, etc.

Las coordenadas del *viewport*, que es el plano de la cámara, no cambian con la resolución de pantalla ni con la orientación, y comienzan en la esquina inferior izquierda (0,0) hasta la esquina superior derecha (1,1). Por ejemplo, si quisiéramos dibujar una mirilla de rifle en mitad de la pantalla en todo momento, lo haríamos en las coordenadas (0.5, 0.5) de las coordenadas del *viewport*.

Por último, las coordenadas del mundo, o de escena, son coordenadas tridimensionales de la escena completa. Estas coordenadas nos ayudarán a ordenar los modelos tridimensionales en la escena, sin importar dónde esté la cámara.



5.3.2. Modelos 3D

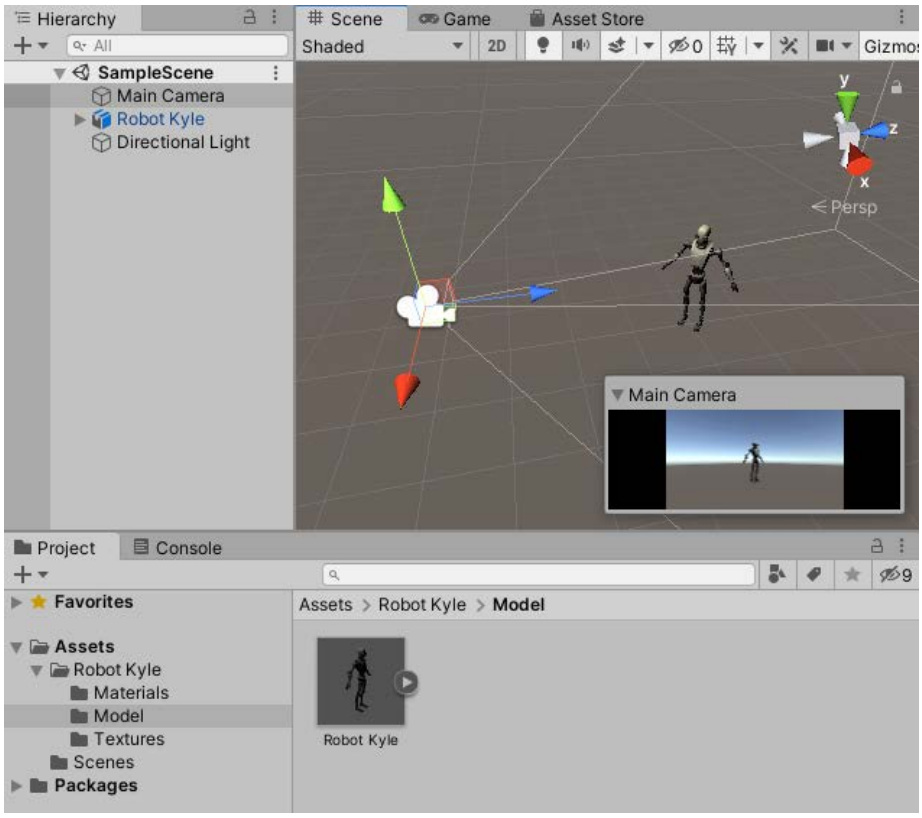
Para mejorar la calidad de nuestro videojuego, podemos utilizar objetos tridimensionales previamente modelados con alguna aplicación de diseño 3D. Por ejemplo, si nuestro videojuego es un simulador de vuelo, podríamos utilizar como enemigos simples esferas asesinas. Sin embargo, obtendríamos mejor resultado si pudiésemos utilizar modelos 3D profesionales de aviones, buques y camiones, los cuales enriquecerían la sensación de realidad de la escena.

Debemos tener en cuenta, además, la posibilidad de importar en nuestro proyecto modelos 3D de otros diseñadores. Mediante el Asset Store de Unity, podemos acceder a recursos para juegos de todo tipo, como por ejemplo, modelos 3D. Existen recursos gratuitos y de pago. No tenemos más que acceder al Store y utilizar el mecanismo de búsqueda para encontrar el recurso que estamos buscando.

Veamos cómo podemos utilizar el Asset Store para importar un nuevo personaje a nuestro juego. Junto a la pestaña de *Scene*, normalmente encontraremos la de *Game* y luego el *Asset Store*. Al pulsar la pestaña, podremos navegar por la web y utilizar su buscador. En el campo de búsqueda escribimos *Robot*; en *All categories*, seleccionamos *3D* y en *Pricing* hacemos clic en *Free assets*. Podemos elegir ese robot tan simpático llamado *Space Robot Kyle*. Dentro de la ficha del elemento, podremos ver todas sus

características, fotos y quizá algún video de animación. Al pulsar el botón *Add to my assets*, el modelo se añadirá a nuestra lista de recursos. Podremos descargarlo mediante el botón *Open* in Unity y mediante el *Package manager*. Una vez descargado, podremos importarlo en cualquiera de nuestros proyectos. Al importarlo veremos una carpeta con su nombre en el árbol de la pestaña *Project*, debajo de la carpeta *Assets*. Para introducir el personaje en escena, arrastraremos el modelo desde su carpeta hacia la ventana de escena. Una vez en escena, podremos modificar sus propiedades, su posición, etc.

Estos diseños podrán incluir texturas, materiales, animaciones, *scripts*… No obstante, también pueden venir sin ellos. Tendremos que adaptarlos a nuestras necesidades, por ejemplo cambiando su tamaño o añadiendo animaciones. Así, gracias a los múltiples recursos del Asset Store, podremos ahorrarnos mucho tiempo en diseño de modelos 3D que, por otra parte, enriquecerán enormemente nuestro juego.

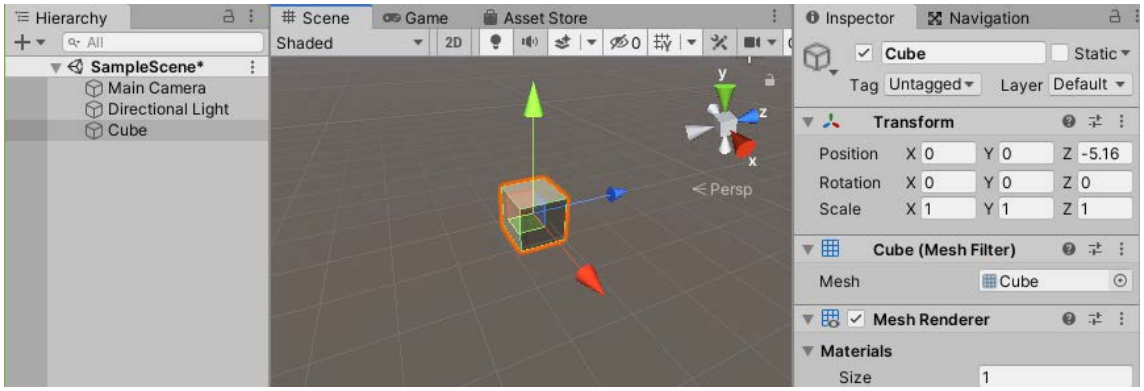


5.3.3. Formas 3D

Los modelos 3D creados con complejas aplicaciones de diseño aportan gran valor a nuestro juego. A pesar de ello, su uso puede ser ineficiente en algunos casos.

Dependiendo de lo complejo que sea el modelo, de la cantidad de vértices y mapeo de texturas o de la cantidad de animaciones, entre otros, un modelo 3D puede ser muy pesado y requerir demasiados recursos del motor gráfico. Ello podría traducirse en que nuestra *app* final se cuelgue por falta de memoria o por no conseguir los *frames* por segundo suficientes (es decir, que las animaciones funcionen a tirones). Si, además, dicho objeto no aporta gran importancia dentro del juego, muy probablemente deberíamos pensar en sustituirlo por una forma tridimensional básica, como un cubo, una espera, una pirámide o un cilindro. Por ejemplo, en el caso un simulador de vuelo, imaginemos que deseamos dibujar una ciudad a nuestros pies mientras la sobrevolamos con nuestro caza. Si modelamos los edificios como prismas cuadrados, en lugar de utilizar modelos perfectos de la torre Chrysler para componer nuestra ciudad, ahorraremos vértices y datos, los cuales colapsarían el motor de renderizado. Con unas buenas texturas, un cubo alargado pasará perfectamente por un rascacielos o un edificio cualquiera. Además, los motores gráficos nos permiten el uso de estas formas 3D primitivas con gran facilidad. De este modo aliviaremos la carga del procesador, a la vez que reduciremos el consumo de memoria.

Veamos un ejemplo. En Unity, selecciona el menú *GameObject* > *3D Object* > *Cube.* En la escena aparecerá un nuevo objeto 3D en forma de cubo. Más tarde podremos añadir texturas, materiales, propiedades físicas, *colliders* o cambiar sus dimensiones desde el *Inspector* y sus atributos.



5.3.4. Transformaciones. Renderización

Ya sea en tiempo de diseño o en tiempo de ejecución, podemos realizar transformaciones en los modelos y objetos que hemos añadido a la escena. Por ejemplo, podemos cambiar el tamaño de las caras de un cubo de manera que adquiera el aspecto

de un edificio de oficinas. Para ello, no tendríamos más que añadir las texturas y materiales necesarios para que su aspecto fuese el requerido. De igual modo, podríamos rotarlo y colocarlo en el punto de la escena donde queremos ese edificio.

Todas esas transformaciones de cada objeto serán procesadas por el motor de render, que hará todo el trabajo duro por nosotros y devolverá una representación bidimensional relativa a la cámara de la escena tridimensional diseñada. En tiempo de ejecución, podremos controlar con nuestros *scripts* las transformaciones que queremos para cada objeto. De este modo, podremos mover los enemigos por la escena, controlar los movimientos mecánicos de otros objetos, como plataformas y ascensores, y responder a la entrada del usuario para mover al jugador y a la cámara, que deberá seguirlo.

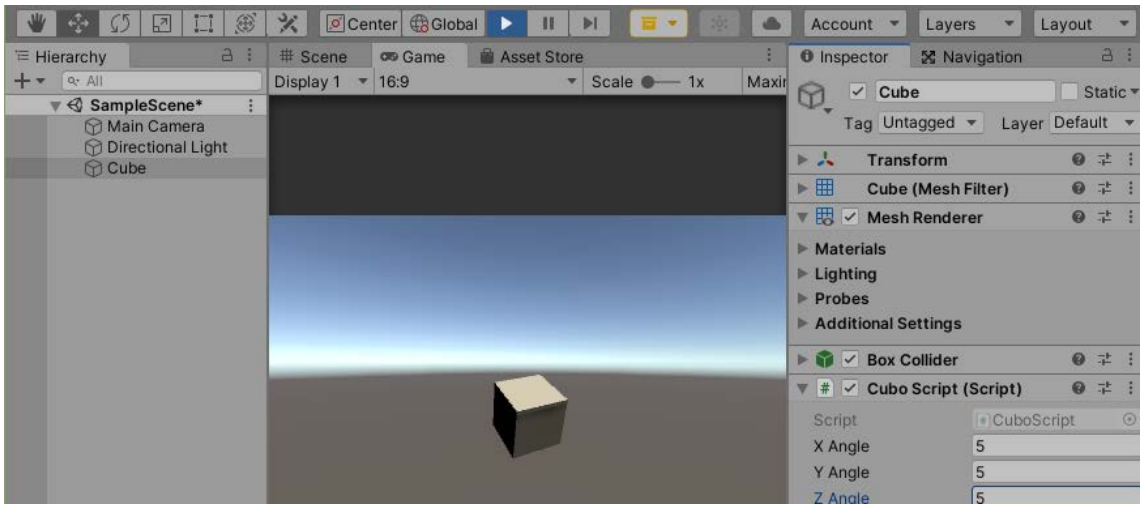
Ya tenemos una idea de cómo añadir elementos a la escena y transformar su posición, rotación y tamaño. Veamos ahora cómo podemos realizar transformaciones en el cubo que añadimos a la escena en el punto anterior, mediante un simple *script*. Para ello, con el cubo seleccionado en la escena, navegaremos por la pestaña del *Inspector*. Bajo todos los elementos del *GameObject*, encontraremos un botón *Add component*. Al pulsarlo, podremos elegir muchos elementos diferentes que podríamos añadir a nuestro objeto. Al final de esta lista, encontraremos *New script*. Escogeremos esta opción, y escribiremos el nombre que nos convenga para el archivo.

Veremos que en la lista aparece un nuevo elemento de tipo *script* con el nombre que le hayamos dado. Si pulsamos sobre el nombre del *script* con el botón derecho, podemos escoger la opción *Edit script* del menú contextual que aparece. Rellenemos el archivo de forma que quede así:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class CuboScript : MonoBehaviour {
   public float xAngle, yAngle, zAngle;

   // Start is called before the first frame update
```

```
    void Start()
    {
        yAngle = 10;
    }

    // Update is called once per frame
    void Update()
    {
        transform.Rotate(xAngle, yAngle, zAngle, Space.Self);
    }
}
```

Hemos añadido las variables *float xAngle, yAngle y zAngle,* que al ser públicas podrán ser accedidas y modificadas desde el *Inspector*. El método Start se ejecuta cuando la escena carga el objeto que lleva el *script*. En este caso, hemos iniciado la variable con un valor de 10, que utilizaremos más tarde en la función *Update*. La función *Update* se llama una vez por cada *frame*, es decir, una vez por cada ciclo de renderizado. En este método, obtenemos el parámetro *transform* del objeto en el que estamos incrustados, y llamamos a la función *Rotate*, de modo que el cubo rotará. Como parámetros, la función *Rotate* admite los ángulos de los tres ejes de coordenadas sobre los que debe rotar, y el espacio sobre el que debe hacerlo: en este caso *Space*.*Self* hará que rote sobre sí mismo, y no sobre las coordenadas del mundo. Guardemos el *script* y ejecutemos la escena pulsando al icono de *Play*; veremos el cubo rotar sobre su eje Y. Ahora podemos cambiar los valores de todos los ángulos y ver cómo reacciona el cubo en tiempo real. El motor de Unity se encargará de la renderización:



5.4. Fases de desarrollo

La **creación de un nuevo juego** exige de una gran cantidad de tareas que se pueden agrupar en distintas **fases**:

1. **Fase de diseño:** es el **paso previo a la programación**. En esta fase es necesario determinar cuáles serán los aspectos relevantes del juego, escoger la temática y el desarrollo de la historia. También es importante establecer cuáles serán las reglas del juego.

   Una vez documentada la historia, es necesario separar el juego en partes. Cada una de estas partes conformará las pantallas del juego. Se debe definir asimismo cuál será el aspecto del menú dentro de las pantallas, así como la colocación de los objetos dentro de la misma.

2. **Diseño del código:** en esta fase se especifican todas las capas de las que se compone el juego. Se trata de separar todos los aspectos básicos del juego de la funcionalidad del mismo. Esto es lo que se conoce como ***framework***.

   El ***framework*** definirá cómo será el **manejo de las ventanas del juego**. Permite asegurar que los objetos ocupan el espacio correcto dentro de la ventana que corresponde. También se encargará del manejo de eventos de entrada de usuario. Estos, en la mayoría de los casos, serán recogidos del teclado o ratón del equipo.

Otra de las tareas del *framework* es el **manejo de ficheros**, en los que se llevarán a cabo las tareas de lectura y escritura como, por ejemplo, guardar las preferencias y puntuaciones del juego.

También determinará el **manejo de gráficos**, donde se establecen los píxeles mapeados en las diferentes pantallas. Es necesario determinar la posición a través de coordenadas de cada uno de los píxeles, así como del color. El **manejo de audio** para poder reproducir, por ejemplo, música de fondo en el juego, también será determinado por el *framework*.

3. **Diseño de los *assets*:** es una de las fases más complicadas y que mayor repercusión tiene en un juego. Hace referencia a la creación de los diferentes elementos o modelos que se pueden utilizar dentro del juego: los personajes, los logos, los sonidos, los botones, las fuentes, etc.

4. **Diseño de la lógica del juego:** en esta fase es donde se define cómo se comportará el juego. Se aplicarán las reglas ya diseñadas, así como la programación del comportamiento de cada uno de los eventos del juego.

5. **Pruebas:** una de las fases más importantes. Es en este momento cuando se realiza una comprobación de toda la aplicación con el fin de valorar el comportamiento del juego y la aplicación correcta del resto de fases.

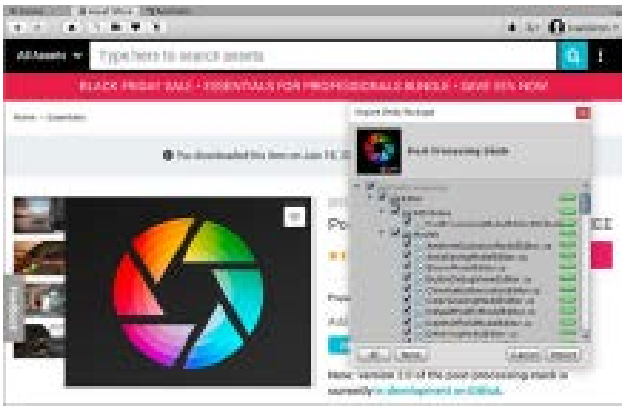
6. **Distribución del juego:** una vez finalizadas las fases de desarrollo, el objetivo es que el producto sea distribuido. Para ello, es necesario exportar este juego de la misma forma que una aplicación.

Efectos de posprocesamiento

El proceso de posprocesamiento puede **mejorar significativamente las imágenes** del producto final, ya que aplicaremos filtros y efectos que dotarán a nuestro diseño de una mayor profesionalidad.

Deberemos añadir los archivos de posprocesamiento a nuestro proyecto de Unity siguiendo los siguientes pasos:

1. Primero tendremos que entrar en la **Asset Store** de **Unity** y bajarnos el *Asset post-processing stack*. Para entrar en la Asset Store, tenemos que ir a *Window* > *Asset Store* o pulsar *Ctrl + 9*.
2. Cuando esté listo, nos aparecerá una pestaña de *Importar*, la cual seleccionaremos. Ya estaremos listos para crear un posprocesado.



3. Tendremos que hacer dos cosas: primero crearemos un perfil de posprocesado (en la pestaña *Project* hacemos clic derecho y después *Create* > *Postprocesing profile*). Veremos que al pulsar este nuevo archivo tendremos varias opciones de posprocesados, como el *fog* o el *motion blur*. Sin embargo, hagamos lo que hagamos no sucederá nada, ya que tendremos que asignarlo. Seguidamente, seleccionamos la cámara y añadimos en componentes *Post-processing behaviour*. Comprobaremos entonces que este dejará un espacio. Ahí añadiremos el perfil de posproceso que hemos creado anteriormente. Eso sería todo.

Motion blur y otros efectos posprocesado

En el *Inspector* del *Post processing profile* encontraremos el efecto ***motion blur***. Este efecto simula el desenfoque de una imagen cuando los objetos principales se mueven más rápido que el tiempo de exposición de la cámara.

El *motion blur* utiliza dos técnicas principales, pero nos centraremos en la **simulación de velocidad de obturación**, la cual imita el desenfoque de una cámara. Esta es costosa y no es compatible con algunas plataformas (como en realidad virtual), pero proporciona un efecto fuerte de desenfoque y tiene alta calidad.

La **profundidad de campo** es otro de los efectos que encontraremos. Este simula las propiedades de enfoque de la lente de una cámara, lo que le aporta un efecto de realidad, pues las cámaras solo pueden enfocarse plenamente en un objeto y aquellos más lejanos o cercanos siempre están algo desenfocados. A este respecto, es un efecto que aporta **altas dosis de realidad** y una sensación de profesionalidad.

Encontramos también el efecto ***anti-aliasing***, que aporta suavidad a los gráficos porque huye del *aliasing* o dientes de sierra. Su finalidad es alisar y redondear los acabados de los polígonos mediante algoritmos diseñados exclusivamente para ello.

En nuestro caso, el algoritmo que veremos será el del *anti-aliasing* rápido y aproximado (FXAA). Es la técnica recomendada para dispositivos móviles y plataformas que no admitan vectores de movimiento. Además, proporciona una compensación entre el rendimiento y la calidad del borde sin difuminar demasiado el acabado de las líneas.

Otro de los efectos que queremos mencionar es la **oclusión ambiental**. Esta busca una similitud con la realidad oscureciendo zonas como pliegues o agujeros. Sin embargo, es recomendable usarlo en equipos de escritorio o en consolas, ya que consume bastantes recursos.

También cabe destacar la **intensidad** o grado de oscuridad producido por defecto; la alta precisión, que alterna el uso de una textura de profundidad de mayor precisión con la ruta de reproducción hacia adelante; y la opción solo ambiente.

Por último, cabe mencionar el efecto **niebla** (*fog*), el cual se utiliza para simular niebla o neblina en imágenes al aire libre, ya que crea un espacio de pantalla que se basa en la textura de profundidad de la cámara.

Valoración y análisis de resultados finales

Para realizar el análisis de los resultados finales, debemos centrarnos en si el producto satisface las necesidades o demandas para las que fue concebido. Además, tenemos que seguir unos pasos concretos para llegar a una conclusión basada en hechos.

- **Análisis objetivo**: se debe realizar un análisis del producto de manera objetiva. Deberemos fundamentar todas y cada una de las reflexiones o conclusiones intentando ser lo más objetivos posibles.
- **Listado de las cosas buenas y malas del producto**: es necesario diferenciar y hacer un listado de los aspectos positivos del producto (es decir, de aquellos que cumplen las expectativas esperadas en cuanto a funcionalidad y calidad y que, por lo tanto, no se deben tocar) y una lista de aquellos aspectos negativos en los que deberíamos introducir mejoras.
- **Ser realistas**: no se trata de hacer un análisis positivista de la situación. Se deben analizar todos los aspectos posibles para evitar volver a cometer los mismos errores y pulir el producto lo máximo posible.
- **Ser constructivos**: aportaremos soluciones reales a los aspectos que no nos gusten o que sean negativos, incidiendo en la mejora de esos aspectos sin caer en la crítica. Si hay algo mejorable, se debería atajar a través de propuestas de mejora que sirvan para finalizar el producto de la mejor manera posible.

Si queremos o necesitamos plasmar todo esto en un texto, deberemos ser concisos y ordenados para que todas las ideas queden expuestas claramente.

Test del producto

El test del producto es una técnica que realizan las compañías a través de **especialistas del mercado**. Se presenta el producto (generalmente un prototipo) a un grupo de consumidores que lo probará y dará su opinión para crear conclusiones.

Los probadores beta o *beta tester* son usuarios con conocimientos avanzados en el campo de los videojuegos o que se dedican a hacer test de las versiones beta, es decir, de las versiones que aún no están finalizadas del todo o que son prototipos. El objetivo es que detecten errores para que los creadores los puedan subsanar.

Asimismo, la fase de testeo del producto puede ser **interna o externa**. Es interna cuando es la propia compañía la que cuenta con varios *beta tester* en plantilla, los cuales suelen encontrarse en el departamento de QA (calidad). Por el contrario, el proceso es externo cuando se buscan *game tester* fuera de la compañía.

Una vez pasada la fase del testeo del videojuego, este debe obtener la *certification testing*, la cual es necesaria para poder comercializarlo. Actualmente, ser *game tester* es un empleo que requiere experiencia y conocimientos avanzados en el sector.

5.5. Propiedades de los objetos: luz, texturas, reflejos, sombras

Adaptación de materiales y texturas motores

En Unity, el renderizado de elementos 3D se realiza con materiales, *shaders* y texturas.

Los **materiales** son definiciones de cómo se debe renderizar la superficie. Los ***shaders*** son *scripts* pequeños que contienen los cálculos matemáticos y los algoritmos para calcular el color de cada píxel renderizado, los cuales se basan en el *input* de iluminación y la configuración del material. Además, cada uno de ellos tiene unas propiedades que aparecen en el *Inspector* cuando se mira un material. Un *shader* define el método para renderizar un objeto y puede especificar diferentes métodos dependiendo del hardware de los gráficos del usuario final. Por otro lado, las **texturas** son imágenes *bitmap* que pueden representar aspectos de la superficie de un material como la rugosidad o el color.

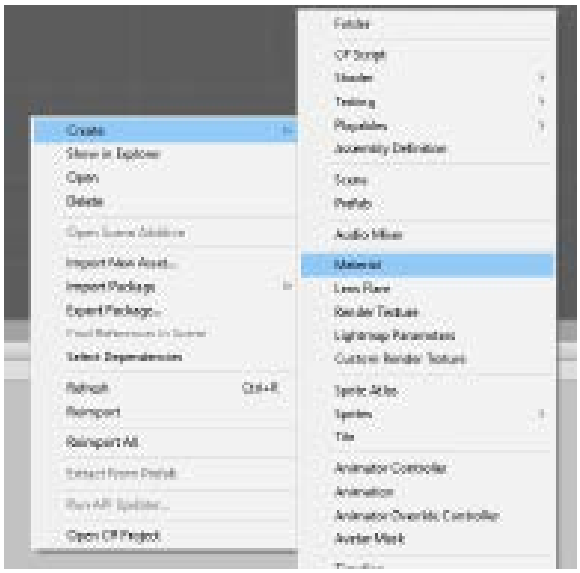
Normalmente, para renderizar la mejor opción es la *Standard shader*, ya que es capaz de renderizar muchos tipos de superficie de manera bastante aproximada a la realidad.

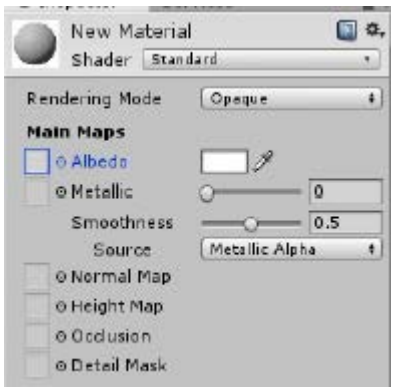
Asimismo, para crear un nuevo material deberemos ir a *Assets* > *Create* > *Material*. Los nuevos materiales se asignan al *Standard shader* y, una vez aplicados a un objeto (para aplicarlo podemos arrastrarlo desde el *Project view* hasta la *scene* o la jerarquía), podremos cambiar las propiedades en el *Inspector*.

Seleccionando el nuevo material. En el *Inspector* podremos escoger un nuevo *shader*, y este nos dictaminará las propiedades que podremos cambiar. Para aplicar una textura a una propiedad, deberemos arrastrarla desde el *Project view* hasta la sección *Main map* y soltarla en el mapa correspondiente.

Unity tiene categorías de *shader* integradas con propósitos concretos (por ejemplo, *Nature* para árboles y terren, o Toon para el renderizado tipo caricatura, entre otros).

Todos estos conceptos y más los veremos en el siguiente módulo.





Efectos de posprocesamiento

El proceso de posprocesamiento puede **mejorar significativamente las imágenes** del producto final, ya que aplicaremos filtros y efectos que dotarán a nuestro diseño de más profesionalidad.

Deberemos añadir los archivos de posprocesamiento a nuestro proyecto de Unity siguiendo los siguientes pasos:

1. Primero tendremos que entrar en la **Asset Store** de **Unity** y bajarnos el *Asset Post Processing Stack*. Para entrar en la Asset Store tenemos que ir a *Window* > *Asset store* o pulsar *Ctrl+9*.
2. Cuando esté listo, nos saldrá una pestaña de *Importar*, la cual seleccionaremos. Ya estaremos listos para crear un posprocesado.

3. Tendremos que hacer dos cosas: primero crearemos un perfil de posprocesado (en la pestaña *Project* hacemos clic derecho y después *Create* > *Postprocesing profile*). Veremos que al pulsar este nuevo archivo tendremos varias opciones de posprocesados, como el *fog* o el *motion blur*. Sin embargo, hagamos lo que hagamos, tendremos que asignarlo antes. Seguidamente, seleccionamos la cámara y añadimos en componentes *Post-processing behaviour*. Comprobaremos que este dejará un espacio. Ahí añadiremos el perfil de posproceso que hemos creado anteriormente. Eso será todo.

5.5.1. Utilización de shaders. Tipos y funciones

Propiedades de los objetos: luz, texturas, reflexión y sombras

Cada uno de los objetos representados dentro de una escena tiene unas determinadas **propiedades**. Estas son:

- **Luz**: es la que permite observar puntos de iluminación dentro de la escena. Esto dotará de vida al juego. La luz, o un punto de luz sobre una parte en particular de una escena, mueve a centrar su atención en ella. Indicará la proyección de la cámara dentro del juego. Es posible añadir diferentes puntos de luz en una escena, así como configurar el color de una luz.
- **Texturas**: reflejan la calidad con la que se pueden apreciar todos los objetos que aparecen dentro del entorno del juego. Forman parte de las texturas, por ejemplo, los **materiales**. Dentro de los materiales se pueden diferenciar algunos, como el agua, el metal, la madera, los tejidos, etc. Para conseguir un efecto real en una textura, a estos materiales se les aplica una serie de algoritmos matemáticos denominados ***shaders***, los cuales permiten definir cuál será el color de cada uno de los píxeles que componen un objeto.
- **Reflejos y sombras**: estos añaden a los objetos una representación más realista. Para conseguir este efecto, se añade una especie de **contorno** a los componentes gráficos. Se define el color de esta sombra y cuál será la distancia aplicada en cada objeto. Las sombras suelen ir acorde a la proyección de la luz, de tal manera que la sombra aparece como un efecto de dicha iluminación. Con los reflejos y sombras es posible establecer la posición que ocupa un objeto dentro de la escena.

Estas son algunas de las propiedades básicas de los objetos. Estas serán configuradas de forma particular para cada objeto en función de la posible interacción dentro de una escena.

5.6. Aplicación de las funciones del motor gráfico. Renderización

Una de las labores de mayor complejidad de un motor gráfico es el **renderizado de los objetos** que componen una determinada escena. El procesamiento de cada uno de ellos requiere de cierta cantidad de recursos gráficos que, en la mayor parte de los casos, son ofrecidos por las tarjetas gráficas.

Se puede definir la renderización como el **proceso de creación de una imagen 2D o 3D real dentro de una escena, aplicando una serie de filtros a partir de un modelo diseñado.**

Algunas de las **propiedades** que definen el proceso de renderizado son:

- **Tamaño**: define el tamaño de la renderización en píxeles. En la mayor parte de los casos, se llevará a cabo sobre texturas.
- ***Anti-aliasing****:* se utiliza para aplicar un filtro de suavizado sobre los objetos que, al ser renderizados, aparecen con formas escalonadas.
- ***Depth buffer***: se encarga de definir la profundidad de los objetos 3D en una escena. Tiene un gran impacto en la calidad de la escena producida.
- ***Wrap mode***: utilizado para definir el comportamiento de las texturas. Por ejemplo, en un terreno se define la repetición de una textura en concreto que será aplicada en toda la escena.

En entornos como Unity, es posible configurar estas propiedades de renderizado a través de la función ***Render texture*.**

5.7. Aplicación de las funciones del grafo de escena. Tipos de nodos y su utilización

Unity ofrece una herramienta sencilla para la organización y gestión de animaciones llamada ***Animator controller***. Esta permite crear dentro de Unity un grafo de acciones para controlar todas las animaciones de un personaje u objeto. Es posible establecer un orden de ejecución en función de algunas de las reglas o condiciones del juego.

De este modo es posible definir el comportamiento de un personaje que de forma normal camina en una dirección y que, al pulsar la barra espaciadora, realizará un salto.

Cada uno de estos nodos será la representación de una acción del personaje. Estos reflejarán las transiciones entre los estados más básicos.

Su utilización suele darse durante el empleo de **movimientos direccionales** del personaje, los cuales se repiten de forma periódica hasta el suceso de otro de los eventos. Estos movimientos suelen ser caminar hacia delante, hacia atrás o en diagonal. Otros nodos definirán estados como la muerte del personaje, caídas o colisiones con otros objetos de la escena, etc.

5.8. Análisis de ejecución. Optimización del código

Durante el desarrollo de un juego, será necesario compilar y depurar el código muchas veces. En las nuevas versiones de Unity, ya no tenemos el IDE integrado (*mono develop*). Como consecuencia, desde hace unos años se utiliza el IDE Visual Studio como el más recomendado para la programación de videojuegos. Este IDE contendrá un conjunto de herramientas que nos permitirá revisar un código lo más limpio posible.

Cuando se está editando un archivo dentro del proyecto, este aparecerá como una pestaña. El editor de texto permite añadir *breakpoints* en los márgenes, al lado de cada una de las líneas de código que se desea. Una vez seleccionados estos puntos de parada, comienza la depuración del código a través del botón ***Debug***. Esto ejecutará el código, quedando parado en el primer punto de parada encontrado en el código. Esto permite ver los valores que han tomado todas las variables hasta ese momento.

Además, es posible navegar entre los distintos puntos de parada para comprobar el correcto comportamiento de la aplicación.

En caso de producirse **errores** en la compilación, Unity contiene un archivo de *logs* denominado ***Debug.log***, donde se almacenarán todos los mensajes mostrados en la consola. Lo más común es que, en caso de existir errores en el código, el propio Unity, al compilar, no permita la ejecución del juego y muestre, en la parte inferior, un mensaje que referencie el error o los errores encontrados.

Otra herramienta que es útil dentro del IDE de Unity es el ***Unity test runner***. Esta herramienta comprueba el código de programación en busca de errores antes de realizar una compilación. Esto puede ser útil para corregir errores de sintaxis, por ejemplo.

A parte de tener todas estas herramientas de depuración, es conveniente que el desarrollador tenga adquiridas una serie de **buenas prácticas de programación y estructuración de código.**

El código tiene que estar lo más limpio posible, lo que ayudará posteriormente a la corrección y mejora de algunas funciones. En proyectos con un desarrollo de código muy extenso, esto puede suponer un problema de optimización muy grande.

Las funciones declaradas deben estar bien definidas y no deberán existir varias funciones cuyo comportamiento sea el mismo.