



Programación multimedia y dispositivos móviles

José Alfredo Múrcia Andrés



Programación multimedia y dispositivos móviles

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del *copyright*.

Diríjase a CEDRO (Centro Español de Derechos Reprográficos, www.cedro.org) si necesita fotocopiar o escanear algún fragmento de esta obra.

Nota: Este libro se atiene al artículo 32 del derecho de cita de la Ley de Propiedad Intelectual de 1996 (R. D. Leg. 1/1996 de 12 de abril).

Derechos reservados © 2022, respecto a la primera edición en español, por:

McGraw Hill/Interamericana de España, S. L.

Basauri, 17 Edificio A, 1^a planta

28023 Aravaca (Madrid)

ISBN digital: 9788448626754

© José A. Múrcia Andrés

Corrección técnica: Vicent Selfa García

Equipo editorial: Cristina García Sánchez y Silvia García Olaya

Corrección de estilo y ortotipográfica: Ester Jiménez Domínguez

Ilustración: Pablo Vázquez Rodríguez, Archivo McGraw-Hill, 123rf

Fotografías: Archivo McGraw-Hill, 123rf



Unidad 1.

Tecnologías para aplicaciones en dispositivos móviles

Propiedad de McGraw-Hill®

1. Dispositivos móviles. Hardware y tecnologías de desarrollo

1.1. Dispositivos móviles

Cuando pensamos en dispositivos móviles, generalmente pensamos en un smartphone. Sin embargo, hoy en día existe una gran diversidad de dispositivos móviles con características muy dispares entre ellos: netbooks, tabletas, relojes inteligentes, y evidentemente, smartphones.

En líneas generales, podemos definir un dispositivo móvil como aquel que posee las características siguientes:

- **Movilidad:** un dispositivo móvil puede ser transportado fácilmente y ser utilizado durante su transporte.
- **Tamaño reducido:** un dispositivo móvil debe ser lo suficientemente pequeño como para facilitar su movilidad y, además, posibilitar la interacción de forma simple, mediante una o las dos manos, sin necesidad de ningún dispositivo externo.
- **Conexión inalámbrica** a una red para poder enviar y recibir datos sin cables, ya sea a través de redes de telefonía móvil o wifi.
- Suelen ser **dispositivos unipersonales**, es decir, asociados a un único usuario.

Además, no debemos olvidar que se trata de dispositivos con **capacidad de procesamiento y memoria**. En los apartados siguientes vamos a analizar el hardware y el software propios de estos dispositivos.

1.2. Hardware de un dispositivo móvil

A. EL PROCESADOR Y EL SOC

Un dispositivo móvil contiene básicamente el mismo hardware que un ordenador, aunque tiene una arquitectura diferente, ya que están condicionados a un menor tamaño del dispositivo. Así pues, cuando hablamos del procesador en un dispositivo móvil, debemos hablar del SoC o *System on a Chip*, un circuito integrado con diferentes partes del sistema, como la propia CPU, la GPU (procesador gráfico), la RAM, la ROM, o los controladores USB o wifi.

Cabe destacar la importancia en el mundo de los procesadores móviles de la empresa británica ARM Holdings, la creadora de la arquitectura con su propio nombre, ARM (Advanced Risc Machines), una arquitectura RISC (Reduced Instruction Set Computer, o Conjunto Reducido de Instrucciones) de 32 y 64 bits que está presente en la mayoría de dispositivos móviles.

En la actualidad, cabe destacar la familia de SoC Snapdragon, de Qualcomm, basada en el conjunto de instrucciones ARM y presente en dispositivos de Motorola (Moto G100), Xiaomi (Redmi Note 10 Pro), Samsung (Galaxy S20 FE) o Vivo (V20).

B. LA MEMORIA RAM

El tipo de memoria RAM que se encuentra dentro del SoC es de tipo dinámica (DRAM), requiere refrescarse periódicamente y es muy rápida. Con la finalidad de reducir el consumo de energía y conseguir un menor calentamiento, esta se ubica cerca de la CPU y la GPU. Esta RAM es compartida por ambas unidades.

C. PANTALLAS EN DISPOSITIVOS MÓVILES

Una de las características determinantes de los smartphones es la pantalla, a través de la cual se realiza la mayor parte de la interacción con el dispositivo.

Existen principalmente dos tipos de tecnologías, con diferentes variaciones y generaciones:

- **LCD o inorgánicas.** De cristal líquido, dejan pasar luz en función de su polaridad, igual que los monitores. Para ello, cuentan con un panel de iluminación situado tras el panel, ya sea de lámparas fluorescentes o LED. Dentro de este tipo encontramos pantallas TFT, de tipo LED, IPS, PLS, o las Retina de Apple.
- **OLED u orgánicas.** Cuentan con materiales orgánicos que emiten luz por sí mismos cuando se les aplica electricidad, de modo que no necesitan un panel de iluminación y pueden apagar o encender los

píxeles de manera independiente. Se trata de pantallas más delgadas que las LCD, al tener menos capas, con un menor consumo, y que muestran mayor contraste y son más brillantes que estas. Además, esta tecnología de fabricación permite que los paneles sean flexibles, dando lugar a teléfonos móviles con pantallas plegables o enrollables. Dentro de esta familia encontramos las pantallas AMOLED, las Super AMOLED y las Dynamic AMOLED.

A parte de la tecnología de fabricación de pantallas, es importante conocer algunas características más de estas:

- **Resolución y densidad.** La resolución de la pantalla hace referencia al número de píxeles que se muestran en esta, y viene determinada por la cantidad de píxeles en horizontal y en vertical. A partir de la resolución podemos conocer la relación de aspecto y la densidad de píxeles por pulgada (ppi). La densidad es una medida que relaciona el tamaño de la pantalla con la resolución. Dos pantallas de distintos tamaños con la misma resolución tendrán densidades diferentes, y, por tanto, diferente calidad de imagen. Lo ideal es que esta densidad sea mayor de 300 o 400 ppi.
- **Relación de aspecto.** Se trata de la relación entre el ancho y el alto de la pantalla. Tradicionalmente, esta relación ha seguido el formato panorámico 16:9, o lo que es lo mismo, tamaños proporcionales a 16 píxeles de ancho por 9 de alto. Por ejemplo, el formato 1920x1080 cumple esta proporción.

Sin embargo, hace unos años empezó a popularizarse la relación 18:9 [e incluso la 19:9], orientada a paneles de mayor tamaño y apenas sin marcos. Esta relación permite que un dispositivo sea menos ancho que otro con la misma diagonal en formato 16:9, con lo que su ergonomía mejora.

D. CÁMARAS

Uno de los componentes a los que más se le suele exigir en un dispositivo móvil es la cámara o cámaras fotográficas. Estas se componen, principalmente, de:

- Un bloque óptico, relativamente limitado, aunque algunos dispositivos permiten la conexión de un objetivo.
- Un sensor, compuesto por una matriz de fotorreceptores que traducen las señales luminosas en eléctricas, obteniendo así el valor para cada punto de la imagen.

Con el tiempo, además, los dispositivos móviles han ido incorporando más cámaras, aparte de la frontal y la trasera, de modo que actualmente podemos encontrar tres, cuatro o hasta cinco cámaras [teleobjetivo, de gran angular, monocromáticas o incluso infrarrojas].

E. SENSORES

Los dispositivos móviles pueden incorporar una gran variedad de sensores con los que podemos obtener información muy variada de nuestro entorno. Entre los sensores que podemos encontrar en un dispositivo móvil destacan:

- GPS [sistema de posicionamiento global]. Proporciona servicios de ubicación, utilizando una señal continua, a los satélites GPS, con lo que permite mostrar en qué lugar del planeta nos encontramos. Este dispositivo da pie a todo un abanico de posibilidades como son los servicios LBS o servicios basados en la localización.
- Acelerómetro y giroscopio. El giroscopio mide los movimientos del dispositivo, gracias a la aceleración angular, e incluso permite detectar pequeños giros, lo que lo hace muy apropiado para aplicaciones de realidad aumentada. Por su parte, el acelerómetro es menos preciso y permite detectar la posición del dispositivo respecto a los tres ejes, X, Y y Z. Con ello, podemos conocer, por ejemplo, si el dispositivo se encuentra en horizontal o vertical, así como detectar giros, vibraciones, la inclinación o colisiones.
- Sensor de luz. Permite detectar la luz ambiental, de modo que el dispositivo es capaz de ajustar el brillo de la pantalla para adaptarse a las diferentes circunstancias lumínicas.
- Sensor de proximidad. Permite detectar la distancia del dispositivo respecto de otros objetos a través de un LED infrarrojo, y permite, por ejemplo, apagar la pantalla cuando nos lo acercamos al oído en una llamada.

Además de estos, podemos encontrar muchos otros, como sensores biométricos (lectores de huella o iris), magnetómetros, barómetros, sensores capacitivos, sensores de infrarrojos, sensores del ritmo cardíaco o de espectro de color, o podómetros.

1.3. Sistemas operativos para dispositivos móviles

El sistema operativo es una pieza fundamental en cualquier sistema informático. Desde la aparición de los primeros dispositivos móviles hasta la actualidad han existido una serie de sistemas operativos, con mayor o menor relevancia, dedicados a estos dispositivos: PalmOS, Symbian, Windows Mobile y Windows Phone, Maemo, Firefox OS, MeeGo, Bada OS, Ubuntu Touch, Tizen o Blackberry OS. Pero sobre todos ellos, han sido Android e iOS los sistemas que se han erigido predominantes en el panorama actual.

A. ANDROID

Android es un sistema operativo, desarrollado por Google y basado en el núcleo de Linux, fabricado específicamente para dispositivos móviles con pantalla táctil: smartphones, tabletas, relojes inteligentes, televisores o incluso algunos coches. La base del sistema se organiza en diversas librerías en C y el Android Runtime (ART), la máquina virtual de Java sobre la que se ejecutan las aplicaciones. Aunque en su origen el lenguaje de programación para el desarrollo en Android ha sido Java, Google adoptó Kotlin como lenguaje de programación oficial en Android, que es más potente, expresivo y capaz de generar bytecode ejecutable directamente en ART.

B. IOS

iOS es el sistema operativo más vendido por detrás de Android, y es propiedad de la multinacional Apple. Fue creado para el iPhone y, posteriormente, adoptado en el iPod touch y el iPad. La base de iOS es MacOS, derivado de Darwin BSD, un sistema operativo de tipo Unix. Aun así, se trata de un sistema cerrado que no permite su instalación en hardware de otros fabricantes. Los lenguajes con los que podremos desarrollar aplicaciones de forma nativa para iOS son Objective-C y Swift.

1.4. Tecnologías para el desarrollo de aplicaciones móviles

El desarrollo de una aplicación que funcione de forma **nativa** en un determinado sistema operativo pasa por el desarrollo mediante sus propias tecnologías. Sin embargo, existen varias tecnologías que con sus ventajas e inconvenientes pretenden abarcar el desarrollo multiplataforma en su sentido más amplio, desde aplicaciones web de tipo responsive hasta aplicaciones compiladas, pasando por las aplicaciones web híbridas o progresivas (PWA).

Las aplicaciones nativas son las que se desarrollan específicamente para el sistema en que se van a ejecutar y aprovechan mejor todos sus recursos. Además, permiten acceder a todas las funcionalidades de las plataformas y disponen de cualquier novedad en el sistema de forma inmediata.

Para el desarrollo nativo en Android utilizaremos Java o Kotlin, mientras que para iOS utilizaremos Objective-C y Swift, generando así código nativo de cada plataforma.

Todo ello da como resultado aplicaciones fluidas y con la mejor experiencia de usuario. Como desventaja, suponen un incremento del coste de producción y de mantenimiento cuando deseamos que estén disponibles en múltiples plataformas, ya que debemos llevar un desarrollo paralelo.

Con tal de minimizar el desarrollo específico para una plataforma, disponemos de varias tecnologías alternativas basadas principalmente en tecnologías web. Vamos a ver cada una de ellas, según su *lejanía o acercamiento* al código nativo.

A. WEBAPPS O APLICACIONES WEB RESPONSIVAS

Se trata de aplicaciones basadas en tecnología web: HTML + CSS + JavaScript, y que para ejecutarse necesitan únicamente un navegador web. El hecho de ser responsivas implica que su interfaz se adapte a cualquier dispositivo. Para este tipo de aplicaciones no es necesario desarrollar nada en código nativo, y son totalmente multiplataforma, puesto que para ejecutarse lo hacen sobre el propio navegador web del sistema operativo. Disponemos, pues, de un único código para ejecutarse en todas las plataformas, con la principal desventaja de que no ofrecen una experiencia de usuario tan buena como las apps nativas ni permiten el acceso a todos los componentes del sistema.



B. APPLICACIONES HÍBRIDAS

Utilizan la terna HTML + CSS + JavaScript para construir un sitio web que se carga dentro de un componente de tipo WebView, que no es más que un navegador sin la barra de navegación ni otras opciones, por lo que presentan la apariencia de una aplicación nativa. Este tipo de aplicaciones ya pueden acceder, a través de este componente, a algunas características del dispositivo, como la ubicación o el acelerómetro. Actualmente, el framework para el desarrollo de aplicaciones híbridas más popular es Ionic, que permite el desarrollo en otros frameworks web como React, Angular o Vue. Otro framework muy popular en su momento fue Adobe Phonegap, pero Adobe abandonó su desarrollo en 2020, en favor de las aplicaciones web progresivas.

C. APPLICACIONES WEB PROGRESIVAS (PWA)

Un poco más cerca de las aplicaciones nativas tenemos las aplicaciones web progresivas, que están revolucionando el panorama actual. Estas aplicaciones incrementan y avanzan en las funcionalidades según el dispositivo móvil en que vayan a ejecutarse, para sacar mayor potencial, y pueden acceder al hardware, trabajar sin conexión o con mala conexión u ofrecer notificaciones del sistema. Existen multitud de frameworks para el desarrollo de PWA, entre los que se encuentran React PWA Library, Angular PWA Framework, Vue PWA Framework, Ionic PWA Framework, Svelte, PWA Builder o Polymer.

D. APPLICACIONES COMPILADAS

Se trata de tecnologías que pretenden utilizar solamente un lenguaje de programación para generar aplicaciones móviles en el código nativo de cada plataforma. Algunas de las tecnologías más utilizadas en este tipo de aplicaciones son las siguientes:

- **React Native y Native Script.** Utilizan como base el lenguaje de programación JavaScript, pero en lugar de construir las interfaces mediante HTML utilizan componentes propios del framework que son compilados a código nativo, haciendo innecesario utilizar un WebView como intermediario.
- **Flutter.** Desarrollado y mantenido por Google, permite el desarrollo de aplicaciones multiplataforma mediante el lenguaje Dart, compilando a código nativo que se ejecuta completamente en el dispositivo. La forma de trabajar de Flutter consiste en diseñar interfaces de usuario mediante widgets integrados en el propio código. Flutter ya cuenta con una serie de widgets predeterminados como botones, barras de navegación, etc.

2. El sistema operativo Android. Herramientas de desarrollo y gestión de dispositivos

2.1. El sistema operativo Android

Android es el sistema operativo para dispositivos móviles más utilizado en la actualidad. Como ya sabemos, se trata de un sistema operativo para dispositivos con pantalla táctil y basado en el kernel de Linux, así como en otros estándares de software abierto.

Los orígenes de Android se encuentran en la empresa de Palo Alto (California) Android Inc, fundada en 2003 por Andy Rubin. Esta empresa empezó con el desarrollo de un sistema operativo para cámaras fotográficas, pero ante la escasa rentabilidad del sector decidieron dar el salto a los dispositivos móviles, dominado en aquel momento por Symbian y Windows Mobile. En 2005 Google adquirió Android Inc; dos años después, junto con la creación de la Open Handset Alliance, un consorcio de varias compañías tecnológicas cuyo objetivo era el desarrollo de estándares abiertos para dispositivos móviles, se anunció la primera versión de Android, Apple Pie. Esta versión empezó a incorporarse en terminales en 2008, y dos años después alcanzó prácticamente la mitad de la cuota de mercado.

A diferencia de otros sistemas, Android se desarrolla de forma abierta, de modo que se puede acceder tanto al código fuente como a la lista de incidencias. El proyecto Android Open Source Project contiene el código fuente de Android, liberado bajo una licencia Apache.

En octubre de 2021 se lanzó la última versión de Android, Android 12, y desde la primera versión se han ido incorporando actualizaciones y funcionalidades al sistema. Además, las facilidades que aporta para el desarrollo de aplicaciones han hecho que actualmente cuente con una gran comunidad de desarrolladores cuyo fruto son más de tres millones de apps en la Google Play Store, el repositorio de aplicaciones oficial de Android.

A. CARACTERÍSTICAS

Algunas de las características más relevantes de Android son las siguientes:

- Adaptable a pantallas y resoluciones muy variadas, con soporte a librerías gráficas 2D y 3D basadas en OpenGL.
- Soporta pantallas multitáctiles de forma nativa.
- Ofrece almacenamiento local a través de una base de datos ligera, como SQLite.
- Soporta un gran abanico de tecnologías de conectividad.
- Incluye un navegador web basado en el motor de renderizado Webkit y el motor JavaScript V8, de Google Chrome.
- Las aplicaciones se programan en Java o Kotlin, y son compiladas a la máquina virtual ART (Android Runtime).
- Incorpora soporte para una gran diversidad de formatos multimedia y streaming.
- Da soporte para una gran cantidad de dispositivos hardware y sensores.
- Dispone de un catálogo muy extenso de aplicaciones a través del servicio Google Play.

B. VERSIONES DE ANDROID

Android ha sufrido una gran transformación desde sus primeras versiones. A modo de resumen, vamos a ver algunas características de las actualizaciones más relevantes.

El sistema inició su camino en 2008, con las versiones 1.0 y 1.1, que incluían apps como Gmail, Maps, Calendar y YouTube. Fue en la versión 1.5, *Cupcake*, cuando se introdujo un nombre de versión y se incluyó una gran cantidad de mejoras en la interfaz, como el teclado en la pantalla, lo que permitió deshacerse de los teclados físicos en el dispositivo. Posteriormente, con la versión 1.6, *Donut*, el sistema comenzó a adaptarse a diferentes tamaños y resoluciones de pantalla.

Las versiones 2.0 y 2.2 (*Eclair* y *Froyo*) introdujeron aspectos como la navegación guiada por voz y la información de tráfico en tiempo real y el dock inferior. La versión 3.0, *Honeycomb*, se lanzó específicamente para dispositivos de mayor tamaño, como las tabletas, y en la versión 4.0, *Ice Cream Sandwich* de 2011, la nueva interfaz, *Holo*, se unificó para tabletas y smartphones, entrando así en la era del diseño moderno.

Pero el mayor cambio estaba por venir con las versiones 5.0 y 5.1, *Lollipop*, en 2014. Google reinventa Android en la que quizá sea la actualización más importante del sistema, introduciendo el estándar *Material Design*, un conjunto de especificaciones de diseño que afectaban tanto al sistema operativo como a la web o las apps. Con *Material* se introducen los diseños con colores planos para dar peso y profundidad a diferentes elementos de la interfaz utilizando sombras, capas o animaciones.

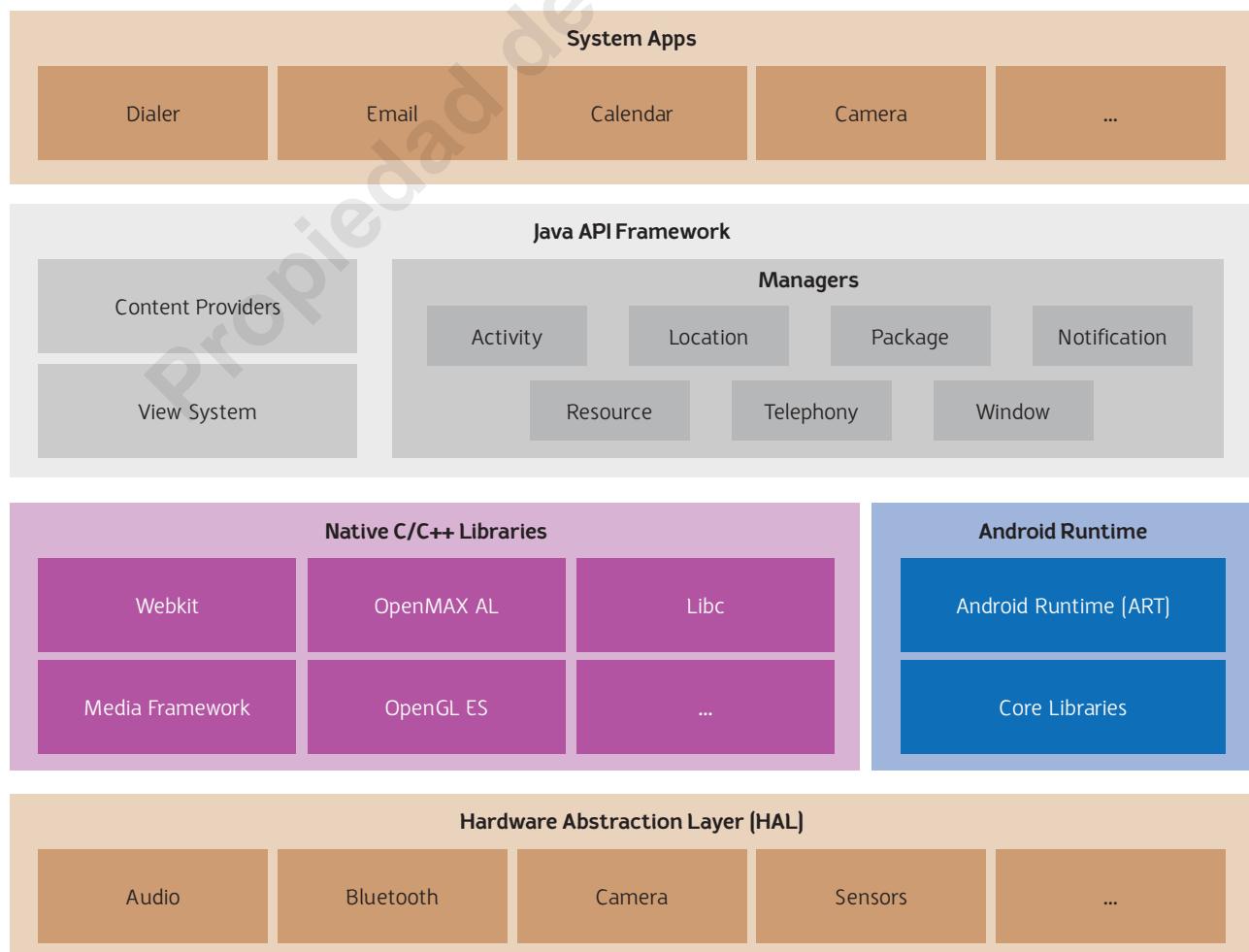
Las versiones posteriores incluyeron algunas mejoras en la interfaz, como el modo de pantalla dividida en Android 7.0/7.1 (*Nougat*) y la pantalla flotante en Android 8/8.1 (*Oreo*). Con Android 10 se abandonaron los nombres clave de versión basados en dulces y se reinventaron algunos aspectos de la interfaz con relación a los gestos y al sistema de navegación.

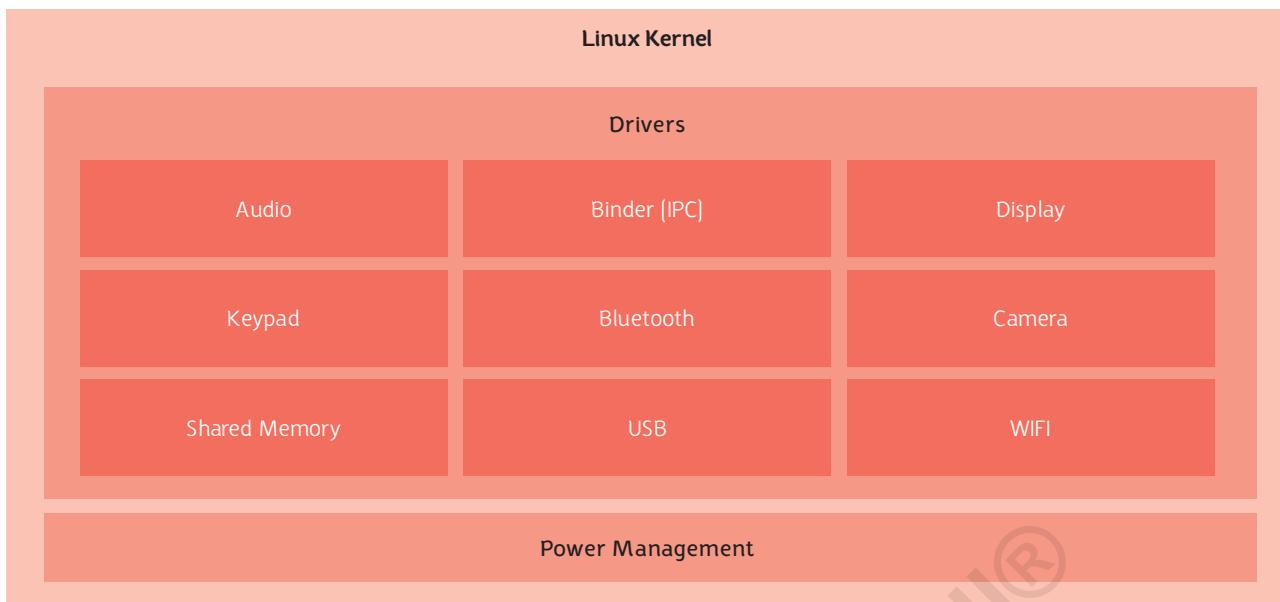
En Android 11, en 2020, llegaron cambios sustanciales en la privacidad, que permiten al usuario dar permisos a las aplicaciones para acceder a la localización, la cámara o el micrófono de forma temporal y cuando se requieran, en lugar de hacerlo permanentemente durante la instalación.

Y en 2021 llegó Android 12, que supone la mayor actualización desde Android 5.0, con la introducción de *Material You*, que supone una personalización del diseño del sistema para ajustarse automáticamente a la configuración del usuario. Esto implica, por ejemplo, que la paleta de colores del tema se adapte al fondo de pantalla elegido por el usuario. Además, internamente ofrece una forma más simple y granular de controlar el modo en que las apps acceden a nuestros datos, a través del nuevo panel de privacidad.

C. ARQUITECTURA DE ANDROID

La arquitectura de Android está compuesta por una pila de capas que toma su base en el kernel de Linux y que se adapta a una gran variedad de dispositivos. Los diferentes componentes del sistema se muestran en la figura siguiente:





Fuente: Android Open Source Project, imagen utilizada conforme a los términos de la licencia Creative Commons 2.5.

Attribution: <https://developer.android.com/guide/platform?hl=es-419>. [COMPONER EN MAQUETA].

En ella, podemos ver los componentes siguientes:

- **Aplicaciones del sistema.** Forman el sistema base y se componen del cliente de correo electrónico, el calendario, la aplicación de SMS, los mapas, la aplicación de cámara, el navegador y la aplicación de contactos o el dialer, entre otras.
- **Framework de aplicaciones (Java API Framework).** Ofrece a las aplicaciones acceso a todas las funciones de Android, así como la posibilidad de publicar y ofrecer sus funcionalidades a otras aplicaciones de forma segura. Todo ello, pensando en la simplificación de la reutilización de componentes del sistema y servicios modulares. Entre estos servicios podemos encontrar:
 - Un sistema de vista Enriquecido, para compilar la interfaz de usuario,
 - Un administrador de recursos, para acceder a los recursos de nuestras aplicaciones, tales como las traducciones, las imágenes o los archivos de diseño,
 - Un administrador de notificaciones, para poder mostrar alertas en la barra de estado,
 - Un administrador de actividad, para administrar el ciclo de vida de las aplicaciones y gestionar la navegación,
 - Proveedores de contenidos, para que las aplicaciones puedan acceder a datos facilitados por otras aplicaciones.
- **Bibliotecas de C/C++.** Son bibliotecas que utilizan algunos componentes del sistema y que se ofrecen a los desarrolladores a través del framework de aplicaciones. Algunas de estas son las bibliotecas de C estándar, las bibliotecas gráficas 3D, las multimedia o la base de datos SQLite.
- **Runtime de Android.** Incluye la propia máquina virtual ART y las bibliotecas base de Java. Las apps ejecutan sus propios procesos como instancias de la propia máquina virtual ART, bajo el paradigma de virtualización de proceso (como la JVM para PC). Los ejecutables para esta máquina virtual se encuentran en formato DEX, un bytecode diseñado específicamente para Android y optimizado para ocupar un espacio de memoria mínimo.
- **Capa de abstracción de Hardware.** Consta de una serie de módulos de biblioteca para los diferentes componentes hardware del dispositivo, como la cámara o los sensores, y ofrece una interfaz a las capas superiores para facilitar el acceso a estos.
- **Kernel de Linux.** Es la base de la plataforma Android, a la que proporciona funcionalidades básicas del sistema como la gestión de procesos, la administración de memoria o de la red, los controladores o la seguridad.



D. APLICACIONES ANDROID

Las aplicaciones Android se programan en Kotlin, Java o C++, y son compiladas por el Android SDK, junto con ficheros con datos y recursos (interfaces, imágenes, etcétera) para generar un fichero .apk, que incluya la aplicación en sí y la información que necesita Android para su instalación.

Android implementa el principio del mínimo privilegio en las aplicaciones, con el que aporta cierta seguridad a estas. Este principio se fundamenta en:

- Android es un sistema operativo multiusuario, puesto que se basa en Linux. La peculiaridad es que para Android cada aplicación será un usuario diferente.
- Cada app tiene un ID de usuario conocido solamente por el sistema, y establece los permisos necesarios para que esta pueda acceder a sus recursos.
- Cada proceso tiene su máquina virtual, de forma que su código se ejecuta de forma independiente. En principio, cada app tendrá su propio proceso.

Con esto, cada aplicación tiene únicamente acceso a los componentes que necesita. De todas maneras, una aplicación puede compartir datos con otras aplicaciones y acceder a servicios del sistema, bien haciendo que dos aplicaciones compartan el ID de usuario bien solicitando permiso al usuario para acceder a datos y recursos del dispositivo (la cámara, el micrófono, la conexión bluetooth, la tarjeta SD, los contactos, etc.).

E. LOS NIVELES DE LA API, LAS BIBLIOTECAS DE COMPATIBILIDAD Y JETPACK

Hemos visto cómo las diferentes versiones de Android introducen cambios sustanciales en cuanto a funcionamiento y prestaciones. A nivel de desarrolladores, más que la versión, nos interesan los niveles de la API. Estos determinan la compatibilidad de las aplicaciones con las versiones de Android. Con cada nueva versión del sistema, se introduce uno o varios niveles de la API, en función de los cuales dicha API nos ofrecerá unas u otras funcionalidades del dispositivo.

Cuando empecemos con el desarrollo de una aplicación Android, deberemos tener en cuenta qué funcionalidades de la API vamos a necesitar y establecer cuál será el nivel mínimo que requerirá nuestra aplicación. Por ejemplo, si vamos a desarrollar una aplicación que queremos que siga los estándares de Material Design, deberemos utilizar un nivel mínimo de API 21, que es la que se introdujo en Android 5.0, o si deseamos funcionalidades presentes en Android 11 o 12, deberemos utilizar los niveles de API 30 y 31, respectivamente.

Con la finalidad de proporcionar funcionalidades nuevas o utilizar funcionalidades equivalentes en versiones anteriores de Android, Google lanzó una capa de compatibilidad mediante las support libraries (bibliotecas de compatibilidad). Actualmente esas bibliotecas se integran en Android Jetpack, un conjunto de bibliotecas que permite a los desarrolladores centrarse en la lógica de la aplicación y seguir las prácticas recomendadas, reduciendo código estándar y produciendo código coherente en las diferentes versiones del sistema.

2.2. Android Studio

Android Studio es el IDE oficial para el desarrollo en Android, y está basado en el IDE IntelliJ de JetBrains. Hasta 2014 se utilizaba Eclipse con el plugin de Herramientas de Desarrollo de Android [ADT].

Las principales características de Android Studio son las siguientes:

- Es un IDE multiplataforma (Windows, Linux y macOS).
- Se distribuye bajo licencia Apache 2.0 (libre).
- Utiliza Gradle para la construcción de paquetes.
- Incorpora un diseñador de interfaces de tipo WYSIWYG (What you see is what you get) que permite la creación de interfaces mediante arrastrar y soltar (drag and drop) sus componentes.
- Incluye una serie de plantillas de diseños comunes para las aplicaciones.
- Permite refactorización de código específica para Android.
- Ofrece soporte al desarrollo para diferentes dispositivos: smartphones, tabletas, televisores o wearables (tecnología incorporada en cosas que se pueden llevar puestas, con las que uno se puede vestir).

- Incorpora un emulador de dispositivos virtuales (Device Manager) para ejecutar, depurar aplicaciones o analizar el rendimiento.
- Incluye soporte para Google Cloud Platform, que permite la integración con varios servicios de Google.

A. DEVICE MANAGER

El administrador de dispositivos incorporado en Android Studio, Device Manager, nos permitirá crear y administrar dispositivos virtuales con Android y también vincular dispositivos físicos reales.

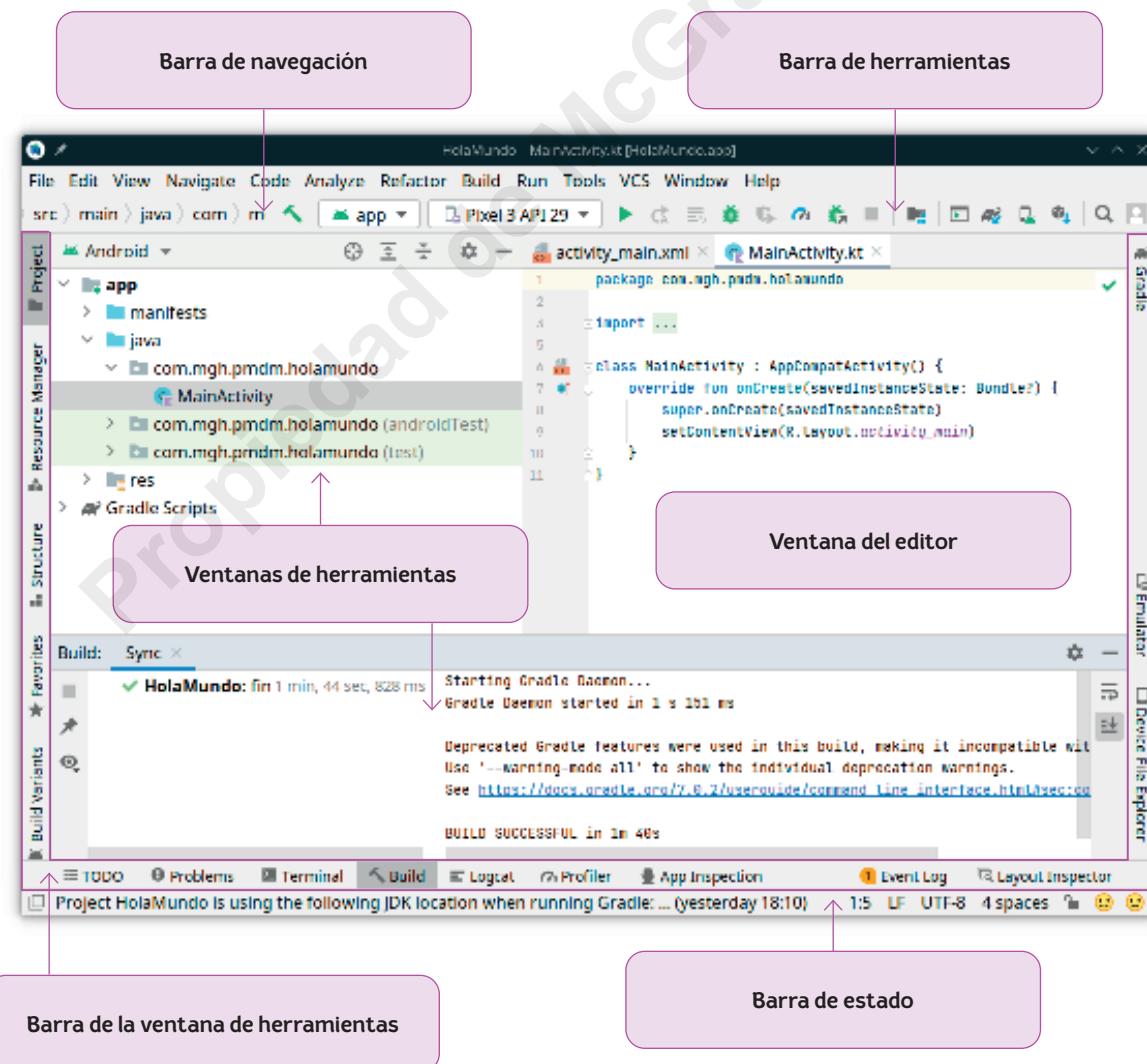
Debemos tener en cuenta que la arquitectura y el sistema para los que vamos a desarrollar nuestras aplicaciones son diferentes de los de nuestro equipo de desarrollo, sin contar que, además, podemos crear aplicaciones para una amplia gama de dispositivos.

Para la depuración de aplicaciones podemos, o bien conectar directamente un dispositivo al equipo y activar el modo desarrollador, o bien utilizar dispositivos virtualizados. La ventaja de la virtualización es que podremos probar nuestras aplicaciones en un gran abanico de dispositivos y configuraciones diferentes. La desventaja es que vamos a tener que crear una máquina virtual para cada uno de los dispositivos que deseemos probar, con el consumo de espacio y memoria que ello implica.

En los siguientes casos prácticos vamos a ver cómo realizar la instalación de Android Studio y cómo gestionar dispositivos en el Device Manager.

B. LA INTERFAZ DE ANDROID STUDIO

Cuando trabajamos en un proyecto, la vista que presenta Android Studio es similar a la siguiente:



Poco a poco iremos descubriendo detalles de esta interfaz, pero, como vemos, no dista mucho de los IDE que ya conocemos. A primera vista, podemos distinguir:

- La **barra de herramientas**, para guardar, abrir, construir o ejecutar el programa, entre otras opciones.
- La **barra de navegación**, para explorar el proyecto.
- La **ventana del editor**, donde tenemos el código o el diseño de la aplicación.
- La **barra de la ventana de herramientas**, que rodea todo el IDE y contiene botones para expandir o contraer ventanas de herramientas individuales.
- Las **ventanas de herramientas**, con varias utilidades como la administración de proyectos, la búsqueda de texto, el control de versiones, la ventana de depuración, etc. En la ventana de la izquierda de *Administración del proyecto* (Pestaña Project), aparecen varias vistas de nuestro proyecto, que nos mostrarán este desde diferentes perspectivas. Las que más utilizaremos será la propia del proyecto (*Project*), la de ficheros (*Project Files*) y, sobre todo, la vista *Android*.

Disponéis de más información sobre la interfaz en la documentación de desarrolladores de Android.

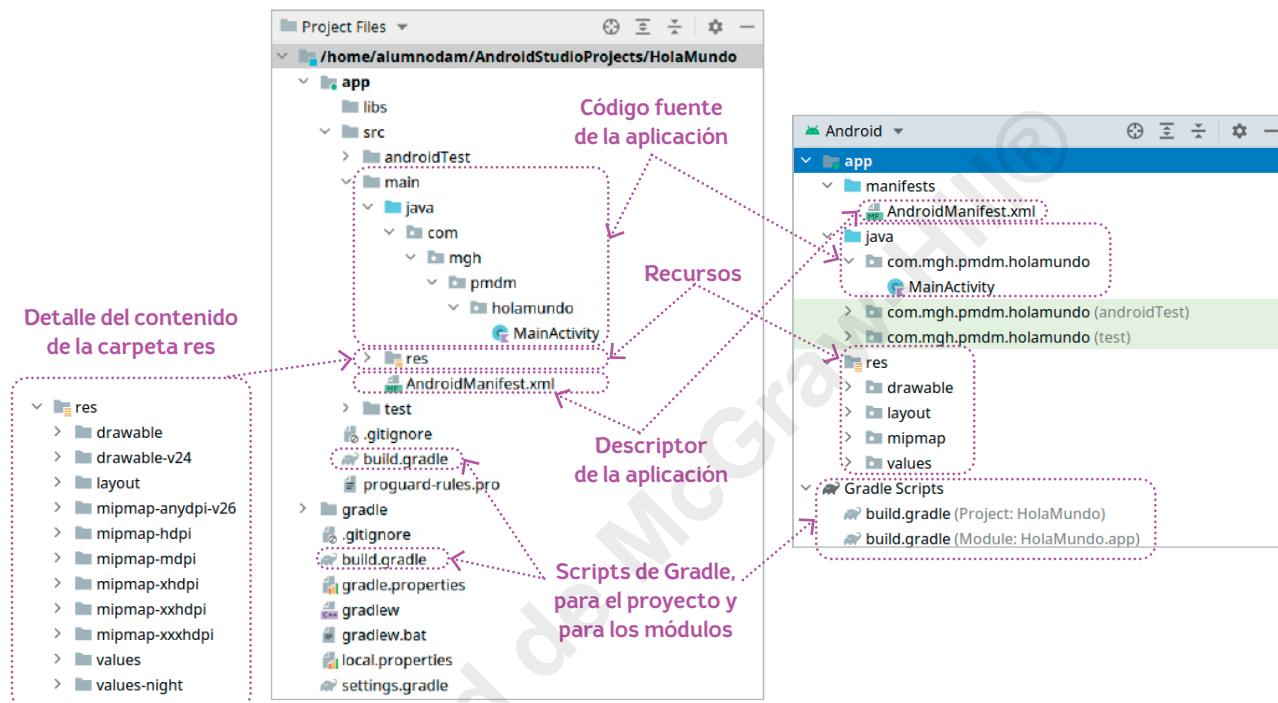
Propiedad de McGraw-Hill®

3. Estructura y componentes de una aplicación Android

3.1. Estructura de un proyecto en Android

Android Studio utiliza Gradle como herramienta de construcción de proyectos. Estos proyectos Android se organizan en módulos, de manera que cada uno de estos módulos será una aplicación diferente. Así, podemos tener bajo un mismo proyecto varias versiones de nuestra aplicación para diferentes tipos de dispositivos (tabletas, wearables, etcétera).

A continuación, vamos a ver la estructura que se genera en un proyecto Android, desde la vista de los ficheros del proyecto (*Project Files*) en la ventana lateral de herramientas, y desde la vista de Android.



En la vista de ficheros del proyecto podemos apreciar la estructura de ficheros típica de un proyecto Gradle, con archivos generales de la aplicación en la carpeta raíz, así como la carpeta propia del módulo de aplicación `app`. Sin embargo, la vista de Android presenta una perspectiva más compacta y práctica de esta información, organizada en cuatro carpetas lógicas: *manifests*, *java*, *res* y *Gradle Scripts*.

Veamos algunos de los elementos más importantes de esta organización:

- Los **scripts Gradle** de construcción del proyecto (`build.gradle`), tanto el general, situado en la raíz, con información común a todos los módulos, como el propio del módulo de la aplicación (`app/build.gradle`). En la vista de Android, se muestran ambos scripts dentro de *Gradle Scripts* y se indica si el script corresponde al proyecto o al módulo.
- Dentro de la carpeta del módulo (`app`) encontramos la carpeta `src`, que contiene el **código fuente** de la aplicación (`app/src/main`). En la vista Android, este se encuentra en la carpeta `java` y, como vemos, se muestra en formato de nombre de paquete, en lugar de mostrar la estructura de directorios.
- La carpeta `app/src/res`, que contiene los **recursos de la aplicación** (imágenes, diseños, cadenas de texto, etc.). Si nos fijamos en el detalle, esta carpeta incluye bastantes subcarpetas, para los diferentes tipos de recursos. En la vista de Android, este contenido se muestra de forma más compacta y organizada, según el tipo de recurso.
- El fichero **descriptor de la aplicación**, `app/src/main/AndroidManifest.xml`, con información asociada a esta. Como veremos, se trata de uno de los ficheros más importantes de nuestro proyecto, ya que define aspectos como el nombre de la aplicación y el paquete, el icono y sus diferentes componentes.

A. SCRIPTS DE GRADLE

Como hemos comentado, un proyecto Android define un script de configuración *build.gradle* general al proyecto, ubicado en la carpeta raíz, y otro ubicado en la carpeta *app*, que hace referencia al propio módulo de la aplicación.

El contenido del script de construcción general tendrá la estructura siguiente:

```
buildscript {  
    repositories {  
        google()  
        mavenCentral()  
    }  
    dependencies {...}  
}  
  
task clean(type: Delete) {  
    delete rootProject.buildDir  
}
```

Como podemos apreciar, simplemente especifica los repositorios generales para descargar paquetes y define la tarea para limpiar el proyecto.

El fichero de construcción que más nos interesaría es, pues, el del propio módulo de la aplicación. De forma simplificada, este fichero tiene la estructura siguiente:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
}  
  
android {  
    compileSdk 31  
  
    defaultConfig {  
        applicationId "com.mgh.pmdm.holamundo"  
        minSdk 21  
        targetSdk 31  
        ...  
    }  
  
    buildTypes { ... }  
    compileOptions { ... }  
    kotlinOptions { ... }  
}  
  
dependencies {  
    implementation 'androidx.core:core-ktx:1.7.0'  
    implementation 'androidx.appcompat:appcompat:1.4.0'  
    implementation 'com.google.android.material:material:1.4.0'  
    ...  
}
```

En primer lugar, el script carga un par de plugins para generar una aplicación Android (`com.android.application`) en Kotlin (`kotlin-android`).

En segundo lugar, observamos que aparece una sección `android`, en la que vamos a definir los aspectos principales de construcción para Android, entre ellos:

- La versión del SDK con la que compilamos la aplicación (`compileSdk`).
- La configuración de esta (`defaultConfig`), con el ID, el SDK mínimo que soportará (`minSdk`) o la versión más alta con la que se ha probado la aplicación (`targetSdk`), entre otros. Estas opciones se configurarán cuando creemos la aplicación.
- Algunos detalles para la construcción y la compilación del proyecto (versiones `release` o `debug` en `buildtypes`, o la versión del bytecode que se generará con `compileOptions`).
- Las dependencias (`dependencies`) del proyecto, entre las que destacan las bibliotecas de Jetpack (`androidx.*`), que incluyen las bibliotecas de compatibilidad (`androidx.appcompat`) y las bibliotecas de componentes Material Design (`com.google.android.material`).

En este punto, debemos hacer una observación respecto a otro tipo de aplicaciones Gradle con las que hemos trabajado previamente, y es que no se ha especificado la clase principal (`mainClass`) y ni siquiera existe la sección `application`. Como veremos a continuación, el punto de entrada lo especificaremos en el fichero `AndroidManifest.xml`.

B. EL FICHERO ANDROIDMANIFEST.XML Y LOS COMPONENTES DE LA APLICACIÓN

El fichero de Manifest es un fichero propio de cada aplicación que contiene información sobre esta. La información se utiliza tanto por las herramientas de creación de Android como por el propio sistema y por Google Play.

Entre la información que vamos a poder encontrar en él, podemos destacar:

- El nombre del paquete de la aplicación, que generalmente coincide con el paquete de Java de las entidades de programación que componen nuestra aplicación, de modo que el sistema sepa dónde encontrarlas.
- Los diferentes componentes de la aplicación y sus propiedades: las actividades (`<activity>`), los servicios, (`<service>`) los receptores de emisiones (`<receiver>`) y los proveedores de contenidos (`<provider>`). Trataremos estos componentes más adelante.
- Los permisos de la aplicación para acceder a recursos protegidos del sistema o a otras aplicaciones, así como los permisos que deben poseer otras aplicaciones para acceder al contenido de esta.
- Las diferentes funcionalidades hardware y software que va a necesitar nuestra aplicación, de modo que Google Play impida su instalación en dispositivos que no posean dichas funcionalidades.

A modo de ejemplo, veamos el `AndroidManifest.xml` de una aplicación tipo Hola Mundo como la que crearemos en el **Caso práctico 1**. La estructura general del fichero es la siguiente:

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

    <application>
        ...
    </application>
</manifest>
```

Como vemos, el elemento raíz `manifest` define:

- El espacio de nombres `android`, con `xmlns:android`, lo que nos permite utilizar posteriormente atributos de tipo `android`:

- El paquete `com.pmdm.holamundo` de la aplicación. Este nombre de paquete será usado por el sistema para nombrar tanto los recursos como los diferentes elementos declarados en el manifest.

Pasamos ahora al elemento `application`, que contiene toda la información sobre la aplicación:

```
<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/Theme.HolaMundo">
    <activity> ... </activity>
</application>
```

Como podemos apreciar, en él se definen varios atributos específicos de Android, como el ícono, la etiqueta o el tema que determina el aspecto de la aplicación. Muchos de estos elementos aparecen precedidos por el símbolo `@`, lo que indica que hacen referencia a recursos de la aplicación.

Como ya hemos comentado, dentro de `application` se definen los diferentes componentes de esta, en este caso con una única actividad:

```
<activity
    android:name=".MainActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Con esto indicamos que la aplicación lleva por nombre `.MainActivity` (realmente es `com.mgh.pmdm.MainActivity`), y que esta va a poder recibir mensajes desde fuera de la aplicación (`android:exported="true"`) o puede ser llamada por otros componentes de esta.

Por otra parte, también se define un elemento `intent-filter`, donde se especifican las acciones a las que va a responder esta actividad. En este caso, se define la acción `android.intent.action.MAIN`, con categoría `android.intent.category.LAUNCHER`. Con `MAIN` indicamos que este es el punto de entrada a la aplicación, y especificando la categoría `LAUNCHER` indicamos que esta actividad debe aparecer en el lanzador de aplicaciones.

Dentro del mismo elemento `application` definiremos el resto de componentes de la aplicación. Los componentes que no se incluyan no serán visibles para el sistema.

C. RECURSOS DE LA APLICACIÓN

La carpeta de recursos de la aplicación (`res`) contiene, organizados en subcarpetas, los diferentes tipos de recursos que son utilizados por esta. Cuando programamos cualquier aplicación, es conveniente mantener los recursos de forma externa a la aplicación, para facilitar su mantenimiento de forma independiente.

Aunque existen varias carpetas físicas para los diferentes tipos de recursos, en la vista de `Android` se presenta una organización lógica de todos estos. Entre estas carpetas lógicas encontramos:

- `drawable/`: Contiene elementos de diseño gráfico que podemos dibujar en la pantalla del dispositivo: no solo imágenes, sino también listas de capas, estados o niveles, entre muchos otros.
- `mipmap/`: Contiene los diferentes iconos de la aplicación, en diferentes densidades (ppp).
- `layout/`: Contiene los diseños de diferentes partes de la interfaz de usuario, en formato XML.

- **values/**: Contiene ficheros en formato XML con valores simples, como cadenas de caracteres, valores enteros o colores.
- **menu/**: Con archivos XML que definen diferentes tipos de menú de la aplicación.
- **animator/** y **anim/**: Archivos XML con diferentes tipos de animaciones.
- **color/**: Archivos XML con listas de estados de color para aplicar sobre diferentes elementos de interfaz y que estos cambien de color según su estado.
- **xml/**: Archivos XML que podemos leer en tiempo de ejecución, con algunas configuraciones especiales.
- **font/**: Carpeta para almacenar tipografías específicas de la aplicación.
- **raw/**: Contiene archivos sin procesar, que deberemos tratar como *streams*.

IMPORTANTE

Cuando compilamos una aplicación, Android procesa todos los recursos de la carpeta res y así genera la clase R, que contiene símbolos con las definiciones de estos recursos.

Esta clase estará dentro del espacio de nombres de nuestra aplicación. Por ejemplo, si una aplicación tiene como nombre de paquete de la aplicación com.mgh.pmdm, la clase con los recursos será com.mgh.pmdm.R.

Por otra parte, además de la clase R de recursos de nuestra aplicación, disponemos de una clase R con los recursos proporcionados por Android, que lleva por nombre android.R.

3.2. Componentes de una aplicación

Como hemos comentado en el punto anterior, las aplicaciones Android tienen principalmente cuatro tipos de componentes que podemos especificar en el archivo de manifiesto: actividades, servicios, receptores de anuncios y proveedores de contenidos.

A. ACTIVIDADES (ACTIVITIES)

Representan las diferentes pantallas de la aplicación y recogen la interacción con el usuario. Cada actividad puede ser un punto de entrada a la aplicación y puede ser requerida por otras aplicaciones, lo que permite reutilizar funcionalidades. Por ejemplo, si queremos enviar una foto mediante una aplicación de mensajería, se utiliza directamente la actividad correspondiente de la aplicación de la cámara en lugar de implementar esta funcionalidad en la aplicación.

B. SERVICIOS (SERVICES)

Los servicios son puntos de entrada general que nos permiten mantener la ejecución de la aplicación en segundo plano y que no proporcionan interfaz de usuario, como pudiera ser un servicio de reproducción de audio o la sincronización de datos desde la red, lo cual permite que el usuario se encuentre interactuando con otra actividad.

C. RECEPTORES DE ANUNCIOS (BROADCAST RECEIVERS)

Se trata de componentes que recogen ciertos eventos del sistema fuera del flujo habitual de las aplicaciones y permiten incluso enviar notificaciones a aplicaciones que no están en ejecución. Algunos ejemplos pueden ser lanzados por el propio sistema, como el aviso de apagado de la pantalla, el de batería baja o el de que se ha hecho una captura de pantalla. Estos componentes tampoco tienen interfaz gráfica, pero pueden crear notificaciones en la barra de estado o servir como puerta de entrada a otros componentes.

D. PROVEEDORES DE CONTENIDOS (CONTENT PROVIDERS)

Permiten compartir contenidos entre aplicaciones de forma segura. Ejemplos de ello son los contactos del usuario o el almacenamiento en el dispositivo.

E. OTROS COMPONENTES DE LA APLICACIÓN

Además de los anteriores componentes, que podemos incluir en el archivo de manifiesto, existen otros componentes clave en el desarrollo de aplicaciones Android. Estos componentes son los siguientes:

- **Vistas [View].** Se conoce como *vista* a cada uno de los componentes de la interfaz de usuario: botones, textos, etcétera. Todos estos elementos serán subclases de la clase *View*, y, aunque pueden crearse dinámicamente desde el código, lo más habitual será definirlos directamente en los archivos de diseño (*Layout*) y dejar que Android los cree por nosotros.
- **Diseños [Layout].** Ya hemos comentado que en la carpeta *layout* se encuentran los diferentes diseños de la interfaz de nuestra aplicación, en formato XML. Estos diseños contendrán las diferentes vistas o elementos de interfaz, tanto de una actividad como de un fragmento (*Fragment*). Pese a que se pueden crear directamente (ya que también son objetos de tipo *View*), lo habitual es generarlos en formato XML dentro de esta carpeta.
- **Fragmentos [Fragment].** Se introdujeron en Android 3.0, con la interfaz *Holo*, para aprovechar el mayor tamaño de las pantallas de las tabletas. Un fragmento consiste en una agrupación de vistas (elementos de la interfaz) que funcionan como un bloque. De este modo, en función del tamaño de la pantalla, una actividad puede constar de uno o varios fragmentos. Además, los fragmentos permiten una navegación entre ellos más ligera que las actividades, y por ello su uso es bastante habitual en las aplicaciones modernas.
- **Intenciones [Intents].** Las actividades, los servicios y los receptores de emisión se activan mediante mensajes asíncronos conocidos como Intenciones (*Intents*), que vinculan en tiempo de ejecución diferentes componentes. Estos mensajes pueden servir para activar un componente específico (intento explícito) o un tipo de componente más genérico (intento implícito), que puede ser tanto interno como externo a nuestra aplicación.

4. El lenguaje Kotlin

4.1. Kotlin. Generalidades del lenguaje

Kotlin es un lenguaje de programación creado por JetBrains en 2011, y en la actualidad desarrollado conjuntamente por JetBrains y Google a través de la Kotlin Foundation. Actualmente es el lenguaje recomendado por Google para el desarrollo de aplicaciones para la plataforma Android.

Algunas de las características más interesantes de Kotlin son las siguientes:

- Es un lenguaje multiplataforma en el sentido más amplio del término, ya que soporta la generación de código para diferentes plataformas, entre las que se encuentra la máquina virtual de Java (JVM), el runtime de Android ART e incluso JavaScript. Además, actualmente JetBrains está trabajando en KMM (*Kotlin Multiplatform Mobile*), lo que permitirá el desarrollo de aplicaciones iOS con prácticamente la misma base de código.
- El hecho de compilar directamente sobre JVM hace que sea totalmente compatible con Java y, por tanto, con sus librerías.
- Es un lenguaje orientado a objetos, con tipado estático e inferencia de tipos.
- Es un lenguaje más conciso, evita código innecesario y repetitivo.
- Es un lenguaje Null Safety, lo que significa que gestiona los valores nulos de forma segura y evita errores de tipo *NullPointerException*.

En este apartado vamos a ver los aspectos más relevantes del lenguaje para empezar a entender el código, y poco a poco iremos ampliando nuestros conocimientos de Kotlin para la programación en Android.

Para el desarrollo de Kotlin en nuestro escritorio necesitaremos el compilador de Kotlin, así como algún IDE que lo soporte, como IntelliJ o VSCode con el plugin de Kotlin. En nuestro caso, seguiremos utilizando Android Studio, ya que desarrollaremos para esta plataforma, aunque los ejemplos que son puramente de Kotlin los podemos probar en el Playground que nos ofrece Jetbrains.

HOLA MUNDO! GENERALIDADES DE KOTLIN

A modo de ejemplo, veamos la estructura de un típico «Hola Mundo!» en Kotlin:

```
/* Bienvenidos a Kotlin */
fun main() {
    // Esta función escribe Hola Mundo por la salida estándar
    println("Hola Mundo!")
}
```

Como podemos ver, se trata de un código bastante sencillo, pero del que ya podemos extraer algunas ideas:

- La función principal (*main*) se declara como una función de primer nivel, sin necesidad de que esté dentro de una clase. Por lo tanto, Kotlin soporta la programación orientada a objetos, pero no requiere, a diferencia de Java, que todo sean clases.
- La función *main* puede recibir argumentos, aunque no es necesario indicarlo.
- Tampoco hay que indicar la visibilidad de la función (*public*), ni si esta es estática (*static*).
- La función *println* está disponible directamente, sin pasar por *System.out*.
- Los signos de punto y coma (*;*) de final de línea son opcionales, y suelen evitarse por convención.
- Los comentarios, tanto de una línea como de varias, se expresan igual que en Java.

4.2. Variables y operadores en Kotlin

En Kotlin todos los tipos de datos son clases, de forma que podemos acceder directamente a sus propiedades y funciones miembro. Recordemos que para esto en Java necesitábamos clases envoltorio para los tipos básicos.

Para definir variables en Kotlin usaremos las palabras reservadas `var` o `val`, de modo que:

- Utilizaremos `var` para definir **variables mutables**, es decir, que pueden cambiar de valor.
- Utilizaremos `val` para definir **variables inmutables** o lo que en Java serían **valores constantes**.

En general, y por temas de rendimiento, se recomienda utilizar valores constantes siempre que sepamos que no van a ser modificados.

Por otro lado, hay que mencionar que, aunque Kotlin posea tipado estático, no obliga a definir el tipo de las variables, siempre y cuando este se pueda **inferir** a partir de su contexto. Por ejemplo:

```
val pi = 3.14 // Variable inmutable, con tipo inferido a Float
val modulo
    modulo="PMDM" // Variable inmutable, inferida a tipo String. Aunque
                    // aparezca en dos líneas, se trata de una única asignación
```

No obstante, si deseamos indicar el tipo de dato en la asignación, podemos hacerlo utilizando las sintaxis siguientes:

```
val nombreVariable: Tipo
val nombreVariable: Tipo = Valor
```

A. TIPOS DE DATOS

Kotlin posee prácticamente los mismos tipos de datos que Java:

- **Tipos numéricos:** Byte, Short, Int, Long, Float y Double.
- **Caracteres:** Char. Los caracteres especiales utilizan secuencias de escape mediante la barra invertida: `\'t`, `\'b`, `\'n`, `\'r`, `\'v`, `\'\", `\'\\`, `\'\$`.
- **Cadenas de caracteres:** String. Soporta también String Templates, o literales de cadena, que pueden contener expresiones que serán evaluadas y su resultado, concatenado a la cadena. Por ejemplo:

```
fun main(){
    val valor="Mundo!"
    println("Hola ${valor}")
    println("$valor té ${valor.length} caracteres")
    val temperatura = 27
    println("Con ${temperatura}° hace ${if (temperatura > 24) "calor" else "frío"})
}
```

- **Clases y objetos:** Como veremos más adelante, Kotlin, aparte de definir clases, nos va a permitir definir objetos directamente. Todas las clases de Kotlin, incluidas las clases que representan tipos básicos, son descendientes directa o indirectamente de la clase Any, que sería el equivalente a la clase Object de Java.

Para trabajar con estos tipos de datos utilizamos los diferentes operadores (de asignación, aritméticos, lógicos, etcétera), que son prácticamente los que ya conocemos de Java.

B. VALORES NULOS O NULLABLE TYPES

Hemos comentado que Kotlin es un lenguaje seguro (null safety), de modo que, entre otras cosas, evita errores del tipo `NullPointerException`. Esto se traduce en que, por defecto, todas las variables en Kotlin van a tener un valor no nulo.

De todos modos, es posible que en ocasiones necesitemos poder especificar que una variable pueda contener explícitamente un valor nulo. En este caso, deberemos definir dicha variable como nullable añadiendo un interrogante al tipo, del modo siguiente:

```
val variable: Tipo?=null
```

Estos son los operadores a utilizar:

- El operador *Safe Call Operator* '?.' Para aportar más seguridad a las variables nullables y poder acceder a propiedades o métodos de un objeto de este tipo, Kotlin nos ofrece el operador *Safe Call Operator*, compuesto por un interrogante y un punto (?.). Este operador nos permitirá acceder a atributos o métodos de un objeto solamente si tiene un valor no nulo. En caso de que este sea nulo, se ignorará, evitando así una excepción de tipo *NullPointerException*. Por ejemplo:

```
println (objeto?.valor) // Si objeto es nulo, se evitará la excepción
```

- El operador *Elvis* '?:' Kotlin también nos proporciona el operador (?:"), conocido como el operador Elvis y compuesto por un interrogante y los dos puntos. Este operador sirve para especificar un valor alternativo cuando la variable es nula. Por ejemplo:

```
val variable2=variable?:0 // variable2 tendrá el contenido de variable o 0
```

- Operador de aserción no-nula '!!' Este operador convierte cualquier valor de tipo nullable a un tipo no nulo, y se utiliza para poder forzar que se lance una excepción *NullPointerException* cuando accedemos a dicho valor nulo. Este operador será necesario en algunas ocasiones en que no podremos asegurar que una variable tenga o no valor.

Veamos un pequeño ejemplo del uso de estos operadores:

```
fun main() {  
    val nombre:String?="Android" // nombre puede ser nulo  
    println(nombre?.length) // muestra 7  
  
    val nombre2:String?=null // nombre2 puede ser nulo  
    println(nombre2?.length) // muestra null  
  
    // Si no utilizamos el operador safe call, provocamos el error:  
    // Only safe (?.) or non-null asserted (!!.) calls are allowed  
    // on a nullable receiver of type String?  
    // println(nombre2.length) // -> Error  
  
    // Ejemplos con el operador Elvis  
    println(nombre?.length ?: -1) // Muestra 7  
    println(nombre2?.length ?: -1) // Muestra -1  
  
    // Ejemplo que fuerza una excepción de tipo NullPointerException  
    println(nombre2!!.length)  
}
```

Podéis ver este código en funcionamiento en el enlace siguiente: <https://pl.kotl.in/gV3A-AfD6>

4.3. Funciones y expresiones lambda

Kotlin contempla que puedan existir funciones que no estén vinculadas como métodos a ningún objeto. Estas funciones se conocen como **funciones de primer orden** o **Top Level Functions**, y pueden ser utilizadas sin la necesidad de obtenerse como instancias de algún objeto.

La definición de una función en Kotlin sigue la sintaxis siguiente:

```
fun nombreFuncion(parámetro1:Tipo1, parámetro2:Tipo2...):TipoRetorno{  
    // Cuerpo de la función  
}
```

Observad que, del mismo modo que para definir variables invertimos el orden del tipo y el valor, aquí también lo invertimos para los argumentos y para el tipo de retorno de la función.

EXPRESIONES LAMBDA

Las expresiones lambda o funciones literales tampoco están asociadas a clases, objetos o interfaces. Su utilidad radica en que pueden pasarse como argumentos a otras funciones, conocidas como funciones de orden superior, Higher-Order functions (no confundir con las funciones de primer orden), y supone una simplificación del bloque de código de una función.

Aunque Java soporta expresiones lambda desde Java 8, en Kotlin son ligeramente distintas. Las principales características de una expresión lambda son las siguientes:

- Se expresa entre llaves {}.
- No tiene la palabra clave fun.
- No tiene modificadores de acceso [private, public, protected], ya que no pertenece a ninguna clase.
- Es una función anónima, es decir, sin nombre.
- No especifica el tipo de retorno, ya que este es inferido por el compilador.
- Los parámetros no se expresan entre paréntesis.
- Podemos asignar una expresión lambda a una variable y ejecutarla.

Veamos algunas formas de crear expresiones lambda.

Expresión lambda sin parámetros y asignada a una variable

- Declaración:

```
val msg = { println("Hola Mundo! Soy una función lambda") }
```

- Invocación:

```
msg()
```

Expresión lambda con parámetros y asignada a una variable

- Declaración: Se utiliza una flecha (->), de modo que en la parte izquierda se indican los argumentos y en la derecha, el cuerpo de la función.

```
val msg = { cadena: String -> println(cadena) }
```

- Invocación:

```
msg("Hola Kotlin!")
```

Paso de expresiones lambda a funciones

Las expresiones lambda pueden usarse como argumentos de otras funciones, conocidas como funciones de orden superior (Higher-Order functions).

Esto tiene varias utilidades, pero quizás la más habitual sea su uso en la gestión de eventos de la interfaz de usuario, como veremos más adelante.

Aquí vemos un pequeño ejemplo de uso de expresiones lambda:

```
fun operacion(x: Int, y: Int, lambda:(Int, Int)->Int):Int{
    return lambda(x, y)
}

fun main(){
    val x=5
    val y=3
    val suma = { x: Int, y:Int -> x + y }
    val resta = {x: Int, y:Int -> x - y }

    println(operacion(x, y, suma))
    println(operacion(x, y, resta))
}
```

Observamos que hemos definido una función llamada *operacion*, que recibe dos enteros, X e Y, y una expresión lambda, y devuelve un entero. Dicha expresión también recibirá dos enteros y devolverá un valor entero. Fijaos en la forma de declarar que un argumento es una función lambda:

lambda: (Int, Int) -> Int

En la función principal definimos las variables x e y, y las expresiones lambda *suma* y *resta*, que reciben dos números enteros y devuelven, respectivamente, su suma y su resta.

Posteriormente, se muestra el resultado de invocar a la función *operacion* pasándole x e y como argumentos y una u otra expresión lambda. El resultado es que se muestra el resultado de la suma por una parte [8], y por otra el resultado de la resta [2]. Observad cómo, mediante estas expresiones, tenemos la capacidad de proporcionar funciones a otras funciones para modificar su comportamiento.

Podéis ver este código en funcionamiento en el enlace siguiente: https://pl.kotl.in/iV_3kCKNL

El nombre de argumento *it*

Cuando una función recibe un único argumento y su tipo puede ser inferido, se genera automáticamente un argumento por defecto que lleva por nombre *it*. Este argumento puede utilizarse para simplificar más las expresiones. Veamos un ejemplo simple:

```
fun operacion(x: Int, lambda:(Int)-> Boolean):Boolean{
    return lambda(x)
}

fun main(){
    val x=5
    // Muestra true o false en función
    // De si el número es par o impar
    println(operacion(x, { it%2==0 } ))
}
```

Vemos que la expresión lambda `{ it%2==0 }` se indica directamente como argumento de la función *operacion*. Dicha expresión tiene un único argumento, por lo que dentro podemos utilizar *it* para trabajar con él.

4.4. Clases y objetos en Kotlin

La forma de declarar clases en Kotlin es bastante similar a la de Java, aunque con una sintaxis más compacta y versátil.

De hecho, aunque no tendría demasiada utilidad, Kotlin permite declarar una clase de la manera más simple posible con:

```
class nombreClasse
```

Para crear una instancia de esta clase, simplemente deberíamos declarar una variable del siguiente modo:

```
val miObjeto=nombreClasse()
```

Observad que en Kotlin no utilizamos el operador new para crear instancias, sino el nombre de la clase como si invocásemos directamente a una función.

En Kotlin existen varias formas de crear una clase que encapsule propiedades y métodos, y diferentes formas de definir el constructor. Vamos a ver una de estas formas, que es utilizando lo que se conoce como **constructor primario**, la forma que utiliza Android para crear clases:

```
class nombreClase constructor(parametro1: Tipo1,...,parametroN: TipoN) {  
    [nivelAcceso] [var | val] propiedad1: Tipo1  
    ...  
    [nivelAcceso] [var | val] propiedadN: TipoN  
  
    init {  
        this.propiedad1=parametro1;  
        ...  
        this.propiedadN=parametroN;  
    }  
  
    [nivelAcceso] fun metodoX(lista_de_parametros): TipoRetorno {  
        // Cuerpo del método  
    }  
}
```

Como vemos, Kotlin permite declarar un constructor en la misma cabecera de la clase, conocido como el constructor primario. Además, se define un bloque **init** que se corresponde a la inicialización de parámetros de este constructor.

Asimismo, Kotlin permite simplificar esta construcción declarando las propiedades directamente en el constructor primario, precedidas de **var** o **val** según sean mutables o inmutables:

```
class nombreClase constructor([var | val ] propiedad1: Tipo1,  
    ...,  
    [var | val ] propiedadN: TipoN) {  
  
    [nivelAcceso] fun metodoX(lista_de_parametros): TipoRetorno {  
        // Cuerpo del método  
    }  
}
```

En esta misma construcción también podemos indicar valores por defecto siguiendo la sintaxis *propiedad1:Tipo1=Valor_por_defecto*. En caso que la clase no tenga modificadores de acceso o anotaciones, incluso puede omitirse la palabra clave constructor.

Veamos algunos detalles más acerca de la definición de clases en Kotlin:

- Las propiedades de la clase pueden ser mutables [var] o inmutables [val], y podemos indicar modificadores de acceso como public, private o protected.
- Esta construcción admite otros constructores secundarios, con diferentes bloques de tipo constructor, y así permite la sobrecarga de constructores.
- En Kotlin no necesitamos definir métodos consultores y modificadores (getters y setters), ya que estos son generados directamente por el compilador para las propiedades públicas. Además, se simplifica su acceso, puesto que se utilizan de forma transparente a través del nombre de cada propiedad. Si la propiedad es mutable, se generará un getter y un setter, mientras que, si es inmutable, únicamente se generará el getter.
- Los métodos se declaran con la misma sintaxis que una función, pero de manera interna a la clase.

Para crear una instancia de la clase anterior y acceder a sus propiedades, lo haríamos de este modo:

```
[ val | var ] objeto=Clase(Valor1, ... , ValorN)  
println(objeto.propiedad1) // Realmente accedemos al getter de propiedad1  
objeto.propiedad2 = valorX // Accedemos al setter, si propiedad2 es mutable
```

A. HERENCIA

Los conceptos de herencia y polimorfismo en Kotlin son prácticamente los mismos que en Java, salvo algunos detalles que mencionaremos a continuación, así como la sintaxis, que varía ligeramente.

La diferencia más importante con respecto a Java es que las clases se definen como final de forma predeterminada. Es decir, si no se especifica nada una clase se considera final y no se podrán crear clases derivadas de ella. Esto sigue una de las buenas prácticas de la ingeniería del software, que promueve que las clases se definan siempre como finales y únicamente se deje como abiertas aquellas que sí tendrán herencia de forma explícita.

Para definir una clase abierta, en Kotlin la declararemos precedida de la palabra clave open. También debemos tener en cuenta que los métodos definidos en una superclase se consideran finales, de modo que para que se puedan sobrescribir necesitan definirse explícitamente como open.

```
open class nombreSuperclase {  
    ...  
    open fun f1(Parametros:Tipos):TipoRetorno{...}  
    open fun f2(Parametros:Tipos):TipoRetorno{...}  
    ...  
}
```

Para indicar que una clase desciende de otra se utilizan los dos puntos. Si deseamos sobrescribir los métodos, utilizaremos la palabra clave override antes de definir el método (a diferencia de Java, en que utilizamos anotaciones):

```
class nombreSubClase : nombreSuperclase() {  
    ...  
    // Sobreescritura del método f1. Por defecto, este método sobreescrito  
    // ahora será open.  
    override fun f1(Paràmetros:Tipos):TipoRetorno{...}
```

```
// Sobreescritura de f2, declarándolo como final
override final fun f2(Parametros:Tipos):TipoRetorno{...}

}
```

B. INTERFACES

Recordemos que una interfaz define un comportamiento común a varias clases que la implementan. Este comportamiento se concreta en una serie de métodos abstractos que las clases que implementan la interfaz deben aplicar.

La forma de definir una interfaz en Kotlin es similar a la definición de una clase, pero utilizando la palabra clave *interface* y sin utilizar constructores:

```
interface IdentificadorInterfaz {

    // Constantes
    val Identificador_1:Tipo_1=Valor_1;
    ...

    // Métodos abstractos
    fun Metodo(parámetro_1:Tipo_1): TipoRetorno_1;
    ...

}
```

Y para definir una clase que implemente la interfaz utilizamos una sintaxis muy parecida a la de la herencia, con la diferencia de que no invocamos al constructor:

```
class NombreClase : NombreInterfaz {

    // Implementación de los métodos de la interfaz
    override fun Metodo(parámetro_1:Tipo_1): TipusRetorno_1{
        // Implementación
    }
    ...
}
```

Además, una clase puede implementar tantas interfaces como desee, separándolas por una coma, aunque solo va a poder extenderse de una clase. A diferencia de Java, aquí la palabra clave *override* es obligatoria para sobrescribir los métodos de la interfaz.

Una interfaz también puede ofrecer una implementación por defecto para los métodos, pudiéndolos utilizar directamente en las clases que implementan la interfaz o sobrescribirlos con *override*. En caso de sobrescribir los métodos por defecto, se puede acceder a la implementación por defecto mediante *super*.

La programación orientada a objetos de Kotlin guarda todavía muchas más cosas interesantes, como la posibilidad de personalizar *getters* y *setters* o la creación directa de objetos sin necesidad de instanciar clases. Todo ello lo iremos viendo poco a poco a lo largo del curso y en los materiales adicionales.

IMPORTANTE

Documentación complementaria sobre Kotlin

En este apartado hemos visto una pequeña introducción al lenguaje Kotlin y a su sintaxis. Puedes consultar la ampliación de estos contenidos en la sección *Material de trabajo/Enlaces web*.

5. Actividades, ciclo de vida y depuración

5.1. Actividades

Como hemos visto, las actividades son uno de los componentes principales de las aplicaciones en Android y representan las diferentes pantallas en que recogemos la interacción del usuario.

Las actividades son objetos de la clase `Activity` o alguna clase descendiente, como `AppCompatActivity`, y son instanciadas por el propio sistema Android. Además, están vinculadas a algún recurso de tipo `Layout`, donde se especifican los diferentes elementos de interfaz (botones, textos, etc.). Estos elementos serán elementos especializados de la clase `View`, conocidos como *Vistas*.

En este apartado vamos a examinar y trabajar el proyecto *Contador*, que deberás haber descargado en la sección Ejemplos para trabajar y analizar.

A. LA CLASE MainActivity

Cuando generamos una aplicación basada en un proyecto de tipo Empty Activity, se genera una clase principal `MainActivity` (fichero `MainActivity.kt`), con el contenido siguiente:

```
package com.mgh.pmdm.contador

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)
    }
}
```

La sintaxis de este código ya nos resulta familiar. En primer lugar, define el paquete `com.mgh.pmdm.contador` de la aplicación, y seguidamente importa un par de clases:

- `androidx.appcompat.app.AppCompatActivity`: Hace referencia a las librerías de compatibilidad (`appcompat`) dentro de Jetpack (`androidx`). De estas librerías importa la clase `AppCompatActivity`, que es la clase base para aquellas actividades que requieran características comunes como la barra de acciones, la barra de herramientas, los modos de navegación o el cambio entre temas claros y oscuros.
- `android.os.Bundle`: Se trata de una clase del sistema que permite mantener el estado de la actividad en la recarga y facilita la comunicación entre actividades. Posteriormente veremos su utilidad.

Después de importar las librerías, se declara la clase principal `MainActivity` como una subclase de `AppCompatActivity` y se sobrescribe el método `onCreate`, que recibe un objeto `nullable` de tipo `Bundle` llamado `savedInstanceState`. Profundizaremos en todo esto más adelante.

De momento, como podéis ver, aunque se trate de la clase que gestiona la actividad principal de la aplicación, no existe un método o una función `main` que la inicie explícitamente. De esto se encargará el propio sistema Android. Recordad que en archivo `AndroidManifest.xml` se registran las actividades de las que se compone la aplicación, y en este registro se indican sus puntos de entrada. Concretamente, sabemos que esta es la actividad principal de la aplicación por la acción `android.intent.action.Main`:

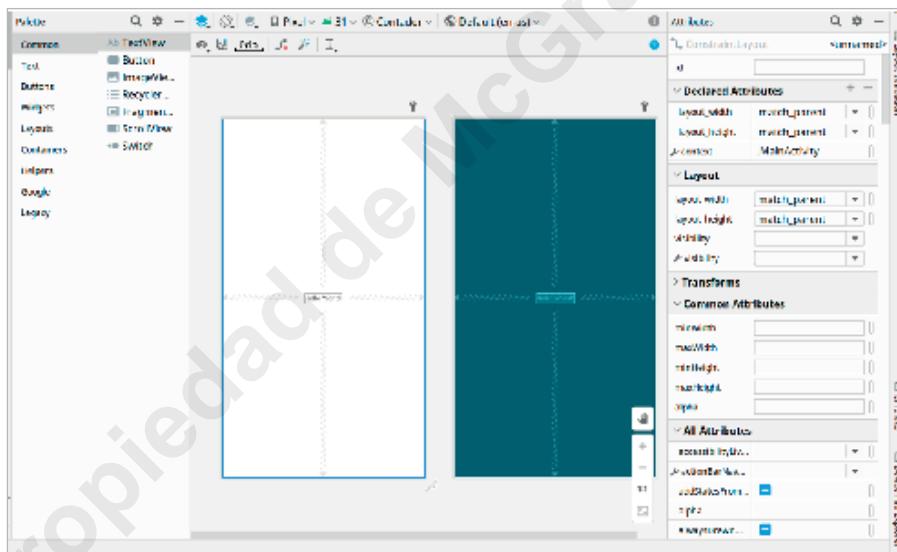
```
<activity
    android:name=".MainActivity"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Más adelante detallaremos el ciclo de vida de una actividad. De momento, podemos avanzar que el método `onCreate` se invoca automáticamente cuando se crea la actividad. Este método contiene, inicialmente, una llamada al método `onCreate` de su superclase (`AppCompatActivity`) y una llamada al método `setContentView`, que establece el contenido de la vista (interfaz) a partir del recurso `R.layout.activity_main`.

B. EL LAYOUT O DISEÑO DE LA ACTIVIDAD

Cuando se crea la actividad y se establece el contenido de la vista, el recurso `R.layout.activity_main` con el que se inicializa hace referencia al fichero `activity-main.xml`, ubicado dentro de la carpeta de recursos `res/layout`.

Estos ficheros XML contienen el diseño con los diferentes elementos de la interfaz de usuario de las actividades. A pesar de ser un fichero XML, Android Studio muestra una interfaz WYSIWYG (What you see is what you get), de forma que vemos el diseño de la pantalla junto con una paleta de componentes que podemos arrastrar y soltar, así como los atributos de cada elemento de la interfaz.



Trabajaremos con este diseñador en la unidad siguiente. Por ahora, observad que en la parte superior derecha este diseñador ofrece tres vistas: Code, Split y Design.

Podemos consultar el código XML de la interfaz con la vista de código (Code), o podemos usar la vista dividida (Split) para ver el código y el diseñador al mismo tiempo.

En el código XML de la interfaz podemos encontrar algunos elementos interesantes:

```
<androidx.constraintlayout.widget.ConstraintLayout
    ...
    tools:context=".MainActivity">
    <TextView
        ...
        android:text="Hello World!">
        ...
    </androidx.constraintlayout.widget.ConstraintLayout>
```

Como podemos ver, consta de un elemento raíz de tipo ConstraintLayout asociado, en principio, a la actividad MainActivity, dentro del cual se define un componente o Vista de tipo TextView con el texto «Hello World!».

C. LA CLASE MainActivity EN EL PROYECTO CONTADOR

En el proyecto Contador que está disponible para descarga se han añadido algunos componentes más a la interfaz y, también, funcionalidad a la clase MainActivity para ilustrar algunos conceptos nuevos.

Como veréis, se trata de una aplicación con una única actividad, compuesta por un contador y un botón que incrementa dicho contador cuando se hace clic en él. Veamos cada una de las partes.

El Layout del contador

El diseño de la actividad consta ahora de una vista de tipo TextView y de un botón (Button):

```
<androidx.constraintlayout.widget.ConstraintLayout ...>
    <TextView
        android:id="@+id/textViewContador"
    ... />
    <Button
        android:id="@+id/btAdd"
    .../>
</androidx.constraintlayout.widget.ConstraintLayout>
```

En estas vistas hemos incluido un atributo identificador, *android:id*, para poder referirnos a las vistas dentro del código. Cuando definimos los atributos con *@+id* indicamos que estamos incorporando un nuevo identificador a los recursos de la aplicación.

La clase MainActivity

El comportamiento de la aplicación se define en la clase principal MainActivity, donde se trata la interacción con el usuario igual que con cualquier interfaz gráfica: asociando una acción a un evento en la interfaz.

En primer lugar, en la clase se define una propiedad contador inicializada a 0:

```
class MainActivity : AppCompatActivity() {
    var contador=0
    ...
}
```

Y se añade el código siguiente dentro del método *onCreate*:

```
// Referencia al TextView
val textViewContador=findViewById<TextView>(R.id.textViewContador)

// Inicializamos el TextView con el contador a 0
textViewContador.setText(contador.toString())

// Referencia al botón
val btAdd=findViewById<Button>(R.id.btAdd)

// Asociaciamos una expresión lambda como
// respuesta (callback) al evento Clic sobre
// el botón
btAdd.setOnClickListener {
    contador++
    textViewContador.setText(contador.toString())
}
```

El método findViewById

Este método pertenece a la actividad y devuelve la vista correspondiente al identificador que se le proporciona como argumento. Estos identificadores, que definimos mediante `@+id` en el *Layout*, se encuentran en la clase *R*.

El método devuelve un tipo genérico, por lo que debe indicarse el tipo de vista de que se trata (`<TextView>` o `<Button>`). De todos modos, si declaramos un tipo para la variable que guardará la referencia, no es necesario indicar el tipo genérico. Por ejemplo:

```
val btAdd:Button=findViewById(R.id.btAdd)
```

Con este método se han obtenido el *TextView* del contador y el botón.

Una vez tenemos las referencias a los objetos de la interfaz, podemos utilizar sus propiedades y métodos, así como asignarles controladores de eventos. Por ejemplo, para establecer el texto del *TextView* del contador usamos el método `setText`:

```
textViewContador.setText(contador.toString())
```

IMPORTANTE

El enfoque actual de Google para referenciar a los elementos de la interfaz o las vistas desde el código es mediante las bibliotecas de vinculación de datos (*Data Binding*) que veremos en la unidad siguiente, en detrimento del método `findViewById`.

Controladores de eventos

Los controladores de eventos se utilizan para indicar las acciones a realizar cuando se produce un evento sobre una vista o un elemento de la interfaz. En el ejemplo hemos capturado el evento *Clic* sobre el botón *btAdd* y establecido un controlador de eventos mediante el método `setOnClickListener`, proporcionándole una expresión lambda con las acciones a ejecutar:

```
btAdd.setOnClickListener {
    contador++
    textViewContador.setText(contador.toString())
}
```

En este caso, se ha incrementado el contador y se ha actualizado la vista. Además, dado que únicamente pasamos esta lambda como argumento, podemos eliminar los paréntesis para que este controlador quede del modo siguiente:

```
btAdd.setOnClickListener {
    contador++
    textViewContador.setText(contador.toString())
}
```

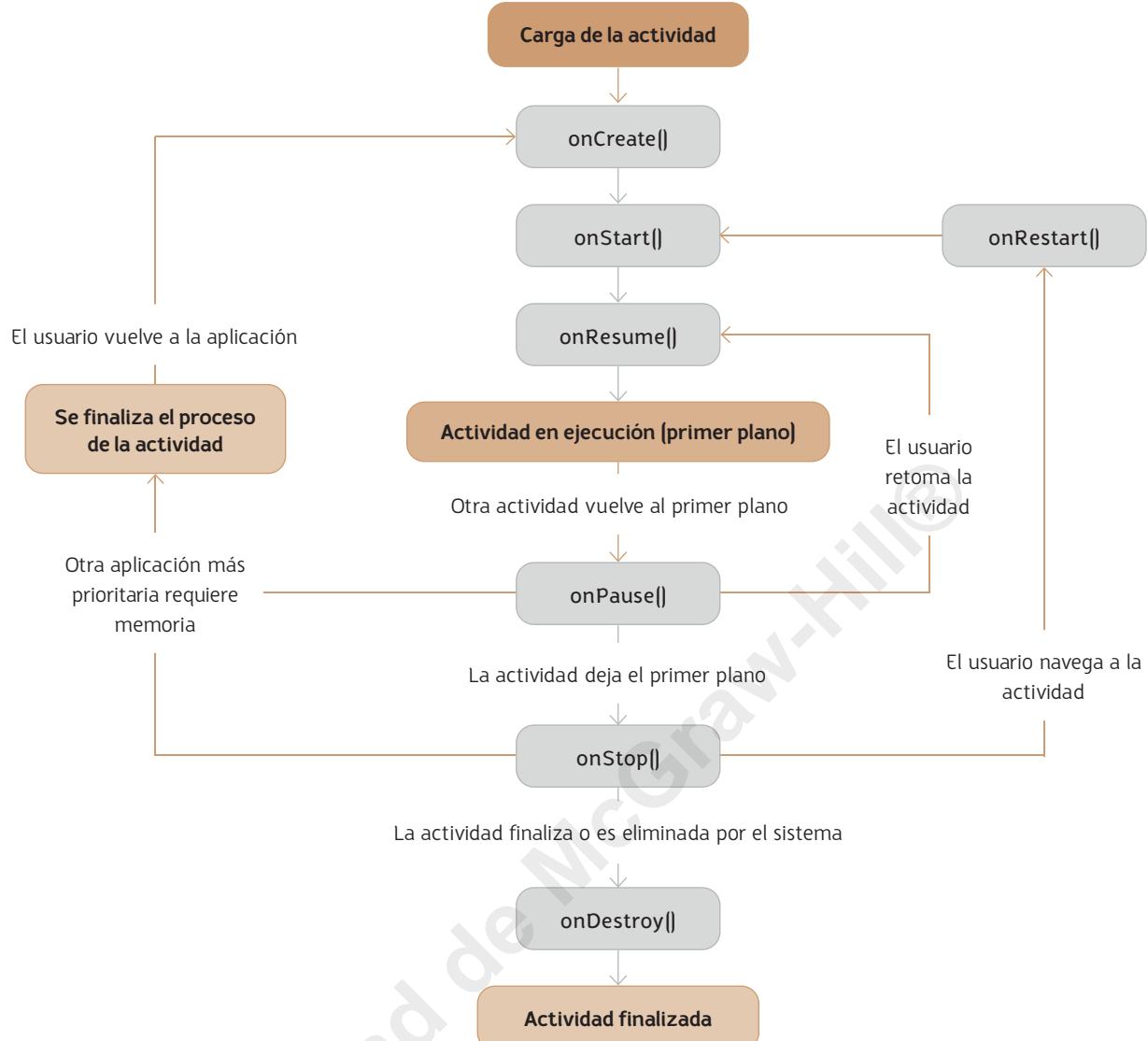
5.2. El ciclo de vida de las actividades

Las actividades y la forma en que estas se inician y relacionan son parte fundamental del modelo de aplicación de Android.

Una aplicación Android puede componerse de varias actividades, y, a pesar de que estas trabajan conjuntamente para dar una experiencia de usuario coherente, en realidad tienen poca relación entre ellas, lo que facilita la invocación de actividades entre diferentes aplicaciones.

Las actividades, a lo largo de su vida útil, pasan por diferentes estados. El usuario de la aplicación puede abrirla, cerrarla, abrir otra aplicación... En estas transiciones entre estados se disparan ciertos eventos, que podemos capturar y gestionar mediante funciones de *callback*.

De forma simplificada, podemos representar este proceso del modo siguiente:



Fuente: Android Open Source Project, imagen utilizada conforme a los términos de la licencia Creative Commons 2.5.
Attribution: <https://developer.android.com/guide/components/activities/activity-lifecycle>

Los *callbacks* que podemos implementar en la clase *Activity* para capturar estos eventos del ciclo de vida son los siguientes:

- **onCreate()**: Se activa en la creación de la actividad y se usa para inicializar sus componentes, enlazar las vistas o vincular los datos. En este método es donde generalmente utilizamos `setContentView()` para establecer el *layout* de la interfaz.
 - **onStart()**: Se activa después del callback `onCreate()`, cuando ya se ha iniciado la actividad y esta pasa a primer plano, volviéndose visible para el usuario.
 - **onResume()**: Se invoca inmediatamente después de `OnStart()` o cuando la actividad estaba pausada y vuelve al primer plano, permitiendo, de nuevo, su interacción con el usuario.
 - **onPause()**: Se invoca cuando la actividad pierde el foco y pasa al estado de pausada (por ejemplo, se ha pulsado el botón *Atrás* o *Recientes*). Cuando el sistema invoca `onPause()`, la actividad todavía es parcialmente visible. Las actividades pausadas pueden continuar actualizando la interfaz si el usuario espera que esto pase (como, por ejemplo, actualizar un mapa de navegación). Este callback no se ha de utilizar para guardar datos ni de la aplicación ni del usuario, hacer llamadas de red o realizar transacciones sobre bases de datos.
 - **onStop()**: Se invoca cuando la actividad ya no es visible para el usuario, bien porque se esté eliminando esta, bien porque se inicie una actividad nueva, bien porque se reanude otra que la cubra.

- **onRestart()**: Se invoca cuando una actividad que estaba detenida pasa de nuevo a estar en primer plano. Este *callback* restaura el estado de la actividad en el momento en que esta se detuvo. Después de este *callback* se invoca siempre *onStart()*.
- **onDestroy()**: Se invoca antes de que se elimine la actividad, y se usa para garantizar que sus recursos se liberan con ella y el proceso que la contiene se elimine.

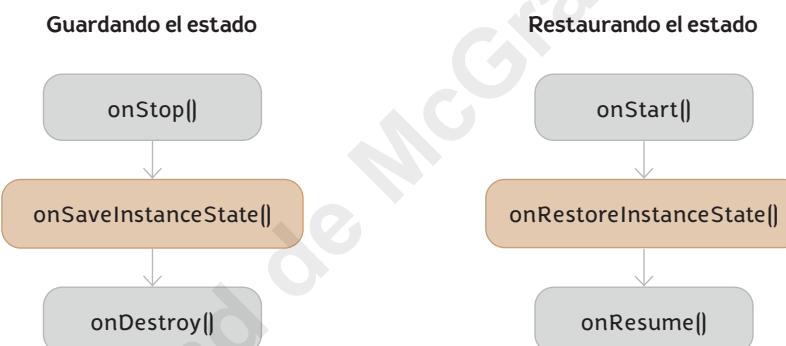
Conocer este ciclo de vida e implementar correctamente los diferentes *callbacks* es de vital importancia para garantizar que el comportamiento de las actividades sea el que se espera y evitar que:

- La aplicación se interrumpa abruptamente cuando el usuario recibe una llamada telefónica o cambia de aplicación.
- Se consuman muchos recursos del sistema cuando no se están utilizando.
- Se pierda el estado de la aplicación cuando el usuario sale de la actividad y vuelve después, o cuando la pantalla cambia de orientación.

MANTENIENDO EL ESTADO DE LAS ACTIVIDADES

Además de parar la actividad, algunas acciones del usuario provocan que esta se destruya, finalizando su ciclo de vida y forzando que se tenga que volver a crear al volver a ella. Casos como hacer *tap* en el botón de atrás o cambios de orientación provocan esta reacción y, con ella, una pérdida en el estado de la actividad.

Para mantener este estado utilizamos los *callback* *onSaveInstanceState* y *onRestoreInstanceState*, heredados de *Activity* y que se invocan antes de destruir la actividad y antes de devolverla al primer plano, respectivamente.



Para guardar el estado se utiliza un objeto de tipo *Bundle* que almacena pares clave-valor. Este tipo *Bundle* se corresponde también con el que se utiliza en el argumento *savedInstanceState* para inicializar la actividad en el método *onCreate*. De hecho, también podríamos restaurar el estado en el método *onCreate*, aunque es más apropiado hacerlo en el método *onRestoreInstanceState*.

Los objetos de tipo *Bundle* admiten varios métodos para almacenar y extraer según el tipo de datos que vamos a utilizar, como *putInt/getInt*, *putString/getString*, etcétera.

Por ejemplo, para guardar un valor entero al cerrar la actividad y restaurarlo al volver a activarla, podemos utilizar el código siguiente:

```
override fun onSaveInstanceState(estado: Bundle) {
    super.onSaveInstanceState(estado)
    // Código para guardar el estado
    estado.putInt("CLAVE", valor)
}

override fun onRestoreInstanceState(estado: Bundle) {
    super.onRestoreInstanceState(estado)
    // Código para guardar el estado
    valor=estado.getInt("CLAVE")
}
```

Actualmente se utilizan también otros tipos de objetos, como el *ViewModel*, que permiten mantener el estado de una actividad de forma independiente a su representación y que esta se actualice de forma reactiva. Estudiaremos estos modelos más adelante.

5.3. Depuración y Logs

Para depurar una aplicación en Android Studio podemos, o bien utilizar las herramientas de depuración del sistema, o bien utilizar el sistema de Logs.

A. HERRAMIENTAS DE DEPURACIÓN DE ANDROID STUDIO

Android Studio nos permite establecer puntos de ruptura (*breakpoints*) en el código fuente haciendo clic en la barra gris de la izquierda del código. Una vez establecido el punto de ruptura, podemos ejecutar esta en modo depuración mediante el ícono correspondiente o el menú *Run > Debug 'App'* (*Shift + F10*).

En el modo depuración, la aplicación quedará suspendida cuando alcance el punto de ruptura y podremos navegar por el código del mismo modo que en otros entornos, mediante las opciones de ejecución paso a paso (*Step Over* o *F8*), *Step Into* (*F9*), etcétera.

B. DEPURACIÓN CON LOGCAT

El fichero Logcat es utilizado por Android para registrar diferentes mensajes del sistema, ya sean errores, avisos u otro tipo de información. Android nos ofrece, a través de la clase *Log*, una API para escribir información en dicho fichero.

Para consultar en tiempo real esta información, Android Studio nos ofrece la ventana de herramienta *Logcat*, donde se muestra el contenido de este fichero y cómo va evolucionando. Esta es una herramienta bastante potente y permite crear filtros de búsqueda, establecer niveles de prioridad, mostrar solamente mensajes de la app o buscar en el registro.

Cuando nuestra app genera una excepción, esta ventana muestra un mensaje y justo después el seguimiento de la pila (*Stack Trace*).

Aunque a partir de la versión 2.2 de Android Studio los mensajes de nuestra aplicación también se muestran en la ventana *Run*, es interesante seguir utilizando *Logcat*, puesto que ofrece más posibilidades en cuanto a la gestión de los mensajes.

Para acceder a la ventana *Logcat* podemos hacerlo, bien a través del menú *View > Tool Windows > Logcat*, bien en la pestaña *Logcat* en la barra de ventanas de herramientas, en la parte inferior.

Para escribir mensajes en el fichero *Logcat* la clase *Log* nos ofrece varios métodos, según el tipo de mensaje que deseemos escribir. De mayor a menor prioridad, estos métodos son los siguientes:

Método	Tipo de mensaje
<code>Log.e(String, String)</code>	Mensaje de error
<code>Log.w(String, String)</code>	Mensaje de advertencia (Warning)
<code>Log.i(String, String)</code>	Mensaje de información
<code>Log.d(String, String)</code>	Mensaje de depuración
<code>Log.v(String, String)</code>	Salida detallada del registro

Como vemos, todos estos métodos necesitan dos argumentos. El primero es un *hashtag* o etiqueta que tendrá el mensaje para facilitar su filtrado, y el segundo será ya el texto del mensaje. Cuando se trata de mensajes del sistema, el *hashtag* es un String corto que indica el componente desde el que se ha originado el mensaje.

Es una práctica habitual en nuestras clases hacer uso de una constante *TAG* para utilizarla como *hashtag*.



Unidad 2.

Desarrollo de aplicaciones Android.

Interfaz de usuario

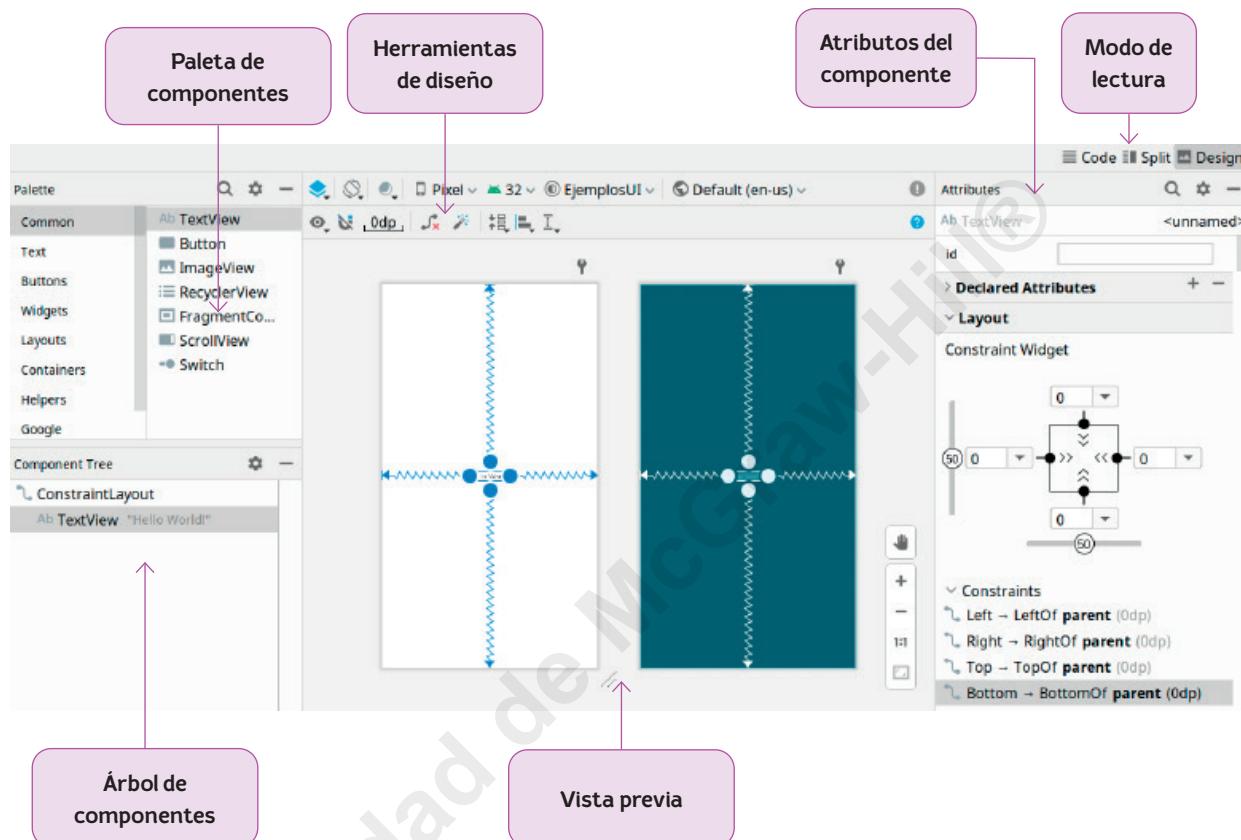


1. Diseñando interfaces de usuario en Android

1.1. El editor de layouts de Android Studio

Para la edición de los ficheros XML que describen interfaces de usuario, Android Studio cuenta con un editor de layouts, al que accedemos directamente cuando editamos uno de estos ficheros.

El editor de layouts permite organizar, crear y modificar de forma gráfica componentes y propiedades de nuestra interfaz de usuario. El aspecto general del editor es el siguiente:



Los diferentes elementos de que consta esta ventana son los siguientes:

- La paleta de componentes, con los diferentes componentes que vamos a utilizar.
- El árbol de componentes, con los diferentes componentes de la interfaz organizados de forma jerárquica.
- El editor de diseño, que muestra el lienzo sobre el que vamos a ir organizando nuestros componentes. Como vemos, muestra dos vistas, la de diseño, y el blueprint o plano técnico. En la parte inferior derecha de este disponemos de controles para el zoom y el desplazamiento lateral.
- Las herramientas de diseño, que permiten ajustar ciertos parámetros de diseño, como la alineación y los márgenes.
- La ventana de atributos del componente, con la que podemos modificar las propiedades de los componentes, como el identificador, el diseño o aspecto, o los atributos según el tipo de objeto, como, por ejemplo, el contenido de un texto.
- El modo de lectura nos permitirá tener diferentes vistas de nuestra interfaz, pudiendo alternar entre el modo de diseño, el código XML o un modo mixto llamado Split.

Dispones de más información en el artículo *Cómo compilar una IU con el editor de diseño* de la documentación oficial de Android, y que puedes consultar en la sección *Material de trabajo/Enlaces*.

Vamos a centrarnos ahora en dos de los aspectos más interesantes de este editor. Por una parte, la paleta de componentes, y, por otra, cómo organizar estos mediante el uso de layouts.

1.2. Diseño de layouts

Los layouts o componentes de diseño son vistas especiales que ayudan a organizar los diferentes elementos de la interfaz. Con la evolución de Android, estos diseños han ido cambiando, pero podemos destacar tres aproximaciones en la forma de organizar los diseños:

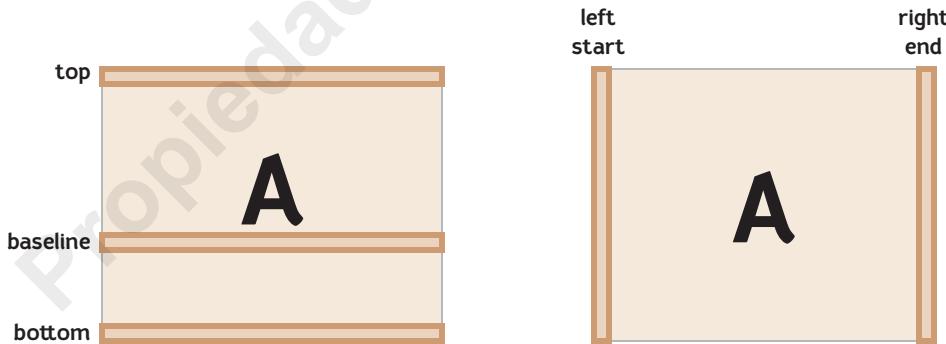
- **Diseño mediante LinearLayouts:** Utiliza contenedores donde se pueden añadir elementos que se ajustan bien en vertical o en horizontal. Para crear interfaces que combinen diseños horizontales y verticales, se utilizan contenedores anidados en otros, lo que provoca que el renderizado de la interfaz no sea óptimo.
- **Diseño mediante RelativeLayouts:** Las posiciones de los diferentes widgets o vistas se describen en referencia a otras vistas, estableciendo una relación padre-hijo y dando lugar a una estructura jerárquica. Esta aproximación ya ha quedado obsoleta.
- **Diseño mediante ConstraintLayout:** Permite gestionar diseños complejos de forma sencilla y lineal utilizando constraints o restricciones. Al tratarse de una estructura prácticamente lineal, es más sencilla y rápida de renderizar, ya que no contiene estructuras anidadas. Este es el tipo de diseño que utiliza Android por defecto cuando creamos una actividad. Veamos con más detalle esta aproximación en el apartado siguiente.

A. DISEÑO DE LAYOUTS CON CONSTRAINTLAYOUT

El componente ConstraintLayout se incorpora en las librerías de compatibilidad de Android Jetpack y reemplaza de una forma más eficiente a los diseños RelativeLayout, ya obsoletos.

Mediante ConstraintLayout podemos posicionar los elementos de la interfaz en función de su relación con otros elementos, ya sean otras vistas, el propio layout o líneas guía. El hecho de utilizar estos diseños hace que se genere un código XML más lineal, con pocos elementos anidados. El diseñador de Android nos prestará mucha ayuda para trabajar con estos diseños.

Toda vista posee cuatro puntos de anclaje: superior, inferior, izquierda y derecha. Teniendo esto en cuenta, podemos definir una constraint o restricción como la relación de cada uno de estos puntos de anclaje con otros elementos, con su contenedor o con líneas guía. Además, también es posible utilizar el Baseline, o línea de la base del texto, para que dos elementos queden alineados verticalmente en función de la línea del texto que contienen en lugar de la vista. Veámoslo gráficamente:



Observamos que para especificar los puntos de anclaje de los lados podemos usar start y end, que suele ser lo más habitual, igual que left y right. Esto se debe a que existen sistemas de escritura, como el árabe, que tienen una direccionalidad de derecha a izquierda, de modo que los puntos de anclaje start y end están intercambiados. En definitiva, la diferencia entre start y end respecto a left y right es que los primeros siguen la direccionalidad de la escritura, y los segundos toman los valores izquierda y derecha directamente.

Así pues, las diferentes restricciones que vamos a poder establecer tendrán en consideración los puntos de anclaje de la propia vista y de otra vista de referencia:

Restricciones en horizontal	
Descripción	Nombre de la restricción
Puntos de anclaje <i>Left</i> y <i>Right</i> de la vista con puntos de anclaje <i>Left</i> y <i>Right</i> de la vista de referencia	<code>layout_constraintLeft_toLeftOf</code> <code>layout_constraintLeft_toRightOf</code> <code>layout_constraintRight_toLeftOf</code> <code>layout_constraintRight_toRightOf</code>
Puntos de anclaje <i>Start</i> y <i>End</i> de la vista con puntos de anclaje <i>Start</i> y <i>End</i> de la vista de referencia	<code>layout_constraintStart_toEndOf</code> <code>layout_constraintStart_toStartOf</code> <code>layout_constraintEnd_toStartOf</code> <code>layout_constraintEnd_toEndOf</code>

Restricciones en vertical	
Descripción	Nombre de la restricción
Puntos de anclaje <i>Top</i> y <i>Bottom</i> de la vista con puntos de anclaje <i>Top</i> y <i>Bottom</i> de la vista de referencia	<code>layout_constraintTop_toTopOf</code> <code>layout_constraintTop_toBottomOf</code> <code>layout_constraintBottom_toTopOf</code> <code>layout_constraintBottom_toBottomOf</code>
Línea de la base de texto de la vista con la línea de base del texto de la vista de referencia.	<code>layout_constraintBaseline_toBaselineOf</code>

Además, para crear restricciones hay que tener en cuenta algunas reglas:

- Cada vista necesitará por lo menos una restricción horizontal y otra vertical. Si utilizamos dos verticales u horizontales, el sistema ajusta la vista automáticamente a ambas.
- Las restricciones, como hemos visto en tablas anteriores, no pueden combinar cualquier tipo de punto de anclaje, sino que deben ser de su mismo tipo: Top-Bottom con Top-Bottom, Start-End con Start-End, etc.
- Solamente puede salir una restricción de un punto de anclaje (no puede estar anclado a varios elementos), pero sí pueden entrar varias restricciones (es decir, utilizarse como referencia).

El código XML resultante para un diseño ConstraintLayout será similar al siguiente:

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

Como vemos, la etiqueta que se utiliza para el ConstraintLayout pertenece a androidx (Jetpack), y el ancho y el alto (`layout_width` y `layout_height`) del componente abarcan todo el contenedor padre (`match_parent`), es decir, toda la ventana de la actividad.

En el artículo *Cómo crear una IU responsive con ConstraintLayout*, que está referenciado en la sección Enlaces web, dispones de más información acerca del Constraint Layout. Además, dedicaremos el caso práctico de este apartado a crear un diseño de este tipo.

B. CONTENEDORES DE TIPO SCROLLVIEW

Cuando el diseño de la interfaz de usuario posee un tamaño superior al de la pantalla del dispositivo, este queda fuera de ella, de modo que algunas vistas serían inaccesibles.

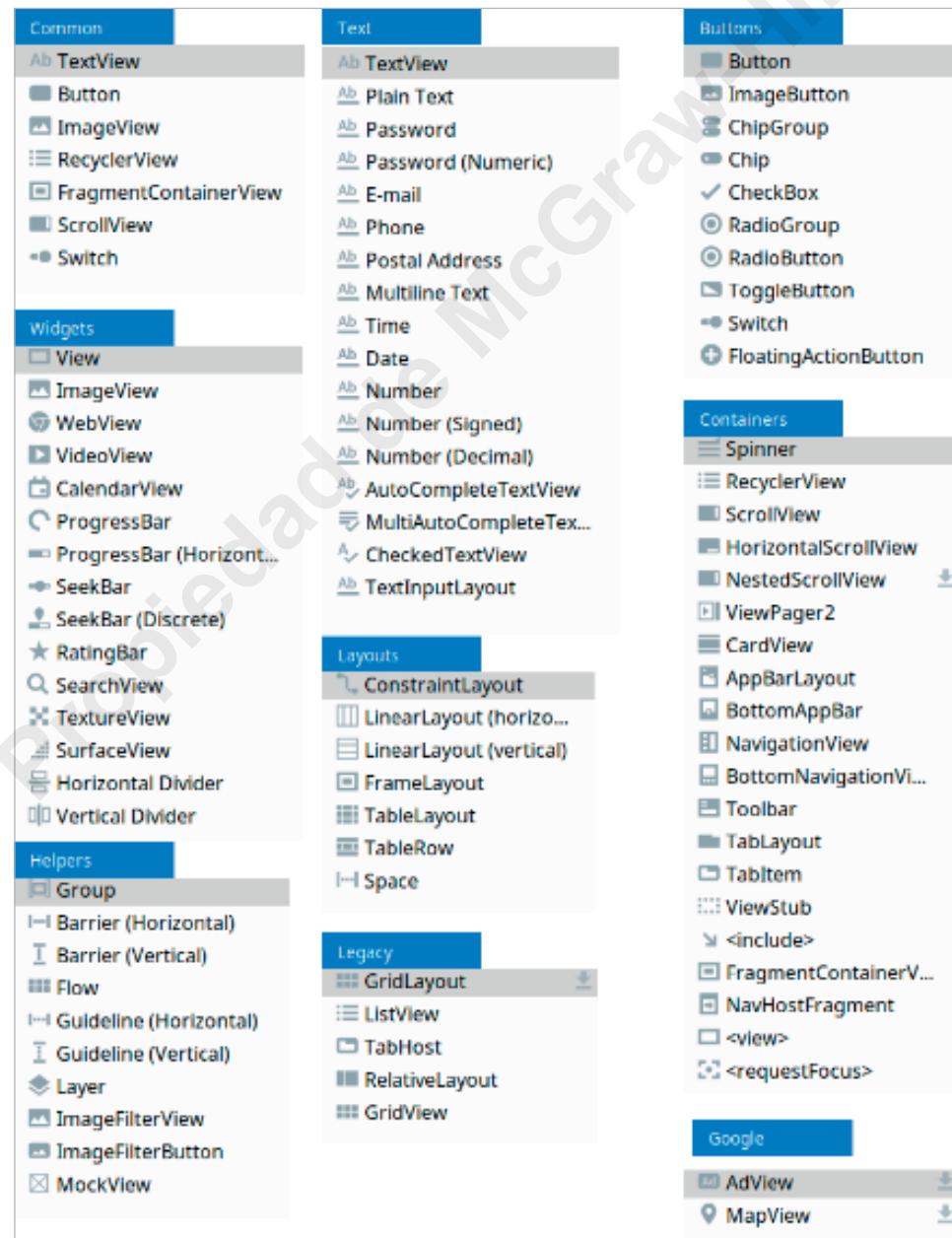
Los contenedores de tipo ScrollView permiten un desplazamiento por la vista que los contenga. Este tipo de componentes únicamente puede tener un hijo directo, y este suele tener otra vista de tipo contenedor, como un ConstraintLayout o cualquiera de los vistos anteriormente.

El componente ScrollView únicamente permite desplazamientos en vertical. Para un desplazamiento en horizontal, utilizaremos HorizontalScrollView.

1.3. Elementos de la interfaz

Todos los elementos de la interfaz de usuario de una aplicación Android son objetos de tipo View o ViewGroup. En general, las vistas u objetos de tipo View se representan en la pantalla del dispositivo y el usuario puede interactuar con ellos. Por su parte, los ViewGroup son objetos que contienen otros objetos de tipo View/ViewGroup. El ConstraintLayout y el ScrollView, por ejemplo, son vistas de tipo ViewGroup.

Android ofrece una gran cantidad de clases de tipo View para los controles más comunes: botones, texto, layouts, etc. Para acceder a estas vistas, Android Studio proporciona la paleta de componentes, que organiza estos en diferentes categorías: elementos comunes, textos, botones, componentes (Widgets), contenedores, diseños (Layouts), asistentes (Helpers), componentes de Google y componentes obsoletos. Los podemos ver en la imagen siguiente:



Como vemos, se trata de una lista de componentes muy extensa y que no vamos a poder tratar en su totalidad. No obstante, veremos aquellas vistas de especial interés, como las de tipo texto, botones, checkboxes o radioButtons, así como algunos textos y visualizaciones de listas.

A. ATRIBUTOS COMUNES A LAS VISTAS

Las vistas poseen atributos comunes que determinan su aspecto y diseño. Estos atributos pueden consultarse y modificarse desde la vista de diseño de interfaces, en la columna Attributes, cuando tenemos una vista seleccionada, o directamente editando el XML.

Algunos de los atributos que usaremos comúnmente serán:

- **El id de la vista**, que servirá para referenciarlo desde el código.
- **Aspectos de diseño**, como layout_width y layout_height, para indicar el ancho y el alto de las vistas. Puede contener el valor wrap_content para ajustar el tamaño al contenido, o valores numéricos para especificar el tamaño exacto. Si se indica un tamaño de 0dp, el tamaño se ajusta automáticamente para satisfacer las restricciones del componente.
- **Restricciones con relación al ConstraintLayout**, para ubicar la vista con relación a otras vistas: layout_constraintLeft_toLeftOf, layout_constraintLeft_toRightOf, etc.

B. VISTAS DE TIPO TEXTO

Las vistas que pueden contener texto dentro de Android son las siguientes:

- **TextView**. Vista que permite mostrar texto al usuario.
- **EditText**. Vista que permite que el usuario introduzca y modifique texto, en diferentes formatos.

Si observamos la paleta de componentes, observaremos que en la categoría Text no se dispone de ningún elemento de tipo EditText, pero en su lugar existen componentes como Plain Text, Password, E-mail, y un largo etcétera.

Todas estas son, en realidad, vistas de tipo EditText que admiten diferentes formatos de entrada. Estos formatos se especifican en su propiedad inputType, la cual determinará tanto el tipo de teclado virtual que se mostrará en pantalla para introducir el texto como el juego de caracteres aceptable y la apariencia del texto.

Por ejemplo, un EditText que vaya a contener un password que únicamente acepte números se expresaría como:

```
<EditText  
    android:id="@+id/Password"  
    android:inputType="text|numberPassword"  
    ...  
/>
```

C. BOTONES

Los botones son vistas que consisten en un texto o un ícono, y que realizan cierta acción cuando el usuario hace clic en ellos. Estos se implementan con la clase Button, que es una subclase de TextView, y la clase ImageButton, que es una subclase de ImageView.

En general, el XML para definir un botón quedaría:

```
<Button  
    android:id="@+id/button"  
    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_toTopOf="parent" />
```

Las propiedades que se incluyen son las siguientes:

- El identificador (id), con valor @+id/button. El prefijo @+id se utiliza para indicar que estamos definiendo un nuevo identificador.
- Las propiedades layout_width y layout_height sirven para indicar el tamaño del botón, con valor wrap_content para que se ajuste al contenido.

La propiedad text contiene el texto del botón. Es aconsejable definir este texto como un recurso de tipo String, dentro del fichero res/values/strings.xml, en la forma:

```
<string name="button_texto">Texto del botón</string>
```

Además, si queremos que incluya un ícono, añadiremos la propiedad drawableLeft:

android:drawableLeft="@drawable/button_icon": con esto estamos añadiendo el recurso de imagen button_icon, ubicado en la carpeta de recursos res/drawable.

Por otro lado, si queremos utilizar un **botón de tipo imagen**, deberíamos usar la etiqueta **<ImageButton>**, incorporando el atributo **src**, con el recurso de la imagen (o **srcCompat** si es una imagen vectorial), y la descripción, con el atributo **contentDescription**.

Respondiendo a eventos

Si queremos que un botón responda a una acción del usuario, podemos utilizar la propiedad android:onClick y especificar un método de nuestra clase asociada:

```
<Button ...  
    android:onClick="metodoEnClase" />
```

Y en nuestra clase definimos el método como público, sin valor de retorno y con la vista (botón) como argumento.

```
fun metodoEnClase(view: View) {  
    ...  
}
```

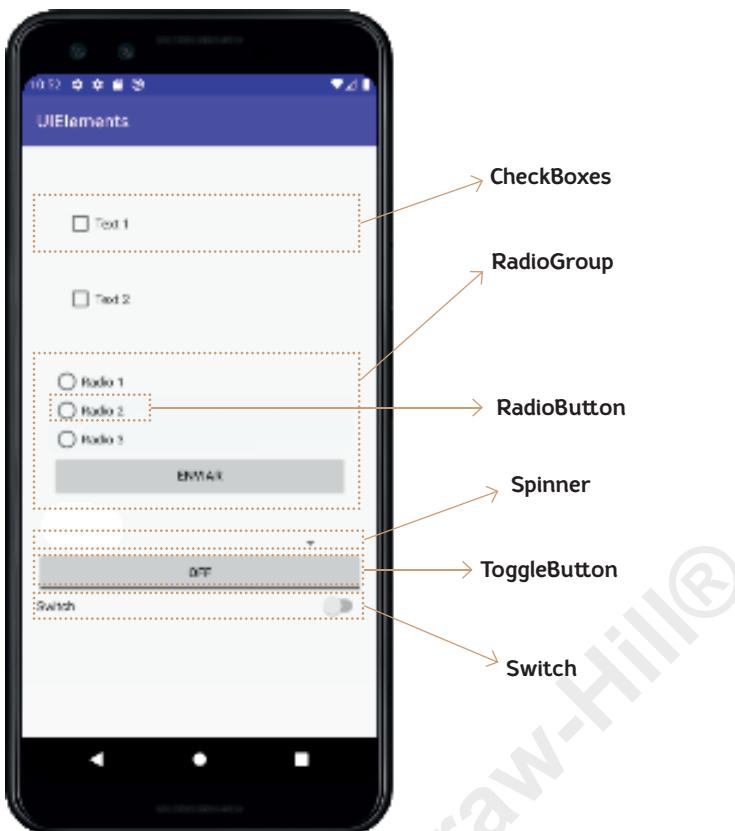
Por otro lado, la opción más habitual para asociar eventos a botones, como ya sabemos, es hacer uso de un receptor de eventos OnClickListener:

```
var boton=findViewById<Button>(R.id.button);  
boton.setOnClickListener {  
    // Callback como respuesta al clic  
}
```

D.CHECKBOXES, RADIOPBUTTONS, TOGGLE BUTTONS Y SPINNERS

Tanto las casillas de verificación (checkboxes) como los botones de selección (radiobuttons) permiten seleccionar opciones entre un conjunto de opciones. La diferencia entre ambas es que las casillas de verificación permiten seleccionar varias opciones, mientras que los botones de selección solamente permiten seleccionar un elemento.

Otros elementos interesantes que trataremos en este apartado son los Spinners que también servirán para seleccionar elementos de una lista, y los ToggleButtons y Switch, que pueden tener estados de activado/desactivado.



Casillas de verificación o Check Boxes

Para crear una casilla de verificación, añadiremos un objeto de tipo checkBox a nuestro diseño. Cada checkbox se gestiona de forma independiente, por lo que registraremos un receptor de eventos para cada uno.

El evento se disparará cuando hacemos clic en un CheckBox es onClick, y puede asociarse directamente a una acción mediante la propiedad android:onClick en el XML o mediante un receptor de eventos [eventListener] capturando el evento onClick. El funcionamiento es el mismo que hemos visto para los botones comunes. Una vez dentro del método que gestiona el evento, podemos acceder a su valor con la propiedad isChecked.

Por otro lado, si desde el propio código queremos cambiar el valor del checkbox, podemos hacerlo con los métodos setChecked(boolean), para indicar un valor concreto, o con el método toggle(), para cambiar su estado.

Botones de selección o RadioButtons

Los botones de selección permiten escoger una opción de entre un conjunto de opciones mutuamente excluyentes. Estos botones se implementan con la vista RadioButton. Para indicar que varios RadioButtons son excluyentes, estos deberán estar dentro de un mismo componente de tipo RadioGroup.

La gestión de eventos es exactamente igual que para los CheckBoxes y para los botones. Cuando hacemos clic en el botón, se dispara el evento onClick, que podemos gestionar directamente en la definición de la etiqueta RadioButton o con un receptor de eventos desde el código. Una vez dentro del método que gestiona el evento, también tenemos acceso a la propiedad isChecked para comprobar que el elemento está seleccionado o no lo está.

Debemos tener en cuenta que el contenedor RadioGroup es una subclase de LinearLayout con orientación vertical, de forma que los RadioButtons que vayamos añadiendo dentro se irán ordenando con esta orientación.

Algunas consideraciones finales:

- Si queremos inicializar un RadioButton o modificar su estado desde el código, podemos utilizar los métodos setChecked(boolean) o toggle().
- Podemos capturar cuándo se modifica la selección en un RadioGroup con el método onCheckedChanged.

Spinners

Los spinners ofrecen una forma rápida de elegir un valor de un conjunto. A diferencia de los RadioButtons, las opciones disponibles no están visibles, sino que se despliegan al hacer clic sobre el spinner, que en todo momento muestra el valor seleccionado. Para añadir un spinner desde el diseñador de interfaces, hay que buscarlo en la paleta *Containers > Spinner*.

El XML que corresponde a un Spinner sería el siguiente:

```
<Spinner  
    android:id="@+id/días_semana"  
    ...  
/>
```

Por otra parte, si un Spinner alberga un conjunto de valores predeterminados, podemos especificarlo como un vector de elementos, en un archivo de recursos en formato XML, y referenciarlos desde el propio Spinner.

Por poner un ejemplo, podemos disponer del archivo XML res/values/estaciones.xml con el contenido siguiente:

```
<?xml version="1.0" encoding="utf-8"?>  
<resources>  
    <string-array name="estaciones">  
        <item>Primavera</item>  
        <item>Verano</item>  
        <item>Otoño</item>  
        <item>Invierno</item>  
    </string-array>  
</resources>
```

De modo que haríamos referencia a él en el fichero de layout dentro de la etiqueta Spinner, con el atributo android:entries:

```
<Spinner  
    android:id="@+id/SpinnesEstaciones"  
    android:entries="@array/estaciones"  
    ...  
/>
```

Finalmente, para acceder a él, usaríamos el método getSelectedItem() del Spinner, que en Kotlin se expresa directamente como selectedItem.

Toggle Buttons

Los Toggle Buttons son botones que permiten alternar entre dos estados. El objeto que gestiona estos controles es ToggleButton. A partir de Android 4.0 (API level 14), se introduce otro tipo de Toggle Button denominado switch, controlado por el objeto Switch y SwitchCompat en las librerías de compatibilidad.

Para modificar su estado, podemos usar los métodos setChecked() o toggle(), y para consultarla podemos usar el método isChecked.

Veamos cómo se reflejan estos elementos en el XML:

- Para el Toggle Button:

```
<ToggleButton  
    android:id="@+id/toggleButton"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="ToggleButton" />
```

- Para el Switch:

```
<Switch  
    android:id="@+id/switch1"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="Switch" />
```

Para detectar la activación de cualquiera de estos botones, usaremos el método setOnCheckedChangeListener() para asignar el callback.

Propiedad de McGraw-Hill®

2. View Binding: Vinculación de vistas

2.1. Vinculación de vistas

Los diferentes elementos o vistas que conforman la interfaz de usuario en los ficheros XML se deben poder referenciar de algún modo desde nuestro código.

Hasta ahora, hemos estado utilizando el método `findViewById` para localizar vistas en el XML a partir del nombre del recurso que deseamos.

Este proceso es sencillo y práctico cuando disponemos de pocas vistas, pero se vuelve complicado y tedioso cuando la cantidad de vistas empieza a crecer.

Con la versión 3.6 de Android Studio (Canary) se introdujo el mecanismo del View Binding o vinculación de vistas, que permite escribir de una forma más sencilla el código que interactúa con las vistas.

La vinculación de vistas consiste en enlazar los diferentes elementos de la interfaz de usuario especificada a los ficheros de diseño XML con nuestro código, mediante la generación de una clase de vinculación para cada archivo XML de diseño presente en el módulo.

De este modo, las instancias de esta clase de vinculación contendrán referencias directas a las vistas que tienen un ID en el Layout, y no será necesario el uso de `findViewById`.

2.2. Activación del View Binding

Para activar el View Binding en un módulo, debemos especificarlo de forma explícita en el fichero de construcción `build.gradle` del módulo en cuestión. Para ello, añadimos a la sección de Android el código siguiente:

```
android {  
    ...  
    buildFeatures {  
        viewBinding true  
    }  
}
```

En versiones de Android Studio con el plugin de Gradle anteriores a la 4.0.x se usa del modo siguiente:

```
android {  
    ...  
    viewBinding {  
        enabled = true  
    }  
}
```

Cuando añadimos este código, Android Studio detecta los cambios en el fichero de construcción y pide sincronizarse.

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly. [Sync Now](#) [Ignore these changes](#)

En el momento en que sincronizamos se crea la clase de vinculación asociada al layout.

EXCLUSIÓN DE LAYOUTS DEL VIEWBINDING

Si no deseamos que se genere la clase de vinculación para un layout concreto, podemos indicarlo en la propia declaración del layout, mediante el atributo tools:viewBindingIgnore="true": Por ejemplo:

```
<androidx.constraintlayout.widget.ConstraintLayout>
    ...
    tools:viewBindingIgnore="true"/>
    ...
</androidx.constraintlayout.widget.ConstraintLayout>
```

2.3. Uso del View Binding

Una vez se ha habilitado el viewbinding y sincronizado el proyecto, podemos importar la nueva clase de vinculación desde nuestro código.

Si, por ejemplo, en nuestro paquete com.mgh.pmdm.ejemplosui tenemos definida la actividad principal en el layout activity_main.xml, para importar los bindings desde el fichero de la clase principal MainActivity.kt deberemos importar la clase de vinculación del modo siguiente:

```
import com.mgh.pmdm.ejemplosui.databinding.ActivityMainBinding
```

Con esto tendremos disponible en nuestra actividad la clase ActivityMainBinding, aunque de momento solo es eso, una clase que vamos a tener que instanciar para poder usarla. Para ello, dentro del método onCreate() de la vista, usaremos el método estático inflate, que nos proporciona la clase de vinculación y al que debemos proporcionarle un argumento de tipo LayoutInflator:

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

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

Hagamos algunas observaciones respecto a este código:

- Hemos definido binding como una propiedad de tipo ActivityMainBinding, de manera que mediante esta tengamos accesible la clase de vinculación en toda nuestra clase MainActivity.
- Esta propiedad se ha definido también como lateinit (Late Initialization), lo que indica que la inicialización de esta variable no se hará en el constructor, sino posteriormente (en nuestro caso, en el método onCreate). Con esto evitamos problemas con los valores nulos.
- El método inflate es un método estático de la clase ActivityMainBinding, y recibe como argumento un objeto de tipo LayoutInflator. Recuerda que, de momento, solamente has importado la clase ActivityMainBinding, pero esta no contiene todavía ninguna referencia a las vistas del layout. La clase LayoutInflater ayuda a instanciar el archivo XML del layout con las vistas correspondientes. Esta clase no debe usarse de forma directa, y en su lugar se usa el método getLayoutInflater() de la clase Activity. Dado que en Kotlin accedemos a los getters y los setters como si se tratase directamente de atributos, aquí toma la forma de layoutInflater. Informalmente, se dice que con esto estamos inflando la vista.

Ya solamente nos queda invocar al método `setContentView()` para activar la vista en la pantalla. Ahora bien, en lugar de proporcionarle como argumento el recurso `R.layout.activity_main`, vamos a proporcionarle el elemento raíz de la vista que hemos inflado. Para acceder a este elemento raíz, usaríamos el método `getRoot()`, o, mejor, la propiedad `root` haciendo uso de la sintaxis de propiedades para los getters de Kotlin. Así pues, la clase quedará:

```
class MainActivity : AppCompatActivity() {  
  
    private lateinit var binding: ActivityMainBinding  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
  
        super.onCreate(savedInstanceState)  
        binding = ActivityMainBinding.inflate(layoutInflater)  
        val view = binding.root  
        setContentView(view)  
    }  
}
```

Con esto, a través del objeto `binding` vamos a poder acceder a las diferentes vistas de la interfaz como si se tratase de propiedades de este objeto, cuyo nombre corresponde al del ID de la vista. Por ejemplo, si tenemos un `TextView` con el ID `NombreUsuario`, podremos acceder a este mediante `binding.NombreUsuario`.

3. Notificaciones y diálogos

3.1. Toasts

Los toast sirven para proporcionar información al usuario, de manera simple, sobre una operación en una pequeña ventana emergente. Esta ventana solo ocupará el espacio requerido para mostrar el mensaje y no bloqueará la actividad actual. Se trata de avisos que desaparecen automáticamente al cabo de un tiempo determinado.

Para mostrar un toast usaremos el método estático `makeText()` de la clase `Toast`, que devolverá el objeto de este tipo. Este método requiere de tres parámetros: el contexto de la aplicación, el mensaje de texto y el tiempo durante el cual el aviso será visible. Una vez hemos inicializado el `Toast`, lo mostramos con `show()`.

En general, este código quedaría del modo siguiente:

```
// Definimos el texto a mostrar
val texto = "Hola, esto es un Toast"

// Definimos la duración
val duracion = Toast.LENGTH_SHORT

// Creamos el Toast
val toast = Toast.makeText(applicationContext, texto, duracion)

// Y lo mostramos
toast.show()
```

Observamos que el primer argumento del método `makeText` es `applicationContext`. Vamos a hacer un pequeño inciso para ver a qué nos referimos cuando hablamos de contexto.

IMPORTANTE

En la documentación de Android se define el `Context` como una clase abstracta que proporciona una interfaz con información global sobre el entorno de nuestra aplicación.

En otras palabras, cuando hablamos de un contexto en el ámbito del desarrollo en Android, nos referimos a eso mismo, al propio contexto en el que estamos presentes, y nos permite conocer el estado de la aplicación e información sobre esta. Además, nos proporciona acceso a ciertos recursos, bases de datos y preferencias, así como al uso de Intents para invocar otras actividades.

Fundamentalmente, existen dos tipos de contexto: el contexto de la aplicación, que nos contextualiza en la propia aplicación, y el contexto de la actividad, que nos contextualiza en la actividad y va ligado al ciclo de vida de esta.

El argumento `applicationContext` que proporcionamos al método `makeText` hace referencia, pues, al contexto de la aplicación, que el `Toast` necesitará conocer para saber dónde mostrarse. Esta referencia realmente se obtiene a través del método `getApplicationContext()`, aunque Kotlin nos da acceso a él como si se tratase de una propiedad.

En principio, para mostrar el `Toast` en la actividad actual podemos utilizar tanto este atributo `applicationContext` como directamente `this`.

Además, también podemos simplificar el código anterior y encadenar el `show` con la creación del `toast`, lo que nos ahorra la variable:

```
Toast.makeText(applicationContext, texto, duration).show()
```

PERSONALIZACIONES

La clase Toast admite algunas opciones de personalización. Por una parte, como ya hemos visto, podemos ajustar el tiempo en que la notificación está presente. En el ejemplo hemos utilizado la constante proporcionada por la clase Toast LENGTH_SHORT. Si deseamos que esta se muestre más tiempo, podemos utilizar la constante LENGTH_LONG.

Por otra parte, si deseamos que el toast se muestre en la parte superior, podemos utilizar el método:

```
setGravity(  
    constant_Gravity:int,  
    desplazamiento_x: int,  
    desplazamiento_y:int)
```

Por ejemplo:

```
toast.setGravity(Gravity.TOP, 0, 0)
```

Se mostrará el toast en la parte superior y sin desplazamientos adicionales en horizontal o vertical. Existen múltiples constantes para el campo Gravity. Aparte de TOP y BOTTOM, podemos usar CENTER, START o END, entre muchas otras.

3.2. Snackbars

Una alternativa a las notificaciones de tipo Toast cuando nuestra aplicación está en primer plano son las barras de notificaciones, que incluyen más opciones y proporcionan una mejor experiencia al usuario.

Estas barras forman parte de las especificaciones de diseño de Material Design, y proporcionan mensajes breves sobre los procesos de la aplicación en la parte inferior de la pantalla. Al igual que los toasts, no deben interrumpir la experiencia del usuario y, en principio, no requieren de interacción con este para desaparecer. Aun así, estas barras admiten ciertas acciones de forma opcional, como veremos a continuación.

Las barras de notificaciones están compuestas por los tres elementos siguientes:



CREACIÓN DE BARRAS DE NOTIFICACIONES

Para crear una barra de notificación, en primer lugar, debemos importar el componente desde la librería Material Design:

```
import com.google.android.material.snackbar.Snackbar
```

Luego, debemos utilizar el método make del componente Snackbar para su creación. Este método admite varias firmas, como, por ejemplo:

```
Snackbar.make(View vista, CharSequence texto, int duracion)
```

Como vemos, recibe tres parámetros: la vista que toma como contexto para poder mostrarse, el texto del Snack, y la duración o el tiempo que permanece visible. Existen otras formas que reemplazan el texto con un identificador de recurso de tipo String, por ejemplo. La duración se puede expresar mediante constantes como Snackbar.LENGTH_LONG, Snackbar.LENGTH_SHORT, Snackbar.LENGTH_INDEFINITE o cualquier valor numérico en milisegundos.

Para asociar una acción a esta barra se usa el método setAction, que presenta la sintaxis siguiente:

```
public Snackbar setAction (int resId | CharSequence text,  
                           View.OnClickListener listener)
```

El primer parámetro de este método es el texto que se mostrará para la acción, y que puede ser cualquier recurso de tipo cadena o una cadena de texto arbitraria. El segundo parámetro que recibe setAction es el callback (listener), que se ejecutará cuando se haga clic sobre la acción. Además, dado que este callback es el último argumento, Kotlin nos permite simplificarlo expresándolo fuera de los paréntesis.

Y finalmente, utilizaremos el método show() para mostrar el Snackbar.

Veamos un ejemplo concreto teniendo en cuenta todo lo anterior:

```
Snackbar.make(binding.root, "Texto del Snackbar", Snackbar.LENGTH_LONG)
    .setAction("Acción") {
        // Respuesta cuando se hace clic en la acción
    }
    .show()
```

Observamos que la vista que estamos utilizando es la raíz de la clase de vinculación generada en el ViewBinding.

Las barras de notificaciones permiten algunas opciones más acerca de la personalización y los estilos de estas. Disponemos de más información en la documentación oficial de Android y de Material Design para estos componentes.

3.3. Notificaciones

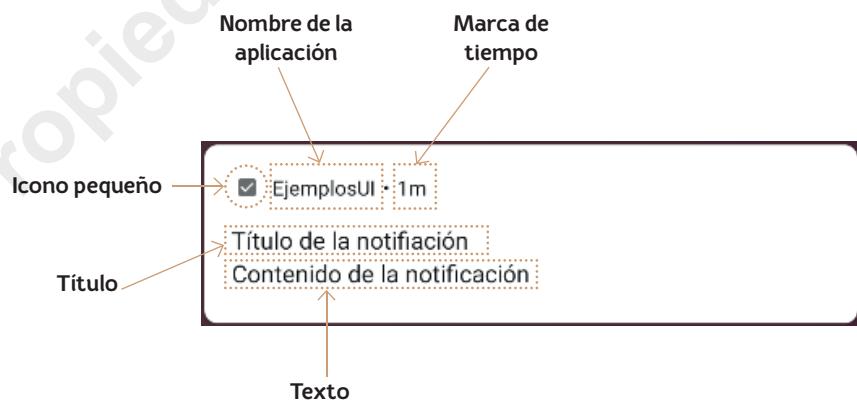
Cuando nuestra aplicación está en segundo plano, pero necesitamos mostrar información al usuario o que este realice alguna acción, usamos notificaciones.

Las notificaciones de Android son mensajes que se muestran fuera de la interfaz de la aplicación y permiten al usuario abrirla o realizar cierta acción directamente desde la notificación.

Como las posibilidades que ofrecen las notificaciones son muy extensas, en este apartado las introduciremos brevemente y aprenderemos a crear notificaciones sencillas que realicen una determinada acción al ser presionadas.

La gestión de las notificaciones se realiza a través de la biblioteca de compatibilidad NotificationCompat y sus subclases, incluidas en Jetpack, así como a través de NotificationManagerCompat.

En su forma más básica (conocida como contraída), una notificación posee los elementos siguientes:



Su diseño viene predeterminado por las plantillas del sistema y, además de lo anterior, puede incluir un ícono más grande (generalmente para mostrar los contactos en aplicaciones de mensajería) o una barra de botones en la parte inferior para realizar otras acciones diferentes de la predeterminada.

A.CREANDO UN CANAL DE NOTIFICACIONES

A partir de Android 8.0, se requiere el registro de un canal de notificaciones de la aplicación en el sistema. Para ello, podemos utilizar el método siguiente:

```
private fun crearCanalNotificaciones() {  
    // Creamos el canal de notificaciones solamente si  
    // nos encontramos en un nivel de API 26+  
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
        // Definimos las propiedades del canal: nombre,  
        // descripción e importancia.  
        val nombre = "Canal de prueba"  
        val descripcion = "Descripción de mi canal"  
        val importancia = NotificationManager.IMPORTANCE_HIGH  
  
        // Creamos el canal con NotificationChannel.  
        // El constructor recibe la cadena con el ID, el nombre  
        // y la importancia.  
        // Una vez creado, utilizaremos la función de ámbito apply sobre él,  
        // mediante la cual asignamos la descripción a dicho canal.  
  
        val canal = NotificationChannel("MI_CANAL", nombre, importancia)  
            .apply {  
                description = descripcion  
            }  
  
        // Y finalmente se registra el canal en el sistema mediante  
        // el servicio de notificaciones del sistema del modo siguiente:  
  
        val notificationManager: NotificationManager =  
            getSystemService(Context.NOTIFICATION_SERVICE)  
                as NotificationManager  
        notificationManager.createNotificationChannel(canal)  
    }  
}
```

El código anterior requerirá que importemos las bibliotecas:

```
import android.app.NotificationChannel  
import android.app.NotificationManager
```

B. CREANDO Y MOSTRANDO NOTIFICACIONES

Con el canal de notificaciones listo, para crear una notificación utilizaremos el método `Builder` de la clase `NotificationCompat` y le proporcionaremos un contexto y el identificador del canal:

```
var builder = NotificationCompat.Builder(applicationContext, CHANNEL_ID)
```

A este builder le podremos incorporar el ícono pequeño mediante el método `setSmallIcon`, el título mediante `setContentTitle`, el texto mediante `setContentText` y la prioridad mediante `setPriority`.

Asimismo, con el método `setContentIntent` le podremos añadir qué acción queremos que se realice cuando se haga clic sobre la notificación, aunque para ello deberemos haber definido un Intent previamente. Este Intent deberemos proporcionarlo al sistema de notificaciones mediante un Intent pendiente (Pending Intent). Los Pending Intents actúan como testigos que les proporcionamos a otras aplicaciones (en este caso, el sistema de notificaciones) para permitirles ejecutar parte de nuestro código.

Si deseamos que la notificación desaparezca cuando la leamos, utilizaremos el método `setAutoCancel()`.

Con todo esto en cuenta, el código quedaría:

```
// Preparamos el ID del canal para la notificación
var CHANNEL_ID = "MI_CANAL"

// Preparamos un Intent explícito para la actividad
val intent = Intent(this, MainActivity::class.java).apply {
    flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
}

// Y el Pending Intent, a partir de este, que le proporcionaremos
// al sistema de notificaciones
val pendingIntent: PendingIntent =
    PendingIntent.getActivity(this, 0, intent, 0)

// Creamos la notificación proporcionándole un contexto y el ID del canal
// y le añadimos todas las propiedades necesarias
var builder = NotificationCompat.Builder(applicationContext, CHANNEL_ID)
    .setSmallIcon(androidx.appcompat.R.drawable.btn_checkbox_checked_mtrl)
    .setContentTitle("Título de la notificación") // Título
    .setContentText("Contenido de la notificación") // Texto
    .setPriority(NotificationCompat.PRIORITY_DEFAULT) // Prioridad
    .setContentIntent(pendingIntent) // Acción en hacer clic
    .setAutoCancel(true) // Para que desaparezca la notificación
```

Como podemos observar, la creación de la notificación y sus propiedades se realizan en la misma instrucción.

El código anterior requerirá que importemos algunas bibliotecas:

```
import android.app.PendingIntent
import android.content.Intent
import androidx.core.app.NotificationCompat
import androidx.core.app.NotificationManagerCompat
```

Finalmente, ya solo nos queda mostrar la notificación mediante el método `notify` de la clase `NotificationManagerCompat`, proporcionándole un ID único para la notificación, así como el resultado del `builder.build()`:

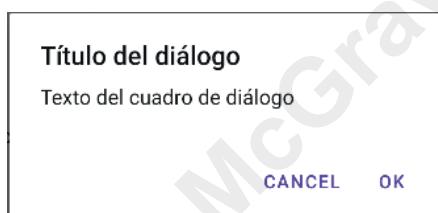
```
val ID_Noticacion=0
with(NotificationManagerCompat.from(applicationContext)) {
    notify(ID_Noticacion, builder.build())
}
```

En este código se usa la función de ámbito `with`. Esta función recibe dos argumentos: el primero es el ámbito o contexto sobre el que se va a actuar. En este caso, este ámbito es el objeto `NotificationManagerCompat` del contexto de nuestra aplicación. Por otro lado, en el ámbito del Notification Manager se invoca la expresión lambda proporcionada como segundo argumento, y que Kotlin permite extraer de los paréntesis. En este caso, esta expresión es la invocación al método `notify`.

3.4. Diálogos

Los diálogos son ventanas pequeñas que sirven para proporcionar avisos al usuario, confirmar acciones o añadir información adicional. Los diálogos no ocupan toda la pantalla, pero hacen que no sea posible la interacción con la pantalla principal hasta que no se realice una determinada acción (se conocen como ventanas modales).

Los diálogos se implementan mediante la clase `Dialog`, de la que deriva la clase `AlertDialog`, que es la que generalmente utilizaremos, y que puede contener un título, un texto, hasta tres botones, una lista de elementos seleccionables o bien un diseño personalizado.



Además, de la clase `AlertDialog` derivan otras clases, como `ProgressDialog`, `DatePickerDialog` y `TimePickerDialog`, para mostrar barras de progreso y selectores de fecha y hora, respectivamente.

Para crear un diálogo vamos a utilizar la clase `Builder`, que es una clase interna de `AlertDialog` y cuyo diseño se corresponde con el patrón de diseño creacional `Builder`. Este patrón permite crear objetos complejos sin necesidad de preocuparnos de detalles en el proceso de construcción.

Así pues, para crear un primer diálogo, haremos:

```
var miDialogo = AlertDialog.Builder(this)
```

Lo único que deberemos proporcionarle como contexto es la propia actividad en que lo definimos.

Para incorporar el título y el texto, utilizaremos los métodos `setTitle` y `setMessage`:

```
miDialogo.setTitle("Título del diálogo")
miDialogo.setMessage("Texto del cuadro de diálogo")
```

Y, además, utilizaremos el método `setCancelable(boolean)` para indicar si el diálogo se puede cerrar desde la parte opaca de detrás del diálogo (`cancelable=true`) o si debemos hacerlo mediante los botones `Ok/Cancel` (`cancelable=false`)

```
miDialogo.setCancelable(false)
```

Para añadir los botones, podemos utilizar los métodos `setPositiveButton` y `setNegativeButton`, a los que proporcionaremos dos argumentos: el primero es la cadena de texto del botón, que puede ser un recurso, y el segundo, una implementación de `DialogInterface.OnClickListener`, con el callback en forma de expresión lambda, que se ejecutará cuando se pulse el botón (evento `onClick()`). Por ejemplo, para añadir el botón `OK`, haríamos:

```
miDialogo.setPositiveButton(android.R.string.ok,
    DialogInterface.OnClickListener {dialog, which ->
    // Acciones cuando se pulsa el botón OK
})
```

Aquí, los parámetros que recibe el callback que gestiona el evento son una referencia al diálogo que ha recibido el evento (dialog) y al elemento sobre el que se ha hecho.

Finalmente, para mostrar el diálogo utilizamos el método show() de este.

Combinando todo lo anterior, el código para crear y mostrar un diálogo que muestre un toast con el botón pulsado podría quedar de forma compacta del modo siguiente:

```
AlertDialog.Builder(this)
    .setTitle("Título del diálogo")
    .setMessage("Texto del cuadro de diálogo")
    .setCancelable(false)
    .setPositiveButton(android.R.string.ok, { dialog, which ->
        Toast.makeText(applicationContext,
            "Clic en Ok", Toast.LENGTH_SHORT).show()
    })
    .setNegativeButton(android.R.string.cancel, { dialog, which ->
        Toast.makeText(applicationContext,
            "Clic al Cancelar", Toast.LENGTH_SHORT).show()
    })
.show()
```

Observamos que hemos concatenado los diferentes métodos en una sola línea y que, además, Kotlin permite simplificar algunas notaciones, como el `DialogInterface.OnClickListener`, de forma que queda un código compacto y bastante más legible.

El código anterior requerirá que importemos algunas librerías más:

```
import androidx.appcompat.app.AlertDialog
import android.content.DialogInterface
```

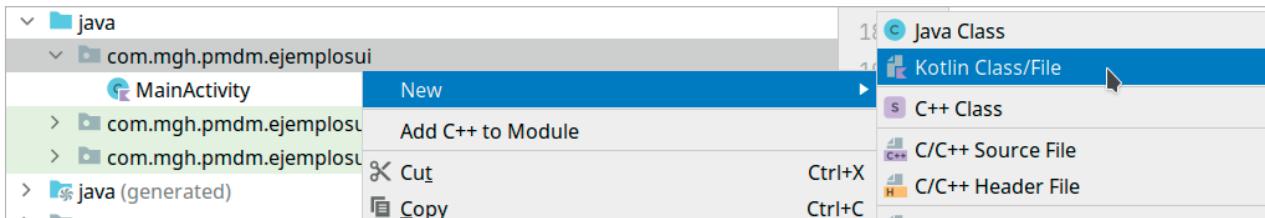
IMPORTANTE

¡Cuidado! Este método es una forma relativamente sencilla de crear un diálogo de alerta. Ahora bien, en el momento en que se redibuja la pantalla, por ejemplo, cuando la giramos, el diálogo desaparece, por lo que tendremos que buscar una manera más práctica para mostrar los diálogos.

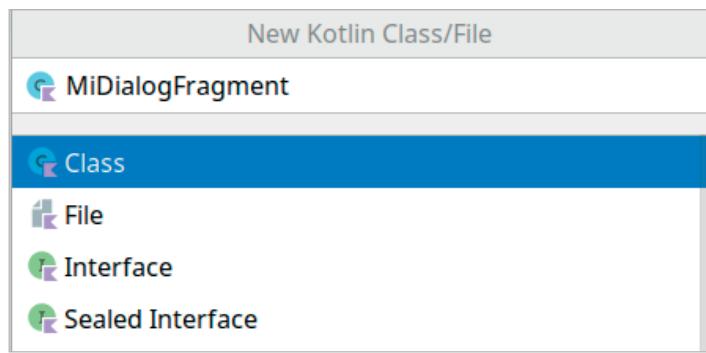
A.DIÁLOGOS CON DIALOGFRAGMENT

La forma habitual de presentar cuadros de diálogo es mediante un contenedor DialogFragment, y con este se proporcionan los controles necesarios para crear y gestionar estos, y se garantiza el control de los eventos del ciclo de vida. De este modo, el diálogo se mantiene cuando el usuario pulsa el botón de volver atrás o gira la pantalla.

Para crear un diálogo de este modo, deberemos crear una nueva clase en Kotlin (botón derecho sobre el proyecto > New > Kotlin Class/File):



Y le asignaremos un nombre, por ejemplo, MiDialogFragment:



Esta nueva clase deberá ser una subclase de DialogFragment. De esta clase, sobrescribiremos el método onCreateDialog.

El código para esta nueva clase será el siguiente:

```
class MiDialogFragment : DialogFragment() {  
    override fun onCreateView(savedInstanceState: Bundle?): View?  
        return activity?.let {  
            val title = "Título del diálogo"  
            val content = "Texto del cuadro de diálogo"  
            val builder: AlertDialog.Builder = AlertDialog  
                .Builder(requireActivity())  
            builder.setTitle(title).setMessage(content)  
                .setPositiveButton(android.R.string.ok) { _, _ ->  
                    // Callback para el «Ok»  
                }  
                .setNegativeButton(android.R.string.cancel) { _, _ ->  
                    // Callback para el Cancel  
                }  
            // Devuelve un AlertDialog,  
            // tal y como lo hemos definido en el builder  
            return builder.create()} ?: throw IllegalStateException("El  
                fragmento no está asociado a ninguna actividad")  
        }  
}
```

Para reconocer las clases que hemos utilizado, previamente deberemos añadir las bibliotecas DialogFragment, Dialog o AlertDialog, entre otras:

```
import android.app.Dialog  
import android.os.Bundle  
import androidx.appcompat.app.AlertDialog  
import androidx.fragment.app.DialogFragment
```

Si analizamos la estructura del método onCreateDialog, veremos que este consta de una única sentencia return:

```
class MiDialogFragment : DialogFragment() {  
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {  
        return activity?.let {  
            ...  
            return builder.create()  
        } ?: throw IllegalStateException  
        ("El fragmento no está asociado a ninguna actividad")  
    }  
}
```

En este código:

- En primer lugar, obtenemos la actividad asociada al fragmento mediante el nombre de propiedad activity (que internamente Kotlin implementa como el getter getActivity()).
- Si la actividad no es nula, utiliza la función de ámbito let para crear el diálogo dentro de esta actividad.
- En caso de que el fragmento no esté asociado a ninguna actividad, se lanzará una excepción del tipo IllegalStateException (estado ilegal). Para ello, utilizamos el operador Elvis (`?:`).

Esta estructura es bastante común en Android, puesto que muchas veces tenemos elementos que nunca sabremos si son nulos o no, y tanto los contextos como los fragmentos pueden serlo.

Por otro lado, debemos tener en cuenta que estamos en un fragmento, y no en una actividad, por lo que no tenemos acceso directo a esta ni a su contexto. Para obtener una referencia a la actividad en la que se encuentra el fragmento, usaremos el método requireActivity().

Para crear el builder:

```
val builder: AlertDialog.Builder = AlertDialog.Builder(requireActivity())
```

Y si añadimos código dentro de los callbacks para generar, por ejemplo, un toast, quedaría:

```
Toast.makeText(requireActivity().applicationContext,  
    "Texto del Toast", Toast.LENGTH_SHORT).show()
```

Finalmente, para usar el diálogo que hemos creado en el fragmento, haríamos lo siguiente:

- Si se invoca desde una actividad, crearemos una instancia del diálogo e invocaremos su método show proporcionándole el administrador de fragmentos asociado a nuestra actividad (supportFragmentManager) y un tag.

```
val miDialogo = MiDialogFragment()  
miDialogo.show(supportFragmentManager, "miDialogo")
```

- Si estuviésemos en otro fragmento, para acceder al administrador de fragmentos de la actividad, en primer lugar deberíamos acceder a esta:

```
val miDialogo = MiDialogFragment()  
miDialogo.show(requireActivity().supportFragmentManager, "miDialogo")
```

B. DIÁLOGOS Y CALLBACKS

En los diferentes ejemplos que hemos visto sobre diálogos, las acciones a realizar cuando el usuario selecciona uno u otro botón vienen definidas en el propio diálogo. Lo habitual será que estas acciones se definan en la propia actividad y se proporcionen de algún modo al diálogo.

En el CPE del final de este apartado veremos cómo resolver este problema mediante la creación de interfaces internas a nuestro diálogo.

4. Navegación entre actividades: Intents y menús

4.1. ¿Cómo definimos un menú?

Los menús se definen en un formato estándar en XML, de manera independiente a la actividad o el fragmento, de forma que después se inyecta en esta (inflate) a través de un objeto de tipo Menu.

Los recursos de menú se ubican dentro del directorio res/menu, e incorporan las etiquetas siguientes:

- <menu>: Es la raíz del menú, y contiene varios elementos de tipo <item> y <group>.
- <item>: Crea un MenuItem, que representa un elemento en un menú y que puede contener otros elementos <menu> anidados para crear un submenú. En este caso, solamente está permitido un nivel de anidamiento. Dentro de los ítems, usaremos propiedades como el identificador [android:id], el icono [android:icon] o el texto [android:title], así como propiedades para especificar cuándo y cómo aparece el elemento en la barra de app, mediante app:showAsAction. Esta propiedad admite los valores siguientes:
 - ifRoom: Si hay espacio, añade el elemento en la barra de la aplicación si hay espacio. En el caso de que no haya lugar para todos los elementos que tengan esta propiedad activada, se mostrarán los que tengan valores del atributo orderInCategory más pequeños. El resto irán al menú expandido.
 - never: No se muestra nunca el elemento en la barra de la aplicación, sino en el menú expandido. Este es el comportamiento por defecto.
 - always: Añade siempre el elemento a la barra de la aplicación. Esta no es una opción recomendable, puesto que puede provocar que varios elementos de la UI se superpongan.
 - collapseActionView: Indica si hay que contraer la vista de la acción asociada al elemento de acción.
- <group>: Permite categorizar diferentes elementos de menú para compartir propiedades, como el estado de una actividad o la visibilidad.

4.2. Añadiendo un menú a una actividad

Para incorporar un menú a una actividad, se usa el método onCreateOptionsMenu(), dentro del cual usamos el método getMenuInflater.inflate() para añadir a la actividad el menú definido en el XML. La propia actividad nos proporcionará el objeto de tipo getMenuInflater mediante el método getMenuInflater(), que utilizaremos en Kotlin como menuInflater.

Este método onCreateOptionsMenu() es invocado por el propio sistema Android cuando se inicia la actividad, de modo que nos proporciona el menú de la aplicación como argumento, antes de añadirlo a la barra de la aplicación, para que podamos modificarlo antes de visualizarlo. En las versiones anteriores a Android 3.0 este menú se creaba al pulsar el botón de menú.

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    val inflater: MenuInflater = menuInflater
    inflater.inflate(R.menu.menu_principal, menu)
    return true
}
```

Además de definir el menú en el XML, también se pueden añadir elementos de menú con add() y recuperar elementos con findItem().

GESTIONANDO LOS CLICS EN EL MENÚ

Cuando se selecciona un elemento del menú de opciones, el sistema invoca el método onOptionsItemSelectedSelected() de la actividad y proporciona el MenuItem seleccionado. Podemos identificar el elemento invocando el método getItemId (o itemId para el acceso en Kotlin), que nos proporcionará el id del elemento de menú definido por android:id en el recurso de menú, de modo que podamos determinar qué acción realizar según la opción seleccionada.

El esquema general que seguiremos en este método será similar al siguiente:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    // Determinamos la acción a realizar según el id del ítem
    return when (item.itemId) {
        R.id.id_entrada_1 -> {
            // Acciones cuando se selecciona la entrada 1
            true
        }
        R.id.id_entrada_2 -> {
            // Acciones cuando se selecciona la entrada 2
            true
        }
        ... // Otras entradas
        // En caso de que no coincida con ningún id, invocamos el
        // método onOptionsItemSelected de
        else -> super.onOptionsItemSelected(item)
    }
}
```

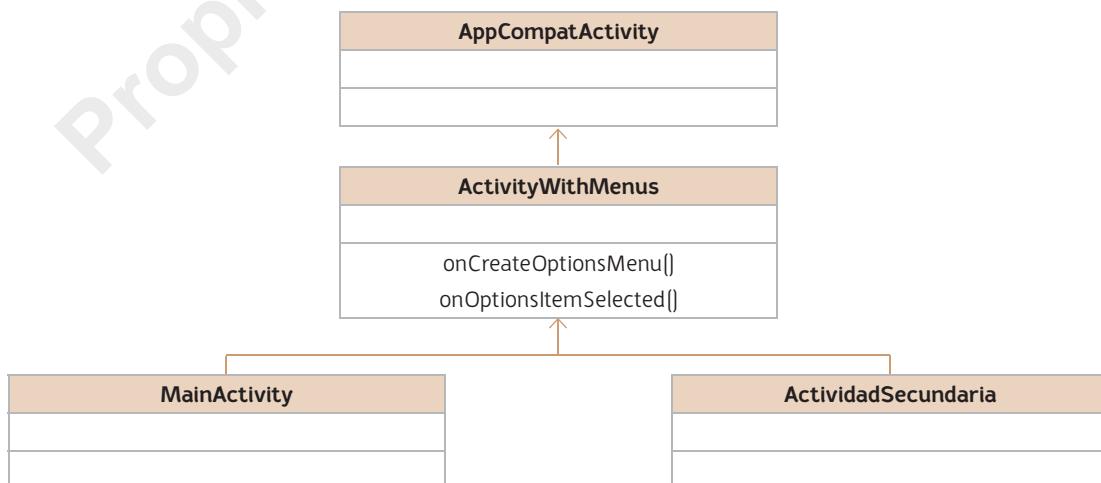
Cuando se controle correctamente una entrada de menú, el método deberá devolver el valor «true», y, si ocurre algún error, devolverá «false». En caso de no controlar el elemento del menú en este método, invocaremos el método onOptionsItemSelected de la superclase.

4.3. Uso de menús comunes a varias actividades

Los menús pueden estar asociados a una o varias actividades. Tal y como se ha trabajado hasta ahora, si quisieramos utilizar el mismo menú en varias actividades, deberíamos copiar el mismo código en todas ellas, con el inconveniente de que, cuando hubiera que añadir, por ejemplo, el control de otra entrada de menú, se deberían modificar todas.

Para facilitar la reutilización de código, podemos crear una nueva clase Actividad que sea la encargada de gestionar el menú implementando los métodos onCreateOptionsMenu() y onOptionsItemSelected(). Cada actividad que comparta el mismo menú de opciones nacerá de esta nueva clase. Así pues, tenemos centralizada la gestión del menú en una única clase, de modo que las modificaciones se realizan en un único punto.

A efectos prácticos, si habitualmente definimos nuestras actividades como extensiones de la clase AppCompatActivity, ahora introduciremos una nueva clase que descienda de AppCompatActivity, que implemente los métodos onCreateOptionsMenu() y onOptionsItemSelected(), y de la que descenderán las otras actividades. Gráficamente, podemos ver esto del modo siguiente:



En este punto, hay que tener en cuenta que Kotlin, por defecto, define las clases como finales, por lo que no admiten herencia.

Para permitir, pues, la herencia en una clase, deberemos definirla explícitamente como abierta, mediante la palabra «open» delante de la definición de esta.

INTENTS Y MEJORAS EN EL MENÚ

Las opciones de menú, sobre todo si se trata de un menú global de la aplicación, servirán generalmente para abrir otras actividades de la propia aplicación. Debemos recordar que con este objetivo Android proporciona el mecanismo de las Intents que ya conocemos: mensajes asíncronos que podemos lanzar a algún componente, sea de nuestra aplicación o de cualquier otra preparada para ello.

Así pues, el código que veremos habitualmente dentro del método `onOptionsItemSelected` seguirá un esquema similar a este:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.itemID_Abrir_ActividadX -> {
            val intent = Intent(this, ActividadX::class.java)
            startActivity(intent)
            true
        }
        ...
    }
}
```

Si dejamos así este código, conseguimos que desde una actividad que contenga el menú se puedan abrir las diferentes actividades en él contempladas, incluida la propia actividad.

Para evitar esto, tenemos dos opciones:

1. Que no se lance la Intent si ya estamos en la actividad que se selecciona.
2. Que se oculte la entrada del menú correspondiente a la actividad en que estamos.

En ambos casos la clase que gestiona el menú deberá conocer en qué actividad nos encontramos. Para ello, en Java usaríamos un atributo static en la clase menú, de modo que fuese compartido por sus subclases. En Kotlin no disponemos de este mecanismo, pero sí de los objetos complementarios o companion objects.

Companion Objects u objetos complementarios

Debemos recordar que Kotlin permite definir directamente objetos que no sean instancias de una clase. Los objetos complementarios se declaran directamente como objetos miembros de una clase, y se comportan del mismo modo que las propiedades estáticas en Java, es decir, están asociados a la clase en sí y no a sus instancias. Con esto, todas las clases que deriven de la clase que contiene el objeto complementario compartirán dicho objeto.

Por tanto, la declaración de la clase con el menú quedaría de la manera siguiente:

```
open class ActivityWithMenus : AppCompatActivity() {
    companion object {
        var actividadActual=0;
    }
    ...
}
```

De este modo, con esta `actividadActual` sabemos en qué actividad nos encontramos (0 en la primera, 1 en la segunda, etc.).

Ahora, cuando tratemos cada opción, podemos determinar mediante este objeto qué hacer:

- Si deseamos que simplemente se lance la Intent, si la actividad es diferente a la actual, escribiríamos:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.itemID_Abrir_ActividadX -> {
            if (actividadActual != 0) {
                val intent = Intent(this, ActividadX::class.java)
                actividadActual = 0;
                startActivity(intent)
            }
            true
        }
        R.id.itemID_Abrir_ActividadY -> {
            if (actividadActual != 1) {
                val intent = Intent(this, ActividadY::class.java)
                actividadActual = 1;
                startActivity(intent)
            }
            true
        }
        ...
    }
}
```

- Si deseamos ocultar la entrada del menú correspondiente a la entrada actual, podríamos hacer:

```
open class ActivityWithMenus : AppCompatActivity() {
    companion object {
        var actividadActual=0;
    }
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        val inflater: MenuInflater = menuInflater
        inflater.inflate(R.menu.menuprincipal, menu)
        for (i in 0 until menu.size()) {
            if (i == actividadActual) menu.getItem(i).isVisible = false
            else menu.getItem(i).isVisible = true
        }
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        return when (item.itemId) {
            R.id.itemID_Abrir_Actividad_X -> {
                ...
                actividadActual=nun;
                ...
            }
            ...
        }
    }
}
```

Otra opción interesante, y similar a esta, sería utilizar isEnabled en lugar de isVisible para desactivar la opción, sin que esta desaparezca del menú.

4.4. Mejorando el comportamiento de las Intents

Observando el comportamiento actual de la aplicación, si, cuando lanzamos una actividad nueva desde el menú, en lugar de presionar el botón *Atrás* para volver a la actividad anterior lo hacemos desde el menú, conseguimos que se abra una nueva actividad.

Esto tiene que ver con la organización de actividades. Android utiliza una pila para gestionar las diferentes actividades de una misma tarea, de modo que cuando el usuario inicia una nueva actividad esta se añade a la pila. Cuando el usuario presiona el botón *Atrás*, la actividad finaliza y se elimina de la pila.

Lo ideal en este caso sería que, cuando el usuario lance una actividad que ya se encuentra en la pila, en lugar de lanzarla de nuevo, reordene la pila para que la nueva actividad se ponga la primera. Esto lo podemos conseguir mediante el flag FLAG_ACTIVITY_REORDER_TO_FRONT en el momento de lanzar la Intent. Con este flag se crearía de nuevo la actividad, si esta no existiera previamente, mientras que, si la actividad ya existe, esta pasará a primer plano, sin crearse de nuevo.

Para conseguir esto, haremos:

```
val intent = Intent(this, NuevaActividad::class.java)
intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);
startActivity(intent)
```

MENÚS CONTEXTUALES

Los menús contextuales proporcionan acciones asociadas a elementos o marcos concretos de la interfaz de usuario, generalmente elementos de tipo colección de vistas, como los ListViews, GridViews o RecyclerView, que veremos en el apartado siguiente.

Hay dos formas de proporcionar acciones contextuales. Por un lado, en un menú flotante, como una lista de elementos de menú que aparece al hacer un clic largo sobre un elemento; por otro lado, en el modo de acción contextual, que muestra en la parte superior de la pantalla una barra de acciones contextuales con elementos de acción. Nos centraremos en el primero de ellos.

Creación de un menú contextual flotante

Para crear un menú contextual flotante, debemos:

1. Indicarle al sistema sobre qué vista queremos que se active un menú contextual. Esto se conoce como registrar la vista, y para ello utilizamos el método registerForContextMenu(). Cuando queremos que el menú se muestre para cada uno de los elementos de una lista, deberemos proporcionarle esta al método registerForContextMenu().

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // Registraremos la vista para que cuando se haga un
    // clic largo sobre ella se muestre el menú contextual
    registerForContextMenu(vista_a_registrar)
}
```

2. Implementar el método onCreateOptionsMenu(), que será invocado por el sistema cuando se haga un clic largo sobre un elemento registrado. En este método definiremos los elementos del menú, generalmente añadiendo un recurso de este tipo.

```
// Cuando se genere un menú contextual sobre una vista
// se invoca el método siguiente para crear el menú
override fun onCreateOptionsMenu(
    menu: ContextMenu?,
    v: View?,
    menuInfo: ContextMenu.ContextMenuItemInfo?) {
    menuInflater.inflate(R.menu.menu_contextual,menu)
    // También podemos añadirle una cabecera
    menu?.setHeaderTitle("Menú contextual")
    super.onCreateOptionsMenu(menu, v, menuInfo)
}
```



Este método funciona de forma similar al método `onCreateOptionsMenu()`, con la diferencia de que, en lugar de invocarse solamente una vez al iniciarse la aplicación, lo hace cada vez que se necesita el menú contextual.

3. Establecer las acciones a realizar en cada opción del menú contextual mediante el método `onContextItemSelected()`, que funcionará de un modo parecido a `onOptionsItemSelected()`:

```
override fun onContextItemSelected(item: MenuItem): Boolean {  
    when(item.itemId) {  
        R.id.id_opcion_X ->  
            // Acciones..  
        ...  
    }  
    return super.onContextItemSelected(item)  
}
```

Opcionalmente, si deseamos controlar cuándo se cierra este menú contextual, podemos hacerlo implementando el método `OnContextMenuClosed()`.

Propiedad de McGraw-Hill®

5. El componente RecyclerView

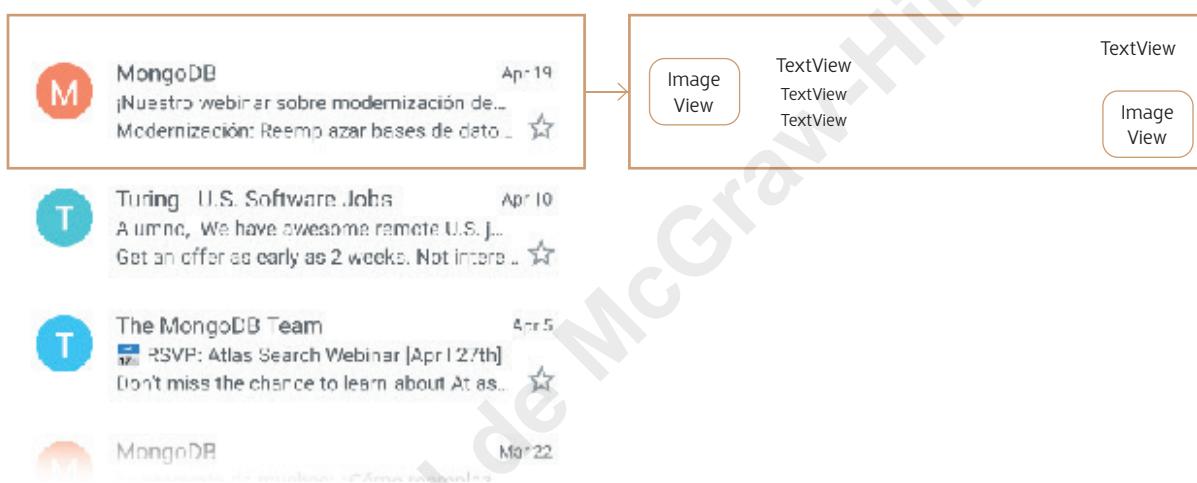
5.1. Listas con RecyclerView

El componente RecyclerView es un Viewgroup muy flexible que permite representar una colección de datos. Estos datos pueden provenir de varias fuentes, como pueden ser ficheros o bases de datos locales o de Internet.

El RecyclerView se compone de una serie de vistas, que pueden organizarse de diferentes formas dentro del componente y permiten el desplazamiento dentro de este. Cada uno de los elementos que se muestran en la lista tendrá un layout determinado con varios elementos de tipo View.

Esquemáticamente, podemos verlo como un contenedor en el que encontramos diferentes instancias de un layout concreto, que puede contener texto, texto e imagen u otros componentes, como, por ejemplo, tarjetas (CardView).

Podemos ver un ejemplo de ello en la imagen siguiente de la app de Gmail, donde se muestra una lista de correos organizados de forma lineal y en vertical. Cada correo se representa mediante una vista concreta, aunque todas ellas siguen el mismo esquema:



Siguiendo con este ejemplo, ahora imaginemos que tenemos una lista con miles de correos. En un ListView, a medida que nos desplazamos por la lista, deberíamos generar una vista nueva, rellenarla con la información correspondiente e incorporarla a la lista. Además, los elementos deberían destruirse cuando ya no estuviesen visibles.

Al usar un RecyclerView, en cambio, las diferentes instancias de los diseños que desaparecen de la pantalla cuando se hace scroll no se destruyen, sino que se reciclan con nueva información para mostrar los elementos que entran en la pantalla. Esto supone una gran ventaja respecto al clásico ListView, ya que el proceso de creación y destrucción de instancias es relativamente pesado.

El funcionamiento general del RecyclerView se muestra en la imagen siguiente:

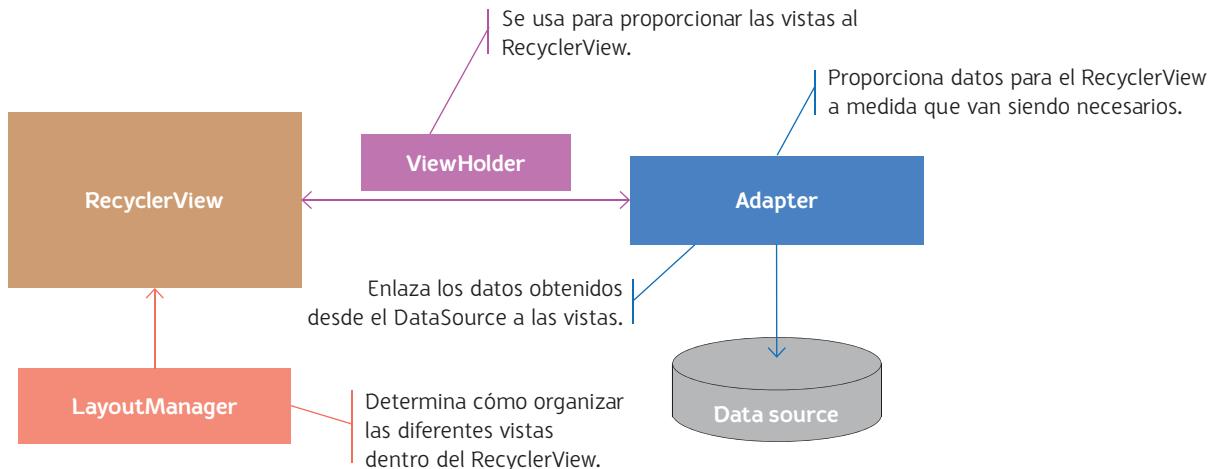


5.2. Elementos para crear un RecyclerView

Para la implementación de un RecyclerView necesitamos varios elementos:

- Un **contenedor general** que muestre de forma ordenada los diferentes elementos. Este contenedor es el propio RecyclerView y se ubicará en el layout de la actividad en la que deba aparecer. La forma de ordenar los elementos dentro del RecyclerView vendrá definida por un Gestor de diseño o LayoutManager.
- Un **diseño para cada uno de los elementos** que queremos mostrar a la lista. Este diseño lo definiremos mediante otro recurso de tipo layout, en un fichero XML.
- Saber cuál es el **conjunto de datos** que queremos mostrar en la lista. Estos pueden provenir tanto de un simple vector de objetos como de una consulta a una base de datos, y se suelen conocer como DataSetCo.

Con el diseño por un lado y los datos por otro, únicamente queda enlazar ambos para que los datos se muestren en el RecyclerView según el diseño creado para cada ítem. De esto se encargará un **Adaptador (Adapter)**, que será el componente más importante del RecyclerView y al que más tiempo dedicaremos en su implementación. Este componente, además, contendrá una clase llamada **ViewHolder**, que se encargará de hacer la correspondencia entre el conjunto de datos **DataSet** y el diseño de la vista de los elementos.



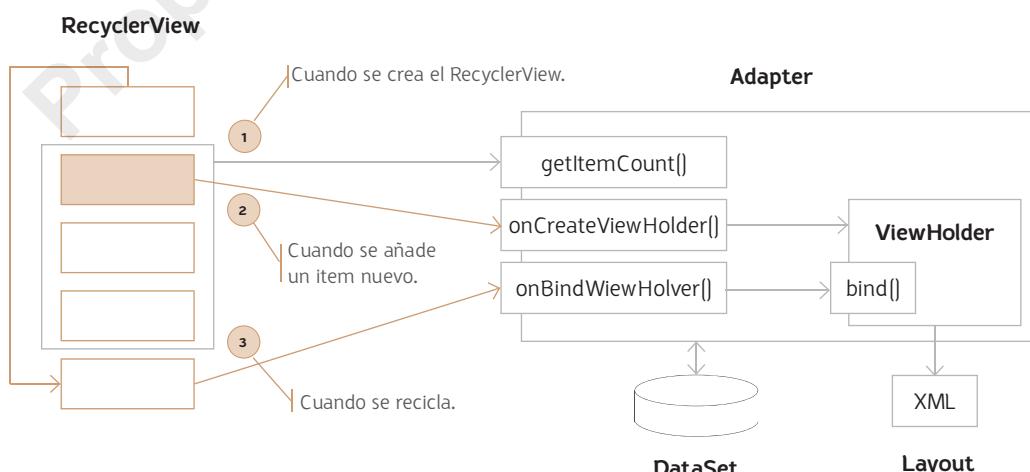
5.3. Creación del adaptador

Como hemos comentado, el adaptador es la clase que conecta el RecyclerView con el conjunto de datos a representar (DataSet). Cada vez que el RecyclerView necesite reciclar un ítem de la lista, utilizará el adaptador para obtener dicho ítem a partir de los datos. Concretamente, usará una clase interna al adaptador denominada ViewHolder, que se encargará de establecer la correspondencia entre este DataSet y el diseño de la vista de los elementos individuales que forman el ítem.

Es importante remarcar que nuestra tarea se va a centrar en preparar el adaptador para que el RecyclerView haga uso de él cada vez que lo necesite.

La interacción entre el componente RecyclerView y el adaptador pasará por los puntos siguientes:

1. Para dibujar la lista de elementos en la pantalla, el RecyclerView pedirá al adaptador el número total de elementos que contiene el DataSet. Esto lo hará mediante el método getItemCount(), que deberemos implementar en el adaptador.
2. Cuando el RecyclerView necesite añadir una nueva instancia de la vista en memoria, invocará el método onCreateViewHolder() del adaptador. En este punto, el adaptador preparará el ViewHolder con el diseño definido en el XML de la vista.
3. Cuando un ítem ya creado se tenga que reutilizar, el RecyclerView pedirá al adaptador que actualice sus datos. Es decir, utilizará el mismo diseño, pero únicamente modificará los datos que se muestran. Esto se consigue sobrescribiendo el método onBindViewHolder() del adaptador.



Así pues, nuestra tarea principalmente se centrará en implementar las clases ViewHolder y Adapter.

A. LA CLASE VIEWHOLDER

Empezando por el ViewHolder, este tendrá un esquema similar al siguiente:

```
class MiViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {  
    // Definición e inicialización de los diferentes recursos  
    // de la vista en el ViewHolder  
    val recurso_1 = itemView.findViewById(id_recurso1) as TextView  
    ...  
    // Enlace de los datos con la vista  
    fun bind(objeto: Clase) {  
        recurso_1.valor = objeto.valor  
        ....  
    }  
}
```

Como vemos, la clase MiViewHolder desciende de RecyclerView.ViewHolder, y se inicializa a partir de la vista de un ítem. Esta vista proporcionada será el layout ya inflado del ítem al que se corresponderá el ViewHolder.

Por otra parte, se implementa el método bind, el cual se encargará de vincular el valor de los diferentes componentes del ítem con los valores correspondientes a un objeto del DataSet. Este método bind se invocará cada vez que el RecyclerView invoque el método onBindViewHolder del Adapter.

B. LA CLASE ADAPTER

La clase Adapter tendrá un esquema similar al siguiente:

```
class MiAdapter: RecyclerView.Adapter<RecyclerView.ViewHolder>() {  
    // Método onCreateViewHolder: Invocado por el RecyclerView cuando  
    // necesita crear un nuevo ítem (ViewHolder)  
    override fun onCreateViewHolder(  
        parent: ViewGroup,  
        viewType: Int): RecyclerView.ViewHolder {  
        // Se crea un LayoutInflater para inyectar el XML  
        val inflater = LayoutInflater.from(parent.context)  
        // Se inyecta el layout (XML) en la vista.  
        val vista=inflater.inflate(R.layout.layoutVista, parent, false);  
        // Y crea un nuevo ViewHolder proporcionándole la vista  
        return miViewHolder(vista)  
    }  
  
    // Método onBindViewHolder: Es invocado por el RecyclerView cuando  
    // necesita reciclar un ítem. Recibe el ViewHolder y la posición en el  
    // DataSet que contiene el objeto a representar.  
    override fun onBindViewHolder(holder: RecyclerView.ViewHolder,  
        position: Int) {  
        // Este método usa el método bind de la clase MiViewHolder  
        (holder as MiViewHolder).bind(objeto_del_dataset_en_posicion);  
    }  
  
    // Método getItemCount: Es invocado por el RecyclerView en su  
    // creación, y le proporciona el número de elementos del DataSet  
    override fun getItemCount(): Int {  
        return tamaño_del_dataset;  
    }  
}
```

C. CREACIÓN DEL RECYCLERVIEW

Una vez preparados el ViewHolder y el Adapter, ya podemos implementar el RecyclerView. Para ello, seguimos los pasos siguientes:

1. Asignación de un LayoutManager al RecyclerView. Este LayoutManager o administrador de diseño será necesario para ordenar las diferentes vistas en el RecyclerView. Existen tres administradores de diseño:
 - LinearLayoutManager, que es el administrador más común y que crea un diseño de lista unidimensional sobre el que se puede hacer scroll.
 - GridLayoutManager, que dispone los elementos en una cuadrícula bidimensional. Según esta se organice de forma horizontal o vertical, este administrador intentará que todos los elementos de cada fila o columna, respectivamente, tengan el mismo ancho y altura.
 - StaggeredGridLayoutManager, que es similar al anterior, aunque sin ajustar el alto o el ancho en filas o columnas, sino que filas y columnas de distintos tamaños pueden compensarse entre sí.



Para asignar este LayoutManager, haremos:

```
// Asociamos el LayoutManager
MiRecyclerView.setLayoutManager(LinearLayoutManager(...))
                           | GridLayoutManager(...) |
                           | StaggeredLayoutManager(...) |
```

2. Podemos indicarle al RecyclerView que su tamaño va a ser fijo, mediante el método setHasFixedSize(true). Este paso no es obligatorio, pero se trata de una optimización, ya que evitamos que el RecyclerView recalcule su tamaño cuando se añadan nuevos elementos.

```
MiRecyclerView.setHasFixedSize(true)
```

3. Finalmente, se crea una instancia del Adaptador y se asigna al Adaptador del RecyclerView:

```
MiRecyclerView.setAdapter(MiAdaptador())
```

5.4. Gestión de Eventos en el RecyclerView

Generalmente, los RecyclerViews, aparte de mostrar datos, permiten interactuar con la lista de elementos. En este apartado vamos a ver cómo realizar esta interacción y responder a los diferentes eventos que se produzcan sobre los elementos de la lista.

En Android, la captura de eventos se realiza en la vista concreta con la que interactúa el usuario. En el caso de un RecyclerView, dicha vista será cada uno de los ViewHolders que lo compongan. En muchas vistas existen objetos de escucha de eventos [event listeners] que posibilitan la interacción; por ejemplo, un Button ya tiene asociados objetos de escucha de eventos para gestionar eventos tales como onTouch(), onClick(), etc. Sin embargo, cuando tenemos vistas personalizadas, como es el caso de los ViewHolders en un RecyclerView, dichos objetos no existen. Por este motivo, la clase View proporciona algunas interfaces internas que ofrecen objetos de escucha de eventos, por lo que las vistas personalizadas que deseen responder a ciertos eventos deberán implementar esas interfaces.

En resumen, si deseamos que un ViewHolder responda ante un cierto evento, el ViewHolder deberá implementar la interfaz correspondiente al objeto de escucha de eventos para dicho evento. Por ejemplo, si deseamos que responda al evento onClick, deberá implementar la interfaz View.OnClickListener del modo siguiente:

```
class MiViewHolder(itemView: View) :  
    RecyclerView.ViewHolder(itemView),  
    View.OnClickListener  
{  
    ...  
}
```

Esta interfaz aporta un objeto de escucha del evento onClick sobre la vista, y nos comprometerá a implementar el método onClick en la clase. Una vez definida la clase de este modo, deberemos indicar en su bloque de inicialización que será la propia clase la que implemente el método onClick del ítem:

```
init {  
    itemView.setOnClickListener(this)  
}
```

Recuerda: Sobre los constructores primarios. El constructor primario de una clase es aquel declarado en el mismo encabezado de la clase. Definido de este modo, este constructor primario no puede contener ningún código, por lo que la inicialización de parámetros en este constructor se realiza mediante bloques de inicialización, con prefijo init.

Con todo ello, el código del ViewHolder será ahora:

```
class MiHolder(itemView: View) :  
    RecyclerView.ViewHolder(itemView),  
    View.OnClickListener {  
    // Definición e inicialización de los diferentes recursos  
    // de la vista en el ViewHolder  
    val recurso_1 = itemView.findViewById(id_recurso1) as TextView  
    init {  
        itemView.setOnClickListener(this)  
    }  
  
    // Enlace de los datos con la vista  
    fun bind(objeto: Clase) {  
        recurso_1.valor = objeto.valor  
        ....  
    }  
}
```

```
override
fun onClick(v: View?) {
    // Implementamos el método onClick de la interfaz OnClickListener.
    // Recibimos la vista que recibe el evento, por lo que a través
    // de ella podemos obtener la información acerca de este.
}
```

Este esquema es adecuado cuando el propio ViewHolder, a través de la información proporcionada en la lista, es capaz de determinar qué hacer. Si, por el contrario, esta vista no contiene información suficiente, debemos indicarle de algún modo al RecyclerView qué hacer. Esto normalmente se consigue mediante callbacks.

GESTIÓN DE EVENTOS EN EL RECYCLERVIEW MEDIANTE CALLBACKS

Cuando el ViewHolder no es capaz de determinar por sí mismo y por la información que recibe de la vista qué hacer, es posible proporcionarle desde el adaptador esta funcionalidad mediante callbacks.

El esquema será similar. La detección del evento se sigue realizando en el ViewHolder, pero la funcionalidad vendrá definida por la actividad que contiene el RecyclerView, y la proporcionará a través del adaptador.

Adaptación del ViewHolder a los callbacks

Para adaptar el ViewHolder, realizaremos los cambios siguientes respecto a la implementación anterior:

1. El ViewHolder no implementará la interfaz View.OnClickListener, puesto que esta funcionalidad vendrá dada por el callback.
2. Eliminaremos el bloque de inicialización, por el mismo motivo.
3. En el método bind, que enlaza con la vista, incorporaremos el callback como argumento. Dado que el itemView que estamos enlazando recibirá los eventos, en este método deberemos asignar el listener que se nos proporciona como callback a la vista.
4. Eliminamos el método onClick de esta clase, puesto que esta funcionalidad ya se proporcionará en el callback.

El esquema resultante seguirá, pues, el patrón siguiente (también se marcan en rojo las líneas eliminadas del esquema anterior).

```
class MiHolder(itemView: View) : RecyclerView.ViewHolder(itemView)

    //, View.OnClickListener { // 1. Ya no se implementa esta interfaz

    val recurso_1 = itemView.findViewById(id_recurso1) as TextView

    /* 2. No necesitamos indicar que será el ViewHolder quien gestione
       los eventos.

    init {
        itemView.setOnClickListener(this)
    }
    */

    // 3. Enlace de los datos con la vista e incorporamos en el bind
    // el callback

    // fun bind(objeto: Clase) {

    fun bind(objeto: Clase, eventListener: (Clase, View) -> Unit) {
        recurso_1.valor = objeto.valor
        ....
        itemView.setOnClickListener{ eventListener(objeto, itemView) }
    }
}
```

```
/*
4. Eliminamos el método onClick, puesto que se proporciona el callback
override
fun onClick(v: View?) {
    4. Eliminamos...
}
*/
```

}

Adaptación del Adaptador a los callbacks

Ya que hemos modificado el método bind del ViewHolder, deberemos modificar también la invocación que se realiza en el Adaptador en el método onBindViewHolder.

Los principales cambios en el esquema del Adapter serán los siguientes:

1. Incorporar un nuevo parámetro en el constructor del adaptador, que será el callback que se encargará de gestionar el evento.
2. Proporcionar dicho callback en la invocación al bind del ViewHolder.

```
//class MiAdapter: RecyclerView.Adapter<RecyclerView.ViewHolder>() {
// 1. Incorporamos el eventListener al constructor primario
class MiAdapter(val eventListener: (Clase, View) -> Unit ): 
    RecyclerView.Adapter<RecyclerView.ViewHolder>() {

    ...
    override fun onBindViewHolder(holder: RecyclerView.ViewHolder,
                                position: Int) {
        // (holder as parkViewHolder).bind(objeto_del_dataset_en_posicion);

        // 2. Incorporamos el eventListener en la llamada a bind
        (holder as MiViewHolder).bind(objeto_del_dataset_en_posicion,
                                      eventListener);
    }
}
```

Adaptación en la creación del RecyclerView

En la actividad en que creamos el RecyclerView, deberemos:

1. Modificar la creación del adaptador proporcionándole una expresión lambda como callback.
2. Añadir el nuevo método [itemClicked], que será el método que gestionará el evento.

```
class MainActivity ... 
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // MiRecyclerView.adapter = MiAdaptador()
    MiRecyclerView.adapter = MiAdaptador(
        {objeto:Clase, v: View -> itemClicked(objeto, v)} )
}

private fun itemClicked(objeto:Clase, v: View) {
    // Acciones a realizar cuando se produce el evento
}
}
```



Unidad 3.

Desarrollo de aplicaciones android.

Persistencia, sensores

Propiedad de McGraw-Hill®

1. Plantilla Basic Activity

1.1. La plantilla de actividad básica

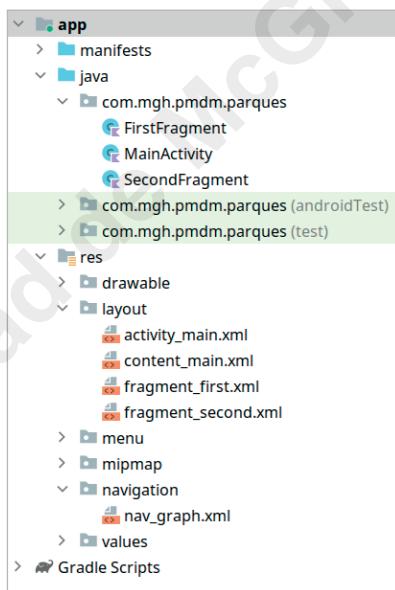
La plantilla de actividad básica puede suponer un buen punto de partida para una aplicación, ya que introduce elementos de interfaz que ofrecen una mejor experiencia de usuario. Los componentes de esta plantilla son los siguientes:

- **Barra de la aplicación o de acción (AppBarLayout):** Barra horizontal ubicada bajo la barra de estado y que contiene elementos como el nombre de la aplicación o el menú.
- **Botón de acción flotante (FloatingActionButton):** Se trata de un elemento definido en las especificaciones de Material Design y que contiene un ícono del stock de Android, concretamente app:srcCompat="@android:drawable/ic_dialog_email". Este ícono lo podremos modificar.
- **Contenido principal:** La parte central de la actividad contiene el contenido principal de la aplicación y, como veremos, usa otros ficheros de diseño para separar la funcionalidad.

Aparte de aportar un aspecto más actual a las aplicaciones, lo más interesante de esta plantilla es que nos va a introducir en el uso de fragmentos y en la navegación entre ellos.

A. ORGANIZACIÓN DE LA BASIC ACTIVITY

La plantilla de actividad básica ofrece organización y separación de contenidos. Cuando generamos un proyecto con una actividad de este tipo, tenemos una estructura de ficheros como la siguiente:



Como vemos, esta estructura contiene cuatro layouts y tres ficheros fuente, además de un nuevo recurso de tipo Navigation.

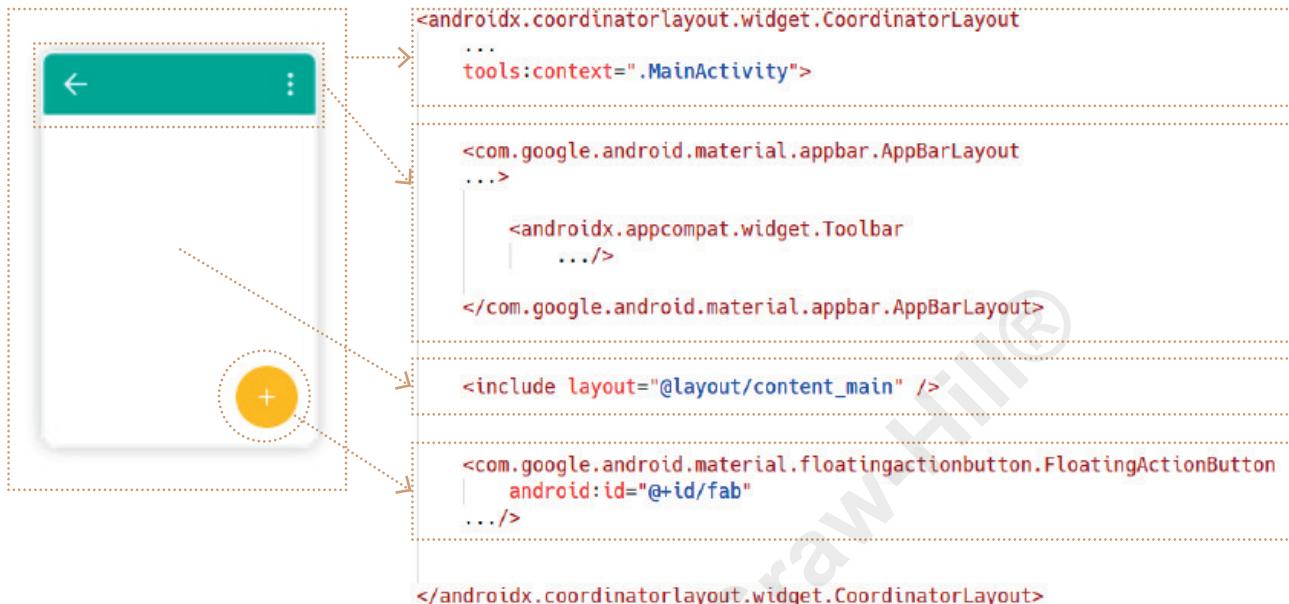
B. LA ACTIVIDAD PRINCIPAL

El layout de la actividad principal, layout/activity_main.xml, se compone ahora de un diseño de tipo CoordinatorLayout. Este componente es un contenedor de vistas (ViewGroup) que tiene como finalidad posicionar los widgets de la aplicación de nivel superior de los que ya hemos hablado: el AppBarLayout y el FloatingActionButton, y proporcionar los mecanismos para que interactúen entre ellos.

Para utilizar este componente deberíamos incorporar como dependencia en el fichero build.gradle el componente CoordinatorLayout. No obstante, dado que partimos de la plantilla Basic Activity, esta dependencia se incluye indirectamente y de forma automática, del mismo modo que la dependencia del componente RecyclerView, dentro de la dependencia de las librerías Material Design.

El componente CoordinatorLayout es parte de Android Jetpack, un conjunto de bibliotecas que nos ayuda a seguir las prácticas recomendadas, reducir código estándar y escribir código que funcione de forma coherente entre los diferentes dispositivos y versiones de Android. Los componentes y las librerías asociadas a Jetpack empiezan por androidx.

De forma esquemática, el código de este layout para la actividad principal puede verse así:



Observamos que el diseño consiste en un elemento CoordinatorLayout, que contiene la especificación de la barra superior mediante AppBarLayout y el botón flotante de acción FloatingActionButton. Las propiedades de estos elementos se pueden modificar directamente sobre el código de la etiqueta o con el editor de propiedades y cambiar, por ejemplo, el color de fondo de la barra o el icono del botón.

Veamos algunos detalles sobre los diferentes elementos.

- **Sobre la barra principal de la aplicación.**
 - El nombre de la aplicación que se muestra en esta barra se especifica en el atributo android:label="@string/app_name" del AndroidManifest.xml. Para modificarlo, deberemos cambiar el recurso de texto app_name en el fichero values/string.xml.
 - En esta barra también aparece el botón derecho con los tres puntos verticales para desplegar el menú oculto. El código para gestionar este menú se encuentra en la misma actividad principal MainActivity.kt, (no hay que crear una superclase), y se gestiona con los métodos que ya conocemos, como onOptionsItemSelected(). Los elementos de este menú se ubican en menu/menu_main.xml.
- **Sobre el botón de acción flotante (FloatingActionButton).** La ventana principal también contiene el botón de acción flotante FloatingActionButton (fab), un elemento definido en Material Design y que contiene un ícono del stock de Android, concretamente app:srcCompat="@android:drawable/ic_dialog_email". El fichero con la clase que controla esta actividad, MainActivity.kt, incluirá dentro del método onCreate un controlador onClick para este botón. De manera predeterminada, cuando hacemos clic en el botón, aparece un snack bar que nos indica su reemplazo por el código que deseamos. El color de este botón utiliza la tonalidad de énfasis del tema. Si queremos modificarlo, cambiaremos el atributo colorAccent del fichero de recursos values/colors.xml.
- **Sobre el contenido principal de la aplicación.** Finalmente, este fichero importa el contenido principal de la aplicación mediante la etiqueta <include layout="@layout/content_main"/>, que incorpora el contenido del fichero layout/content_main.xml.

En líneas generales, podríamos resumir los puntos clave de esta plantilla así:

- El layout de la actividad principal no define ninguna vista.
- Las vistas se incluyen al fichero content_main.xml, que a su vez se incorpora al layout principal con un include.
- De este modo, las vistas del sistema se mantienen separadas de las vistas exclusivas de la aplicación.

1.2. Navegación

Cuando hablamos de navegación en Android, nos referimos a la capacidad de los usuarios para desplazarse a través de diferentes elementos de contenido de la aplicación. Android Jetpack incorpora el componente Navigation, que permite implementar esta navegación, ya sea mediante botones o mediante otros elementos más complejos como los paneles laterales de navegación.

Este componente consta de tres partes fundamentales:

- Un **grafo de navegación (nav_graph)**, especificado como un recurso XML en la carpeta *navigation*, y que centraliza toda la información relacionada con la navegación.
- El **host de navegación (NavHost)**, que es un contenedor vacío donde se ubicarán los diferentes destinos del grafo de navegación. De forma predeterminada, el componente Navigation contiene la implementación NavHostFragment, que se encarga de intercambiar los destinos de los fragmentos.
- El **administrador de navegación o NavController**, que gestiona la navegación de la aplicación dentro de un NavHost.

IMPORTANTE

El componente Navigation está pensado para aquellas aplicaciones que tienen una actividad principal que contiene varios destinos, todos ellos en forma de fragmentos. La actividad principal tendrá un grafo de navegación asociado con un NavHostFragment, que será el contenedor donde se intercambiarán los distintos destinos. Si tenemos una aplicación con diferentes actividades, cada actividad puede contener un grafo de navegación.

A.FRAGMENTOS

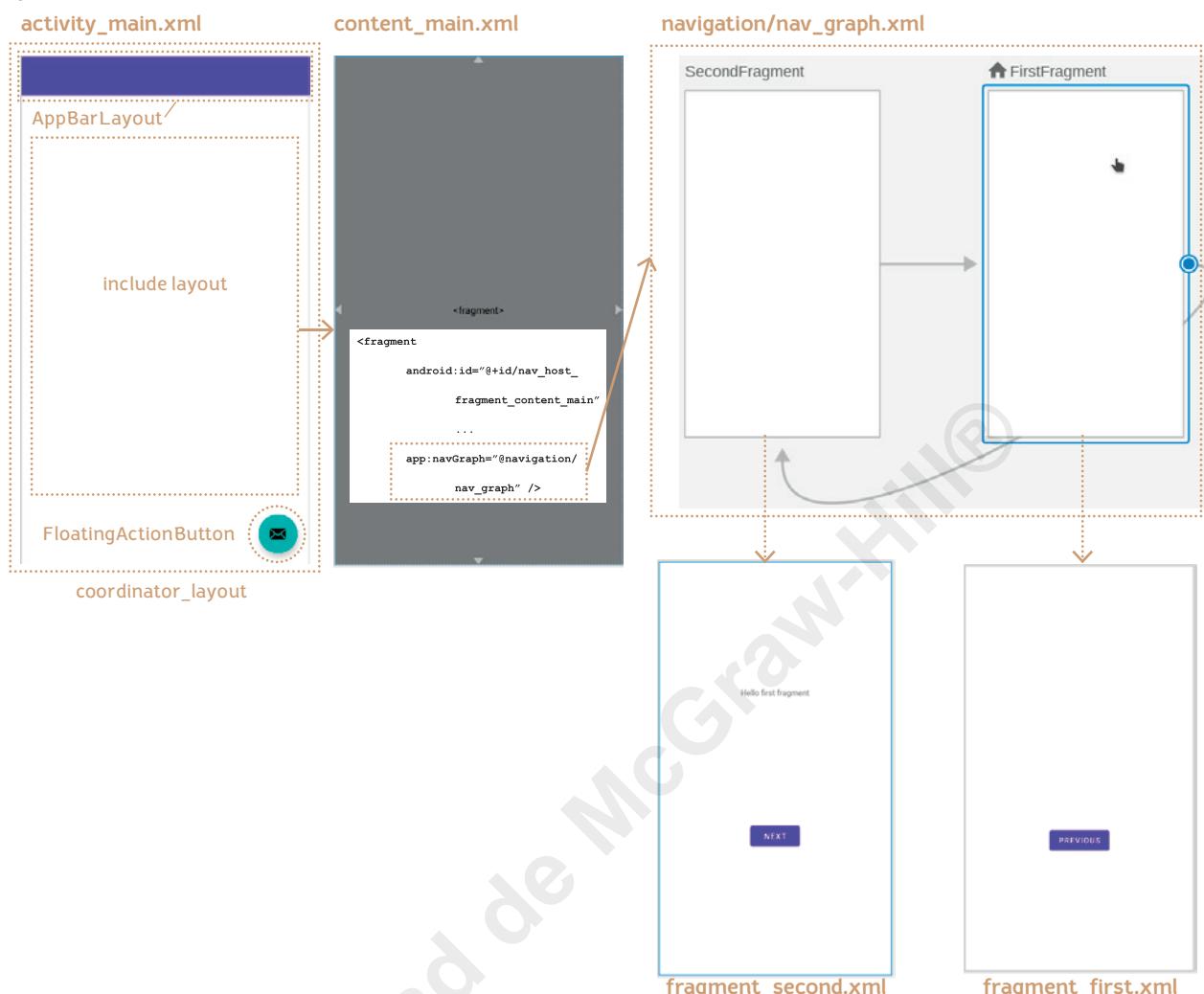
Los fragmentos representan partes reutilizables de la interfaz de usuario. Disponen de un diseño y de un ciclo de vida propios, y gestionan sus propios eventos.

Los fragmentos aparecieron para aprovechar el espacio que ofrecían dispositivos de visualización más grandes que un teléfono móvil, como las tabletas, de modo que una misma aplicación pudiera ofrecer diferentes organizaciones de las vistas según el tamaño de pantalla.

Tanto el componente Navigation de Android Jetpack como otros como BottomNavigation o ViewPager están pensados para funcionar con fragmentos. Como hemos comentado, un fragmento dispondrá de su propio diseño y clase que lo gestione, pero debe estar vinculado a una actividad.

B. NAVEGACIÓN Y FRAGMENTOS EN LA PLANTILLA BASIC ACTIVITY

Teniendo en cuenta los conceptos sobre navegación y fragmentos, podemos entender mejor el esquema siguiente con la arquitectura y el funcionamiento de la plantilla Basic Activity:



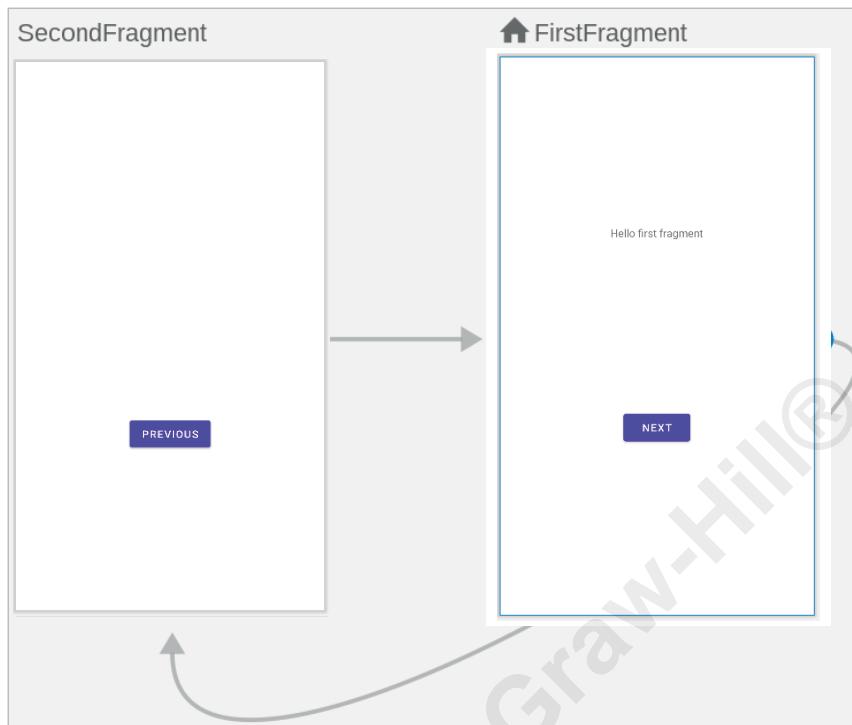
Como vemos, el contenido principal se almacena en el diseño layout/content_main.xml. Este ya contendrá un diseño de tipo ConstraintLayout, en el que únicamente se incluye una vista de tipo fragment. De esta etiqueta destacamos los atributos siguientes:

- android:id="@+id/nav_host_fragment_content_main": Tiene el identificador del fragmento.
- android:name="androidx.navigation.fragment.NavHostFragment": Indica el nombre de la implementación de nuestro NavHost. En este caso, se utiliza la implementación por defecto basada en fragmentos NavHostFragment.
- app:navGraph="@navigation/nav_graph": Asocia el host de navegación NavHostFragment con un grafo de navegación, donde se especificarán los diferentes destinos de navegación a los que podemos llegar.
- app:defaultNavHost="true": Sirve para interceptar correctamente el botón Atrás del sistema. Si tuviésemos varios hosts en un mismo diseño (por ejemplo, dos paneles), deberíamos tener en cuenta que solamente un NavHost puede ser el predeterminado.

Con esto, en el content_main.xml tendremos el fragmento que servirá de contenedor para la navegación (host de navegación) y que será gestionado por la implementación NavHostFragment y el grafo de navegación especificado en el fichero navigation/nav_graph.xml.

C. EL GRAFO DE NAVEGACIÓN

El fichero navigation/nav_graph.xml define una etiqueta de tipo navigation, dentro de la cual se indican dos etiquetas más de tipo fragment con los fragmentos que compondrán la navegación y las transiciones entre ellos. Gráficamente lo podemos ver del modo siguiente:



Vemos que solo aparecen dos fragmentos: FirstFragment, marcado con el icono de una casa a su lado, que indica que es el fragmento de inicio, y SecondFragment. Además, los unen dos flechas, y que representan el sentido de la navegación.

Veamos algunos detalles del XML para entenderlo todo mejor:

- En primer lugar, en el hashtag navigation definimos el identificador e indicamos cuál es el fragmento de inicio:

```
<navigation xmlns:android="http://schemas.android.com/apk/res/android"  
...  
    android:id="@+id/nav_graph"  
    app:startDestination="@+id/FirstFragment">
```

- Dentro de esta etiqueta, añadimos los diferentes fragmentos indicando su identificador, su nombre (la clase que los gestionará), la etiqueta (label) y su diseño (layout):

```
<fragment  
    android:id="@+id/FirstFragment"  
    android:name="com.ieseljust.pmdm.myapplication.FirstFragment"  
    android:label="@string/first_fragment_label"  
    tools:layout="@layout/fragment_first" >
```

- Y, a su vez, dentro de cada fragmento, disponemos de una nueva etiqueta action, que representará las transiciones que puede haber desde el fragmento. En este caso, la acción action_FirstFragment_to_SecondFragment tendrá como destino el fragmento SecondFragment. Es decir, cuando se realice esta acción, el contenido de FirstFragment será reemplazado por el de SecondFragment en el contenedor o host de navegación.

```
<action  
    android:id="@+id/action_FirstFragment_to_SecondFragment"  
    app:destination="@+id/SecondFragment" />
```

1.3. Detalles de implementación

Una vez vistos los diseños implicados en la navegación de la actividad básica, ahora nos queda por ver todo el código asociado en las diferentes clases implicadas.

A. ACTIVIDAD PRINCIPAL: MAINACTIVITY

La actividad principal consiste en una actividad de tipo AppCompatActivity, que ya incorpora el databinding. Además, también contiene una referencia al objeto de configuración de la barra de la aplicación (appBarConfiguration), que se utilizará en un par de métodos:

```
class MainActivity : AppCompatActivity() {  
  
    private lateinit var appBarConfiguration: AppBarConfiguration  
    private lateinit var binding: ActivityMainBinding  
  
    ...  
}
```

El método onCreate inyecta los layouts a los bindings y establece el contenido principal de la vista de la aplicación:

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

Por otro lado, dentro de este mismo método se invoca al método setSupportActionBar() heredado de la clase AppCompatActivity, que establece la barra de herramientas como la barra de la aplicación de la actividad:

```
setSupportActionBar(binding.toolbar)
```

En otros tipos de actividades, como la Empty Activity, hemos usado la barra por defecto y, por ello, no hemos tenido que establecerla explícitamente. El principal motivo para incorporar ahora la barra de forma explícita es que vamos a añadirle el controlador de navegación para poder hacer visible el botón de volver atrás en la navegación.

Esto lo conseguimos mediante una referencia al administrador de navegación, o NavController, asociado al contenedor principal de los fragmentos (nav_host_fragment_content_main), definido en el fichero layout/content_main.xml. Con esta referencia, obtenemos el grafo de navegación (navController.graph) y lo asignamos a la barra mediante la creación de un objeto de configuración de tipo AppBarConfiguration. Con esto, inicializamos la barra de navegación con este controlador de navegación y la configuración que hemos creado:

```
val navController = findNavController(R.id.nav_host_fragment_content_main)  
appBarConfiguration = AppBarConfiguration(navController.graph)  
setupActionBarWithNavController(navController, appBarConfiguration)
```

Finalmente, el método onCreate asocia un gestor de eventos al evento del clic sobre el botón de acción flotante, que, en este caso, muestra un elemento de tipo Snackbar:

```
binding.fab.setOnClickListener { view ->  
    Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)  
        .setAction("Action", null).show()  
}
```

Por otro lado, esta clase principal implementa los métodos `onCreateOptionsMenu` y `onOptionsItemSelected`, que ya conocemos para gestionar el menú, el layout del cual podemos encontrar en `menu/menu_main.xml`.

El último método que se define en esta clase, y que servirá para gestionar la acción de volver hacia atrás (o arriba) en la navegación, es `onSupportNavigateUp()`:

```
override fun onSupportNavigateUp(): Boolean {
    val navController =
        findNavController(R.id.nav_host_fragment_content_main)

    return navController.navigateUp(appBarConfiguration)
        || super.onSupportNavigateUp()
}
```

La función principal de este método es invocar el método `navigateUp` del administrador de navegación `navController` de nuestro contenedor principal del fragmento. El método `navigateUp` recibe la configuración de la barra de aplicación (`appBarConfiguration`) y devuelve un valor lógico indicando si se ha completado o no dicha navegación. Nuestro método `onSupportNavigateUp` devolverá este mismo valor o, en su defecto, el valor de retorno del método `onSupportNavigateUp` de su clase padre.

B. ANALIZANDO EL CÓDIGO DE LOS FRAGMENTOS

Ya solo nos queda analizar el código de los fragmentos en sí. Si analizamos cualquiera de los dos ficheros para la gestión de los fragmentos (`Firstfragment.kt` o `Secondfragment.kt`), veremos que estos ahora derivan de la clase `Fragment` y no de `Activity`:

```
class FirstFragment : Fragment() {...}
```

Después definimos los `dataBindings`, pero, en lugar de definirlos como un `lateinit`, los definimos mediante propiedades de respaldo (`backing properties`). Es decir, definimos la propiedad (`de respaldo`) `_binding` como nullable, y la inicializamos directamente como null modificando su método de acceso `binding get()` (al que invocaremos con `binding`), de forma que devuelva el valor de la propiedad de respaldo `_binding`, con la posibilidad de que dé una excepción si este valor es nulo:

```
private var _binding: FragmentFirstBinding? = null
// This property is only valid between onCreateView and
// onDestroyView.
private val binding get() = _binding!!
```

Para entender bien esto, además de las propiedades de respaldo, debemos introducir algunos detalles del ciclo de vida de los fragmentos. A pesar de que irán ligados al ciclo de vida de la actividad, los fragmentos también tienen su propio ciclo de vida, y responden a determinados callbacks que se invocan en este ciclo. Los principales callbacks que se deberán implementar en todo fragmento son los siguientes:

- `onCreate()`, invocado por el sistema cuando se crea el fragmento.
- `onCreateView()`, invocado cuando el fragmento debe generar la interfaz, de forma que muestra una vista.
- `onPause()`, invocado cuando el usuario abandona el fragmento.

En caso de que algunos de estos métodos no especifiquen cuándo definimos un fragmento, se usarán directamente los métodos de la clase padre `Fragment`.

En los fragmentos generados en la plantilla de actividad básica, los métodos que se implementan en los fragmentos son `onCreateView` y `onViewCreated`. Este último se dispara inmediatamente después de `onCreateView` y antes de restaurar cualquier estado en el fragmento. Asimismo, se implementa el callback `onDestroyView`, que se invoca cuando la vista que se generó en el `onCreateView` se desvincula del fragmento, de modo que la próxima vez que el fragmento se deba mostrar se genere la vista de nuevo.

Veamos el código de estos métodos:

```
override fun onCreateView(  
    inflater: LayoutInflater, container: ViewGroup?,  
    savedInstanceState: Bundle?  
) : View? {  
    _binding = FragmentFirstBinding.inflate(inflater, container, false)  
    return binding.root  
}
```

Recuerda que este método crea la vista del fragmento. Para ello se usa el binding, de manera que inyecte el diseño definido en layout/fragment_first.xml y devuelva la raíz de este. Fijateos en que para hacer la asociación utiliza la propiedad de respaldo _binding, pero accede a la raíz a través del getter binding.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
    super.onViewCreated(view, savedInstanceState)  
  
    binding.buttonFirst.setOnClickListener {  
        findNavController()  
            .navigate(R.id.action_FirstFragment_to_SecondFragment)  
    }  
}
```

Una vez generada la vista del fragmento, asociamos al evento onClick del botón buttonFirst la acción de navegar del primer fragmento al segundo. Para ello, accedemos al administrador de navegación (mediante findNavController) y, desde este, invocamos el método navigate indicando el identificador de la acción action_FirstFragment_to_SecondFragment, que es la que realiza la transición entre el primer fragmento y el segundo, y está definida en el fichero navigation/nav_graph.xml.

```
override fun onDestroyView() {  
    super.onDestroyView()  
    _binding = null  
}
```

Cuando se desvincula la vista de la actividad y se destruye el fragmento, se asigna el valor nulo a la propiedad de respaldo del binding, que ya volveremos a generar cuando volvamos a acceder a este fragmento.

2. Componentes de la arquitectura de Android y MVVM

2.1. Componentes de la arquitectura de Android

En el ciclo de conferencias Google IO 2017, Google presentó su guía de arquitectura de aplicaciones definiendo los principios clave sobre los que debería asentarse una aplicación Android, estableciendo como estándar el patrón MVVM y presentando los bloques de construcción para su implementación: los componentes de arquitectura de Android.

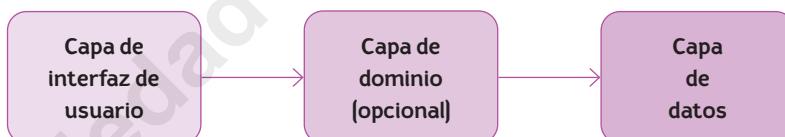
A. PRINCIPIOS DE LA ARQUITECTURA DE APLICACIONES ANDROID

Los principios que propone la guía de arquitectura de Android son los siguientes:

- **Separación de problemas.** Según este principio, en Activities y Fragments únicamente hay que mantener la lógica correspondiente a la interacción con el usuario y con el sistema, de manera que estas clases se mantengan lo más limpias posible. Recordemos que Android, cuando necesita recursos, puede eliminar de la memoria actividades o fragmentos que no se encuentren en primer plano.
- **Controlar la interfaz de usuario a partir de modelos de datos.** De acuerdo con este principio, los modelos que representan los datos de la aplicación deben ser independientes de la interfaz de usuario y de su ciclo de vida. Además, es interesante dotar a estos modelos de persistencia para que no se pierda información cuando el sistema elimine procesos.

La propuesta de Google para mantener estos principios es una arquitectura multicapa, en la que cada aplicación tenga, al menos, dos capas:

- La **capa de interfaz de usuario**, que se encarga exclusivamente de mostrar los datos de la aplicación en la pantalla.
- La **capa de datos**, con la lógica de negocio de la aplicación y la gestión de los datos, los cuales expone a la interfaz.
- De forma opcional, se plantea una **capa de dominio** que simplifique y reutilice las interacciones entre la interfaz y las capas de datos.



Fuente: Diagrama de una arquitectura de app típica. Imagen publicada como obra cultural gratuita en [Developer.android.com](https://developer.android.com) bajo licencia Creative Commons Attribution 2.5 (CC BY 2.5).

B. COMPONENTES DE LA ARQUITECTURA

Con el fin de alcanzar los principios anteriores, Google introduce los Componentes de la arquitectura, que ayudarán a la separación de problemas y harán que nuestras aplicaciones sean más fáciles de testar y de mantener.

Estos componentes son cuatro, cada uno con una función específica, y trabajan de forma conjunta para ofrecer una arquitectura sólida. Estos componentes son Lifecycle, LiveData, ViewModel y Room.

- **Lifecycle.** Se trata de una clase asociada a Actividades y Fragmentos que mantiene información sobre el estado del ciclo de vida de estos. Esto permite que otros objetos puedan observar y hacer un seguimiento de dicho estado.
- **LiveData:** Es un contenedor de datos observables optimizado para ciclos de vida. Esto significa que respeta los ciclos de vida de las actividades o los fragmentos, de modo que solo actualiza los observadores cuando el estado del ciclo de vida de los observados se encuentra activo.
- **ViewModel:** Es el componente alrededor del cual gira la arquitectura propuesta, cuya traducción literal vendría a ser Modelo de la Vista. Se trata de una clase que gestiona los datos relacionados con la interfaz de usuario de manera optimizada para los ciclos de vida. Uno de los detalles más importantes es que no elimina la clase ViewModel cuando el sistema decide eliminar una actividad o un fragmento por

falta de recursos, por lo que sus datos se conservan ante cambios de configuración, como una rotación de pantalla.

- **Room:** Se trata de una biblioteca de persistencia que actúa como capa de abstracción para la base de datos SQLite de Android. La trataremos en detalle en el apartado siguiente.

2.2. El modelo MVVM

El patrón Modelo-Vista-ViewModel (o MVVM) es un patrón de arquitectura basado en flujos observables que sigue los principios y la arquitectura de aplicación propuesta por Google; por lo tanto, se trata del patrón propuesto para estructurar las aplicaciones en Android.

Este patrón se divide en tres componentes:

- El **Modelo** (DataModel), que se encarga de obtener los datos desde cualquier fuente de datos y de exponerlo a los ViewModels.
- La **Vista**, que se encarga de mostrar y gestionar la interfaz de usuario, de modo que muestra al usuario información proporcionada por capas inferiores e informa a estas de sus acciones.
- El **ViewModel**, que está ubicado entre los dos anteriores y que se encarga de obtener los datos del modelo y de exponerlos a la interfaz mediante eventos que las vistas puedan observar.

Así pues, en este modelo la vista se corresponde a la actividad o el fragmento, y será la responsable de la visualización de los datos, pero de una forma más reactiva o automatizada, gracias al componente ViewModel, que preparará los datos para que estos sean observables en la Vista a partir del Modelo.

Las clases ViewModel y AndroidViewModel

Para crear un ViewModel no tendremos más que crear una clase que descienda de esta:

```
class MiViewModel() : ViewModel() { ... }
```

Si nuestro ViewModel, además, va a necesitar el contexto de la actividad o del fragmento asociado, deberá definirse como subclase de AndroidViewModel:

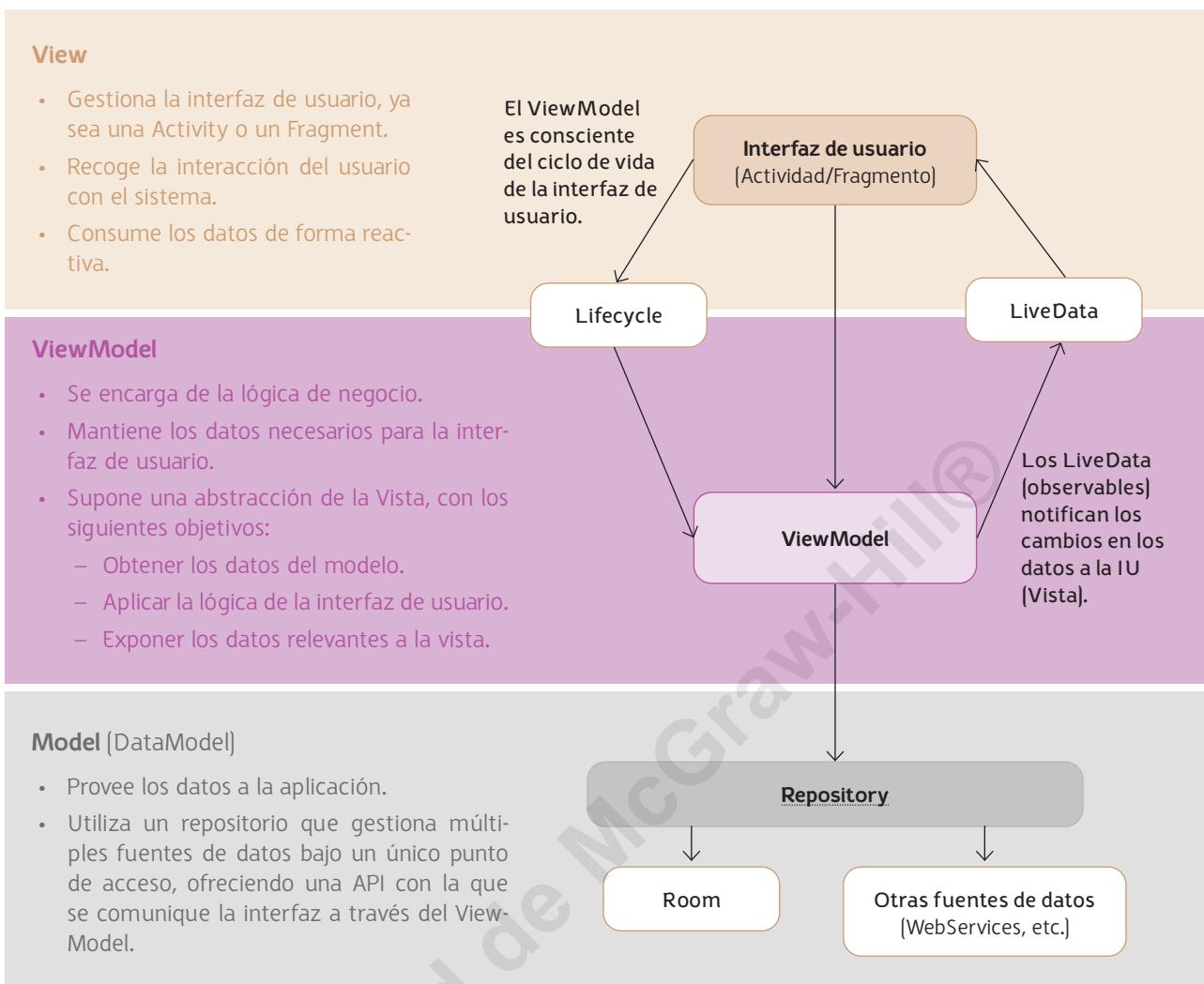
```
class MiViewModel(application: Application) : AndroidViewModel(application) { ... }
```

Luego, para asociar un ViewModel a una vista usaremos un ViewModelProvider, que será el encargado de instanciar el ViewModel de forma adecuada y de proporcionárnoslo.

La principal ventaja de este patrón es que los cambios son reactivos, es decir, la interfaz se modifica como reacción a los cambios en el ViewModel, de forma automática, así como el ViewModel reacciona a los cambios producidos en la interfaz y modifica su estado al instante.

Además, este modelo resuelve también el problema de pérdida de estados que ocurre cuando Android decide liberar memoria destruyendo actividades o fragmentos, o cuando se produce una rotación de pantalla, ya que es el ViewModel quien mantiene este estado.

Gráficamente, e introduciendo los componentes de arquitectura de Android, podemos ver este modelo MVVM del modo siguiente:



Como vemos, la capa Vista está representada por Activities y Fragments que observan elementos de tipo LiveData en los cuales se reflejan los cambios en el modelo. Asimismo, el ViewModel es consciente del estado del ciclo de vida [Lifecycle] de la vista y así mantiene la consistencia en el modelo ante cambios de configuración de la vista o la interacción del usuario.

Cuando la vista necesita datos, los pide al ViewModel, que accederá a la capa de modelo para proporcionárselos mediante los LiveData observables. Observamos que el **ViewModel** en ningún momento hará referencia directa a la vista, por lo que la actualización de esta será exclusivamente a través de los LiveData.

Centrándonos en el modelo, observamos que aparece una nueva entidad: el **Repository**. Este repositorio no es ninguna clase especial de Android, sino una clase cuya función será obtener los datos de todas las posibles fuentes, ya sea una base de datos SQLite a través de Room o servicios web. Cuando se le pida información, el repositorio buscará en la fuente de datos adecuada y ofrecerá estos datos, generalmente a través de LiveData, al ViewModel.

EL PATRÓN OBSERVABLE. IMPLEMENTACIÓN DE LOS LIVEDATA

Hemos hablado sobre los cambios reactivos en la interfaz de usuario del patrón MVVM y comentado que esta reactividad se consigue a través los LiveData, que son elementos observables, pero ¿qué significa esto exactamente?

El patrón observable es una de las bases tanto de los LiveData como de los componentes conscientes del Lifecycle, y consiste en que un objeto puede notificar a un conjunto de observadores cualquier cambio de estado. De este modo, las actividades y los fragmentos que observen un LiveData recibirán actualizaciones sobre cualquier modificación y podrán actualizarse de forma automática.

Como hemos comentado, el LiveData, además de tratarse de un contenedor de datos observables, está optimizado para ciclos de vida. Esto significa que solamente actualizará aquellos observadores que se encuentren en componentes (fragmentos, actividades o servicios) que tienen un estado de ciclo de vida activo (STARTED o RESUMED).

Generalmente, definiremos los LiveData en el ViewModel, de modo que puedan ser observados desde la vista. La forma de definir un LiveData será la siguiente:

```
class MiViewModel() : ViewModel() {  
    val miLiveData: MutableLiveData<T> by lazy {  
        MutableLiveData<T>()  
    }  
    ...  
    private fun setLiveData(valor: T) {  
        miLiveData.value = valor  
    }  
}
```

Hagamos algunas observaciones respecto a este código:

- Hemos definido la variable inmutable miLiveData como MutableLiveData<T>. La clase LiveData es una clase abstracta, y por ello puede instanciarse, así que para crear este tipo de objetos utilizamos la clase MutableLiveData, que implementa toda la funcionalidad necesaria.
- El genérico <T> es el tipo de datos que vamos a almacenar en el LiveData.
- Observamos que, aunque el LiveData esté definido como inmutable, su contenido sí que puede variar.
- La inicialización de la variable miLiveData se ha definido como lazy. Este es un concepto similar al lateinit, e indica que la inicialización de la variable se realizará durante la primera vez que se haga uso de ella. Lazy es una función de Kotlin, la cual recibe una función lambda o de orden superior como argumento. Esta función será la que inicialice la variable en su primera invocación. En el resto de invocaciones, retornará el valor calculado inicialmente.
- Para darle valor al contenido del LiveData debemos acceder a su propiedad value. Arriba, hemos creado una función privada setLiveData, a la que le pasamos un valor de tipo T que se asigna al valor del LiveData.

Observadores

Los observadores pueden registrarse en objetos que implementen la interfaz LifecycleOwner, es decir, que posean un ciclo de vida. De este modo se permite desvincular al observador cuando el objeto pase a DESTROYED en su ciclo de vida.

Tanto actividades como fragmentos implementan esta interfaz, por lo que pueden registrar observadores a objetos LiveData. Cuando la actividad o el fragmento en que se definieron se elimina, estos observadores son borrados y se elimina así cualquier riesgo de fuga de memoria. Aquí lo interesante es que los ViewModels no se destruyen, de manera que cuando una actividad o un fragmento eliminado por falta de memoria o por cambios de configuración se vuelve a crear puede vincularse de nuevo a los LiveData y mantener el estado.

Los observadores suelen crearse en los métodos onCreate u onCreateView de las actividades o los fragmentos. Cuando la actividad o el fragmento se inicia, recibe los valores más recientes de los LiveData que está observando.

Veamos en el siguiente código de ejemplo, de forma genérica, cómo estableceríamos un observador para los LiveData definidos anteriormente en una actividad:

```
class MiActividad: AppCompatActivity() {

    // Definimos la instancia del ViewModel como lateinit
    private lateinit var miFragmentViewModel: MiViewModel

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

        // Observamos los LiveData, mediante su método Observe
        miViewModel.miLiveData.observe(viewLifecycleOwner,
            Observer { variable ->
                // Actualización de la interfaz
            })
    }
}
```

Como vemos, este método Observe recibe dos parámetros: el objeto viewLifecycleOwner y el Observer. El primero informa al ViewModel del estado del ciclo de vida del fragmento, para que le envíe o no notificaciones según su estado; el segundo es un objeto que implementa la interfaz Observer. Esta interfaz implementa un único método, onChange, que se invoca cuando los datos del LiveData observado sufren alguna modificación. Dentro de este método, en la interfaz realizaremos todos los cambios que sean necesarios en función de los datos que hayamos obtenido.

Aquí se aplican las **SAM Conversions** de Kotlin para simplificar la notación, de modo que no hace falta indicar el nombre del método onChange, ni siquiera el constructor Observer:

```
miViewModel.miLiveData.observe(viewLifecycleOwner, { variable ->
    // Actualización de la interfaz
})
```

SAM Conversions, Conversiones SAM

Cuando queremos utilizar una interfaz que contiene únicamente un método, Kotlin permite utilizar una expresión lambda en lugar de definir una clase anónima para su implementación. Esto se conoce como conversiones SAM (Single Abstract Method), y permiten a Kotlin convertir de forma transparente expresiones lambda cuya firma coincida con la del método de la interfaz en una instancia de una clase anónima que implementa la interfaz.

Actualización de los LiveData

La clase MutableLiveData ofrece los métodos públicos setValue() y postValue() para editar el valor almacenado en los LiveData. Se utiliza setValue si nos encontramos en el hilo principal, y postValue si lo hacemos desde algún subproceso.

Por otra parte, dado que Kotlin nos permite utilizar los métodos accesores como si se tratase de propiedades, podemos utilizar directamente value tanto para consultar el valor como para modificarlo.

Así pues, desde cualquier actividad o fragmento, podemos acceder al valor del LiveData pasando por el ViewModel:

```
miViewModel.miLiveData.value=valor
```

Este cambio de valor, como todo cambio de valor en un LiveData, enviará una notificación a todos sus observadores.

3. El componente Room

3.1. SQLite y Room

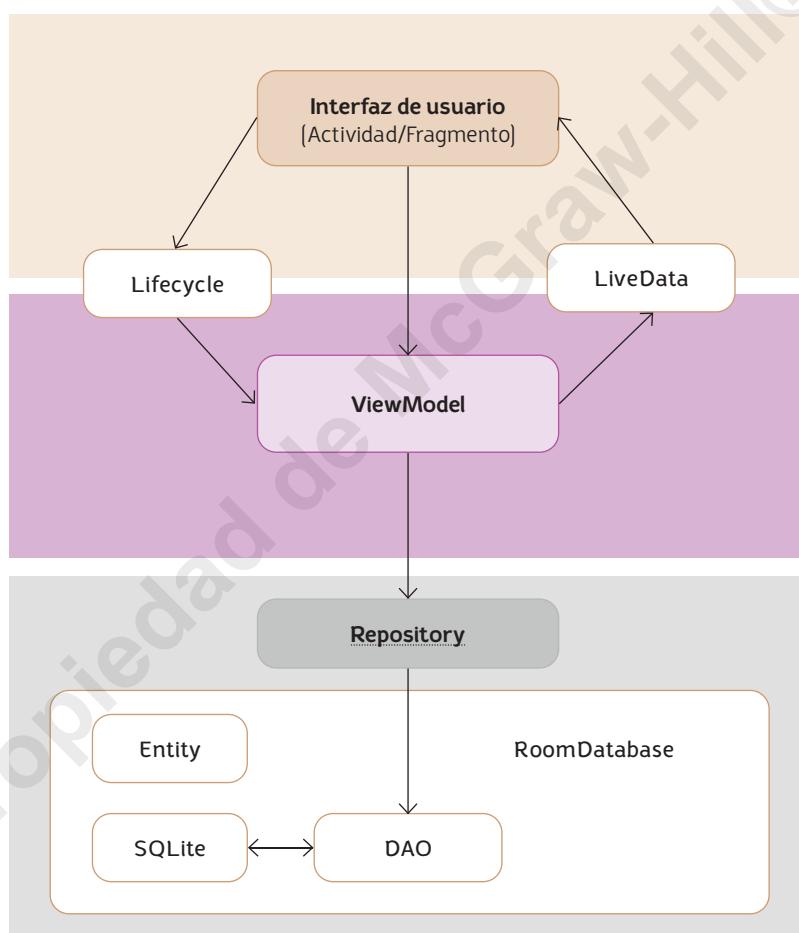
El sistema Android dispone de una base de datos interna, SQLite, que permite su administración mediante el lenguaje SQL.

Antes de que Google propusiera sus componentes de arquitectura de aplicaciones, lo habitual era trabajar directamente con SQLite o utilizar algún ORM (Object Relational Mapping) como SugarORM o ActiveAndroid.

A partir de la publicación de esta arquitectura de aplicaciones, la recomendación de Google es utilizar el ORM Room para acceder a SQLite.

Room supone una capa de abstracción sobre la base de datos SQLite, y proporciona una serie de objetos de acceso a datos (DAO, Data Access Objects) que nos permite trabajar con entidades de las bases de datos.

La imagen siguiente muestra los componentes principales de la arquitectura, en la que destacan los componentes de Room:



Ya conocemos el funcionamiento del repositorio, del ViewModel y de la vista: la vista se encarga de presentar los datos al usuario actualizando estos al ViewModel de forma reactiva mediante observadores, y el ViewModel accede a la API del repositorio, que unifica el código de acceso a los datos.

Introducimos ahora el componente de arquitectura Room. Este se articula a través de la base de datos RoomDatabase, en la cual se definen las diferentes entidades (Entity) que representan las tablas de la base de datos, así como los objetos de acceso a datos (DAO), que ofrecen una API de acceso a la base de datos SQLite.

¿Cómo trabaja Room?

El ORM Room trabaja mediante anotaciones para definir los diferentes elementos que gestiona. Estas anotaciones se convertirán en código de acceso a la base de datos en tiempo de compilación. Para ello, necesitaremos que Room pueda generar código por nosotros.

Esto lo conseguiremos con la herramienta KAPT (Kotlin Annotatin Processing Tool), que permite procesar anotaciones y generar código. Configuraremos esta herramienta usando el plugin kotlin-kapt en el fichero build.gradle del módulo. Veremos cómo hacerlo en el caso práctico.

3.2. Entidades

Las entidades (Entities) son clases que nos ayudan a representar los objetos a almacenar en la base de datos. Cada entidad se corresponde con una tabla, y cada instancia de la entidad, con una fila de la tabla.

Para definir una clase como entidad, debemos anotarla con @Entity y definir los campos que se corresponden a las columnas de la tabla, incluyendo uno o más campos para la clave principal.

IMPORTANTE

Los campos de una entidad deben ser accesibles desde fuera de la clase para que Room pueda acceder a ellos. Así pues, deben declararse como públicos u ofrecer los getters y setters correspondientes.

Por otra parte, en SQLite, tanto los nombres de tablas como los nombres de las columnas no distinguen entre mayúsculas y minúsculas.

Veamos en la tabla siguiente las anotaciones más usuales para la definición de entidades y sus propiedades:

Anotación	Se aplica a	Significado	Propiedades de la anotación
@Entity	Clase	Define la clase como una entidad/tabla de la BBDD. Por defecto, toma el nombre de la clase como nombre de la tabla.	tableName: Permite indicar un nombre para la tabla que sea diferente al de la clase. primaryKeys: Indica uno o varios campos que formarán la clave primaria. Se usa generalmente para las claves compuestas. ignoredColumns: Indica qué campos no van a almacenarse en la base de datos. Se usa generalmente cuando se heredan campos de otra clase.
@PrimaryKey	Propiedad	Define la propiedad o las propiedades como clave principal de la tabla.	autogenerate: Propiedad booleana que le indica a Room si debe generar automáticamente las CP.
@ColumnInfo	Propiedad	Permite incorporar información adicional a la columna. Por defecto, las columnas toman el mismo nombre que la propiedad, y mediante ColumnInfo podemos modificarlo.	name: Define el nombre de la columna en la base de datos.
@Ignore	Propiedad	De forma predeterminada, Room crea una columna por cada propiedad de la clase. Con @Ignore, indicamos a Room que no deseamos que se genere una columna para el campo en cuestión.	

En el enlace a la documentación de Android *Cómo definir datos con entidades Room* puedes consultar otras anotaciones para aspectos más avanzados, como definir tablas virtuales de respaldo para agilizar la búsqueda de texto completo (FTS) o índices.

3.3. Objetos de acceso a datos (DAO)

Para mantener el principio de separación de problemas, el acceso y la manipulación de la información a las entidades no se realiza de forma directa, sino a través de objetos de acceso a datos (DAO o Data Access Objects).

Para ello, los DAO se especifican con interfaces o clases abstractas con anotaciones mediante las cuales indicaremos a Room qué operaciones se pueden realizar sobre la base de datos. El hecho de utilizar interfaces o clases abstractas permite a Room, a través de la herramienta de procesado de anotaciones, implementar sus métodos en el tiempo de compilación.

Sin embargo, en los casos más sencillos habitualmente utilizaremos interfaces. En cualquier caso, deberemos anotar la interfaz o la clase abstracta como @Dao.

Los métodos que podemos definir en un DAO pueden ser de dos tipos: de **conveniencia**, que permiten realizar operaciones de inserción, actualización y borrado de filas sin necesidad de código SQL, y los métodos de **búsqueda**, que permiten personalizar nuestras consultas SQL.

Veamos en la tabla siguiente las anotaciones más comunes utilizadas en la creación de DAO:

Anotación	Se aplica a	Significado
@Dao	Interfaz o clase abstracta	Define la interfaz o la clase abstracta como un objeto de acceso a datos. Ej: <code>@Dao interface PersonaDao {...}</code>
@Insert	Método	Define un método de conveniencia para insertar entidades en las tablas correspondientes de la base de datos. Admite la propiedad de anotación onConflict para gestionar conflictos. Ej: <code>@Insert fun insertPersona(persona: Persona)</code>
@Update	Método	Define un método de conveniencia para actualizar filas de una tabla. Estos métodos admiten instancias de entidades de datos como argumentos. Se usará la clave principal para hacer coincidir las instancias con las filas de la tabla. <code>@Update fun updatePersonas(vararg persona: Persona)</code>
@Delete	Método	Define un método de conveniencia que permite borrar filas específicas de una tabla y proporciona las instancias de entidades a borrar. <code>@Delete fun deletePersonas(vararg persona: Persona)</code>
@Query	Método	Define un método de búsqueda con el que podemos especificar la instrucción SQL a ejecutar y exponerlas como un método DAO. Puede utilizarse tanto para búsquedas (SELECT) como para cualquier tipo de operación CRUD más compleja. La consulta se indicará como argumento de la anotación. Además, podemos hacer consultas parametrizadas utilizando en la consulta los argumentos de la función antepuestos por dos puntos (:). <code>@Query("SELECT * FROM persona WHERE nombre = :nombre") fun getPersonaByName(name: String): Array<Persona></code>

Disponemos de más información acerca de las clases DAO, así como para realizar búsquedas más avanzadas y tratar datos de forma especial o paginar resultados, en la documentación oficial de Android Cómo acceder a los datos con DAO de Room, en la sección de enlaces.

3.4. Creación de la base de datos

Una vez tenemos definidas las entidades y los DAO para acceder a estas, debemos generar la clase para la base de datos, en la cual indicaremos con qué entidades y DAO puede trabajar. Para ello utilizaremos la anotación `@Database`, con las propiedades `entities` y `version`. Este atributo `version` se utiliza para identificar la versión de la base de datos que utilizamos. Si modificamos el esquema de la base de datos, también deberemos modificar el número de versión, además de cambiar la realización de la migración entre una y otra versión.

Si deseas conocer más acerca de las migraciones entre versiones de la base de datos, puedes leer el documento *Cómo migrar bases de datos de Room*, de la documentación oficial de Android, referenciado en la sección de enlaces.

Con ello, ahora la definición de la clase quedará del modo siguiente:

```
@Database(entities = [Entidad::class], version = 1)
abstract class NombreBD : RoomDatabase() {
    abstract fun metodoAccesoDAO(): ClaseDAO
}
```

Observamos que esta clase ofrece un método de acceso a las clases DAO que trabajan con las entidades, y así aporta una API para las capas superiores del modelo. Generalmente el repositorio accede a ella, aunque, si no se utiliza un repositorio, podría utilizarse directamente desde la capa del ViewModel.

Por otro lado, observamos que debe ser una clase abstracta, por lo que no podrá instanciarse de forma directa. A tal efecto, se utiliza el método `Room.databaseBuilder`, de modo que Room cree la instancia por nosotros. Además, dado que el proceso de creación de una instancia de `RoomDatabase` es bastante costoso, y rara vez necesitaremos varias instancias de esta, suele utilizarse el patrón Singleton para ello.

Existen varias formas de implementar este patrón. A modo de ejemplo, veamos cómo implementarlo mediante un objeto (`DataBaseBuilder`) que muestra cómo crear la instancia de la base de datos:

```
object DatabaseBuilder {
    private var INSTANCE: NombreBD? = null
    fun getInstance(context: Context): NombreBD {
        if (INSTANCE == null) {
            synchronized(ParksDB::class) {
                INSTANCE = buildRoomDB(context)
            }
        }
        return INSTANCE!!
    }

    // Constructor privado
    private fun buildRoomDB(contexto: Context) =
        Room.databaseBuilder(
            contexto.applicationContext,
            NombreBD::class.java,
            "nombre-db"
        ).build()
}
```

El funcionamiento es bastante simple si analizamos bien el código.

- Disponemos de un campo privado que es la instancia de la base de datos (`INSTANCE`).
- El objeto tiene un constructor privado `buildRoomDB`, que, a través del método `databaseBuilder` de Room (no debemos confundirlo con el objeto `DatabaseBuilder` que estamos creando), crea la instancia a la base de datos, a partir del contexto de la aplicación, el nombre de la clase de la base de datos

(NombreBD) y el nombre del fichero en que esta se guardará (nombre-db). A partir de este Builder, mediante el método build, obtenemos nuestra base de datos.

- Para obtener una instancia, se utiliza el método getInstance del objeto DatabaseBuilder. Este crea la instancia de la base de datos si no existe y, si ya existe, nos la devuelve.

IMPORTANTE

Los accesos a la base de datos deben realizarse en un hilo de ejecución distinto del hilo principal de la aplicación, en el que se ejecuta el renderizado de la interfaz.

La base de datos admite el método allowMainThreadQueries() antes de lanzar el build() para forzar que el acceso a la base de datos se realice dentro del hilo principal. No obstante, esta opción no es recomendable, ya que una operación costosa sobre la base de datos o algún error puede congelar la interfaz de usuario y dejarla inaccesible.

Para realizar las consultas en hilos de ejecución, utilizaremos corutinas, como veremos en el apartado siguiente.

ACCESO A LA BASE DE DATOS

Accederemos a la base de datos a través del objeto DatabaseBuilder que hemos creado, el cual nos proporcionará la instancia de la base de datos mediante el método getInstance, con la que accederemos a las clases de acceso a datos (DAO) y a los métodos que estas implementan:

`DatabaseBuilder.getInstance(context).metodoAccesoDao().Consulta()`

Estas operaciones, como hemos comentado, habitualmente se realizarán en el repositorio, el cual ofrece una serie de métodos a las capas superiores (API), de forma que separa el acceso a los datos del acceso a las capas superiores.

3.5. Corrutinas

Las consultas a la base de datos deben hacerse en un hilo de ejecución diferente del hilo principal, ya que este es el encargado del renderizado de la interfaz y cualquier demora en las peticiones a la base de datos enlentecería la interfaz.

Existen muchas formas de conseguir esto: mediante corutinas, LiveData o flujos (Flow), o con otras librerías como RXJava o Guava. En nuestro caso, utilizaremos corutinas y, posteriormente, LiveData.

En la documentación de Android se definen las corutinas como «patrones de diseño de simultaneidad, que se pueden ejecutar de forma asíncrona». Por otro lado, la definición de corutina que nos ofrece la documentación propia de Kotlin comenta que las corutinas son un concepto similar a los hilos de ejecución, en el sentido que definen un bloque de código que se ejecuta de forma concurrente con el resto de código, aunque **no están vinculadas a ningún hilo en particular**, de modo que pueden suspender su ejecución en un hilo y reanudarse en otro.

Así pues, mediante corutinas vamos a poder definir tareas que se ejecutarán en segundo plano y a través de diferentes hilos de ejecución con tal de no bloquear el hilo principal.

Existen cuatro conceptos básicos cuando hablamos de corutinas: el scope, el builder, el dispatcher y withContext.

A. SCOPE (CORROUTINESCOPE)

Define un ámbito (scope) de ejecución de la corutina, el cual puede ser una corutina que se ejecute a lo largo del ciclo de vida de la aplicación completa o estar restringida a un ciclo de vida más corto, como una actividad o un fragmento. Los scope más usuales en Android son los siguientes:

- GlobalScope: Define un ámbito de ejecución global, de modo que la corutina se mantiene en ejecución durante todo el ciclo de vida de la aplicación.
- LifeCycleScope: Define un ámbito de ejecución restringido al ciclo de vida de la actividad o el fragmento, de modo que, una vez este se destruya, la corutina finalizará su ejecución, lo que evitará fugas de memoria y optimizará recursos.

- **ViewModelScope:** Define el ámbito de ejecución vinculado a un ViewModel, de modo que la corutina deja de ejecutarse cuando el ViewModel se destruye.

B. BUILDER

Los builders de corutina son métodos del Scope que nos permiten definir el código en que se ejecutará dicha corutina.

Nos centraremos en el builder por defecto, el builder launch, que define un bloque de corutina dentro del cual podemos ejecutar funciones de suspensión.

Funciones de suspensión

Son funciones que durante su ejecución bloquean la ejecución de la corutina y devuelven a esta un resultado. Su particularidad es que se pueden ejecutar tanto en el hilo actual como en uno diferente, según cómo se configuren.

Para definir una función de suspensión, incluiremos en su declaración la palabra reservada «suspend» antes de la palabra «fun». Es importante tener en cuenta que una función de suspensión únicamente podrá ejecutarse dentro de un bloque de corutina o dentro de otra función de suspensión.

C. DISPATCHERS Y WITHCONTEXT

Como hemos comentado, una función de suspensión puede ejecutarse en diferentes hilos. Esto se consigue definiendo un dispatcher que indique en qué hilo se van a ejecutar las corrutinas o las funciones de suspensión.

Cada Scope tiene asociado un dispatcher predeterminado, aunque podemos modificarlo mediante la función especial withContext. Normalmente desearemos que una corutina se ejecute en el hilo principal y las tareas bloqueantes de esta se ejecuten en otro, de modo que podamos disponer de los resultados en el hilo principal. Principalmente, contamos con tres tipos de dispatchers:

- **Dispatchers.Main:** Todo se ejecuta en el hilo principal, excepto aquellos fragmentos en los que se cambie de contexto de forma explícita mediante withContext.
- **Dispatchers.Default:** Es el dispatcher predeterminado, y se utiliza para tareas que requieren de cálculos intensivos cuando deseamos que estos se ejecuten fuera del hilo principal. Con él, se pueden lanzar tantos hilos como cores tenga la CPU.
- **Dispatchers.IO:** Se usa para aquellas tareas que implican operaciones de entrada y salida de datos, y que bloquean el proceso mientras se espera la respuesta de algún subsistema de E/S: lecturas de ficheros, peticiones de red o acceso a bases de datos, entre otros. Este Dispatcher podrá lanzar por defecto hasta 64 hilos de ejecución.

A continuación, vemos un código a modo de ejemplo de los conceptos expuestos. En primer lugar, definimos una función de suspensión del modo siguiente:

```
suspend fun funcionDeSuspension() : Tipo {  
    // Operaciones potencialmente bloqueantes  
    // ...  
    return resultado  
}
```

Observamos que únicamente hemos indicado la función como suspend. Esta solo podrá utilizarse dentro de otra función de suspensión o en un bloque de corutina. Por ejemplo:

```
GlobalScope.launch(Dispatchers.Main) {  
    // Tareas que se ejecutan en el hilo principal  
  
    val resultado=withContext(Dispatchers.Default){  
        // Tareas que se ejecutarán fuera del hilo principal  
        funcionDeSuspension()  
    }  
}
```

Como vemos, en el ámbito global se ha utilizado un builder de corrutina que se ejecutará en el hilo principal, aunque para la ejecución de la función de suspensión se cambia el contexto.

3.6. Acceso asíncrono a la base de datos

Dado que las operaciones de acceso a la base de datos pueden ser costosas y llegar a bloquear el hilo principal, las realizaremos de forma asíncrona, y para ello utilizaremos corrutinas y LiveData del modo siguiente:

- Declararemos los métodos de conveniencia que realizan operaciones CRUD sobre la base de datos (Select, Insert, Update o Delete) como funciones de suspensión. Esto afectará tanto a las clases de acceso a datos (DAO) como a los métodos del repositorio que las utilicen.
- El código de la aplicación en que se utilicen estas funciones usará bloques de corrutina para su invocación. Generalmente, si realizamos estas operaciones desde el ViewModel, utilizaremos el Scope específico para el ViewModel, así como el dispatcher específico para las operaciones de entrada y salida.
- Por otra parte, para los métodos de consulta (@Query) del DAO utilizaremos LiveData, de modo que las consultas a la base de datos sean reactivas. Así, el resultado se guarda en un LiveData, que será observable, y cada vez que se produzca un dato en la base de datos y se modifique el LiveData este lo notificará automáticamente a la interfaz.
- Debemos tener en cuenta que modificaremos mediante el método `postValue()` cualquier LiveData dentro de un hilo de ejecución que no sea el principal.

En el caso práctico extendido de este apartado veremos todos estos conceptos en acción y dotaremos a nuestra aplicación de persistencia.

4. Acceso a recursos del sistema

4.1. Intents a actividades con valor de retorno

Como sabemos, los Intents son el mecanismo que ofrece Android para invocar a otras actividades, bien de la propia aplicación, bien de una aplicación diferente. Esta invocación se realiza proporcionando el Intent en cuestión al método `.startActivity`, que inicia la actividad sin esperar ningún valor de retorno.

En ocasiones, es posible que, cuando lancemos un Intent, esperemos algún valor de retorno, como, por ejemplo, una foto de la aplicación de la cámara. En estos casos el método `.startActivity` no es suficiente. Hasta la versión 1.2.0 del componente `androidx.activity:activity` (`activity-ktx` para Kotlin), que gestiona las actividades, se utilizaban los métodos `startActivityForResult` y `onActivityResult` para invocar un Intent con valor de retorno y recoger el resultado asíncronamente.

A partir de la versión 1.2.0 del componente de actividades estos métodos se declaran obsoletos, en favor de la API `ActivityResult` que, entre otras cosas, resuelve algunas de las deficiencias que presentaba el modelo anterior cuando trabajaba con fragmentos. Esta API proporciona componentes para registrar, iniciar y procesar resultados de los Intents.

REGISTRO DE CALLBACKS PARA OBTENER RESULTADOS

Recordemos que los Intents trabajan de forma asíncrona, de modo que no podemos lanzarlos y esperar un resultado como si se tratase de invocaciones síncronas. El mecanismo que usaremos para gestionar este asincronismo serán los callbacks.

Ahora bien, la forma de declarar estos callbacks será un tanto especial. Cuando iniciamos otra actividad, ya sea para obtener un resultado o no, es posible que, por falta de recursos, el sistema elimine la propia actividad que lanzó el Intent. Cuando la segunda actividad devuelva los resultados a la primera, esta deberá tener disponibles los callbacks antes incluso de lanzar el método `onCreate` de la actividad.

La API de `ActivityResult` propone registrar los callbacks, de forma incondicional cada vez que se cree la actividad, en la misma definición de la clase separando los callbacks que esperan el resultado de los métodos del ciclo de vida.

Para registrar un callback desde una actividad o un fragmento, se usa la API `registerForActivityResult()`, que recibe dos parámetros:

- **Un objeto de tipo `ActivityResultContract`** que se trata de un contrato en que se especifican las conversiones de tipos desde y hacia los Intents; es decir, qué entrada necesitamos para producir el resultado y qué tipo de resultados esperamos. La API proporciona una serie de registros predeterminados para acciones básicas como obtener una foto o solicitar permisos, a pesar de que podríamos crear contratos personalizados.
- **Un objeto de tipo `ActivityResultCallback`** consistente en una interfaz que contiene el método `onActivityResult()` con el callback que lanzará al disponer de los resultados de la actividad.

El objeto que nos devolverá `registerForActivityResult()` es de tipo `ActivityResultLauncher`, que hará de cargador y ofrece un método `launch` para lanzar la actividad.

En general, el funcionamiento será el siguiente:

1. Se registra el callback para obtener el cargador, donde se describe qué hacer cuando se reciba el resultado (por ejemplo, si pedimos una foto de la cámara, actualizamos un `ImageView`).
2. Despues, con este cargador se lanza el Intent (por ejemplo, obtener una foto).
3. Android abrirá la actividad destino (por ejemplo, la aplicación de la cámara), capturará la imagen y devolverá la información solicitada a la actividad invocando el método '`onActivityResult()`', en el cual hemos definido los callback.

Es posible invocar a `registerForActivityResult()` de forma segura antes de crear el fragmento o la actividad, de modo que se puede utilizar directamente en la sección de declaración de variables miembro, para que así ya estén disponibles cuando se cree la actividad.

En general, el esquema para hacer el registro sería el siguiente:

```
private val cargadorActividad = registerForActivityResult(StartActivityForResult()) {  
    result: ActivityResult -> {  
        // Cuerpo de la función de callback, donde disponemos  
        // de los resultados a Result.  
    }  
}
```

Debemos tener en cuenta que, aunque parezca que el método registerForActivityResult tenga un único argumento, su segundo argumento es el callback, expresado como función lambda fuera del paréntesis y siguiendo la guía de estilo de Kotlin.

En el Caso práctico 1 veremos un ejemplo sencillo para enviar datos a una actividad y que esta nos devuelva un resultado utilizando esta API.

4.2. Intents de uso común

Cuando queremos llevar a cabo una acción genérica del dispositivo, como hacer una foto, solemos utilizar Intents implícitos, en los que no especificamos el componente de aplicación que se iniciará, sino únicamente la acción que deseamos. Con este Intent, el sistema resuelve qué aplicación puede gestionar esta acción e inicia la actividad correspondiente. En caso de que haya más de una aplicación que pueda gestionar el Intent, aparecerá un diálogo para escoger qué aplicación utilizar.

Android dispone de un gran abanico de Intents implícitos. En este apartado solo vamos a ver algunos de ellos, pero podemos consultar el artículo *Intents comunes*, de la documentación de Android, referenciado en los enlaces, para ver todas las posibilidades e incluso saber cómo crear filtros para que nuestras aplicaciones respondan a acciones genéricas.

Entre estas acciones, podemos destacar ACTION_IMAGE_CAPTURE y ACTION_VIDEO_CAPTURE, para capturar una imagen o un vídeo desde una aplicación de cámara, abrir un tipo de fichero concreto con ACTION_OPEN_DOCUMENT, ACTION_VIEW, para mostrar una ubicación en un mapa o abrir una web, o ACTION_PICK, para seleccionar un contacto de la agenda.

ACCEDIENDO A LOS FICHEROS DEL DISPOSITIVO

A modo de ejemplo, veremos cómo obtendríamos una imagen desde el dispositivo mediante la acción ACTION_OPEN_DOCUMENT. Con esta acción, podemos solicitar abrir un fichero gestionado por otra aplicación. Esta acción nos devuelve una URL sobre la que nuestra aplicación tendrá permiso de lectura al fichero. La principal diferencia con la otra acción para obtener contenidos ACTION_GET_CONTENT es que esta última crea una copia del recurso en nuestra aplicación. Con ACTION_OPEN_DOCUMENT, esta copia no es necesaria.

En primer lugar, definiríamos un cargador del modo siguiente:

```
private val cargadorImagen = registerForActivityResult(ActivityResultContracts.StartActivityForResult()) { result: ActivityResult ->  
    val resultCode = result.resultCode  
    val resultData: Intent? = result.data  
  
    if (resultCode == Activity.RESULT_OK) {  
        if (resultData != null) {  
            val uri: Uri? = resultData.data  
            try {  
                // Procesamos la respuesta, por ejemplo, establecer la URI  
            } catch (e: Exception) {  
                Log.e("CargadorImagen", "Error procesando la respuesta", e)  
            }  
        }  
    }  
}
```

```
// recibida como la imagen de un ImageView
binding.imageView.setImageURI(uri)
} catch (e: IOException) {
    e.printStackTrace()
}
}
}
```

Como vemos, sigue el esquema propuesto en el punto anterior, y el resultado se obtiene a través de un Intent que contiene la URI de la imagen como datos, la cual se puede asignar directamente a un ImageView.

Por lo que respecta al código para lanzar el Intent, cuando quisiéramos cargar la imagen simplemente deberíamos crear el Intent para obtener la imagen y utilizar el cargador creado para lanzarlo y obtener el resultado:

```
val openDocumentIntent = Intent(Intent.ACTION_OPEN_DOCUMENT)
openDocumentIntent.type = "image/*"
cargadorImagen.launch(openDocumentIntent)
```

Observemos algunos detalles:

- El Intent se ha definido con la acción ACTION_OPEN_DOCUMENT, lo que sería equivalente a definir un Intent vacío (`val intent=Intent()`) y añadirle después la acción:
`intent.setAction(Intent.ACTION_OPEN_DOCUMENT)`.
- Hemos especificado el tipo MIME del fichero que queremos cargar incorporando el atributo type al Intent: `intent.type = "image/*"`
- Este código también podría haberse escrito mediante la función de ámbito apply:

```
val intent = Intent().apply {
    setAction(Intent.ACTION_OPEN_DOCUMENT)
    type = "image/*"
}
```

IMPORTANTE

Acerca de los permisos. El framework de acceso al almacenamiento nos permite interactuar con proveedores de documentos, ya sean volúmenes externos (como una tarjeta SD), almacenamiento interno o la nube. Este framework permite que los usuarios hagan uso de un selector del sistema para seleccionar los documentos específicos, y que la aplicación los pueda manipular. Puesto que el propio usuario elige los ficheros a los que puede acceder la aplicación, **no se requiere ningún permiso del sistema**.

4.3. Obteniendo una imagen de la cámara

Ahora vamos a ver cómo hacer uso de la cámara para obtener fotos, en lugar de seleccionar imágenes del dispositivo.

Una de las principales diferencias respecto al caso anterior residirá en los permisos. Además del acceso a la cámara, se deberá poder escribir en el sistema de archivos para guardar las fotografías.

Por este motivo, para esta tarea se requerirán dos cargadores: uno para comprobar los permisos y otro para hacer las fotos.

Pero antes, vamos a ver cómo configurar un proveedor de almacenamiento para nuestras apps.

A. PREPARANDO UN PROVEEDOR DE ALMACENAMIENTO

Los proveedores de contenidos eran, junto con las actividades, los servicios y los receptores de anuncios, uno de los principales componentes de Android. La función principal de estos, como su nombre indica, es proveer de contenidos a las aplicaciones y así poder permitir compartir contenido.

Los contenedores de contenidos se representan mediante la clase ContentProvider, de la cual deriva FileProvider, que nos permite acceder a los ficheros de nuestra aplicación con una URI de tipo content:// y teniendo en cuenta los permisos.

Para utilizar un proveedor de almacenamiento en nuestra aplicación, igual que hacemos con las actividades, deberemos configurar este en el fichero AndroidManifest.xml, mediante el código:

```
<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>
```

Como vemos, usamos la etiqueta provider y, de ella, destacamos los atributos siguientes:

- El atributo android:name especifica el tipo de proveedor, en este caso FileProvider.
- El atributo android:authorities especifica la lista de autoridades [direcciones URI para localizar los datos] que proporcionarán los datos del proveedor. Dado que debe tratarse de un valor único, es buena idea obtenerlo a partir del ID de la aplicación y finalizarlo con .provider.
- El atributo exported = "false" indica que otras aplicaciones con un ID diferente al de la aplicación no pueden acceder a los contenidos del proveedor.
- El atributo android:grantUriPermissions = "true" permite otorgar permisos de forma temporal sobre el proveedor a quienes lo soliciten.

Además de estos atributos, hay que definir los directorios que contendrán los ficheros disponibles en el proveedor. Esto se hará con una lista de <paths/> que indicaremos en otro fichero XML (res/xml/file_paths.xml). Para vincular este fichero a nuestro proveedor se utiliza el elemento <meta-data> dentro del <provider>. Tal y como hemos visto más arriba, indicamos el nombre del recurso android.support.file_PROVIDER_PATHS y la referencia al fichero XML @xml/provider_paths.

Hecho esto, habrá que crear el fichero res/xml/file_paths.xml, con los paths a utilizar, el cual contendrá un elemento raíz de tipo <paths> en el que se especifiquen las diferentes ubicaciones para los recursos, con la sintaxis siguiente:

```
<elemento-de-ruta name="nombre" path="path" />
```

Los diferentes elementos de ruta indican al FileProvider qué URI de contenido se le pueden solicitar, haciendo referencia tanto al almacenamiento externo como al interno. Veamos en la tabla siguiente los diferentes elementos de ruta, a qué recursos hacen referencia y qué métodos podemos utilizar para obtener las URI:

Elemento de ruta	Recursos	Métodos de acceso desde código
<files-path/>	Ficheros en el directorio de almacenamiento interno de la aplicación.	<code>Context.getFilesDir()</code>
<cache-path/>	Caché del almacenamiento interno de la aplicación.	<code>getCacheDir()</code>
<external-path/>	Raíz del área de almacenamiento externo.	<code>Environment.getExternalStorageDirectory()</code>
<external-files-path/>	Raíz del área de almacenamiento externo de la aplicación.	<code>Context.getExternalFilesDir()</code>
<external-cache-path/>	Raíz de la caché del área de almacenamiento externo de la aplicación.	<code>Context.getExternalCachesDir()</code>
<external-media-path/>	Raíz del área de almacenamiento externo de los medios [media]. API>=21	<code>Context.getExternalMediaDirs()</code>

Por ejemplo, si especificamos:

```
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <external-filas-path name="external_filas" path=". />
</paths>
```

en nuestro código, necesitaremos del método `getExternalFilesDir()` para acceder a los ficheros de nuestra aplicación en el almacenamiento externo.

B. OBTENIENDO PERMISOS PARA HACER FOTOS

Para poder utilizar la cámara, necesitamos dos permisos: para acceder al dispositivo de la cámara y para acceder al almacenamiento. En primer lugar, para que la aplicación pueda solicitar permiso al usuario, deberemos indicarlo en el fichero AndroidManifest.xml con las líneas siguientes:

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

Una vez con el Manifest configurado, ya podemos realizar solicitudes de permiso desde nuestro código mediante la acción `ACTION_REQUEST_PERMISSIONS`, a través de un `ActivityResultLauncher`. Así pues, deberemos definir un cargador siguiendo el esquema habitual, pero ahora utilizando un contrato de tipo `ActivityResultContracts.RequestMultiplePermissions()`, que nos sirve para lanzar la acción de comprobación de permisos.

De este modo, el registro del callback tendrá el esquema siguiente:

```
val CameraPermissionRequest = registerForActivityResult(
    ActivityResultContracts.RequestMultiplePermissions())
{
    // callback..
}
```

Este registro nos ofrecerá un método `launch` para cargar la acción, aunque ahora el contrato `ActivityResultContracts.RequestMultiplePermissions()` requerirá que le proporcionemos al método un vector de Strings con los permisos que queremos solicitar. En nuestro caso, el permiso para acceder a la cámara y el permiso para escribir en el almacenamiento externo quedarían del modo siguiente:

```
cameraPermissionRequest.launch(arrayOf(  
    android.Manifest.permission.CAMERA,  
    android.Manifest.permission.WRITE_EXTERNAL_STORAGE))
```

El cargador realizará la comprobación de permisos y, cuando esta esté lista, se invocará el callback indicado en el cargador. En este callback recibiremos una lista de pares, con los permisos solicitados y la resolución sobre si estos han sido concedidos o no por el usuario. Esta lista tendrá la forma siguiente:

```
{android.Manifest.permission.CAMERA=true,  
 android.Manifest.permission.WRITE_EXTERNAL_STORAGE=true}
```

Así pues, en el callback podremos comprobar si los permisos se han concedido o no para poder tomar imágenes con la cámara. Con esto, el esquema completo quedaría del modo siguiente:

```
val CameraPermissionRequest = registerForActivityResult(  
    ActivityResultContracts.RequestMultiplePermissions())  
    ) { permissions ->  
        if (permissions.getOrDefault(android.Manifest.permission.CAMERA, false)  
            && permissions.getOrDefault(  
                android.Manifest.permission.WRITE_EXTERNAL_STORAGE, false) )  
            // Acciones si se han concedido los permisos  
  
        else {  
            // Acciones si no se han concedido los permisos  
        }  
    }
```

Para comprobar los permisos se usa el método `getOrDefault` del diccionario. Este método devuelve el valor de una clave del diccionario (Map), en nuestro caso «true» o «false», y retorna un valor por defecto si la clave no está en el diccionario, en este caso «false».

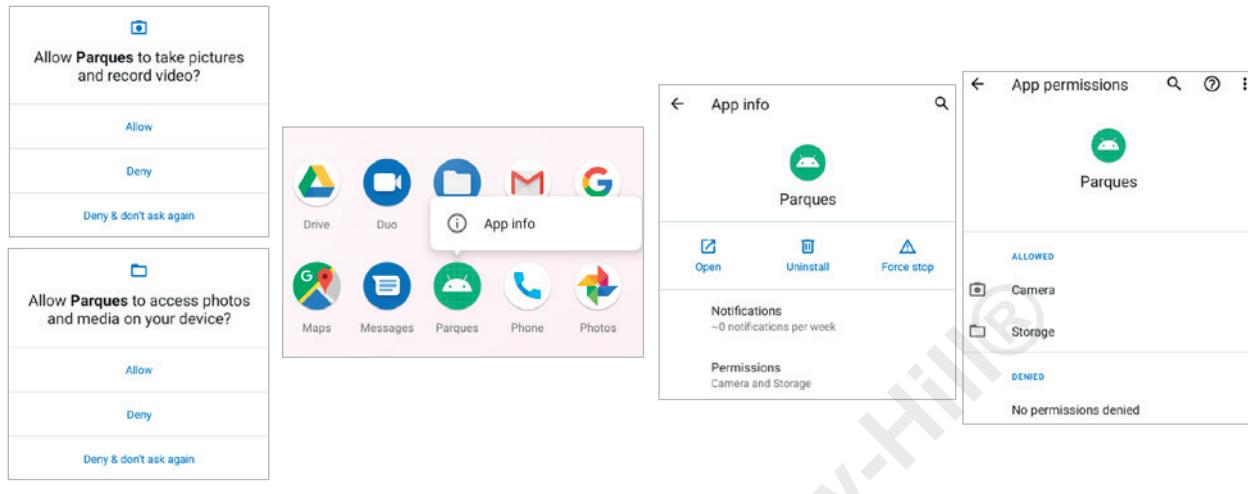
Para utilizar el método `getOrDefault`, necesitamos que el nivel mínimo del API que utilizamos sea el 24. Por defecto, Android Studio genera los proyectos con el nivel mínimo del SDK a 21, por lo que deberíamos modificar este parámetro en el fichero `build.gradle` del módulo.

```
defaultConfig {  
    ...  
    minSdk 24  
    ...  
}
```

IMPORTANTE

La primera vez que se hace esta comprobación de recursos, estos se solicitan al usuario, y el sistema recuerda esta decisión para no pedirlos cada vez.

Una vez hecho esto, podemos consultar y modificar los permisos de la aplicación haciendo un clic largo sobre el ícono del cargador de esta en el menú de aplicaciones.



C.USANDO LA CÁMARA

En la sección anterior hemos visto cómo solicitar permisos de acceso a la cámara y al almacenamiento, pero todavía no hemos visto cómo acceder a esta para tomar fotografías.

La acción que debe invocarse para tomar una fotografía es android.media.action.IMAGE_CAPTURE, y el contrato que requerirá el cargador para esta acción será de tipo ActivityResultContracts.TakePicture. Este contrato nos pedirá una URI, que será la ubicación del fichero con la imagen capturada.

Para obtener una URI correspondiente al almacenamiento externo, usaremos el método getExternalFilesDir, que nos proporcionará un path en el área de almacenamiento externo, acorde a lo indicado en el elemento de ruta <external-files-path>.

Para ello, el código en general será el siguiente:

- Obtención del directorio de imágenes de la aplicación del almacenamiento externo:
`val dir=getExternalFilesDir(Environment.DIRECTORY_PICTURES)`
- Creación de un fichero temporal en este directorio, con un prefijo y una extensión (por ejemplo, jpg):
`val file=file.createTempFile("prefijo", ".jpg", dir)`
- Obtención de la URI del fichero a través del proveedor de almacenamiento ('FileProvider'):
`uriFichero=FileProvider.getUriForFile(this,
applicationContext.packageName + ".provider", file)`
- Y, finalmente, lanzamiento del cargador para que realice la acción de tomar la fotografía:
`cargadorCamera.launch(uriFichero)`

El cargador para capturar una imagen de la cámara tendrá el aspecto siguiente:

```
private val cargadorCamera = registerForActivityResult(  
ActivityResultContracts.TakePicture())  
) { result:Boolean ->  
    if (result) {  
        // ... Imagen disponible uriFichero  
    }  
}
```

Como vemos, utilizamos el contrato para lanzar una foto (TakePicture), y el resultado que recibimos en el callback es un valor de tipo lógico. Este valor nos indica si la imagen se ha tomado con éxito o no. En caso de ser «true», ya tendremos disponible la imagen en la URI que proporcionamos en la invocación al cargador (método launch).

Propiedad de McGraw-Hill®



Unidad 4.

Desarrollo de aplicaciones móviles multiplataforma.

Flutter



Propiedad de McGraw-Hill®



1. Introducción a Flutter

1.1. ¿Qué es Flutter?

Flutter es un SDK y un framework de código abierto creado por Google en 2017 y orientado al desarrollo de aplicaciones multiplataforma (Android, iOS, escritorio o web), con la idea de obtener, con el mismo código de base, aplicaciones compiladas en el código nativo específico de cada plataforma.

A pesar de que el núcleo está creado con C++, Flutter utiliza el lenguaje de programación Dart, desarrollado también por Google, en 2011, como alternativa a JavaScript y con la idea de suplir algunas deficiencias de este. Dart se ejecuta sobre la máquina virtual DartVM, que permite dos tipos de compilación: JIT y AOT. La compilación JIT (just-in-time) tiene lugar durante la propia ejecución en la máquina virtual de Dart, mientras que la compilación AOT (ahead-of-time) o compilación con anticipación se realiza de forma previa a la ejecución y al lenguaje nativo de cada plataforma.

Otro de los pilares de Flutter es el motor de renderizado 2D Skia. Flutter no utiliza los componentes de interfaz nativos de cada plataforma, sino sus propios componentes, y los renderiza mediante Skia, por lo que ofrece un aspecto y un comportamiento prácticamente equiparable a los nativos.

Todo ello aporta a Flutter las características siguientes:

- Stateful hot reload, o recarga en caliente con estado, es gracias a la compilación JIT y permite realizar cambios en el código y **visualizarlos de forma instantánea** en las vistas sin necesidad de volver a cargar todo el contexto gráfico.
- **Rendimiento prácticamente nativo**, ya que el código compilado de modo OAT genera código nativo.
- Permite crear, de forma sencilla y eficaz, **aplicaciones con interfaces gráficas vistosas**, expresivas y con un comportamiento prácticamente nativo gracias a Skia. Este motor permite el renderizado de hasta 60 fps (fotogramas o frames por segundo), lo que implica, además, animaciones de gran calidad.

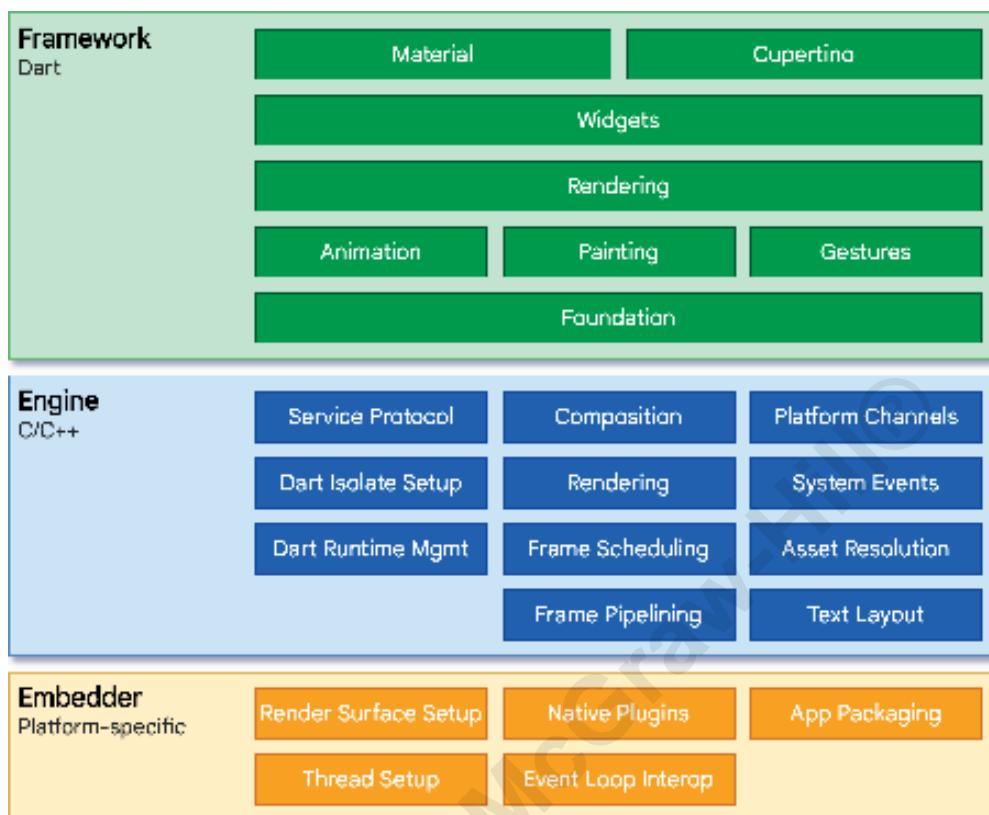
De todos modos, debemos tener en cuenta que no estamos haciendo desarrollos puramente nativos con las herramientas proporcionadas por Google o Android, de modo que:

- No disponemos inmediatamente de las últimas características de los sistemas operativos, ya que estas se incorporan con prioridad a los entornos nativos.
- Las aplicaciones multiplataforma son más pesadas y menos escalables que las nativas, ya que deben estar preparadas para interactuar con diferentes sistemas operativos.
- Tienen un rendimiento ligeramente inferior a las aplicaciones nativas puras, dado que no están tan optimizadas como las compiladas con las herramientas nativas.

Una opción que también es viable, teniendo en cuenta los pros y los contras, son los desarrollos híbridos, con una parte común realizada en Flutter y otra más específica desarrollada de forma nativa.

1.2. Arquitectura de Flutter

La arquitectura de Flutter se compone de un sistema extensible de capas independientes, donde cada capa superior depende de la capa subyacente. Podemos ver esta arquitectura en el diagrama siguiente, extraído de la documentación oficial de Flutter:



Fuente: Imagen publicada como obra cultural gratuita, bajo licencia Creative Commons Attribution 4.0 International (CC BY 4.0) por Flutter.dev: <https://docs.flutter.dev/resources/architectural-overview>

- **Framework.** La capa superior, desarrollada completamente en Dart, proporciona a los desarrolladores un framework moderno y reactivo al lenguaje. Consta de cuatro subcapas:
 - La superior, con las bibliotecas Material y Cupertino, que implementan los controles en los diferentes lenguajes de diseño de Android e iOS.
 - La de widgets, que implementa de forma reactiva los diferentes tipos de controles y su composición.
 - La de representación, que gestiona el árbol de componentes y se encarga de su representación.
 - Las clases fundamentales básicas y las bibliotecas de animación, colores y renderizado.
- **Engine o motor de Flutter.** El framework se sustenta en el motor de Flutter, desarrollado en C y C++. Esta capa es la responsable del renderizado de cada frame, y proporciona la implementación a bajo nivel de la API principal de Flutter: el motor gráfico Skia, la representación de texto, la gestión de la entrada y la salida, o el soporte a la accesibilidad, entre otros.
- **Embedder o integrador.** Principalmente se encarga del empaquetado en formato nativo y de la integración de la aplicación en el sistema operativo subyacente.

1.3. Herramientas del SDK

El SDK de Flutter puede descargarse de su propia web, tal y como veremos en el Caso práctico 1. Flutter organiza el SDK en torno a canales o ramas de desarrollo. Principalmente, disponemos de dos canales:

- **El canal beta**, que incorpora las nuevas funcionalidades y las compatibilidades con cada plataforma, para que los desarrolladores las prueben.
- **El canal estable**, donde se incorporan las nuevas funcionalidades una vez han sido probadas y se consideran estables o con poca probabilidad de producir errores.

Como veremos en el caso práctico, la instalación consiste en descargar el SDK y configurar correctamente el PATH para que podamos acceder a las herramientas que este proporciona.

La herramienta principal del SDK es el comando **flutter**, una potente herramienta de la línea de comandos a través de la cual podremos acceder a todas las funcionalidades del SDK.

Los principales usos de este comando serán:

- **flutter help**. Muestra la información de ayuda del comando, así como los diferentes subcomandos que podemos utilizar, organizados en tres categorías principales: herramientas del SDK, herramientas de creación y construcción de proyectos, y herramientas para el acceso a los dispositivos virtuales.
- **flutter doctor**. Para su correcto funcionamiento, Flutter necesita de otras herramientas, como los SDK de Android o iOS, Java o emuladores. Para comprobar todos estos requisitos, Flutter proporciona la herramienta **flutter doctor**, que se encarga de analizar nuestro sistema y comprobar qué componentes nos faltan para poder utilizar el SDK. Cuando lanzamos la herramienta, se presenta un informe con aquellos componentes que han instalarse o configurarse y cómo hacerlo.
- **flutter create, flutter build, flutter run o flutter clean**. Existen varios subcomandos para la gestión de proyectos, pero estos serán los que más utilizaremos para la creación, construcción, ejecución o limpieza del proyecto, respectivamente.
- **flutter devices, flutter emulators**. Entre las diferentes herramientas de gestión de dispositivos, encontramos **flutter devices** y **flutter emulators**, que sirven para mostrar los diferentes dispositivos conectados o emulados en los que podemos lanzar nuestras aplicaciones.

1.4. Integración con el IDE

Aunque Flutter requiere de algunas herramientas de Android Studio, podemos utilizar diferentes editores de código. El equipo de Flutter oficialmente ofrece integración de este con Android Studio/IntelliJ, Visual Studio Code y Emacs.

La integración de Flutter con los diferentes IDE se realiza a través de plugins. Veamos el caso para los editores Android Studio y Visual Studio Code.

A. INTEGRACIÓN DE FLUTTER CON VISUAL STUDIO CODE

Los plugins que necesitaremos principalmente para el desarrollo de Flutter en Visual Studio Code serán los siguientes:

- **Flutter**, que implementa el soporte para la edición, refactorización, ejecución y recarga en caliente de aplicaciones móviles desarrolladas en Flutter.
- **Dart**, que ofrece el soporte para el lenguaje Dart en VS Code, con herramientas para la edición, la depuración y el resaltado de sintaxis del lenguaje. Este plugin es dependencia directa del plugin de Flutter.

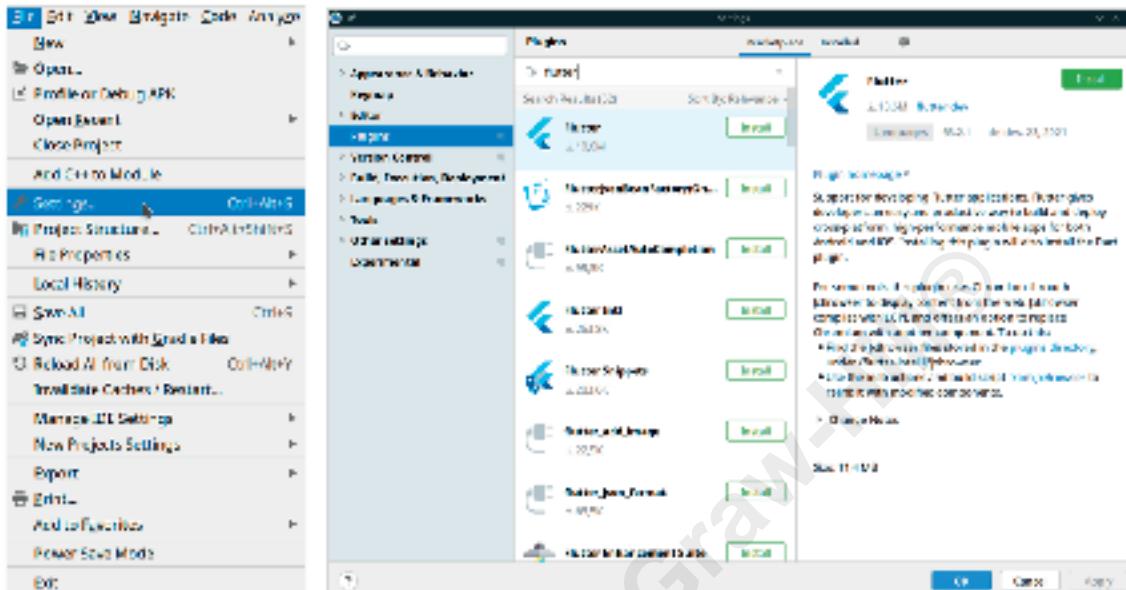
Aunque estos son los plugins necesarios para el desarrollo en Flutter, existen otros que nos pueden resultar útiles, como:

- **Pubspec Assist**, para incorporar fácilmente las dependencias de nuestro proyecto al fichero de configuración,
- **Awesome Flutter Snippets**, con plantillas de clases y métodos que permiten un desarrollo más ágil, ayudando en la creación de componentes.

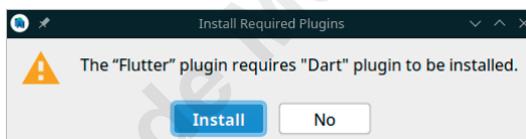
Existen también algunos paquetes de extensiones, como Flutter ExtPack, que únicamente arrastra Flutter y Dart, y otros, como Flutter Development Extensions Pack, Ultimate Flutter Extension Pack y Complete Flutter Extension Pack, que incluyen una gran cantidad de extensiones relacionadas con el desarrollo en Flutter.

B. INTEGRACIÓN DE FLUTTER CON ANDROID STUDIO

Para desarrollar con Flutter en Android Studio necesitaremos un complemento específico, que podemos encontrar en el menú de Ajustes (*File > Settings*), dentro de la sección *Plugins*.



Este plugin requiere del plugin de Dart para Android Studio, tal y como se indica en su instalación:



1.5. Proyectos en Flutter

Para crear un nuevo proyecto en Flutter, se utiliza el comando **flutter create**. Por ejemplo:

```
flutter create nombre_proyecto
```

Con esto se genera el esqueleto de una aplicación con todos los parámetros por defecto, una aplicación que compilará a Android e iOS con los lenguajes Kotlin y Swift, respectivamente. Si, por ejemplo, deseásemos generar una aplicación únicamente para Android, podríamos utilizar el comando siguiente:

```
flutter create -t app --platforms android --android-language kotlin nombre_app
```

Por otra parte, tanto VS Code como Android Studio permiten la creación de proyectos desde su interfaz, como veremos en los casos prácticos extendidos.

ESTRUCTURA DEL PROYECTO

Un proyecto en Flutter se organiza en diferentes directorios y ficheros de configuración y código fuente. Los más importantes son los siguientes:

- El directorio `android`, con el proyecto completo de Android, con su estructura habitual (módulo `app`, scripts de Gradle, Manifests, etc.). Una vez hecha la compilación del proyecto, nuestro código Flutter se integrará en este proyecto Android, de modo que podremos abrirlo de forma independiente, ejecutarlo y depurarlo como una aplicación Android.
- El directorio `ios`, con el proyecto completo para iOS. Si deseásemos abrir la aplicación para su compilación, modificación o depuración, necesitaríamos XCode.

- Directorio lib, que contiene el código fuente Dart de nuestra aplicación Flutter, que será compilado a código específico de Android e iOS. La clase principal será main.dart y será el punto de entrada a nuestra aplicación.
- Fichero pubspec.yaml, que es el archivo de configuración del proyecto Flutter en formato YAML, donde se incluyen las dependencias a bibliotecas y se especifican recursos de imágenes, fuentes, audio o vídeo.

Propiedad de McGraw-Hill®

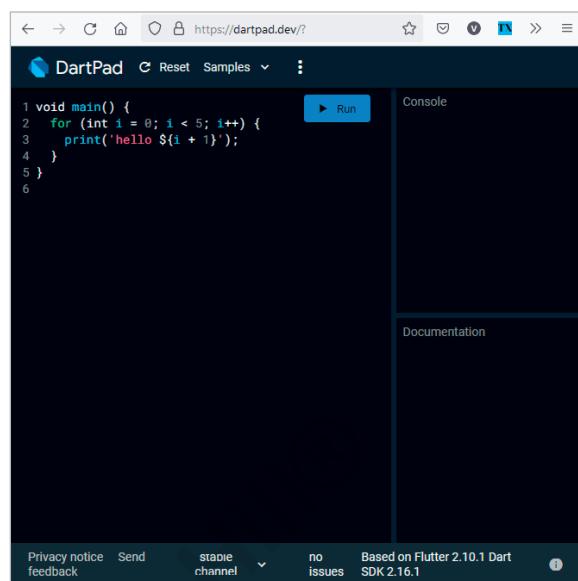
2. El lenguaje de programación Dart

2.1. Trabajando con Dart

El SDK de Flutter integra la máquina virtual de Dart, mediante la cual vamos a poder lanzar nuestros programas. Un programa Dart no es más que un archivo con extensión `.dart` ejecutable con:

```
$ dart nombre_de_fichero.dart
```

Alternativamente, también podemos trabajar con el playground de Dart, que, además, tiene la ventaja de incluir en la misma ventana el editor de código, con resaltado de sintaxis y detección de errores, la salida del programa, documentación y algunos ejemplos, tanto con Dart como con Flutter. Para más información al respecto, podemos consultar la sección *Material de trabajo/Enlaces*.



The screenshot shows the DartPad interface at <https://dartpad.dev/>. The code editor contains the following Dart code:

```
void main() {
  for (int i = 0; i < 5; i++) {
    print('Hello ${i + 1}');
  }
}
```

The 'Run' button is highlighted. To the right, there's a 'Console' panel which currently displays nothing. Below the editor, there's a 'Documentation' panel that also appears empty. At the bottom of the page, there are links for 'Privacy notice', 'Send feedback', 'stable channel', 'no issues', and 'Based on Flutter 2.10.1 Dart 2.16.1'.

2.2. Conceptos básicos

Como otros muchos lenguajes, Dart utiliza la función `main` como punto de entrada en un programa. Partiremos de un clásico «Hola mundo!» personalizado:

```
void main(List<String> args) {
  // Ejemplo de Hola Mundo!
  if (args.length > 0)
    print("Hola ${args[0]}");
  else
    print("Hola mundo!");
}
```

Vemos algunos detalles de Dart en este código:

- La función `main` no devuelve ningún valor, y la palabra reservada `void` suele utilizarse, aunque no es obligatoria.
- Esta función principal puede recibir argumentos en forma de lista de Strings. Observad que la sintaxis para definir estos es similar a la de Java. En caso de que no vayamos a utilizar los argumentos, podemos utilizar directamente `void main()`.
- Los comentarios se expresan como en otros lenguajes, con `//` y `/* ... */`, según sean de una línea o multilínea.
- Para mostrar un mensaje por pantalla utilizamos la orden `print()`. Cuando utilizamos variables dentro del texto, a pesar de que podemos utilizar el operador `+` para encadenar literales, se recomienda el uso de la interpolación de cadenas (string interpolation), mediante el símbolo del `$` (`$_variable`) o `{}$` si tenemos que acotar la interpolación (`${variable.propiedad}`).
- Las sentencias Dart acaban con punto y coma `;`.
- Podemos acceder a las listas directamente con el operador `[]`.



2.3. Variables y tipos de datos

Los tipos de datos soportados por Dart son los siguientes:

- Numéricos: enteros (`int`) y decimales (`double`)
- Cadenas de caracteres: `String`
- Valores lógicos: `bool`
- Colecciones de objetos: listas (`List`), conjuntos (`Set`) y diccionarios (`Map`)

Para declarar una variable con Dart, podemos utilizar `var`, de forma que el tipo de dato se asigne de forma automática, o indicar directamente el tipo. Una vez se asigna un tipo a una variable, este ya no se puede modificar. Vemos algunos ejemplos:

```
var dia1='jueves'; // Infiere el tipo a String
String dia2='martes'; // Definimos un String
int numero=42; // Definimos un entero
bool tenemos_clase=true; // Definimos un valor lógico
```

Para definir constantes utilizamos la palabra reservada `const`, con la que declaramos un valor en tiempo de compilación que será inmutable y no podrá ser reasignado:

```
const modulo='PMDD'
```

Además, también podemos declarar datos como `final`, para indicar que no podrán ser reasignados. La diferencia con las constantes es que un objeto declarado como final, aunque no pueda ser reasignado, es mutable, es decir, puede cambiar sus propiedades internas. Por ejemplo, si definimos una lista como constante, no podremos añadirle elementos, pero, si la declaramos como final, aunque no podamos asignarle otra lista, podremos modificar elementos.

A. NULL SAFETY Y TRATAMIENTO DE LOS NULOS

Dart es un lenguaje de tipado seguro; esto significa que, cuando declaramos una variable de algún tipo, el compilador garantiza que los valores asignados sean de ese tipo. A pesar de que el tipado es obligatorio, indicar el tipo de una variable es opcional, puesto que, si no se hace, su tipo se infiere a partir de los valores.

Desde la versión 2.12 (marzo de 2021), Dart soporta también Null Safety, de modo que, por defecto, una variable no podrá contener valores nulos a no ser que se especifique lo contrario, ahorrando así los problemas derivados de valores nulos.

Veamos los diferentes operadores con que podemos tratar los nulos con Dart:

Operador	Descripción	Ejemplo
Declaración de valor nullable (?)	Indica explícitamente que una variable puede contener valores nulos.	<code>int? variable=null;</code>
Operador de aserción nula (!)	Se utiliza cuando queremos asignar un nullable a una variable no nullable. Con esto le indicamos a Dart que somos conscientes de ello.	<code>int variable2=variable1!</code>
Operador nulo (??)	Devuelve el valor que le indicamos a la parte izquierda siempre y cuando no sea nulo. En caso contrario, nos devolverá la expresión de la derecha.	<code>var nombre; print(nombre ?? "Anónimo");</code>
Asignación consciente de nulos (null aware assignment) (??=)	Asigna un valor a una variable que tenga valor nulo.	<code>variable1 ??= 10; // Asignamos 10 en el caso que sea null</code>
Acceso consciente de nulos (null-aware access) (?.)	Evita que se lance una excepción cuando se accede a una propiedad o un método de un objeto que puede ser nulo.	<code>String? cadena; print (cadena?.length); // Devuelve null</code>

B. LISTAS

Dart utiliza el principio del *mismo nivel de abstracción*, que indica que un código limpio no tiene que mezclar instrucciones de alto nivel con instrucciones de bajo nivel en la implementación de la lógica, para así facilitar la lectura del código.

Podemos decir que Dart es un lenguaje de alto nivel que no recurre a estructuras de bajo nivel, como puedan ser los vectores, sino que directamente utiliza listas para representar colecciones ordenadas de elementos.

Aunque podemos utilizar las listas en Dart como si se tratara de vectores, hay funciones de más alto nivel que nos permiten trabajar más cómodamente con ellas.

En el fichero descargable *listas.dart* dispones de ejemplos con diferentes formas de definir y de tratar listas.

C. SETS (CONJUNTOS)

Otra colección de elementos interesantes de Dart son los Sets o conjuntos, que, a diferencia de las listas, no mantienen los elementos indexados y evitan elementos duplicados.

En el fichero descargable *sets.dart* dispones de ejemplos con Sets.

D. MAPS (DICCCIONARIOS)

Un diccionario es una estructura de datos que almacena pares clave-valor, como un JSON, y se expresa entre llaves:

```
Map notas;  
  
Notas = {"PMDM": 8, "AD": 9, "PSP": 9, "DI": 7};
```

En el fichero descargable *maps.dart* dispones de algunos ejemplos con declaraciones y manipulación de Maps.

En cuanto al **mapeado de estructuras**, Dart permite utilizar el método map para transformar unas colecciones en otras. Con este método obtenemos otro mapa con cada uno de los elementos transformados. Para ello, habrá que proporcionarle una función anónima que se aplicará a cada uno de los elementos para crear el elemento nuevo.

En el fichero descargable *mapeado.dart* dispones de un ejemplo de mapeado de estructuras.

2.4. Funciones en Dart

Dart admite funciones que no estén vinculadas a un objeto como método. La declaración de funciones se hace de forma muy parecida a C o Kotlin:

```
[TipoRetorno] NombreFuncion( [ListaArgumentos] ) {  
    // Cuerpo de la función  
    ...  
}
```

Como vemos, tanto el tipo de retorno como la lista de argumentos son opcionales. Si no indicamos el tipo de retorno, se asume que es **void**.

A. FUNCIONES ANÓNIMAS Y FUNCIONES FLECHA

Dart admite funciones sin nombre o funciones anónimas, que no pueden ser invocadas pero que se pueden utilizar como argumentos para otras funciones.

Por su parte, las funciones flecha o arrow functions permiten abreviar la declaración de una función cuando esta consta únicamente de una línea, de modo que no utilizan ni las llaves ni la palabra return.

En el fichero descargable *funciones_anonimas_flecha.dart*, dispones de algunos ejemplos.

B. ARGUMENTOS OBLIGATORIOS, OPCIONALES Y CON NOMBRE

Las funciones Dart admiten tres tipos de argumentos:

- Argumentos **obligatorios**.
- Argumentos **opcionales**, que indicaremos con `[]`, y que pueden tener o no valor por defecto [si no se especifica será `null`].
- Argumentos **opcionales con nombre**, para los que necesitamos indicar el nombre cuando se proporciona el argumento; son bastante frecuentes en los constructores de componentes en Flutter. Dado que son opcionales, podemos asignarles un valor por defecto. Por otro lado, si se requieren de forma obligatoria, se indica con `required`. Este tipo de argumentos, igual que en Kotlin, deben ir después de los argumentos posicionales.

Asimismo, cuando se utilizan varios argumentos con nombre, solemos escribirlos en varias líneas, para facilitar la legibilidad. Veremos su uso en la creación de elementos de interfaz.

En el fichero descargable *argumentos_funciones.dart* dispones de varios ejemplos sobre el paso de argumentos a funciones.

2.5. Programación estructurada en Dart

La programación estructurada Dart se basa en las estructuras condicionales y de repetición habituales: if... else, switch, for, forEach y while. Además, también soporta el operador condicional ternario (?).

En el fichero descargable *programacion_estructurada.dart*, dispones de algunos ejemplos con estas estructuras de control.

2.6. Programación orientada a objetos

La orientación a objetos es de gran importancia en Dart, y sobre todo en Flutter, puesto que en estos conceptos se basará todo el diseño de interfaces mediante widgets.

A. CREACIÓN DE CLASES Y CONSTRUCTORES

La sintaxis básica para crear una clase Dart es bastante parecida a otros lenguajes, como Java:

```
class NombreClase {  
    Tipo1 propiedad1=valor_por_defecto1;  
    Tipo2 propiedad2=valor_por_defecto2;  
    ...  
  
    // Constructor (opcional)  
    NombreClase(Tipo1 arg1, Tipo2 arg2,...){  
        propiedad1=arg1;  
        propiedad2=arg2;  
    }  
}
```

Vemos algunos detalles. Por un lado, el constructor de la clase es opcional. En caso de que este no se declare, Dart utiliza un constructor predeterminado sin argumentos. Cuando la clase sí que debe contar con un constructor, este es un método con el mismo nombre que la clase y sin tipo, tal y como hacemos en Java.

Por otro lado, las propiedades, para evitar problemas con valores nulos, deben inicializarse en el constructor o ser declaradas nullables (?).

Para crear un objeto invocamos a su constructor. Aunque se puede utilizar también la palabra reservada new, esta no es obligatoria:

```
NombreClase objeto=new NombreClase(param1, param2);  
NombreClase objeto=NombreClase(param1, param2);
```

B. EL MÉTODO ToString

Cuando intentamos mostrar un objeto por pantalla, se invoca al método **toString** de la clase. Generalmente, este método mostrará que se trata de una instancia de una clase, por lo que si deseamos mostrar una representación personalizada del objeto deberemos sobreescribir el método **toString**:

```
@override  
String toString() {  
    return 'Propiedad1: ${this.propiedad1}, Propiedad2: ${this.propiedad2}';  
    // Ojo con indicar solamente $this.propiedad1,  
    // si no utilizamos las llaves, intentaría imprimir  
    // la propia clase invocando el método toString de  
    // manera recursiva infinitamente.  
}
```

C. SIMPLIFICACIÓN DEL CONSTRUCTOR

El constructor admite una notación simplificada indicando las propiedades internas, con **this** dentro de la propia declaración:

```
NombreClase (this.propiedad1, this.propiedad2);
```

De este modo, además, al inicializar las propiedades en la declaración no necesitamos declararlas como nullables.

D. CONSTRUCTORES CON ARGUMENTOS CON NOMBRE

También es bastante habitual pasar los parámetros de inicialización del constructor por nombre, en lugar de hacerlo de forma posicional, tal y como vimos en las funciones:

```
class NombreClase{  
    ...  
    NombreClase ({  
        required this.propiedad1,  
        required this.propiedad2  
    });  
}
```

Como en las funciones, el uso de la palabra reservada **required** indica la obligatoriedad de incluir el argumento para evitar valores nulos. Si no utilizamos **required**, deberíamos indicar que es una propiedad nullable o proporcionarle un valor por defecto, ya sea en su definición o en los parámetros del constructor:

Sea cual sea, de este modo creamos un objeto con:

```
NombreClase objeto = NombreClase(  
    propiedad1: valor1,  
    propiedad2: valor2,  
    ...  
)
```

Asimismo, como indicamos los argumentos por nombre, no importa el orden en estos se proporcionen.

E. MÚLTIPLES CONSTRUCTORES CON NOMBRE

Dart no soporta sobrecarga de constructores, pero sí permite utilizar constructores con nombre, que también se pueden usar para añadir claridad. Para definir un constructor con nombre, utilizaremos un punto para separar el constructor del nombre en sí:

```
NombreClase.constructor_con_nombre1(lista_argumentos){...}
```

```
NombreClase.constructor_con_nombre2(lista_argumentos){...}
```

F. INICIALIZACIÓN CON DICCIONARIO

Uno de los aspectos más interesantes y habituales en Dart es la creación de objetos a partir de un diccionario o JSON, como, por ejemplo:

```
final objetoJSON = {  
    propiedad1: valor1,  
    propiedad2: valor2  
}
```

Para ello, se crea un constructor que se inicialice a partir de un JSON, del modo siguiente:

```
class NombreClase{  
    ...  
    NombreClase.fromJSON(Map <Tipo1, Tipo2> objetoJSON):  
        this.propiedad1 = objetoJSON['propiedad1'] ?? "valor_por_defecto1",  
        this.propiedad2 = objetoJSON['propiedad2'] ?? "valor_por_defecto2";  
    ...  
}
```

En este caso se recibe un mapa (`objetoJSON`) como argumento, que utilizamos para inicializar las propiedades de la instancia. Observamos que se trata de una única instrucción, donde las inicializaciones se realizan después de la cabecera, separadas de esta por dos puntos (:), y separadas por comas entre ellas. Al tratarse en la misma instrucción tanto la declaración como la inicialización, evitamos problemas con los valores nulos.

Del mismo modo, en este caso hay que fijarse en que se ha creado el constructor con nombre (`fromJSON`), para diferenciarlo de un constructor estándar, y remarcar que los datos se obtienen a partir de un JSON. Ahora, para crear un objeto con este constructor, escribimos:

```
NombreClase miObjeto = NombreClase.fromJSON(objetoJSON)
```

G. MÉTODOS ACCESORIOS: GETTERS Y SETTERS

Dart no contempla palabras reservadas como `public` o `private`. De manera predeterminada, toda propiedad que declaramos será pública. Si deseamos que una propiedad sea privada, lo indicaremos en su nombre, empezándolo con un guion bajo _.

```
class NombreClase{  
    Tipo _propiedadPrivada;  
}
```

A diferencia de otros lenguajes, el encapsulamiento en Dart se realiza a nivel de biblioteca, por lo que una propiedad privada va a seguir siendo visible por todas las clases de la biblioteca.

Cada aplicación en Dart es una biblioteca, con las diferentes clases que la componen, y su distribución se realiza mediante paquetes.

En Dart podemos declarar los métodos accesorios con `get` y `set`, que nos van a permitir acceder a propiedades privadas de una clase desde fuera de la biblioteca, y, además, pueden utilizarse para generar propiedades derivadas.

Crearemos getters y setters de la forma siguiente:

```
tipoRetorno get nombrePropiedad {  
    return expresión;  
}  
  
set nombrePropiedad (Tipo argumento) {  
    // Actualización de las propiedades internas  
}
```

H. HERENCIA

Para que una clase pueda heredarse, necesita tener un constructor vacío, sin argumentos, que será el constructor por defecto que utilizarán las subclases. El resto de constructores no se heredarán.

```
class SuperClase{  
    String propiedad1;  
  
    // Constructor por defecto  
    SuperClase():propiedad1="Valor por defecto en la superclase";  
  
    // Constructor con nombre (no se hereda)  
    SuperClase.fromString(String s): this.propiedad1=s;  
}
```

Definiremos una subclase mediante la palabra **extends**:

```
class Subclase extends SuperClasse { ... }
```

Con esto podemos crear instancias de la subclase con:

```
// Instancia de la subclase
Subclase scl=Subclase();
print("Propiedad 1 de la subclase: ${scl.propiedad1}");
```

En cambio, no podemos utilizar el constructor con nombre **fromString** de la superclase en la subclase:

```
Subclase sc2 = Subclase.fromString("Prueba"); // Error, el constructor from-
String no se hereda!
```

Pero sí que podemos definir el constructor en la subclase e invocar el constructor de la superclase desde este:

```
class SubClase2 extends SuperClase {
    // Constructor con nombre que invoca el constructor
    // con nombre de la clase paro.
    SubClase2.fromString(String s): super.fromString(s);
}
```

Ahora podemos invocar:

```
Subclase sc2 = Subclase.fromString("Prueba");
```

A.CLASES ABSTRACTAS

Como sabemos, las clases abstractas no pueden ser instanciadas, y se utilizan para definir clases derivadas que tienen que implementar necesariamente los métodos indicados en la clase abstracta. Para indicar una clase abstracta, utilizamos la palabra clave **abstract**. Por ejemplo:

```
abstract class Figura{
    int posx;
    int posy;

    void calculaArea(){
        print('Cálculo por defecto');
    }
}
```

B.INTERFACES Y MIXINS

Dart introduce el concepto de interfaz implícita, según el cual cualquier clase puede actuar como interfaz. Para declarar que una clase implementa los métodos de otra clase, se utiliza la palabra reservada **implements**.

```
class ClaseQueImplementaInterfaz implements ClaseQueDefineLaInterfaz{
    // Implementación de los métodos de la clase que define la interfaz
}
```

Finalmente, hay que decir que Dart tampoco soporta la herencia múltiple, pero introduce el concepto de Mixin. que es muy parecido y que se consigue con la palabra reservada **with**.

Por ejemplo, si tenemos definidas las clases A, B y C:

<pre>class A{ void fA(){ print ("Método fA"); } }</pre>	<pre>class B{ void fB(){ print ("Método fB"); } }</pre>	<pre>class C{ void fC(){ print ("Método fC"); } }</pre>
---	---	---

Podemos definir el mixin **claseMixta** como una clase derivada de la clase A, pero mezclada con las clases B y C:

```
class ClaseMixta extends A with B, C {}
```

Con esto indicamos que ClaseMixta deriva de la clase A con B y C, es decir, que tiene accesibles las propiedades y los métodos de las clases A, B y C.

2.7. Programación asíncrona con Dart

El asincronismo hace referencia a un modelo de programación donde es posible que determinadas operaciones retornen el control de la ejecución al programa que las ha invocado antes de su finalización.

Los lenguajes ofrecen diferentes tipos de mecanismos para tratar este tipo de programación, como pueden ser los threads en Java o las corrutinas en Kotlin, entre otros muchos. Dart y Flutter utilizan la clase Future y async/await para trabajar funciones de forma asíncrona.

Los Futures definen tipos de datos asociados a tareas asíncronas que no se resuelven de forma inmediata, como, por ejemplo, pedir información a un servidor, que sabemos que se obtendrán en algún momento, pero no cuándo.

Para ver un ejemplo que simule una función asíncrona, definimos la función siguiente:

```
Future<String> funcionAsincrona() {  
    return Future.delayed(Duration(seconds: 2), () {  
        print("Estamos en funcionAsincrona");  
        return 'Valor de retorno';  
    });  
}
```

Esta función devuelve un objeto de tipo Future transcurridos dos segundos, que contendrá un String.

Para forzar una pausa, utilizamos el método **delayed** de la clase **Future**, que recibe dos argumentos:

- El primero es un objeto de tipo **Duration**, que se inicializa con un argumento opcional, de nombre **seconds** y que indica los segundos de la pausa. Este constructor de **Duration** admite también otras inicializaciones con minutos, horas, etc.
- El segundo es una función anónima, a modo de callback, que se lanzará una vez transcurrido el tiempo. En este caso, esta función muestra el texto Estamos en funcionAsincrona y retorna el texto Valor de retorno. Este texto devuelto será el que contendrá el String contenido en la clase Future.

Para invocar a esta e intentar mostrar el resultado, podríamos hacer lo siguiente:

```
var a=funcionAsincrona();  
print(a);
```

Pero esta imprimirá **Instance of '_Future<String>'**, puesto que lo que realmente devuelve la función es un Future, no un String. Para obtener el valor, hemos de esperar a que este se resuelva, es decir, a que el String del Future tenga valor. Esto se consigue mediante el método **then** de la clase Future, del modo siguiente:

```
Future<String> a=funcionAsincrona();  
a.then( ( String fecha ) {  
    print(fecha);  
});
```

Este método **then** se invocará cuando la función asíncrona devuelva el valor, y reciba, ahora sí, un dato de tipo **String** con el resultado de la función. Se suele decir que la variable **a** se resuelve en este momento.



ASYNC/AWAIT

Si quisieramos esperar a que se ejecutara la función antes de seguir con la ejecución del programa, es decir, realizar una ejecución de forma síncrona, deberíamos invocar la función precedida de la palabra reservada **await**:

```
String a = await funcioAsincrona();  
print(a);
```

Con esto, esperaríamos que terminase la función asíncrona y recibiríamos el String directamente, en lugar del Future. Cuando utilizamos un await dentro de una función, esta función se debe declarar como **async**:

```
void testAsync3() async {  
    print("Hola");  
    String a= await funcioAsincrona();  
    print(a);  
    print("Adiós");  
}
```

En el fichero descargable funciones_asincronas.dart dispone de estos ejemplos sobre funciones asíncronas.

3. Widgets

3.1. Todo es un widget, pero... ¿qué es un widget?

Flutter sigue el principio de «Todo es un widget»: cualquier elemento de la interfaz, ya sea un diseño o un componente de interacción con el usuario o un botón, es un widget.

Los widgets son clases de Dart cuyos constructores pueden contener argumentos tanto posicionales como con nombre, y que nos sirven para componer nuestras interfaces gráficas en Flutter. Este tipo de objetos, además, pueden anidarse para crear una estructura en forma de árbol que representa la interfaz, y pueden ser widgets sin estado (Stateless) o con estado (Stateful), según si sus propiedades pueden mutar o no. Por ejemplo, un botón que no suele cambiar sería un widget sin estado, mientras que un formulario sería un widget con estado. Estos últimos son los que permiten la interacción con el usuario. Cuando el estado de un widget cambia, también cambia la representación del árbol de widgets, y Flutter determina cuáles son los cambios mínimos a realizar en el renderizado para hacer posible la transición entre estados.

Todos los elementos importantes de interfaz incluyen widgets que se corresponden a los diseños de componentes de Android e iOS, así como también a los elementos de interfaz nativa en aplicaciones de escritorio.

A la hora de crear una interfaz, podemos crear nuestros propios widgets, aunque lo más habitual será partir de cualquiera de los múltiples widgets de la librería de Flutter y ampliar su funcionalidad.

IMPORTANTE

Frameworks declarativos e imperativos

Flutter es un framework declarativo. Esto significa que la vista (los widgets) se genera y actualiza según ciertos datos enlazados con ella; cuando los datos cambian, la vista también lo hace. Estos datos con los que se enlaza la vista definen el estado de esta. Aparte de Flutter, utilizan esta aproximación declarativa frameworks web como ReactJS o VueJS.

Frente a los frameworks declarativos se encuentran los frameworks imperativos, que actualizan las vistas mediante instrucciones imperativas desde el propio código. Este ha sido el mecanismo tradicional en Android e iOS, donde definimos la interfaz en ficheros XML y la actualizamos desde el código. Con Jetpack Compose para Android y SwiftUI para iOS, tanto Google como Android ofrecen también la posibilidad de trabajar de forma declarativa para sus sistemas.

3.2. Un «Hola mundo!» con widgets

A modo de ejemplo, veamos cuál sería el código para mostrar un «Hola Mundo!» muy básico:

```
import 'package:flutter/material.dart';

void main() {
  runApp(
    const Center(
      child: Text(
        'Hola Mundo!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

Observemos algunos detalles:

- La función principal, como en todo proyecto Dart, sigue siendo main. Esta función invoca la función runApp, que es la encargada de injectar el widget raíz del árbol y vincularlo a la pantalla. Este widget raíz cubrirá la totalidad de esta.
- En este caso el widget raíz es Center, un widget de tipo contenedor que centra el contenido. Al tratarse del elemento raíz y cubrir toda la pantalla, su contenido estará centrado.
- La forma de indicar que un widget contiene uno o varios widgets anidados es mediante el parámetro con nombre child (o children, si es más de uno). En este caso el widget raíz tiene un widget hijo de tipo Text.
- El widget Text se construye con un primer parámetro posicional y obligatorio con Text, que corresponde al texto que queremos mostrar, y un parámetro con nombre textDirection, que indica la dirección de ese texto, en este caso de izquierda a derecha (ltr o left-to-right). Este parámetro no nos hará falta luego.

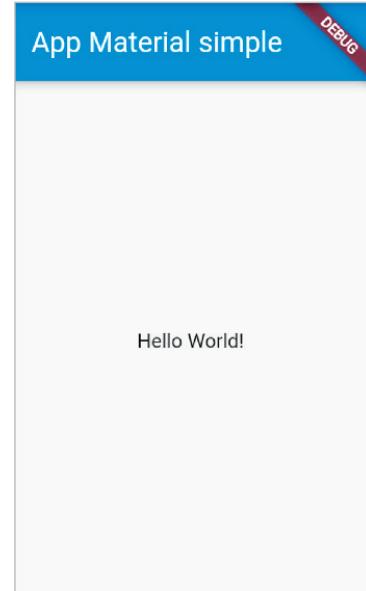
El hecho de utilizar const delante del constructor de un widget es una recomendación del framework, para optimizar el rendimiento en el proceso de renderizado.

«HOLA MUNDO!» CON MATERIAL

En Flutter podemos utilizar el widget MaterialApp como elemento raíz para crear una aplicación de tipo Material, con la barra superior y el cuerpo.

```
import 'package:flutter/material.dart';

void main() {
  runApp(MaterialApp(
    title: 'App Material simple',
    home: Scaffold(
      appBar: AppBar(
        title: const Text('App Material simple'),
      ),
      body: const Center(
        child: Text('Hola Mundo!'),
      ),
    )));
}
```

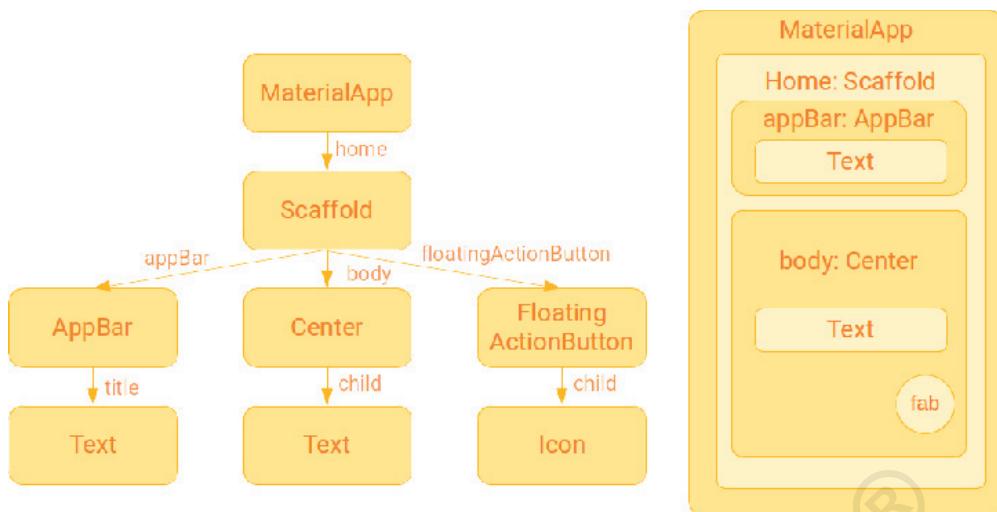


Para indicar que un proyecto va a usar las librerías Material de Flutter, se indica en el fichero pubspec.yaml con:

```
flutter:
  uses-material-design: true
```

Una aplicación Material, como vemos, consta de un título (title) y una página inicial (home), consistente en un componente de tipo armazón (Scaffold) de la aplicación. En nuestro ejemplo esta estructura contiene una barra de aplicación (appBar) y un cuerpo (body), aunque, podría contener además un botón de acción flotante (FloatingActionButton).

Este widget Material podría representarse con la estructura siguiente y su correspondiente árbol de componentes:



3.3. Organización de componentes

Aunque se podría generar todo el árbol de widgets directamente en el método main, lo habitual y lo más cómodo será ir creando los widgets de forma independiente.

Nuestro trabajo al programar aplicaciones en Flutter será crear nuevos widgets a partir de los existentes. Estos serán subclases de StatelessWidget o de StatefulWidget, según tengan o no un estado. Cuando creamos un widget, debemos implementar su función de construcción build, que nos describirá el widget en términos de otros widgets.

Este hecho de dividir un widget en varios componentes vendrá determinado por la funcionalidad que deseemos darle, y, sobre todo, por las posibilidades de reutilización de cada uno.

Como podemos apreciar, el hecho de combinar en un mismo código tanto la lógica del programa como la capa de presentación puede llevar a código muy anidado y confuso si no se trabaja con cuidado.

Así pues, también es bastante lógico que, a medida que crezca nuestra aplicación, tengamos la necesidad de dividir estos componentes en diferentes archivos. Dentro de la carpeta lib podemos crear la estructura de directorios y ficheros que deseemos para organizar nuestro código, aunque esta suele venir determinada en gran medida por la arquitectura o los patrones de diseño que se estén usando. Cuando desde dentro de un archivo necesitemos utilizar componentes definidos en otros, deberemos incluir la clase MyApp().

```
void main() {  
  runApp const MyApp();  
}
```

En la sección de descargas *Ejemplos para trabajar y analizar* dispones del proyecto hola_mundo_widgets.zip, con este «Hola Mundo!» dividido en varios ficheros.

WIDGETS SIN ESTADO: CREACIÓN Y CICLO DE VIDA

En el proyecto anterior, hola_mundo_widgets.zip, hemos creado algunos widgets propios a partir de widgets predeterminados, como, por ejemplo, la clase MyHomePage, del modo siguiente:

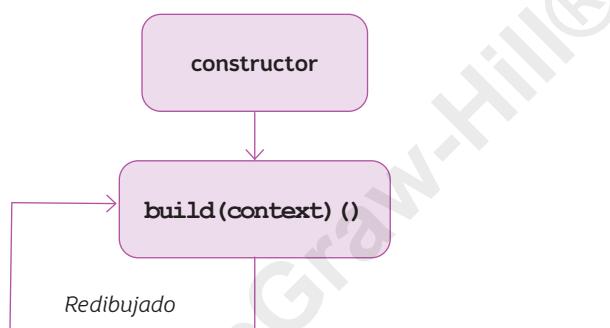
```
class MyHomePage extends StatelessWidget { ... }
```

Vamos a analizar brevemente su estructura. Si utilizamos el snippet **StatelessWidget** del plugin Awesome Flutter, podemos ver la estructura para crear estos tipos de widgets:

```
class name extends StatelessWidget {  
  const name({Key? key}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return ....;  
  }  
}
```

Como puntos más destacados, vemos que se trata de una clase que desciende de la clase StatelessWidget, que tiene un constructor, y que sobrescribe el método build, el cual declara el aspecto que tendrá el widget.

Ahora veamos en el diagrama siguiente cuál es el ciclo de vida de este tipo de widgets sin estado:



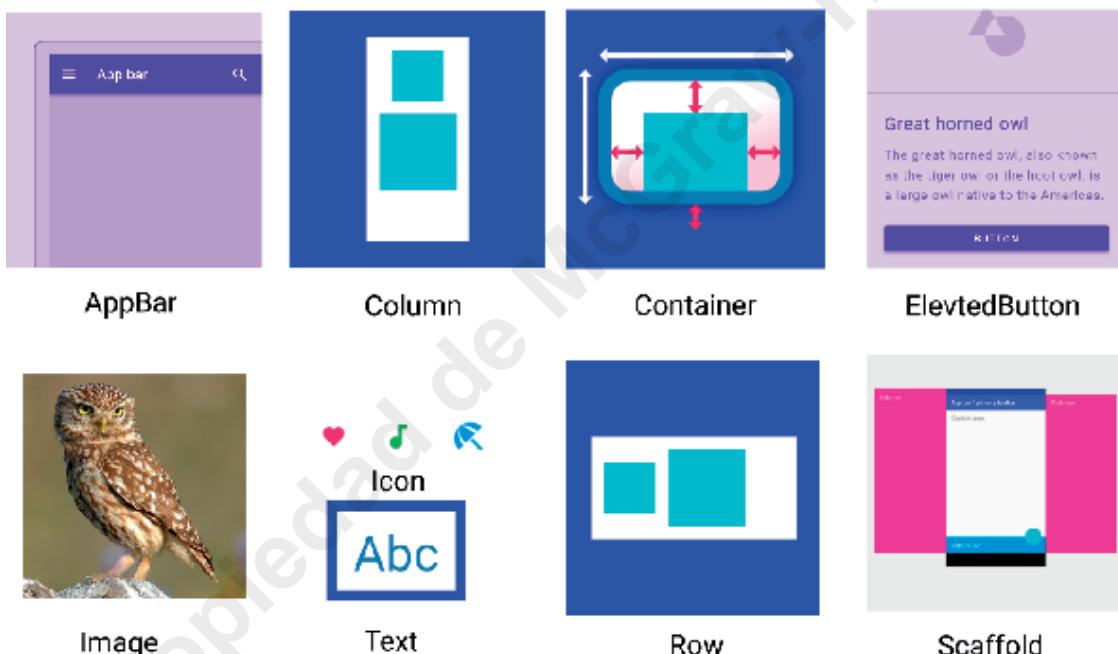
Se inicia con su constructor, al que se le pueden proporcionar argumentos. Una vez creado, cuando este se introduce en el árbol de widgets, se invoca al método build(), heredado de StatelessWidget y que sobrescribimos para establecer su diseño.

Tras ser creado, este no se destruye, sino que se vuelve a dibujar, invocando de nuevo al método build() cada vez que sea necesario, generalmente, cuando un nodo ascendente del árbol de widgets debe redibujarse.

3.4. Widgets básicos

Como ya hemos comentado, Flutter dispone de una gran cantidad de widgets prediseñados en sus librerías. En la documentación oficial de Flutter se muestran los widgets básicos siguientes:

- **Text.** Para crear una cadena de texto con estilo.
- **Row y Column.** Para crear diseños flexibles tanto en horizontal (Row) como en vertical (Column). Su diseño se basa en el modelo de layout flexbox de CSS para la web.
- **Stack.** Permite apilar los widgets, uno encima de otro, en el orden en que estos se renderizan. Para posicionar los widgets en relación con los bordes del Stack, podemos utilizar el widget Positioned. Estos widgets se basan en el modelo de layout de posicionamiento absoluto del CSS para la web.
- **Container.** Crea un elemento visual rectangular, que puede ser decorado con un BoxDecoration, un fondo, un borde o una sombra, así como tener márgenes, fondos o restricciones aplicadas al tamaño.
- **Scaffold.** Estructura de un diseño típico de una aplicación Material. Además de los componentes ya tratados, cuenta con la posibilidad de incluir menús de navegación o elementos emergentes, como los snackbars.
- **ElevatedButton.** Componente de tipo botón de Material Design que arroja sombra por su elevación.
- **Image.** Muestra una imagen.



Esta es solamente una pequeña muestra de los widgets más básicos de Flutter. En la documentación oficial disponible desde la sección *Material de trabajo/Enlaces web* podéis encontrar documentación, tanto en inglés como en español, sobre todos los widgets disponibles.

3.5. Interacción con el usuario: gestos

Los gestos recogen las acciones que el usuario realiza en la interfaz, de manera que proporcionan interactividad a nuestras aplicaciones.

Para ello, existen widgets preparados para la interacción, como pueden ser los botones. Estos widgets, en su inicialización, pueden incluir el argumento con nombre onPressed, consistente en una función de callback que se ejecutará cuando el usuario haga el gesto de tap sobre el widget.

Por ejemplo, para un widget de tipo ElevatedButton:

```
ElevatedButton(  
    onPressed: () {  
        // Cuerpo del callback  
    })
```



Por otra parte, también podemos utilizar el componente GestureDetector para detectar gestos genéricos sobre los widgets, como el swipe, que se produce al deslizar el dedo por la pantalla, o los escalados. Este detector proporciona callbacks opcionales para tratar cada una de las acciones.

Para crear un GestureDetector, el método build que construye nuestro widget deberá devolver un GestureDetector con los gestos a tratar y el contenedor en cuestión. Veamos un pequeño ejemplo para detectar un swipe sencillo a la derecha dentro de un contenedor:

```
Widget build(BuildContext context) {  
  return GestureDetector(  
    onPanUpdate: (details) {  
      // swipe a la derecha  
      if (details.delta.dx > 0) {  
        // Acciones  
      }  
      child: Container();  
    }  
}
```

En el apartado de enlaces dispones del artículo *How to detect Swipe in Flutter*, en el que se explican formas más complejas y precisas de detectar el swipe.

En la sección de descargas *Ejemplos para trabajar y analizar* dispones del fichero comprimido gestos.zip, que contiene un proyecto con varios ejemplos de interacción con el usuario.

3.6. Widgets con estado

Flutter es un framework declarativo, donde los diferentes componentes de la interfaz (widgets) se pueden construir y actualizar en función de un estado. Desde este punto de vista, un widget podría entenderse como la representación gráfica de un estado.

Como ya hemos comentado, los widgets en Flutter pueden ser con y sin estado. En widgets sin estado esta representación se realizará directamente mediante el método build, y en widgets con estado se hará a través de su estado asociado.

Veámoslo un poco más claro mediante código. El snippet **StatefulWidget**, del plugin Awesome Flutter, nos proporciona la plantilla básica para un widget con estado, que nos servirá para ver su estructura:

```
class name extends StatefulWidget {  
  name({Key? key}) : super(key: key);  
  
  @override  
  _nameState createState() => _nameState();  
}  
  
class _nameState extends State<name> {  
  @override  
  Widget build(BuildContext context) {  
    return Container();  
  }  
}
```

Fijémonos en algunos detalles:

- La clase que define el widget desciende ahora de StatefulWidget, lo que indica que contiene un estado mutable. El constructor, igual que los widgets con estado, para poder referenciarse dentro del árbol de widgets tendrá también una propiedad que será la clave (key).

- La clase sobrescribe el método `createState()` de la clase `StatefulWidget`. Este método es invocado por el framework cuando el widget se inserta por primera vez en el árbol de widgets, de forma que se crea una instancia del estado `_nameState`. Con ello, cada vez que se redibuja el widget, el framework reutilizará esta instancia, de modo que el estado no se pierde. Este mecanismo sería el equivalente a cuando mantenemos los datos en un `ViewModel` de Android, independiente de la vista.

Como ejemplo básico de este tipo de widgets, y de cómo modificar el estado, podemos echarle un vistazo a la aplicación del contador de ejemplo que proporciona Flutter.

En la sección de descargas *Ejemplos para trabajar y analizar* dispones del fichero comprimido contador.dart, con el código simplificado y comentado del contador.

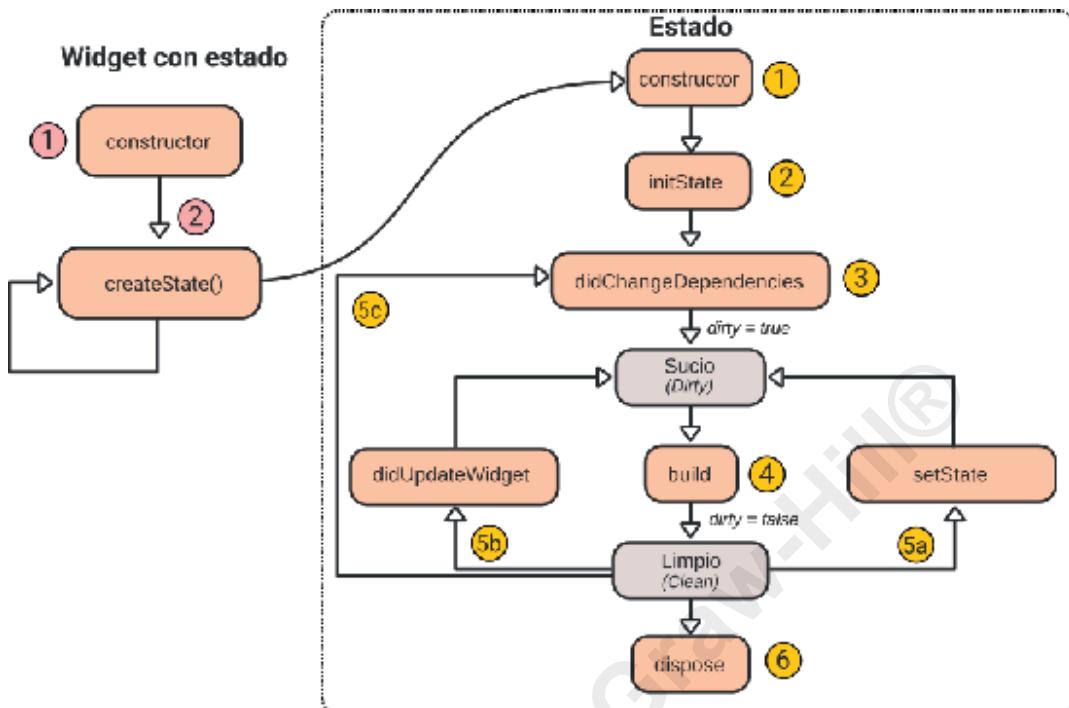
Como podemos ver, en el ejemplo se sigue esta misma estructura, a la que se incorpora, dentro del estado, una propiedad entera, con el contador, así como un método `_incrementCounter` que invoca al método `setState` proporcionándole el modo de actualizar el estado (incrementando el contador) en forma de callback. Este método implicará un cambio de estado, y, en consecuencia, el redibujado del widget.

```
class Contador extends StatefulWidget {  
  ...  
  @override  
  _EstadoContador createState() => _EstadoContador();  
}  
  
// Clase de estado  
class _EstadoContador extends State<Contador> {  
  int _counter = 0;  
  
  void _incrementCounter() {  
    setState(() { // Indica un cambio de estado. Ejecuta build de nuevo.  
      _counter++;  
    });  
  }  
  
  @override  
  Widget build(BuildContext contexto) { // Redibuja el widget... }  
}
```

EL CICLO DE VIDA DE LOS WIDGETS Y DE SU ESTADO

El ciclo de vida de un widget con estado es similar al ciclo de vida de un widget sin estado. Los cambios importantes se encuentran, como veremos, en el ciclo de vida del propio estado.

Veamos el ciclo de vida de un widget junto con su estado en el diagrama siguiente:



Ciclo de vida del widget con estado

1. Se inicia invocando a su constructor.
2. Se invoca al método createState para crear el estado asociado. En este momento se inicia el ciclo de vida de este.

Ciclo de vida del estado

1. Se inicia mediante la llamada a su constructor por parte de createState en el widget asociado. Dado que el widget no está todavía en el árbol, no deberíamos realizar aquí las inicializaciones que dependan del contexto.
2. Se lanza el método initState, donde generalmente se realizan operaciones de inicialización de datos o estos se obtienen desde fuentes externas, como una base de datos o la red. Cuando sobreescrivimos este método, debemos invocar al constructor de su superclase para realizar tareas de inicialización. Este método se ejecutará solamente una vez.
3. El método didChangeDependencies se invoca una vez, inmediatamente después de initState. Este método se volverá a lanzar únicamente cuando se deba hacer alguna inicialización en que intervenga un InheritWidget. En este punto, Flutter marca este widget con el flag `dirty=true`, para indicar que necesita reconstruirse.
4. Si se necesita una reconstrucción del widget (`dirty=true`), se invoca el método build() para determinar los widgets que deberá renderizar nuestro widget. Después de esto, el estado se marca como limpio, con `dirty=false`.

En este punto ya tenemos el widget insertado en el árbol y renderizado acorde a su estado, por lo que se considera que está limpio. A partir de este momento pueden pasar varias cosas que vuelvan a establecer este flag como dirty (sucio) que impliquen una reconstrucción de este.

- 5a. Si ocurre algún evento (como un tap en un botón) que modifique el estado, se invoca al método setState(), que vuelve a marcar el estado con `dirty=true`, para que vuelva a reconstruirse en el método build, proporcionándole el estado más reciente.

- 5b. Si un widget ascendiente pide que se reconstruya su descendencia y esto implica al widget en cuestión, se invoca al método `didUpdateWidget`, proporcionándole el widget anterior como argumento y marcando el estado como sucio, para que vuelva a reconstruirse.
- 5c. Si el widget depende de un `InheritedWidget` y este widget heredado cambia, se invoca al método `DidChangeDependencies`, reconstruyendo también el widget. Los widgets heredados serían un tercer tipo de widget que permite a un widget descendiente acceder directamente a su estado, sin necesidad de ir ascendiendo en el árbol de widgets hasta llegar a él.
6. Si el widget no va a utilizarse más, se invoca a `dispose()` para destruirlo. En este método se deberán detener las animaciones, cerrar conexiones, etc.

En la sección de descargas dispones del fichero `CicloVidaContador.dart`, con el ejemplo del contador modificado para mostrar las etapas y callbacks del ciclo de vida de los widgets.

3.7. FutureBuilder

Cuando trabajamos con funciones o métodos asíncronos en nuestras aplicaciones, la interfaz de usuario debe responder de forma adecuada a los diferentes eventos y resultados que se producen proporcionando el comportamiento que se espera de ella y evitando bloqueos de la misma.

Para gestionar interfaces con aplicaciones asíncronas, disponemos del widget `FutureBuilder`. Se trata de un widget con estado que es capaz de generarse a sí mismo como respuesta a algún evento asíncrono.

Si utilizamos el snippet `FutureBuilder`, veremos que proporciona el código siguiente:

```
FutureBuilder(  
  future: Future,  
  initialData: InitialData,  
  builder: (BuildContext context, AsyncSnapshot snapshot) {  
    return ;  
  },  
) ,
```

Observamos que el constructor de `FutureBuilder` recibe tres argumentos con nombre:

- **Future**. Representa el tipo de dato resultado de una operación asíncrona, que generalmente se obtendrá a partir de un método o una función que devuelva un `Future`.
- **InitialData**. Es el valor inicial hasta que se resuelve el `Future`.
- **Builder**. Se trata del constructor del widget como tal, y es el único argumento requerido. Consiste en una función anónima que se invocará cuando se haya resuelto el `Future`. Esta función recibe el contexto y un objeto de tipo `AsyncSnapshot` con información sobre la resolución del `Future`. Este snapshot contendrá, entre otros, el flag `hasData`, que informa de la resolución del `Future` y de que los datos se encuentran en el componente `data` del propio snapshot.

Veamos un pequeño ejemplo de uso. Disponemos de la función asíncrona siguiente, que devuelve un mensaje pasados dos segundos:

```
// Función asíncrona  
Future<String> funcionAsincrona() async {  
  await Future.delayed(Duration(seconds: 2));  
  return 'Valor de retorno';  
}
```

Para construir un widget de tipo FutureBuilder que se actualizase utilizando esta función, en primer lugar, utilizaríamos un widget con estado que definiese un valor `_value`, de tipo `Future<String>`, asignado este al valor que retornara la función asíncrona:

```
class _EjemploFutureBuilderState extends State<EjemploFutureBuilder> {  
    late Future<String> _value;  
  
    @override  
    initState() {  
        super.initState();  
  
        // El estado viene dado por lo que devuelva  
        // la función asíncrona  
        _value = funcionAsincrona();  
    }  
    ...  
}
```

Dentro de esta clase se incluiría el método `build` para construir el widget y, dentro de este, se usaría el `FutureBuilder`. Este widget, de forma simplificada, tendría los componentes siguientes:

```
FutureBuilder<String>(  
    future: _value,  
    initialData: 'Valor inicial',  
    builder: (BuildContext context, AsyncSnapshot<String> snapshot) {  
        if (snapshot.connectionState == ConnectionState.waiting) {  
            // Contenido mientras se está a la espera  
            ...  
        } else if (snapshot.connectionState == ConnectionState.done) {  
            // Contenido cuando se tienen los datos: Actualización del widget  
            ...  
        }  
    }  
}
```

Advertimos que se establece un valor inicial para el texto (`initialData`), y se establece que el `future` que vamos a utilizar es el correspondiente al valor asociado al estado, que es resultado de la función asíncrona. En el builder, mientras el estado de conexión del snapshot esté a la espera, se mostrará un contenido, y, cuando el valor del `future` se haya resuelto y el estado del snapshot sea `done`, se actualizará el widget con el valor de retorno.

En la sección de descargas *Ejemplos para trabajar y analizar* dispone del fichero comprimido `ejemplo-FutureBuilder.zip` con este proyecto completo.

4. Navegación, listas y bibliotecas

4.1. Navegación y rutas

Generalmente, la interfaz de una aplicación consta de diferentes pantallas, entre las que podemos navegar y compartir información. Cuando trabajamos con Android, navegamos entre pantallas mediante Intents, para abrir otras actividades, o mediante la navegación entre fragmentos.

En Flutter esta navegación entre pantallas se basa en el widget Navigator, que mantiene el historial de las pantallas que se han visitado en forma de pila.

Las rutas en Flutter son una abstracción para las distintas pantallas, que permite el enrutamiento entre ellas. Veamos un sencillo ejemplo con dos pantallas. La primera tiene la estructura siguiente:

```
class Pantalla1 extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(  
        title: Text('Primera Pantalla'),  
      ),  
      body: Center(  
        child: ElevatedButton(  
          child: Text('Ir a pantalla 2'),  
          onPressed: () {  
            // Completaremos este  
            // código después  
          },  
        ),  
      );  
    }  
}  
  
class Pantalla2 extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return Scaffold(  
      backgroundColor: Colors.indigo,  
      appBar: AppBar(  
        title: Text("Segunda Pantalla"),  
      ),  
      body: Center(  
        child: ElevatedButton(  
          child:  
            Text('Volver a Pantalla 1'),  
          onPressed: () {  
            // Completaremos este  
            // código después  
          },  
        ),  
      );  
    }  
}
```

Observamos que se trata de pantallas similares, con una barra de título y un botón ElevatedButton, y lo que se pretende es que desde la primera pantalla se pueda abrir la segunda, y desde esta segunda se pueda volver a la primera.

Para poder hacer esto, necesitamos añadir rutas con nombre hacia ambas pantallas en nuestra aplicación, lo que se consigue mediante la propiedad routes en el constructor de una aplicación MaterialApp:

```
class MyApp extends StatelessWidget {  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Ejemplo rutas',  
      initialRoute:  
        '/', // Indica la ruta inicial, es decir, la pantalla por defecto  
      routes: { // La ruta raíz (/) es la primera pantalla
```

```
'/' : (context) => Pantalla1(),
'/pantalla2' : (context) =>
    Pantalla2(), // la ruta /pantalla2 lleva a la segunda pantalla
},
);
}
}
```

Como vemos, para navegar a una u otra pantalla no tenemos más que indicarlo con su ruta respectiva. Esto se consigue gracias al objeto Navigator y a sus métodos push y pop. Concretamente, utilizaremos pushNamed para apilar una ruta con nombre en la pila de ventanas, y pop, para eliminar la ruta que se encuentra encima del todo de la pila del Navigator. Así pues, el código correspondiente a cada botón será el siguiente:

```
ElevatedButton( // Botón en Pantalla 1
    child: Text('Ir a pantalla 2'),
    onPressed: () {
        // Navegación hacia la segunda pantalla. Proporcionamos el
        // context y el nombre de la ruta a la pantalla
        Navigator.pushNamed(context, '/pantalla2');
    },
)
ElevatedButton( // Botón en pantalla 2
child: Text('Volver a Pantalla 1'),
    onPressed: () {
        // Vuelta a la primera pantalla.
        // El método pop recibe el contexto.
        Navigator.pop(context);
    },
)
```

Con esto, además, en la barra de aplicación de la pantalla 2 dispondremos de un ícono de volver, que tiene el mismo resultado.

Es importante observar que en este caso el constructor de MaterialApp no recibe el argumento home, ya que este vendrá dado por las diferentes rutas, y el contenido que se mostrará inicialmente vendrá determinado por la ruta indicada en InitialRoute.

En la sección de descargas *Ejemplos para trabajar y analizar* dispone del fichero rutas.dart con este ejemplo completo.

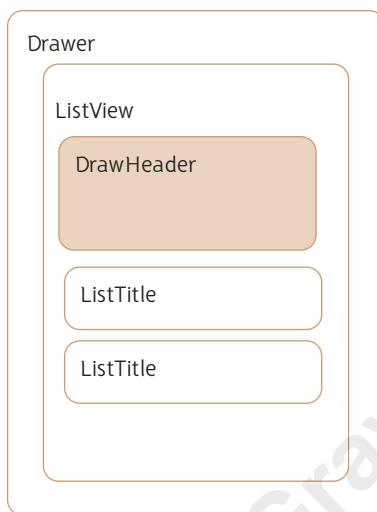
A.DRAWER

El componente Drawer es, junto con los Tabs, uno de los principales mecanismos de navegación entre pantallas en una aplicación Material Design. Como ya sabemos, un Drawer consiste en un menú lateral de navegación que permite desplazarse entre diferentes pantallas, y se compone principalmente de una cabecera y una lista de opciones.

En Flutter, el componente Drawer se puede proporcionar como argumento con nombre al constructor del Scaffold, para generar así el esqueleto de una aplicación Material, junto a la barra superior, el cuerpo y el botón de acción flotante.

```
Scaffold(  
    drawer: MyDrawer(),  
    appBar: AppBar(  
        title: Text('...'),  
    ),  
    body: ...  
)
```

La estructura general de este componente será similar a la siguiente:



Observamos que el componente Drawer tiene un único widget hijo, que es un ListView. Con esto podemos mostrar las diferentes opciones en que el usuario puede hacer scroll si no hay suficiente espacio vertical. Esta lista contiene un primer elemento de tipo DrawHeader con la cabecera del Drawer, y diferentes ListTitle con las distintas opciones. Estos widgets internamente se compondrán de otros widgets como textos e imágenes.

Además, cada ListTitle tendrá un manejador asociado al evento `onTap`, dentro del cual gestionaremos la navegación en sí.

Para esta navegación deberemos, por una parte, tener en cuenta en qué ruta nos encontramos y hacia qué rutas podemos ir, y, por otra, cómo vamos a gestionar la pila de las diferentes ventanas.

Para obtener la ruta en que nos encontramos, haremos uso de la clase abstracta `ModalRoute` del modo siguiente:

```
String currentRoute =  
    (ModalRoute.of(context)?.settings.name).toString();
```

Con esto, obtenemos la ruta más cercana a nuestro contexto y accedemos al componente `settings.name` de esta, lo que devolverá el nombre de la ruta más cercana a nuestra ubicación actual.

Únicamente deberemos comparar la ruta actual con la ruta a la que deseamos ir, y solo navegar si son diferentes:

```
title: const Text('Pantalla 1'),  
    onTap: () {  
        String currentRoute =  
            (ModalRoute.of(context)?.settings.name).toString();  
        if (currentRoute != "/") {  
            Navigator.of(context).pushReplacementNamed('/');  
        }  
    },
```

En la sección de descargas *Ejemplos para trabajar y analizar* dispones del fichero `drawer.dart` con este ejemplo completo.

4.2. Listas

Una de las tareas más comunes en el desarrollo de aplicaciones consiste en mostrar listas de datos. A este respecto ya vimos componentes como el RecyclerView en Android. En Flutter utilizaremos para este fin el widget ListView, que consiste en una lista desplazable de widgets en disposición lineal, bien horizontal, bien vertical. Aunque hay otros widgets con características similares, como el GridView, el ListView es el más utilizado.

La forma más simple de crear un ListView es la siguiente:

```
ListView(  
  children: [  
    child widget1,  
    child widget2,  
    ...  
  ],  
  scrollDirection: [Axis.horizontal | Axis.vertical],  
  reverse: [true | false ]  
)
```

Como vemos, el constructor recibe un argumento de tipo children, con todos los ítems que contendrá la lista, en forma de widgets, así como la dirección de desplazamiento (scrollDirection), que puede ser horizontal o vertical, y el orden en que deseamos que se muestren los diferentes elementos.

Además de estos parámetros, el constructor admite varios argumentos con nombre, como puede ser el padding, para añadir relleno entre los elementos. Podemos consultar este argumento y otros en la documentación oficial y en español sobre el ListView.

A. LISTVIEW.BUILDER()

Aunque la anterior sea la forma más simple de crear listas, esta que ahora presentamos muestra solamente una cantidad finita de ítems. Cuando deseemos construir listas de una forma más dinámica, se requerirá algún mecanismo más avanzado.

La forma de hacerlo es mediante el constructor con nombre ListView.builder(). Este constructor requiere de la propiedad itemBuilder, que consiste en una función anónima que devuelve la composición de cada elemento nuevo. Asimismo, es importante, aunque no estrictamente necesario, proporcionar un valor a la propiedad itemCount, con la que informamos al ListView del número de ítems, para mejorar el rendimiento en el desplazamiento de la lista.

Si utilizamos el snippet listViewB del plugin Awesome Flutter Snippets en VSCode, este generará el código de base siguiente.

```
Listview.builder(  
  itemCount: 1,  
  itemBuilder: (BuildContext context, int index) {  
    return ;  
  },  
) ,
```

A partir de este, podemos generar nuestra lista completa. Por ejemplo, dada esta lista:

```
List<String> mylist = ["item 1", "item 2", "item 3"];
```

A partir de ella, podemos generar un nuevo ListView de tarjetas (Card) con el contenido siguiente:

```
ListView.builder(  
    itemCount: mylist.length,  
    itemBuilder: (BuildContext context, int index) {  
        return Card(  
            child: Center(  
                child: Padding(  
                    padding: const EdgeInsets.all(8.0),  
                    child: GestureDetector(  
                        child: Text(mylist[index]),  
                        onTap: () => {  
                            ScaffoldMessenger.of(context).showSnackBar(SnackBar(  
                                content: Text("Seleccionado ${mylist[index]}"))  
                        },  
                    )));  
    },  
)
```

Como vemos, la propiedad itemCount se inicializa con la longitud de la lista, y el itemBuilder, que recibe el índice del elemento que va a dibujar de la lista, devuelve una tarjeta (Card) con un widget de centrado, que contiene un espaciado (Padding) con un texto, envuelto por un GestureDetector. Esto nos sirve para detectar cuándo se pulsa sobre el elemento de la lista y, así, mostrar un Snackbar indicando el elemento de la lista que se pulsó.

En la sección de descargas *Ejemplos para trabajar y analizar* dispone del fichero listas.dart con este ejemplo completo.

B. EL WIDGET LISTTILE

En el ejemplo anterior hemos generado nuestros propios diseños para las filas de una lista. Esto es útil si deseamos crear un diseño concreto, pero, generalmente, será suficiente con un diseño de fila estándar. Esto se consigue mediante el widget ListTile, que ofrece una fila que puede contener un título, un subtítulo e iconos al principio y al final. A modo de ejemplo, veamos la sintaxis para crear un widget de tipo ListTile con estos elementos:

```
ListTile(  
    title: Text("Título del ítem"),  
    subtitle: Text("Subtítulo"),  
    tileColor: Colors.color_de_fondo,  
    leading: Imagen_de_cabecera,  
    trailing: Imagen_final  
)
```

En la sección de descargas *Ejemplos para trabajar y analizar* dispone del fichero listas2.dart con el ejemplo de las listas usando el componente ListTile.

4.3. Trabajando con paquetes y bibliotecas

Flutter admite el uso de bibliotecas o paquetes compartidos, desarrollados por terceros tanto para Flutter como para Dart, lo que nos permite crear aplicaciones sin tener que programar ciertas funcionalidades desde cero.

A pesar de su corto recorrido, el framework Flutter cuenta con una gran cantidad de librerías, tanto del propio framework como de terceros creadas por la comunidad de desarrolladores.

A. BIBLIOTECAS, PAQUETES Y COMPLEMENTOS

Como ya se ha comentado anteriormente, todo proyecto en Flutter constituye una **biblioteca**, por eso el código se encuentra ubicado en el directorio lib. Todos los elementos públicos de los archivos Dart ubicados en esta carpeta formarán parte de la biblioteca.

Por otra parte, cuando hablamos de paquetes en Dart nos referimos a cualquier directorio que contenga un fichero pubspec.yaml, en el que se describen diferentes aspectos de este, como sus metadatos, su entorno y sus dependencias. Cuando estos paquetes Dart además contienen aplicaciones Flutter, se dice que son paquetes Flutter. Aparte de este fichero, un paquete puede contener pruebas, imágenes y cualquier otro tipo de recurso.

Los **paquetes** serán, pues, la forma de distribuir nuestras bibliotecas, sea una aplicación o una biblioteca como tal. Podemos, por tanto, distinguir dos tipos de paquetes:

- **Paquetes de aplicaciones**, que contienen una aplicación en sí, es decir, una biblioteca con el código, y en él, una función main para iniciar la aplicación.
- **Paquetes de biblioteca**, que no contienen una función main, sino una colección de funciones, definiciones de tipos, clases o propiedades que pueden ser útiles para otras aplicaciones.

Nos quedan por describir los **plugins** o complementos. Cuando un paquete implementa cierta funcionalidad de la plataforma haciéndola disponible para las aplicaciones, se dice que es un plugin. Se trata, pues, de paquetes que están escritos específicamente para la plataforma e implementan funcionalidades tales como la capacidad de acceder a la cámara o a cualquier sensor que el dispositivo posea.

La web pub.dev sirve como repositorio de paquetes Dart/Flutter y permite su búsqueda mostrando qué paquetes son compatibles con Flutter y cuáles son los sistemas nativos que los soportan.

B. AÑADIENDO PAQUETES. UN EJEMPLO PRÁCTICO: GEOLOCATOR

Añadimos un paquete a nuestra aplicación mediante el fichero pubspec.yaml, concretamente en la sección de *dependencies*.

A modo de ejemplo útil, utilizaremos el plugin de geolocalización geolocator, para acceder a la ubicación del dispositivo (consulta el enlace correspondiente a su descarga en la sección *Material de trabajo/Enlaces web*).

The screenshot shows the 'geolocator' package page on pub.dev. At the top, it displays the package name 'geolocator' in blue, followed by its version '8.0.5', the number of stars '2744', the number of commits '130', and the word '100% compatible'. Below this, there's a brief description: 'Geolocation plugin for Flutter. This plugin provides a cross-platform (iOS, Android) API for generic location (GPS etc.) functions.' There are also links for 'View on GitHub' (3 days ago), 'View on pub.dev', 'Flutter', 'Android', 'iOS', 'MacOS', and 'Web'.

Como podemos ver, cuando buscamos una librería en esta página, aparte del nombre, la descripción y la versión, también se indica si está adaptada a Null Safety, si se trata de una librería de Flutter o de Dart, y los sistemas con los que esta es compatible. En el caso de la biblioteca geolocator, esta se encuentra en la versión 8.0.5 en el momento de escribir estas líneas, pero es posible que la versión sea superior cuando las leas. Además, es compatible con Android, iOS, MacOS y Web.

Para añadir esta biblioteca a nuestra aplicación, añadiríamos la línea siguiente a las dependencias del fichero pubspec.yaml:

dependencies :

```
# [Otras dependencias de la aplicación]  
geolocator: ^8.0.5
```

Como vemos, también debemos especificar el número de versión junto con el nombre del paquete. Una vez hecho esto, desde la terminal, y en el directorio raíz del proyecto, ejecutamos:

```
flutter packages get
```

Esta instalación también se puede hacer directamente, sin modificar el fichero pubspec.yaml con lo siguiente:

```
flutter pub add geolocator
```

Con esta orden se busca el paquete geolocator, se añade automáticamente a las dependencias del paquete, y se descarga.

Recordemos que los paquetes descargados no son parte de nuestro proyecto, sino que se encuentran en una caché compartida por todos los proyectos en nuestra carpeta de usuario.

Configuración de geolocator para Android

Dado que se trata de un paquete que contiene una implementación por plataforma, vamos a tener que adaptar el proyecto específico de cada una de ellas. En nuestro caso, adaptaremos únicamente el proyecto de Android, ubicado en la carpeta *android* de nuestro proyecto Flutter.

Los pasos a realizar para la adaptación son los siguientes:

- En primer lugar, comprobamos que el fichero gradle.properties está configurado para utilizar las librerías AndroidX, añadiendo, si no existieran, las líneas siguientes:

```
android.useAndroidX=true  
android.enableJetifier=true
```

- En segundo lugar, debemos modificar el fichero AndroidManifest.xml de la aplicación para que esta pida al usuario los permisos necesarios para acceder a la geolocalización. Para ello incorporamos como hijos directos de la etiqueta manifest:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.example.mi_tiempo">  
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"  
    />  
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"  
    />  
    <application...>
```

Finalmente, ya solo nos quedará importar la librería en nuestros ficheros dart, con la declaración import correspondiente, indicando el paquete y el fichero a importar:

```
import 'package:geolocator/geolocator.dart';
```

Una vez tenemos la librería importada, podremos utilizar la clase Geolocator para obtener información relativa a la geolocalización. Entre las diferentes funcionalidades que ofrece, encontramos:

- **Geolocator.isLocationServiceEnabled()**. Función asíncrona que indica si el servicio de geolocalización está activo.
- **Geolocator.checkPermission()**. Comprueba si se tienen permisos de acceso al servicio de geolocalización. Estos pueden ser:
 - LocationPermission.always. El permiso está concedido incluso cuando la app está ejecutándose en background.

- LocationPermission.denied. El permiso no está concedido, pero se puede pedir acceso al usuario.
- LocationPermission.deniedForever. El permiso está denegado de forma persistente, de modo que no pide acceso al usuario. Si se desea modificar, debe hacerse desde la configuración de la aplicación.
- LocationPermission.unableToDetermine. No se pudo saber el estado de los permisos sobre geolocalización. Solamente será aplicable a entornos web.
- **Geolocator.requestPermission()**. Muestra el diálogo del sistema para pedir permiso al usuario sobre la geolocalización.
- **Geolocator.getCurrentPosition()** Geolocator.getCurrentPosition(). Retorna de forma asíncrona la posición actual proporcionada por el servicio de geolocalización, con sus respectivas coordenadas de latitud y longitud.

Finalmente, dado que el paquete contiene código específico de la plataforma, deberemos recompilar y reiniciar la aplicación, ya que un hot reload o hot restart solamente afecta al código dart.

Propiedad de McGraw-Hill®



Unidad 5.

Introducción a los videojuegos. Motores.

Introducción a Unity





1. Introducción

1.1. ¿Qué es un videojuego?

Tal y como podemos deducir de su nombre, un videojuego (*video + juego*) no es más que un juego de video; un juego que se muestra a través de una pantalla, ya sea un televisor, un monitor, un teléfono móvil, una tableta o unas gafas de realidad virtual. Este juego, además, permite que una o varias personas interactúen con él mediante los controles correspondientes.

Desde un punto de vista más pragmático, podemos ver un videojuego como una película de animación que aporta una experiencia interactiva al usuario, quien con sus acciones hace evolucionar una historia.

Como vemos, esta interacción entre el usuario y el videojuego se realiza principalmente a través de dispositivos, donde mostrar las imágenes al usuario, y de controles, a través de los cuales el usuario interactúa con el juego. Además de esto, el sonido desempeña un papel importante en los videojuegos, y también podemos encontrar otros dispositivos, conocidos como periféricos hapticos, que transmiten sensaciones al usuario, como vibraciones o fuerzas.

1.2. Historia de los videojuegos

La historia de los videojuegos puede ocupar centenares de páginas. En este apartado, vamos a hacer una breve descripción de las etapas y los hitos que consideramos más importantes en la historia de los videojuegos. En los enlaces recomendados disponéis de varios recursos audiovisuales donde podréis profundizar más en esta historia.

A. LOS INICIOS (1950-1970)

Podríamos ubicar el inicio de la historia de los videojuegos en la década de los cincuenta. Con la aparición de los primeros ordenadores, también empezaron a crearse programas lúdicos y a plantearse los primeros algoritmos basados en juegos como el ajedrez, el juego de nim, el tres en raya o las damas.

En esta época aparecieron las primeras aproximaciones a lo que serían los videojuegos, como el *Tennis For Two* de William Higinbotham (1958), haciendo uso de un osciloscopio, o el *Spacewar!* (1962), de Steve Russell junto con Wayne Witaenem y Martin Graetz, estudiantes del MIT (Instituto Tecnológico de Massachusetts), y que consistía en un duelo espacial para dos jugadores creado sobre el primer PDP-1 que llegó a la institución.

En esta época también vio la luz la *Brown Box* (1967), de Ralph Baer, que supuso el primer prototipo de videoconsola doméstica.

B. LA ECLOSIÓN (1970-1978)

Aunque todo empezó en las décadas anteriores, la década de los setenta supuso una primera revolución con el inicio de la industrialización de los videojuegos. Todo empezó con *Galaxy Game*, una adaptación del *Spacewar!* para que funcionara con monedas. La idea fue de Bill Pitts, un estudiante de la Universidad de Stanford, quien fundó con Hugh Tuck la Computer Recreations Inc., en 1971.

Un año después, en 1972, el prototipo *Brown Box* de Ralph Baer dio lugar a *Magnavox Odyssey*, convirtiéndose en la primera videoconsola de la historia.

Ese mismo año, la compañía Atari lanzó el videojuego *Pong*, una versión mejorada y con sonido inspirada en un juego de ping-pong que llevaba de serie la consola *Magnavox Odyssey*, y empezó a comercializarlo en los salones recreativos; sembró la semilla de lo que se convertiría en la industria de las máquinas recreativas de la década. Tres años después, en 1975, Atari empezó a portar *Pong* a las videoconsolas domésticas con *Telegames Pong*.

En esta década también se incorporaron los procesadores digitales a las máquinas de videojuegos, abriendo todo un mundo de posibilidades.

Más títulos que aparecieron entonces fueron *Breakout* (1976), *Death Race* (1976) o *Night Driver*, así como videojuegos para los primeros ordenadores personales, como *Microchess* (1976) o *Adventureland* (1978), entre otros.

También fue a finales de la década de los setenta cuando se empezó a definir el mercado de las videoconsolas domésticas, con la *Atari 2600* al frente.

C. LA EDAD DE ORO (1978-1983) Y LA DÉCADA DE LOS OCHENTA

Con la aparición de *Space Invaders* (1978, Taito Corporation) de *Toshihiro Nishikado*, y de la mano de la industria japonesa, empezó la que algunos autores han descrito como «la edad de oro de los videojuegos». El interés general por los videojuegos crecía cada vez más, y los salones recreativos empezaron a popularizarse a lo largo y ancho del planeta. En las salas recreativas aparecieron títulos como *Pacman* (Namco, 1980), *Galaxian* (Namco, 1979), *Donkey Kong* (Nintendo, 1981), *Track & Field* (Konami, 1983) o *Gauntlet* (Atari, 1985).

En 1979 surgió también la primera compañía de videojuegos independiente, que no disponía de una consola propia y desarrollaba juegos para terceros. Se trata de Activision, la cual a principios de los ochenta lanzó juegos como *Checkers*, *Skiing*, *Dragster* o *Boxing*. Esta empresa sigue viva hoy en día con títulos como *Call of Duty*, entre muchos otros.

En el año 1983 se produjo una crisis en la industria del videojuego en EE.UU. causada por la diversidad de ordenadores y consolas, que generaba juegos de baja calidad. Atari llegó a perder el 90% de sus beneficios en dos años. En 1982 lanzó el videojuego de E.T., considerado el peor videojuego de la historia; este llevó, en parte, a la división y la posterior venta de Atari en 1984.

Esta crisis en EE.UU. favoreció el mercado japonés, que tomó el relevo y llevó la industria del videojuego a un período de madurez. En ese momento aparecieron títulos como *Super Mario Bros* (Nintendo, 1985), *Tetris* (Nintendo, 1989), *Legend Of Zelda* (Nintendo, 1987), *John Madden Football* (1988) o *SimCity* (Brøderbund Software Infogrames, 1989).

Asimismo, la década de los ochenta supuso un auge en el mercado doméstico, con las consolas de 8 bits (o de tercera generación) como la Nintendo Famicom/NES, la Sega Mark III/Master System o la Atari 7800, y los ordenadores domésticos como el ZX Spectrum, el Amstrad CPC, el MSX, el Apple II o el Commodore Amiga.

LA EDAD DE ORO DEL SOFTWARE ESPAÑOL (1983-1992)

Coinciendo con la segunda mitad de la década de los ochenta, aparece lo que se conoce como la edad de oro del software español. Esta tuvo sus inicios en la empresa Indescomp, que juntó dos grandes desarrolladores como Paco Suárez y Paco Portal (Paco & Paco), desarrolladores de La Pulga, la cual obtuvo un gran éxito en el mercado inglés. Esta empresa fue el origen de otras como Dinamic (hoy FX Interactive), con los hermanos Nacho y Víctor Ruiz, de la que destacan títulos como *Game Over*, *Phantomas*, *Army Moves*, *Fernando Martín* o el mismísimo *PC Fútbol*; Opera Soft (Goody, *La abadía del crimen*, *Sol Negro*, *Livingstone Supongo*); Erbe (*Operation Wolf*, *Chicago's 30*); Topo Soft (*Mad Mix Game*, Emilio Butragueño, *Silent Shadow*, *Black Beard*) o Made in Spain (*Fred*, *Sir Fred*, *El Misterio del Nilo*).

D. LA DÉCADA DE LOS NOVENTA: DE LA REVOLUCIÓN MULTIMEDIA (1987-1996) AL 3D

A finales de la década de los ochenta, con los procesadores de 16 bits aumentaron las capacidades multimedia de los ordenadores y videoconsolas, que podían integrar audio, gráficos, animación y texto.

En este contexto, Sega lanzó la consola Sega Genesis Megadrive, en 1989, con 16 bits y con el videojuego *Sonic the Hedgehog* (1991) por bandera, desbancando a Nintendo en el panorama internacional. Empezó así la guerra de las consolas. Nintendo reaccionó con su Super NES (1990) y videojuegos como *Super Mario World* (1990), *Super Mario Kart* (1992), *Donkey Kong Country* (1994) o *Star Fox* (1993). Por otro lado, y en paralelo a esta guerra con Sega, Nintendo abrió un nuevo mercado con la primera consola portátil, la Game Boy.

Ya en la segunda mitad de la década de los noventa empezaron a aparecer nuevos géneros en el sector de los videojuegos, que siguen vigentes hoy en día, como la estrategia en tiempo real: *Dune 2* (1992), *Warcraft* (1994) o *Starcraft* (1998); y, también, First Person Shooter, a partir de *Doom* (1993).

En el ámbito de las recreativas, el género de la lucha tomó nuevos aires con juegos como *Street Fighter II* (1991) o *Mortal Kombat* (1993), el primer videojuego en utilizar fotografías de personas reales como personajes, y uno de los videojuegos más polémicos por su violencia.

Además siguieron apareciendo nuevas generaciones de consolas (de quinta generación), basadas en 32 bits principalmente, que empezaron a explotar las capacidades 3D: PlayStation (1994), Sega Saturn (1994), y la que rompió con los 32 bits, Nintendo 64 (64 bits). En este momento aparecieron títulos como *Tomb Raider* (1996), *Turok: Dinosaur Hunter* (1997), *Super Mario 64*, *Final Fantasy VII* (1997), *Metal Gear Solid* (1998), *Resident Evil* (1996), *Castlevania: Symphony of the night* (1997), *Legend of Zelda: Ocarina Of Time* (1998), *Unreal* (1998) o *Half-Life* (1998).

E. DÉCADA DEL 2000: INTERNET

Los videojuegos llegan a un período de madurez en la década del 2000, y se expanden gracias a Internet y a la conexión a esta de las videoconsolas de sexta generación, que abre las puertas a los juegos multijugador y de mayor sociabilidad. Videojuegos como *Ultima Online* (1997), *Unreal* (1998) o *Half-Life* (1998), que aparecieron a finales de los noventa, ya ofrecían la capacidad multijugador a través de la red; ahora se suman a ellos juegos como *The Sims* (2000) o *Halo: Combat Evolved* (2001).

También es interesante destacar el videojuego *Grand Theft Auto III*, el primero de acción/aventura en mundo abierto.

En este contexto surgen los juegos de tipo MMOG (juegos masivos multijugador) como *World of Warcraft* (2004). El tipo MMORPG, el juego de rol masivo multijugador más jugado de la historia, así como su forma de distribución, aprovechaba Internet. En 2003, Valve lanzó Steam, la primera plataforma de distribución de videojuegos.

F. ÉPOCA MODERNA (2006 - ACTUALIDAD)

Este período podríamos decir que está marcado por nuevos dispositivos como los teléfonos móviles y todas sus posibilidades, así como por nuevos paradigmas en lo que respecta a dispositivos de control y realidad aumentada.

En 2007, Apple lanzó el iPhone, y un poco más tarde, en 2008, abrió su App. Este modelo ha sido seguido por el resto de plataformas, disponemos de Stores donde los juegos pueden descargarse de forma gratuita y ser monetizados mediante micropagos u ofrecer precios más asequibles a los bolsillos de los usuarios. Algunos de los videojuegos para dispositivos móviles más destacados son *Super Mario Run* (Nintendo, 2016), *Fire Emblem Heroes* (Nintendo, 2017), el conocido *Pokémon Go* (2016) o el popular *Angry Birds* (2009).

A principios de los años 2000 apareció un nuevo género: el de los juegos musicales, con el *Guitar Hero* por bandera. Además, surgieron nuevas videoconsolas en las que se renovó por completo la interacción con el usuario, como es el caso de la Wii o la Nintendo Switch.

Las redes sociales también jugaron un importante papel en esta época, ya que muchos juegos se integraron en estas, como es el caso de *Farmville* (2009) o *Clash Royale* (2016), aprovechando su capacidad de viralización para atraer nuevos jugadores.

Todo ello condujo al panorama actual, muy diverso en cuanto a videojuegos. Las tendencias actuales incluyen películas interactivas como *Heavy Rain* (Quantic Dream, 2010), *The Walking Dead* (Telltale Games, 2012), *Black Mirror: Bandersnatch* (Netflix, 2018) o el *Gato Caco* (Netflix, 2022); la realidad aumentada, con *Skylanders Battle* (Activision, 2016) o *Pokémon Go* (Niantic, 2016), y las plataformas de videojuegos como servicio, como Steam o los eSports, con eventos de gran popularidad y el mercado de videojuegos indie.



1.3. Conceptos sobre videojuegos

Vamos a ver algunos conceptos relacionados con el mundo de los videojuegos, con los que trabajaremos a lo largo de esta unidad y en la siguiente.

- **Sistema de juego.** Se trata de un término común al lenguaje de los juegos (no solo de los videojuegos), y hace referencia al conjunto de reglas que lo rigen de una manera coherente. Por ejemplo, el juego del tres en raya tendrá un sistema de juego concreto, así como un RPG (Role Playing Game) tendrá sus propias reglas.
- **Mecánicas del juego.** Hace referencia a los medios ofrecidos al jugador para intervenir en el estado del juego, realizando diferentes acciones sobre los objetos del juego según las reglas establecidas, para alcanzar el objetivo de este.
- **Gameplay o jugabilidad.** Se trata de un concepto habitual en las reseñas de videojuegos. El DRAE lo define como la «Facilidad de uso que un videojuego ofrece a los usuarios». Este concepto puede resultar un tanto abstracto, pues se refiere a aquellas propiedades que pueden describir la experiencia del jugador ante un determinado sistema de juego. Ideas como el control, la accesibilidad o la capacidad de entretenimiento estarían relacionadas con la jugabilidad.
- **Potenciadores o Power-ups.** Son objetos del juego que aportan capacidades especiales instantáneas, como inmunidad, mayor velocidad, más fuerza, etc.
- **HUD o Head-Up Display.** Es la información que se muestra en la pantalla durante el juego: las puntuaciones, el tiempo restante, el número de vidas... y que puede mostrarse combinando imagen (íconos) y texto.
- **Assets o recursos.** Hace referencia a todos los elementos que componen el juego: modelos, texturas, imágenes, scripts de código, sonido, animaciones, etc.
- **Frame o fotograma.** Entendiendo un videojuego como una experiencia visual interactiva, un fotograma sería cada una de las imágenes que componen las animaciones. Debemos tener en cuenta que las imágenes que se representan en la pantalla son planas, en 2D. Para crearlas, se debe determinar el valor de cada píxel en función de todos los elementos que hay en una escena, considerando la iluminación, las transparencias o los reflejos. Este proceso comporta muchos cálculos, que especialmente en el caso de los videojuegos deberemos simplificar, puesto que se realiza en tiempo real.
- **Framerate.** Indica la frecuencia a la cual se muestran los diferentes frames, y suele expresarse en Frames Per Second (fps). A mayor frame rate, tendremos mayor fluidez en las animaciones. El estándar en cine es de 24 fps y en animación suele ser de 30 fps; en el caso de los videojuegos esta ratio se sitúa entre los 30 y los 60 fps, siendo este último valor el deseable.
- **Texturas.** Imágenes 2D (bitmap) que podemos mapear sobre un objeto 3D simulando su superficie (metal, madera, etc.).
- **Sprite.** Un sprite es un tipo de bitmap que se crea en la pantalla del ordenador a partir de una imagen. Aplicado al mundo de los videojuegos 2D, son un conjunto de imágenes que representan de forma gráfica personajes, objetos o parte de ellos, y crean efectos de movimiento o cambios de estado. Relacionándolo con el concepto anterior, podríamos decir que se trata de una textura asociada a un objeto 2D.

1.4. Géneros

Los videojuegos han ido evolucionando, evidentemente, en función de las capacidades de cómputo y las capacidades gráficas, pero también según han ido apareciendo nuevas mecánicas y formas de aplicar las reglas del juego, dando lugar así a diferentes géneros que clasifican los videojuegos de acuerdo con una jugabilidad y una mecánica concretas. Establecer una clasificación por género de videojuegos puede ser una tarea bastante subjetiva, y hay que tener en cuenta que existen juegos que no se adaptan completamente a un género o que pertenecen a varios.

Una posible clasificación de los videojuegos podría ser la siguiente:

- **Juegos de acción.** Requieren de las habilidades y la coordinación del jugador. Dentro de estos, podemos encontrar:
 - **Juegos de plataformas.** Juegos en los que el personaje se desplaza por superficies y plataformas evitando obstáculos y enemigos. Ej. *Mario Bros, Sonic the Hedgehog*.

- **Terror y supervivencia (Survival Horror).** Juegos de terror donde el jugador debe sobrevivir hasta el final. Ej. *Resident Evil*.
- **Beat'em Up.** Juegos clásicos de peleas donde el jugador se desplaza por un escenario enfrentándose a enemigos. Ej.: *Golden Axe, Double Dragon I/II/III, Renegade*.
- **Juegos de lucha.** Juegos donde dos oponentes luchan entre ellos. Ej.: *Street Fighter I/II, Tekken*.
- **Hack'n'slash.** Juegos que enfatizan el combate masivo. Ej.: *God of War, Attack on Titan*.
- **Juegos de disparos (Shooters):** Algunos autores lo consideran un subgénero de los juegos de acción. En este tipo de videojuegos el personaje usa armas de fuego para abrirse camino en el juego. Dentro de este género encontramos:
 - **First Person Shooters (FOS).** Juegos de disparos con vista en primera persona. Ej. *Doom, Call of Duty*.
 - **Third Person Shooters (TPS).** Juego de disparos con vista en tercera persona, se ve al personaje desde atrás. Ej. *Tomb Raider*.
 - **Matamarcianos.** Juego de disparos en perspectiva en 2D, donde manejamos una nave espacial (o, en general, cualquier vehículo en el aire). Un ejemplo clásico sería *Space Invaders*.
 - **Shoot'em Up.** Con una dinámica similar a los juegos beat'em'up, pero basados en disparos, el personaje avanza por los escenarios, generalmente en perspectiva lateral, disparando. Ejemplos de este subgénero serían *Contra* (Konami, 1987) o *Metal Slug* (SNK, 1996).
- **Juegos de estrategia.** Basados en conflictos estratégicos, el jugador puede escoger múltiples acciones en cada momento. Suelen dividirse en juegos de estrategia en tiempo real (*Age Of Empires, Warcraft*) o de estrategia por turnos (*Civilization, Total War*).
- **Juegos de rol.** Juegos donde el jugador controla uno o varios personajes que van aumentando sus habilidades y poderes a lo largo de una historia. Se caracterizan por la interacción con los personajes y una historia más profunda. Como ejemplos, podemos mencionar *Final Fantasy, Dungeons & Dragons* o *Dragon Quest*.
- **Juegos de aventura.** En estos juegos se controla a un personaje a lo largo de una historia. Suelen contar con acción, elementos narrativos, exploración o resolución de puzzles. Dentro de este género podemos encontrar la aventura conversacional, basada principalmente en texto y que fue muy popular en los ochenta (*Carvalho: Los pájaros de Bangkok*, Dinamic 1988); la aventura gráfica, que usa el ratón en lugar de introducir las acciones por teclado (*Monkey Island, Maniac Mansion*), y el subgénero acción-aventura, que mezcla ambos géneros (*The Legend of Zelda*).
- **Juegos de simulación.** Simulan actividades o situaciones del mundo real. Dentro de este género, encontramos:
 - **Simulación de vehículos.** Se simula la conducción de diferentes tipos de vehículos. Como ejemplos, destacamos *Gran Turismo* o *Flight Simulator*.
 - **Simulación de construcción.** Se gestiona la construcción de proyectos, sea ciudades (*SimCity*), imperios (*Civilization*), sandbox (*Minecraft*), política (*Democracy*) o gestión deportiva (*PC Futbol, Football Manager*).
 - **Simulación de vida.** El jugador controla formas de vida artificial. Suelen clasificarse en videojuegos de simulación social, como *The Sims*, de mascotas virtuales, como los populares *Tamagotchi* o *Pou*, o de simulación biológica, como *SimLife*.
- **Juegos deportivos.** En estos se simula cualquier tipo de deporte del mundo real. Como ejemplos, podemos citar *FIFA* (fútbol), *Pro Evolution Soccer* (fútbol), *NBA Live* (baloncesto) o *Tony Hawk's Pro Skater* (skate), entre muchos otros.
- **Juegos de competición.** Videojuegos que pueden considerarse dentro del ámbito de los deportivos, y que consisten en carreras, generalmente con vehículos a motor, en las que el jugador debe llegar a la meta en un tiempo determinado o antes que sus contrincantes. Podemos citar como ejemplos *Mario Kart* o *GTA*.
- **Juegos de puzzles o de agilidad mental.** Son juegos donde se requiere de habilidad y razonamiento lógico y espacial. Algunos ejemplos son *Tetris, Monument Valley...*
- **Party Games.** Son juegos con un gran componente social, donde varios personajes son controlados por personas en un juego de tablero y participan en múltiples mini juegos. Ejemplos de este género son toda la saga *Mario Party, Rayman, Sonic Shuffle* o *Among Us*.
- **Juegos musicales.** Requieren de habilidades musicales o de baile (*Guitar Hero, Just Dance, Singstar*).

2. Motores de videojuegos

La complejidad en el desarrollo de videojuegos ha ido creciendo a lo largo de la historia. En sus inicios, y a principios de la década de los ochenta, los juegos se programaban desde cero y, además, requerían de un desarrollo específico para cada plataforma. Esto implicaba hacer grandes inversiones por parte de las empresas desarrolladoras. Recordemos que en 1983 se produjo una crisis en la industria americana por todo ello; entonces, las empresas empezaron a desarrollar sus propios motores para uso interno, de manera que pudiesen reutilizar código desarrollado previamente.

A partir de videojuegos como *Wolfenstein 3D*, *Doom* o *Quake*, los primeros FPS (First Person Shooter), en la década de los noventa se empezaron a distribuir los propios motores y, con el tiempo y la maduración de las tecnologías, empezaron a utilizarse en otras áreas como la medicina o las simulaciones.

El motor que se usó para el desarrollo de estos tres FPS, y tras diferentes reescrituras, dio lugar a la primera versión del motor Unreal, en 1998. Un poco más tarde, en 2005, vio la luz el motor Unity3D.

Con todo esto, ya podemos entender por qué los motores son hoy en día una herramienta imprescindible a la hora de desarrollar videojuegos, pues facilitan muchas tareas comunes en dicho desarrollo.

Podemos entender un motor de videojuegos, o game engine, como un conjunto de herramientas de desarrollo que permiten el diseño, la creación y la representación de un videojuego.

Otra de las ventajas del uso de motores de juegos es que hacen posible el desarrollo de videojuegos para múltiples plataformas, desde consolas hasta juegos de PC, web o dispositivos móviles.

2.1. Componentes

Como hemos comentado anteriormente, un motor de videojuegos es un conjunto de herramientas que facilitan la creación de videojuegos. Vamos a ver cuáles son las herramientas o componentes del motor:

- **El editor.** Es el elemento con el que tendremos un contacto más directo. Con él creamos las escenas, importamos los diferentes recursos; editamos los scripts (programas), las animaciones, los modelos, la interfaz y el sonido, o previsualizamos el propio juego.
- **Motor de físicas.** Esta funcionalidad aplica los principios de la física a los diferentes elementos de un videojuego y sus interacciones. Los cálculos se simplifican y optimizan, ya que se requieren para el renderizado de cada frame. Este componente hace posible que un objeto simule atributos físicos como el peso, volumen, la gravedad o la aceleración.
- **Motor de sonido.** Es el encargado de cargar las pistas de audio, adaptarlas y reproducirlas de forma sincronizada.
- **Motor de scripting.** Los diferentes objetos de un juego tienen definido su comportamiento mediante scripts, escritos en algún lenguaje de programación.
- **Motor gráfico.** Seguramente es el componente más relevante de un motor de juegos, el encargado de realizar los cálculos necesarios para mostrar las imágenes 2D y 3D en la pantalla, teniendo en cuenta aspectos como la iluminación, los polígonos o las texturas.

Además, un motor también incorpora otros componentes: generadores de sistemas de partículas, editores de terrenos o gestión de LOD (Level Of Detail).

2.2. Motores de juegos

En este apartado veremos algunos de los motores de juegos más relevantes o que mayor influencia han tenido.

- **Unreal Engine.** Creado por Epic Games en 1998, tuvo una nueva versión en 2014 (Unreal Engine 4) y se espera la próxima versión, Unreal Engine 5, para 2022. Unreal se utiliza en empresas como Electronic Arts y Ubisoft, y utiliza C++ para su programación.
- **CryEngine.** Creado por Crytek en 2004, tiene una filosofía muy unida a la investigación, con gráficos ultrarealistas y físicas muy avanzadas. Su mayor éxito fue *Far Cry* (2004).
- **Unity 3D.** Fue desarrollado por Unity Technologies en 2005 con disponibilidad para múltiples plataformas. Su principal propósito es hacer lo más accesible posible el desarrollo de contenido interactivo 2D

y 3D, por lo cual proporciona licencias gratuitas para desarrolladores independientes. Este motor utiliza C# para su programación.

- **Godot.** Fue creado en 2007 por Juan Linietsky y Ariel Manzur. Se trata de un motor 2D y 3D multiplataforma, distribuido bajo licencia de software libre, y que permite las exportaciones a PC, móviles y web. Aunque está desarrollado en C y C++, los scripts se crean en GDScript, muy parecido a Python.
- **Amazon Lumberyard.** Basado en CryEngine, se trata de un motor multiplataforma, gratuito e integrado con AWS, lo que permite construir y almacenar los juegos en los servidores de Amazon. Permite el desarrollo para plataformas Windows, PS4, XBox One, y limitado para iOS y Android. Aunque todavía se encuentra en fase beta, ya se han desarrollado varios videojuegos con este motor.

Respecto a otros motores propietarios o profesionales, podemos nombrar Frostbite Engine, de Digital Illusions CE, Decima Engine o Luminous Engine.

2.3. El motor Unity

Como ya hemos comentado, Unity fue creado por Unity Technologies en 2005, y engloba diferentes componentes para la gestión de físicas 2D y 3D, sonido, animaciones o renderizado, entre otras. Unity permite generar proyectos para varias plataformas, tanto móviles como de escritorio, web o videoconsolas, y ofrece varios servicios, una vasta documentación y una amplia comunidad de usuarios.

El lenguaje de programación que utiliza este motor es C#, muy parecido a Java y desarrollado por Microsoft. Con el tiempo, el lenguaje se ha estandarizado y se han creado implementaciones libres. Una de estas implementaciones, Mono, es en la que se basa Unity.

A.PLATAFORMAS

El editor de Unity está disponible en los sistemas operativos más comunes: Windows, MacOS y Linux. Podéis encontrar los requerimientos de hardware y software para la versión 2020.3 (LTS) en la documentación oficial de Unity.

En general, se requiere de una tarjeta gráfica con soporte de DirectX 10 (Shader Modelo 4.0) para Windows; OpenGL 3.2 o superior en Linux, y un procesador de 64 bits con el juego de instrucciones SSE2.

En cuanto a las plataformas a las que es capaz de exportar, encontramos dispositivos de todo tipo: PC, móviles, videoconsolas, dispositivos de realidad virtual y aumentada, y un largo etcétera. Podéis consultar el listado completo en la misma página de documentación.

B. UNITY ID. LICENCIAS

Para trabajar con Unity se requiere de una cuenta de usuario (Unity ID) con la que vincular los diferentes servicios, compras o descargas que se realicen.

Respecto a las licencias, Unity plantea un modelo de suscripción con las opciones siguientes:

Personal	Plus	Pro	Empresa
Gratis	399 USD /año por puesto	1.800 USD /año por puesto	4.000 USD /mes por 20 puestos
Comienza a crear con la versión gratuita de Unity.	Más funcionalidad y recursos para potenciar proyectos.	Soluciones completas para que los profesionales creen y operen.	Éxito a gran escala para organizaciones grandes con objetivos ambiciosos.
Elegible si los ingresos o los fondos son inferiores a 100.000 USD en los últimos 12 meses.	Elegible si los ingresos o los fondos son inferiores a 200.000 USD en los últimos 12 meses.	Si los ingresos o fondos han sido superiores a 200.000 USD en los últimos 12 meses, hay que usar un plan Pro o Empresa.	20 puestos mínimo. Si los ingresos o fondos superan los 200.000 USD en los últimos 12 meses, hay que usar un plan Pro o Empresa.

Podéis encontrar más información sobre las licencias y sus características en la comparativa de planes de la documentación de Unity. Dentro de los planes personales, también existe la opción de utilizar licencias para estudiantes, con algunas ventajas más que la licencia personal.

En el caso práctico de este apartado veremos paso a paso cómo crear nuestro Unity ID.

2.4. Unity Hub

Unity Hub es la herramienta que permite gestionar, de manera centralizada, cuentas de Unity, licencias, proyectos y diferentes instalaciones (versiones) del editor de Unity, junto con los diferentes módulos de este.

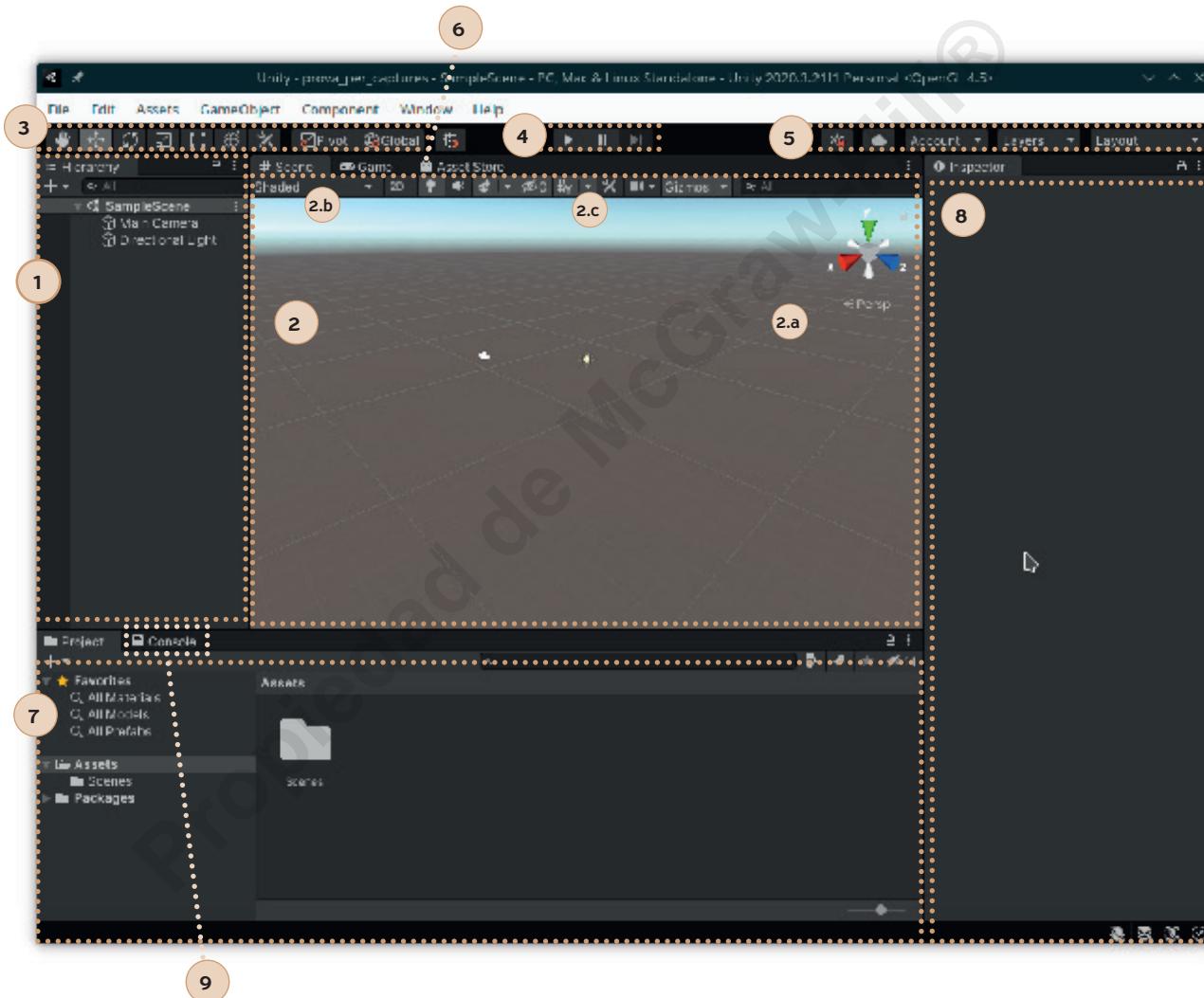
En el caso práctico de este apartado veremos un pequeño tutorial con la instalación y uso de Unity Hub, y aprenderemos a gestionar con él nuestras licencias y componentes de Unity.

2.5. El editor de Unity

El componente con el que más a menudo trabajaremos en Unity es el editor, donde modelaremos y daremos forma a los diferentes componentes de nuestro juego.

A. VISTAS DE UNITY

La interfaz de Unity presenta el aspecto general siguiente, dividido en varias pestañas o vistas:



- 1. Ventana de Jerarquía (Hierarchy View).** Contiene el árbol de la escena, con todos los elementos que la componen ordenados jerárquicamente. Si un nodo es padre de varios componentes, los cambios que realicemos en este afectarán a todos los nodos hijos. Podemos añadir elementos, con el botón + o con el botón derecho del ratón, ordenarlos, establecer relaciones jerárquicas u ocultarlos y volver a mostrarlos mediante el ícono lateral de un ojo. Por defecto, se muestra una escena de ejemplo (*Sample Scene*) con un objeto de tipo cámara principal (*Main Camera*) y, si nos encontramos en un proyecto 3D, un objeto de luz direccional (*Directional Light*). En el caso de los proyectos 2D, solo tendremos la cámara.
- 2. Vista de escena (Scene View).** Esta es la escena más importante del editor, donde se muestran los diferentes elementos que componen la escena, y que podemos manipular de forma interactiva. Para movernos a través de esta ventana podemos hacerlo de diferentes formas: mediante el teclado (cursores), el ratón (botón derecho) o ambos (botón derecho + teclas WASD QE), como si de un FPS se tratase. Además al hacer clic en algún elemento del árbol de jerarquía, se centra la vista en el objeto seleccionado.

A parte de los objetos, en esta vista también tenemos algunas herramientas adicionales:

- **2.a) Gizmo.** Muestra la relación entre el sistema de coordenadas del mundo que estamos creando y la cámara, y nos permite cambiar entre las diferentes vistas de la escena [alzado, planta y perfil] haciendo clic sobre los ejes, así como, también, cambiar la proyección de la cámara.
- **2.b) Modos de vista de la escena.** La vista de escena contiene un desplegable con varios modos de visualización de esta:
 - **Shading mode.** Establece el tipo de sombreado para la representación de objetos en la escena. Podemos elegir entre *Shaded*, con las superficies sombreadas, *Wireframe*, con las mallas 3D, y *Shaded Wireframe* con sombreado y las mallas 3D superpuestas.
 - **Miscellaneous.** Muestra varias opciones de visualización, como la activación de luces direccionales, el canal alfa (transparencia), la superposición de superficies, texturas...

El resto de opciones permiten otros tipos de visualizaciones relativas principalmente a la iluminación, la proyección de sombras, etc. Podéis consultar más acerca de los modos de visualización en la documentación oficial.

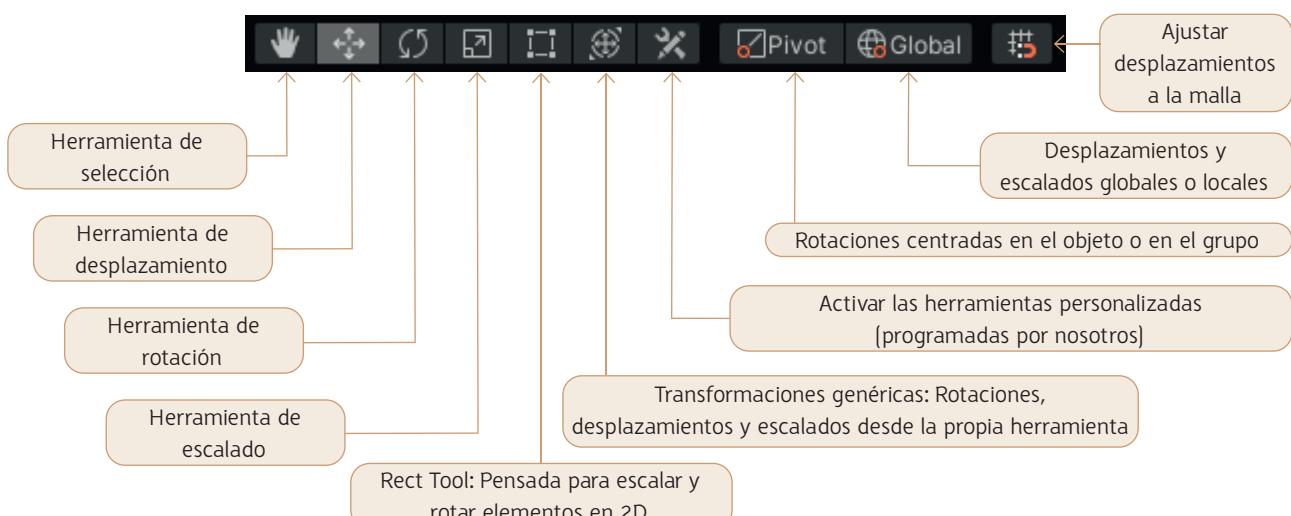
- **2.c) Opciones de vista.** Siguiendo con la barra de control de la vista de la escena, tenemos varias herramientas más para trabajar:
 - Botón 2D para comutar entre la vista 2D y 3D.
 - Botón para comutar entre la iluminación de la escena (con las luces que tengamos definidas) o utilizar una iluminación por defecto.
 - Botón para habilitar o deshabilitar los efectos de audio de la escena.
 - Botón para mostrar u ocultar otros aspectos de visualización, como el cielo artificial (skybox), las partículas, la niebla (fog) o materiales animados, como pueda ser el agua. Podemos habilitarlos e inhabilitarlos individualmente, con la flecha desplegable, o en conjunto, con el mismo botón.
 - Botón para habilitar o deshabilitar las opciones de visibilidad de la jerarquía. Cuando en el árbol de la escena ocultamos algunos elementos, aquí se indica cuántos elementos hay ocultos y podemos habilitar o inhabilitar este ocultamiento.
 - Botón para ajustar las opciones de visualización de la malla del suelo.

En la parte derecha de esta barra tenemos las herramientas siguientes:

- Botón para mostrar el panel de herramientas del editor de componentes (inicialmente vacío).
- Menú de ajustes de la cámara, donde ajustar los planos lejano y próximo, el campo de visión, la velocidad, etc.
- Menú de gizmos. Los gizmos son iconos y representaciones de los objetos de la escena, como la cámara o las luces. Desde este menú podemos desactivarlos, cambiarlos de modo o ajustar algunas propiedades.
- Caja de búsqueda para buscar elementos en la escena, por nombre o tipo.

3. Herramientas de transformación.

Permiten transformar los objetos y hacer varios ajustes sobre estas transformaciones.

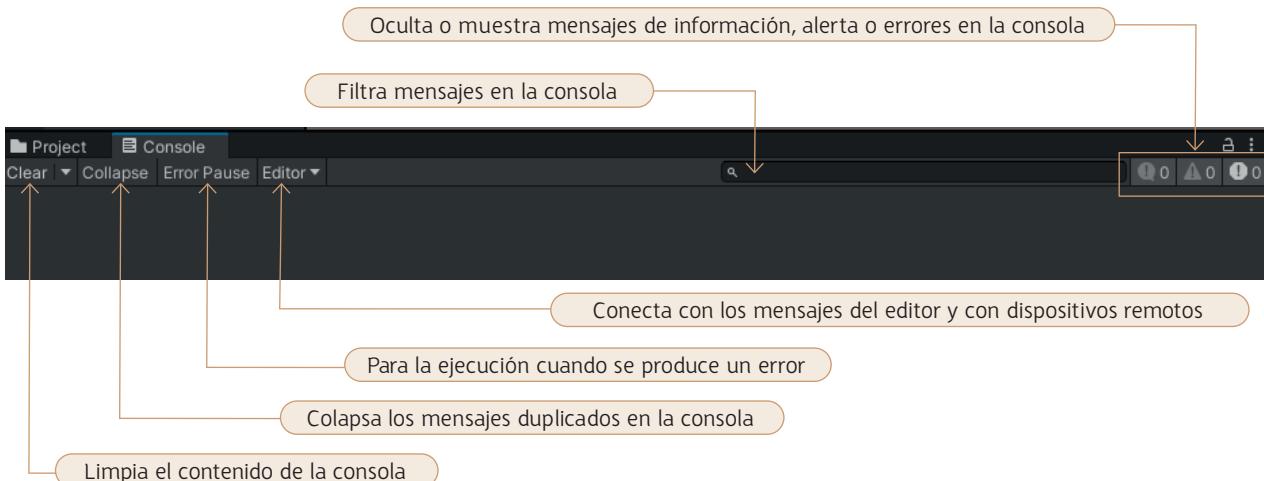


4. **Botones de reproducción.** Permiten ejecutar o parar el juego, pausarlo, o avanzar frame a frame. Durante la ejecución del juego hay que tener en cuenta que estaremos en la vista *Juego*, no en *Escena*, por lo que todos los cambios que hagamos afectarán a la ejecución actual y no a la escena en sí. Para no confundir ambos modos, se suele establecer un color diferente para el modo de ejecución desde las preferencias.
5. **Otras opciones** sobre control de versiones, herramientas y servicios de Unity, información de nuestra cuenta, capas y selección de la disposición de la interfaz de Unity.
6. **Vista de juego.** Es una representación de lo que ve el jugador. En la disposición por defecto, esta vista se encuentra en forma de pestaña al lado de la vista de escena. En otras disposiciones, esta vista está visible en todo momento. Si la activamos, en ella encontramos las opciones siguientes:
 - **Display.** Muestra la vista de diferentes pantallas (*Displays*) asociadas a diferentes cámaras. Cuando añadimos una cámara nueva, es útil alinear esta con la vista actual mediante la combinación de teclas **Ctrl + Shift + F**.
 - **Aspect Ratio.** Determina la relación entre el alto y el ancho en la cámara. Por defecto, utiliza *Free Aspect*, de forma que se ajusta a la ventana del editor en que estamos trabajando. Esta opción no es recomendable, y lo más habitual es trabajar con una relación de aspecto que se corresponda con el dispositivo sobre el que vayamos a lanzar la aplicación (16:9, Full HD, etc.).
 - **Scale.** Para seleccionar la escala que se muestra en la ventana de juego. Es útil cuando utilizamos una resolución mayor que la que permite nuestro monitor o ventana.
 - **Maximize on Play.** Maximiza la pantalla de juego cuando entramos en modo juego.
 - **Mute.** Silencia los efectos de sonido de la escena.
 - **Vsync.** Permite forzar el uso de la sincronización vertical en la ventana del juego. Con ello, no se dibuja un frame hasta que no termine de dibujar el anterior, evitando imágenes erróneas.
 - **Stats.** Muestra información estadística sobre el rendimiento del juego.
 - **Gizmos.** Permite activar en la vista del juego los gizmos de luz, colisionadores, etc.



7. **Vista de proyecto.** Muestra los recursos (*assets*) que están disponibles en el juego. Se organiza en carpetas, y se corresponde al directorio *Assets* del proyecto del juego. Si examinamos las carpetas físicas de nuestro proyecto, observamos que dentro hay más carpetas y ficheros, como los ficheros de tipo Meta con la metainformación que gestiona Unity sobre los objetos, como la versión o el identificador. Es importante realizar la gestión de los ficheros del proyecto desde el propio Unity y no desde el sistema de ficheros, puesto que Unity necesita gestionar toda esta información.
8. **Inspector.** Se trata de un panel dinámico donde el contenido cambia en función del objeto seleccionado, mostrando sus propiedades o componentes y permitiendo su modificación.

9. Consola. Muestra los mensajes de registro de Unity y de nuestro programa. Presenta las opciones siguientes:



B. PERSONALIZACIÓN DEL ENTORNO

Como toda aplicación, podemos ajustar el entorno a nuestras preferencias y necesidades. Algunas de las opciones interesantes que encontraremos son las siguientes:

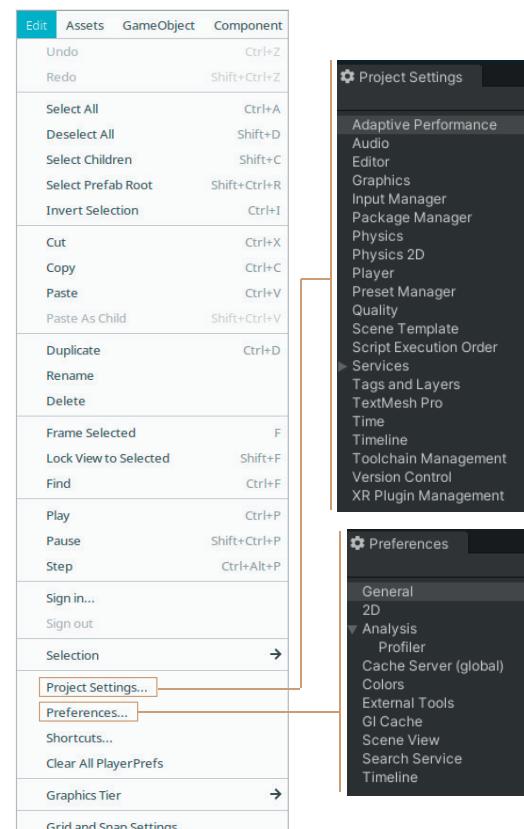
- Maximizar una de las ventanas.** Para maximizar o minimizar ventanas activamos la opción *Maximize*, bien en el menú contextual que nos aparece al hacer clic con el botón derecho del ratón sobre el título, bien utilizando el menú con los tres puntos a la derecha de la ventana.
- Cerrar la pestaña seleccionada.** Se lleva a cabo desde el mismo menú contextual, con la opción *Close Tab*. También podemos añadir nuevas ventanas con *Add Tab*.
- Unity permite también arrastrar las pestañas, para ubicarlas en otras partes o dejarlas como flotantes.

Si deseamos guardar la disposición que hemos configurado, podemos hacerlo mediante *Save Layout* en el menú *Layout* de la parte superior derecha.

C. BARRA DE MENÚ

En la parte superior de la interfaz de Unity disponemos de un menú general. Muchas de las opciones que contiene ya están integradas en los distintos paneles para facilitar su acceso. Las diferentes opciones generales de este menú son las siguientes:

- File.** Para gestionar los proyectos y las escenas, así como para hacer los ajustes de la construcción del juego (*Build Settings*).
- Edit.** Con las opciones de edición habituales (copiar, pegar, selección), junto con las opciones de reproducción o acceso a la configuración general y del proyecto, entre otras.
- Assets.** Para gestionar los recursos de nuestro proyecto. Es prácticamente el mismo menú que el que obtenemos al pulsar el botón derecho sobre la vista de los recursos.
- GameObject.** Contiene las mismas posibilidades que el menú contextual de la vista de la escena para añadir objetos, y también opciones para organizarlos.
- Component.** Muestra, clasificados por categorías, los diferentes componentes que podemos asociar a nuestros objetos (físicas, etc.).
- Window.** Muestra los diferentes paneles y ventanas de Unity, organizados por categorías.
- Help.** Con enlaces a la documentación.



D. GESTIÓN DE LAS PREFERENCIAS GENERALES Y DEL PROYECTO

En el menú *Edit* existen dos entradas para gestionar, por un lado, las preferencias de Unity y, por otro, la configuración del proyecto:

PROJECT SETTINGS		
Project Settings	Adaptative Performance	Información de rendimiento, batería o temperatura de un dispositivo móvil conectado
Adaptive Performance	Audio	Volumen, modos de sonido, plugins...
Editor	Editor	Configura ajustes globales para el editor de Unity
Graphics	Graphics	Configura el modo en que se generan los gráficos
Input Manager	Input Manager	Configura el sistema de entrada
Package Manager	Package Manager	Configura aspectos avanzados de nuestro proyecto
Physics	Physics	Configura el motor de físicas 3D
Physics 2D	Physics 2D	Configura el motor de físicas 2D
Player	Player	Configura la construcción del juego para las diferentes plataformas configuradas
Preset Manager	PresetManager	Plantillas de configuración por defecto de diversos recursos
Quality	Quality	Opciones de calidad del proyecto (sombreado, renderizado, etc.)
Scene Template	Scene Template	Permite ajustar las propiedades por defecto de las nuevas escenas
Script Execution Order	Script Execution Order	Permite cambiar el orden de ejecución de los scripts
Services	Services	Permite configurar ciertos servicios para atraer y retener usuarios o para monetizar el juego
Tags and Layers	Tags and Layers	Permite gestionar capas y etiquetas del juego
TextMesh Pro	TextMesh Pro	Instala componentes adicionales para TextMesh Pro, un sistema avanzado de renderizado de texto
Time	Time	Permite modificar aspectos relacionados con la escala de tiempo del juego
Timeline	Timeline	Permite ajustar el framerate
Toolchain Management	Toolchain Management	Para el soporte a construcciones en Linux
Version Control	Version Control	Permite hacer visibles o invisibles los ficheros de metainformación para el control de versiones
XR Plugin Management	XR Plugin Management	Gestión de plugins de realidad virtual y aumentada

PREFERENCES		
Preferences	General	Comportamiento general de Unity
General	2D	Tamaño de la caché para la caché de sprites en proyectos 2D.
2D	Analysis	Preferencias de la herramienta Profiler, para obtener información sobre el rendimiento
Analysis	Cache Server	Configura un servidor de caché para optimizar la recarga de assets
Profiler	Color	Personalización de colores de la interfaz
Cache Server (global)	External Tools	Configura las herramientas externas para edición de código, assets, SDK, etc.
Colors	GI Cache	Caché del sistema de iluminación global
External Tools	Scene View	Configura la vista de escena
GI Cache	Search Service	Para configurar las búsquedas de objetos, proyecto o escenas
Scene View	Timeline	Para los ajustes de tiempo de secuencias de audio, efectos de partículas o gameplay
Search Service		
Timeline		

En la documentación oficial de Unity podemos obtener información más detallada sobre todas estas opciones.

E. EL GESTOR DE PAQUETES

Otra opción interesante que podemos encontrar dentro de Unity es el gestor de paquetes, *Window > Package Manager*, desde donde podremos actualizar los paquetes instalados (publicidad, realidad virtual, físicas, etc.) o instalar otras herramientas en forma de paquetes. Como en todo sistema de paquetes, estos pueden depender unos de otros.

3. Trabajando con Unity 3D

Ahora que ya nos hemos familiarizado un poco con el entorno y el editor de Unity, tomaremos contacto con algunos de los conceptos y elementos que lo componen, y con los que trabajaremos a lo largo de estas unidades.

3.1. GameObjects y sus componentes

A. GAMEOBJECTS

Los GameObjects son los objetos más importantes de Unity, ya que se trata de las piezas que componen el juego: personajes, objetos... Todos los elementos que aparecen en una escena, incluyendo la cámara, las luces y el sonido, son GameObjects.

Para agregar un nuevo GameObjects, podemos hacerlo a través de la entrada *GameObject* del menú principal o desde la jerarquía de la escena, mediante el botón + o el menú contextual. Los diferentes tipos de GameObjects que podemos encontrar se agrupan del modo siguiente:

Create Empty	Shift+Ctrl+N	Empty/Empty Child/Empty Parent: GameObject vacío.
Create Empty Child	Shift+Alt+N	2D Object: Objetos 2D (en proyectos 2D) como sprites, mapas, etc.
Create Empty Parent	Shift+Ctrl+G	3D Object: Objetos 3D simples como cubos, esferas, cápsulas, cilindros, u otros más complejos como árboles y terrenos.
2D Object	→	Effects: Efectos, principalmente sistemas de partículas, pequeñas imágenes 2D para simular fluidos, humo, fuego o nubes.
3D Object	→	Light: Luces, que pueden ser direccionales, puntuales o de áreas.
Effects	→	Audio: Fuentes de Audio y reproducción de Video.
Light	→	Video: UI: Con diferentes elementos para crear interfaces de usuario.
Audio	→	Camera: Para añadir nuevas cámaras.
Video	→	
UI	→	
Camera	→	

Los GameObjects que tengamos en la escena se mostrarán en la ventana de jerarquía de esta, desde donde podremos combinarlos y organizarlos, e, incluso, establecer relaciones jerárquicas entre ellos, así como eliminarlos, copiarlos o crear duplicados.

B. COMPONENTES

Los GameObjects por sí mismos no poseen ninguna funcionalidad, son sus componentes los que le proporcionan estas, entendiendo, por tanto, un GameObject como un contenedor de componentes. Los componentes asociados a un GameObject los podremos ver en la ventana del Inspector. Todo GameObject, posee, como mínimo, un componente de tipo Transform, para indicar la posición, la rotación y la escala del GameObject. Si añadimos un objeto vacío (Empty), únicamente tendrá este componente Transform. En cambio, si añadimos, por ejemplo, un cubo en la escena, tendremos un componente de tipo Mesh Filter y otro del Mesh Renderer, el primero lee la información sobre el modelo 3D y lo pasa al segundo para que lo represente teniendo en cuenta la iluminación. Este cubo también tendrá un componente de tipo Box Collider, que se encarga de definir los límites del objeto para la detección de colisiones.

Podemos añadir componentes mediante el botón *Add Component* en el Inspector o desde el menú *Components*. Por ejemplo, el componente Rigidbody sirve para que se aplique el motor de física sobre el objeto.

Los valores de un componente se pueden modificar. Si queremos volver a los valores por defecto, podemos reiniciar el componente mediante la opción *Reset*.

También hay que tener en cuenta que existen componentes que dependen de otros, y los incorporan automáticamente, así como existen otros componentes que crean conflictos entre ellos. Por ejemplo, un Rigidbody y un Rigidbody 2D, ya que no podemos aplicar físicas 2D y 3D sobre el mismo objeto.

Podemos ver los diferentes componentes existentes en la entrada *Components* del menú principal, ordenados por categorías. Iremos viendo, poco a poco, los componentes que vayamos necesitando.

3.2. Prefabs

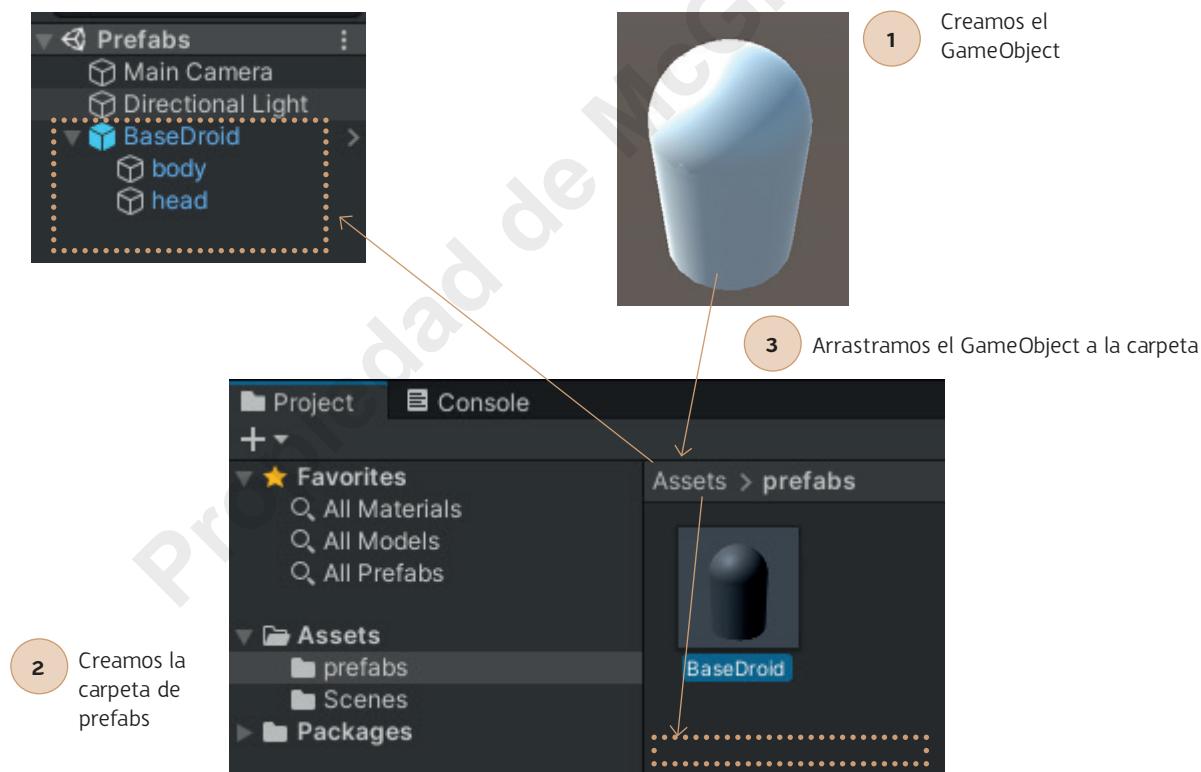
El sistema de prefabs de Unity permite crear, configurar y guardar, en forma de recurso (asset) reutilizable, GameObjects completos, con todos sus nodos hijos y sus componentes.

Básicamente, se trata de convertir en una plantilla (o una clase) un GameObject creado previamente, para poder crear otras copias a partir de él. Lo que realmente hacemos cuando creamos un objeto a partir de un prefab no es una copia, sino una instanciación del GameObject. Este mecanismo de instanciación podrá realizarse también desde el código, facilitando así la generación dinámica de objetos en el juego.

Otra de las principales ventajas de los prefabs es que los cambios que se hagan en el prefab se propagan de forma automática a todas sus instancias, lo cual es imposible en duplicados.

TRABAJANDO CON PREFABS

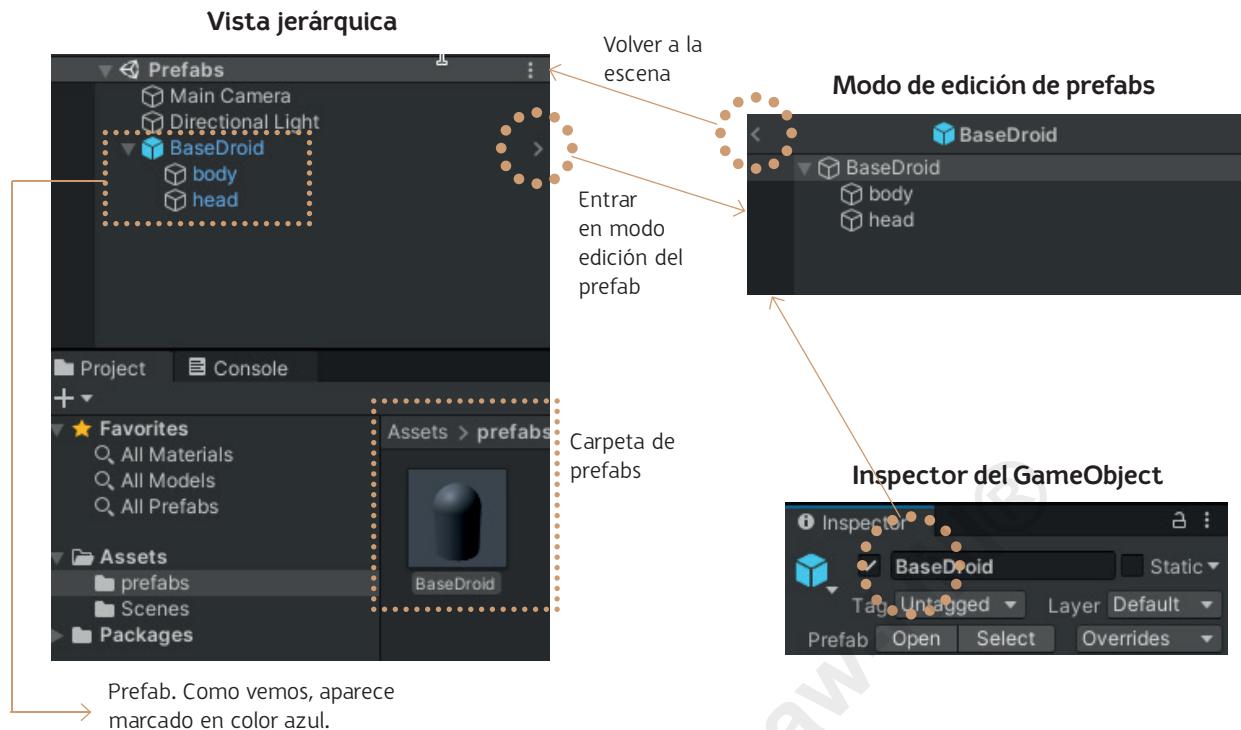
Para crear un prefab, se parte de un objeto generado en la escena que queremos utilizar como plantilla. Bastará con arrastrar el objeto a la ventana del proyecto para crear el prefab. Aunque podamos guardarlo en cualquier lugar, es preferible generarla en una carpeta específica para los prefabs, y, dentro de esta, organizarlos por categorías.



Ahora, para usarlo deberemos arrastrar y soltar el prefab a la escena o a la jerarquía de la escena.

Si lo que deseamos es editar un prefab, debemos hacerlo desde el modo prefab, un modo especial del editor que permite hacer modificaciones centradas en el prefab, y de forma independiente a la escena. Podemos acceder a este modo con el icono en forma de flecha al lado del prefab en el árbol de la escena, como hemos comentado más arriba, haciendo doble clic en el Asset del prefab desde la carpeta donde se encuentra o manteniendo el prefab seleccionado y haciendo clic en *Open Prefab* desde el Inspector.

Una vez hechas las modificaciones correspondientes en el prefab, podemos volver al modo de escena mediante la flecha de volver atrás que aparece en la jerarquía junto al nombre del prefab.



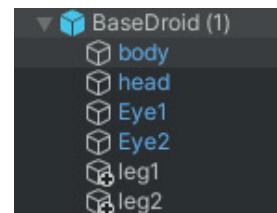
Mientras trabajamos en Unity con prefabs, es importante considerar lo siguiente:

- **Instance Override.** Cuando editamos un prefab, debemos tener en cuenta que estamos modificando el propio prefab, es decir, la plantilla, y los cambios que hagamos se verán reflejados en todas las instancias.

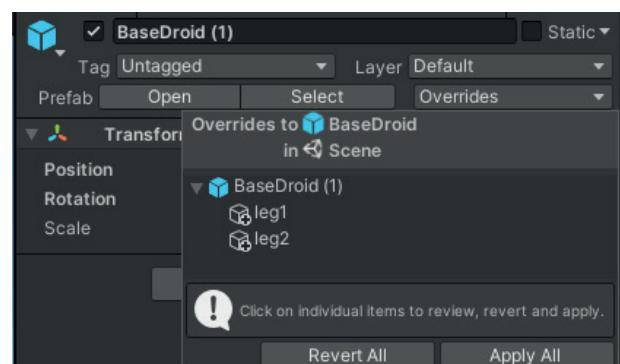
No obstante, es posible aplicar variaciones en las instancias que hayamos generado a partir del prefab, bien aplicando cambios en los valores de las propiedades, bien añadiendo o eliminando objetos hijos. Lo que no podremos hacer es eliminar GameObjects o componentes que sean hijos del prefab.

Si añadimos nuevos GameObjects a una instancia de un prefab, veremos que en el árbol de herencia aparece un símbolo (+) al lado de dicho GameObject, para indicar que este no pertenece al prefab original y que se añadió después.

Del mismo modo, si añadimos componentes adicionales a las instancias, se marcará también con una línea azul en el inspector. En caso de que modifiquemos algún valor de los componentes, estos se marcarán en negrita.



Además, si deseamos que las modificaciones que hemos incorporado a una instancia del prefab se apliquen, total o parcialmente al prefab original (Instance Override), podemos hacerlo mediante las diferentes opciones que están disponibles en la parte superior del Inspector, para revertir los cambios que hemos hecho, o aplicarlos a todo el prefab, de forma que se extiendan a las otras instancias.

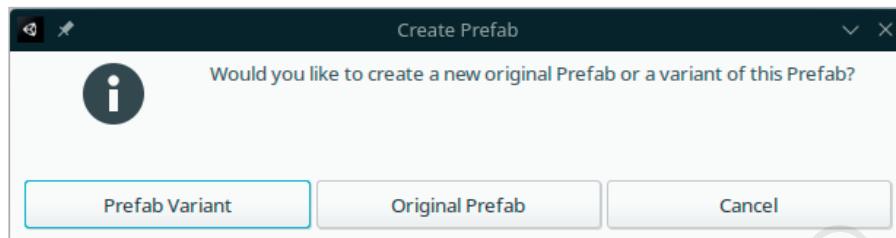


- **Anidamiento de prefabs.** Desde la versión de Unity 2018.3, podemos arrastrar prefabs dentro de otros, como si se tratara de un GameObject más. Una vez anidemos los prefabs, estos estarán enlazados, pero podrán seguir modificándose de forma independiente.
- **Variantes.** Por otro lado, cabe la posibilidad que deseemos que un prefab tenga pequeñas variaciones respecto de otro, como una especie de prefab que herede de otro pero que incorpore algunas variaciones. Hasta ahora hemos visto cómo personalizar las instancias de un prefab y cómo anidar prefabs. Si hicierámos una comparación con la programación orientada a objetos, sería como crear objetos con identidad propia (instanciación) y crear objetos en combinación de otros (anidamiento).

Ahora lo que deseamos es crear subclases a partir de otras. Esto lo conseguimos con el concepto de *variante*. Las variantes son prefabs que tienen ciertas diferencias respecto al prefab de base. Es similar a los overrides de las instancias, pero con los prefabs.

La utilidad de las variantes es obvia, puesto que es el medio con el que podemos crear objetos con una base común, pero de diferentes tipos y comportamientos.

Para crear una variante de un prefab, una vez tengamos añadidas las nuevas propiedades, simplemente lo arrastraremos a la carpeta *Prefabs*, de forma que el sistema nos pregunte qué queremos hacer, si crear una variante o modificar el prefab original.



Al crear una nueva variante, vemos que el ícono del prefab modifica su cuadro azul, sombreando una de sus caras.



- **Desempaquetado de prefabs.** Finalmente, si deseamos, a partir de un prefab, añadir a la escena algunos de los diferentes GameObjects que lo componen, deberemos desempaquetar el prefab. Para ello, disponemos de las opciones *Unpack*, para desempaquetar el prefab padre, y *Unpack Completely*, para desempaquetar por completo tanto padres como hijos.

3.3. Etiquetas y capas

- **Etiquetas.** Las etiquetas permiten marcar los GameObjects para diferenciarlos entre sí. Cuando hacemos un duplicado de un objeto o creamos una nueva instancia a partir de un prefab, se genera un nuevo nombre para la copia, pero las etiquetas que tuviera asignadas permanecen. De este modo, es más fácil identificar así todas las copias de un mismo tipo de objeto.

Para asignar una etiqueta a un GameObject, lo seleccionamos y desplegamos las etiquetas del desplegable *Tag* en el *Inspector*. Si deseamos añadir nuevas etiquetas, podemos hacerlo mediante la opción *Add Tag* del mismo menú de las etiquetas o desde la opción *Tags and Layers* de las preferencias.

- **Capas.** De forma similar a las etiquetas, permiten agrupar objetos por categorías y definir a qué objetos afectan determinados sistemas, tales como la iluminación, la navegación o la física. La interacción entre capas se determinará con la matriz de colisiones, ubicada en la configuración de físicas del proyecto.

3.4. Escenas

Las escenas suponen una forma de organizar lógicamente un juego. Una escena contiene los diferentes GameObjects que componen una parte concreta del juego. Las escenas pueden representar niveles, pero también las diferentes pantallas de la interfaz de usuario: pantalla de inicio, selector de niveles, transiciones, etc.

Las escenas son un recurso más de nuestro proyecto, por lo que se guardan en la carpeta *Assets* del proyecto, concretamente dentro de la subcarpeta *Scenes*. Por defecto, cuando creamos el proyecto, esta carpeta contiene la escena por defecto *SampleScene*, con una cámara e iluminación global. En la parte práctica de este apartado veremos cómo gestionar múltiples escenas en un mismo proyecto.



3.5. Asset Store

Asset Store es una tienda de recursos de diferentes tipos: código, texturas, materiales, modelos, animaciones, utilidades, etc., tanto de pago como gratuitas. Podemos acceder a ella a través de su web y, también, con el menú *Window > Asset Store* del editor de Unity.

Dentro del sitio, encontramos diferentes secciones para poder navegar por las categorías de recursos: 3D, 2D, herramientas, audio, plantillas, efectos...

Además, si accedemos con nuestra cuenta, en la parte superior de la vista disponemos de un menú, desde el cual podemos acceder a los assets que hemos descargado, a nuestros favoritos (aunque no los hayamos descargado) o al carrito y al resto de herramientas de Unity.

Para incorporar un asset a nuestro proyecto desde Asset Store, podemos hacerlo tanto desde la parte web como desde el propio Unity. Si lo buscamos en el portal web del Asset Store, tendremos el enlace para abrirlo en Unity; una vez en la ventana de la Asset Store en Unity, ya podemos descargarlo e instalarlo.

Propiedad de McGraw-Hill®

4. Mecánicas del juego: Entrada y movimiento cinemático

4.1. Funcionamiento de un videojuego

El funcionamiento de un videojuego consta de tres partes muy diferenciadas: la inicialización, el bucle del juego y la liberación:

- **Inicialización.** Se inicializan y cargan los diferentes componentes de la escena, librerías, objetos, scripts y variables, así como una precarga de los recursos que se necesiten en un momento dado: imágenes, modelos, texturas, tipografías o sonidos.
- **Bucle de juego (game loop).** Es el bucle principal, que controla diferentes procesos como la lógica, la aplicación de físicas, el render o la interacción entre objetos, y genera, en cada iteración, un fotograma.
- **Liberación.** Se liberan los recursos utilizados: memoria, almacenamiento temporal, etc.

MECÁNICAS DEL JUEGO Y LA CLASE MONOBEHAVIOUR

Anteriormente hemos hablado de un videojuego como una experiencia interactiva. Esto significa que el usuario puede interactuar con el juego y ser partícipe en el desarrollo de este. Esta interactividad se consigue, en gran parte, gracias a las mecánicas del juego.

Podemos definir las mecánicas como las acciones que realiza el jugador y que modifican el estado del juego. Este estado del juego viene definido, entre otras cosas, por la posición y las características de los diferentes elementos (GameObjects) que lo componen. Las mecánicas del juego describirán, pues, qué puede hacer el jugador, cómo lo puede hacer y bajo qué reglas.

Estas mecánicas se articulan a través de la clase **MonoBehaviour**, y se asociarán a los diferentes GameObjects a través de un componente de tipo **script**.

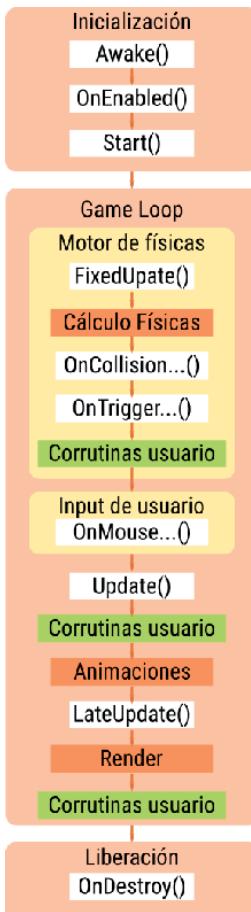
Como se ha mencionado arriba, el bucle del juego es el encargado de realizar diferentes tareas para implementar la lógica del juego de la forma más rápida y eficiente posible. Las tareas serán, por citar algunas, comprobar las acciones del usuario, el cálculo de físicas, la detección de colisiones o el renderizado.

IMPORTANTE

¿Qué relación tienen las mecánicas con el bucle del juego y con la clase MonoBehaviour?

El bucle del juego, como hemos comentado, pasa por diferentes etapas. Cuando deseemos aplicar determinadas mecánicas a un GameObject, deberemos asociarle un componente de tipo script. Dicho script será una clase derivada de **MonoBehaviour**, que contendrá métodos concretos (event functions) que serán invocados en las diferentes etapas del bucle del juego. Estos métodos están definidos en la clase **MonoBehaviour** y conforman lo que se conoce como ciclo de vida del script. De este modo, podremos controlar el objeto en los diferentes momentos del bucle del juego.

En la documentación oficial de Unity se presenta un esquema muy completo acerca del orden en la ejecución de estos métodos. De forma simplificada, veámoslo con las fases y los métodos que más comúnmente usaremos.



En esta imagen se muestran las tres fases (inicialización, bucle de juego y liberación) de las que hemos hablado, y en cada una de ellas se invocan varios métodos de MonoBehaviour.

- **Inicialización.** Se crean los objetos y se invocan, una sola vez, los métodos Awake() y Start(). Awake se invoca tanto si el objeto está habilitado o deshabilitado, y Start cuando está habilitado. El método OnEnable se lanza cuando el objeto pasa de deshabilitado a habilitado.
- **Bucle de juego (Game Loop).** Se ejecutan varias fases en él, y en cada iteración se produce un fotograma. En la primera fase se realiza otro bucle para la aplicación del **motor de físicas**, con las actualizaciones de FixedUpdate(), cálculos y detección de colisiones, principalmente. Después se recoge la entrada o **Input** de usuario, y se invoca al método Update(), con el que gestionamos movimientos cinemáticos (no físicos). Y finalmente, en este bucle, se aplican las animaciones, se invoca a LastUpdate() y se hace el renderizado del fotograma. Como vemos, dentro de este bucle y en varios momentos se pueden invocar ciertas corutinas de usuario, funciones que se utilizan para gestionar eventos asíncronos.
- **Liberación.** Cuando un objeto del juego es eliminado, se invoca a su método OnDestroy(). Esto no implica que el bucle de juego finalice, sino que lo haga el ciclo de vida de ese GameObject.

A la vista del esquema anterior, podemos diferenciar tres bloques principales o tipos de scripts:

1. Scripts para la gestión del Input, que se encargarán de detectar la entrada del usuario (pulsación de teclas, movimiento o clics del ratón, etc.).
2. Scripts de movimiento, donde se gestionan animaciones y se aplican transformaciones de movimiento, tanto cinético (físico) como cinemático (no físico).
3. Scripts para detectar colisiones entre objetos.

Además, los diferentes GameObjects, en sus scripts, es posible que hagan referencia a otros objetos de la escena, por lo que también deberemos establecer mecanismos para posibilitar la comunicación entre ellos.

4.2. Gestión del Input del usuario

Vamos a ver algunos métodos, tanto de la clase MonoBehaviour como de la clase Input, para detectar la interacción por parte del usuario.

A.EVENTOS SOBRE LOS GAMEOBJECTS

En primer lugar, en la tabla siguiente se muestran diferentes métodos que podemos utilizar en la clase MonoBehaviour para detectar eventos de ratón sobre un GameObject.

Gestión del ratón	
Método	Descripción
void OnMouseDown () {}	Clic del ratón
void OnMouseDrag () {}	Arrastre del ratón
void OnMouseEnter () {}	El ratón entra en cierta región
void OnMouseExit () {}	El ratón sale de una región
void OnMouseOver () {}	El ratón se mueve sobre una región

B. MÉTODOS DE LA CLASE INPUT

La clase Input es la encargada de gestionar la interacción con el usuario a través de los dispositivos de entrada. Su gestión suele realizarse dentro de los métodos `Update()` y `FixedUpdate()` de la clase `MonoBehaviour`.

En la tabla siguiente vamos a ver algunos métodos de la clase Input, que devuelven «cierto» o «falso» según se esté pulsando un botón del ratón o no. Los identificadores (`id`) de estos botones serán 0, 1 y 2, para el botón izquierdo, el central o el derecho, respectivamente.

Entrada del ratón	
Método	Descripción
<code>bool Input.GetMouseButtonDown(id)</code>	El botón se pulsa por primera vez
<code>bool input.GetMouseButton(id)</code>	El botón se mantiene pulsado
<code>bool input.GetMouseButtonUp(id)</code>	Se deja de pulsar el botón

Para la gestión del teclado, podemos utilizar, entre otros, los métodos siguientes que nos devuelven «cierto» o «falso» según se esté pulsando una tecla o no:

Entrada del ratón	
Método	Descripción
<code>bool Input.GetKeyDown (código_tecla) {}</code>	El botón se pulsa por primera vez
<code>bool Input.GetKey (código_tecla) {}</code>	Devuelve «cierto» cuando se empieza a pulsar una tecla
<code>bool Input.GetKeyUp (código_tecla) {}</code>	Devuelve «cierto» si se deja de pulsar una tecla

Los diferentes códigos de teclas están definidos dentro de Unity en la enumeración `KeyCode`, que podemos consultar en la referencia de este.

Código	Tecla
<code>KeyCode.A</code>	Tecla A
<code>KeyCode.D</code>	Tecla D
<code>KeyCode.W</code>	Tecla W
<code>KeyCode.S</code>	Tecla S
<code>KeyCode.Space</code>	Barra espaciadora
<code>KeyCode.UpArrow</code>	Flecha del cursor hacia arriba
<code>KeyCode.DownArrow</code>	Flecha del cursor hacia abajo
<code>KeyCode.RightArrow</code>	Flecha del cursor a la derecha
<code>KeyCode.LeftArrow</code>	Flecha del cursor a la izquierda

Por otro lado, también disponemos de botones y ejes virtuales que se pueden configurar desde la ventana del Input Manager [`Edit > Project Settings > Input Manager`].

Estos botones virtuales son `Fire1`, `Fire2` y `Fire3`, y están asociados por defecto a las teclas `Control`, `Alt` y `Cmd`. Para comprobar su estado, usamos métodos como `GetButton`, `GetButtonUp` o `GetButtonDown` de la clase `Input`, que nos devuelven «cierto» o «falso» según si se está pulsando el botón o no.

En cuanto a los ejes virtuales, se usa el método `GetAxis` de la clase `Input`, que devuelve un valor de tipo `float` entre -1 y 1, siendo 0 el estado de reposo. Los ejes `Horizontal` y `Vertical` están asociados en el Input Manager a las teclas `W`, `A`, `S` y `D`, y a las flechas del cursor, y `MouseX` y `MouseY` están asignados al movimiento del ratón.

Podemos encontrar todas las propiedades y métodos de esta clase en la documentación oficial de Unity.

En el proyecto de ejemplo de la unidad dispone de la escena `Eventos`, donde puedes ver en acción algunos de estos métodos, así como una aplicación del uso de etiquetas.

4.3. Scripts de movimiento

El movimiento de los diferentes GameObjects es fundamental para llevar a cabo la interacción con el usuario. Este movimiento viene definido por el componente Transform del GameObject, el cual podrá ser modificado directamente a través de la alteración de sus valores de posición, rotación o escalado, o mediante el motor de físicas, definiendo el GameObject como un objeto físico.

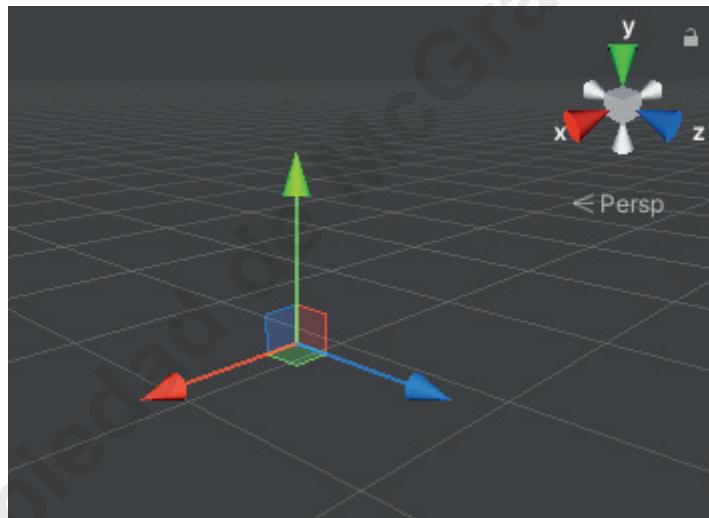
Hablamos, pues, de dos tipos de movimientos: movimiento cinemático, cuando actuamos directamente sobre el componente Transform del GameObject, y movimiento cinético o físico, cuando actúa sobre él el motor de físicas. En este apartado nos centraremos en el movimiento cinemático, y dejaremos el movimiento físico para el siguiente.

A. VECTORES

Antes de adentrarnos en los scripts de movimiento, debemos recordar algunos conceptos sobre física y aritmética de vectores.

En primer lugar, para representar el espacio físico real utilizamos un espacio euclídeo, en el que definimos un origen de coordenadas y una base vectorial formada por dos o tres vectores, según nos encontremos en dos o tres dimensiones. Los vectores son secuencias de números reales (dos números para espacios 2D y tres números para espacios 3D) que representan un segmento de recta dirigido y que dan formato a magnitudes físicas, como la posición, la velocidad o la aplicación de una fuerza.

En la imagen siguiente vemos el sistema de coordenadas de Unity, cuyo origen es el punto [0,0,0] y en el que los ejes X y Z definen el plano del suelo, y el eje y define la altura.



Los vectores se caracterizan por tener:

- Un origen o punto de aplicación sobre el que actúa el vector.
- Un módulo, que es la longitud del vector.
- Una dirección indicada por la orientación de la recta en el espacio.
- Un sentido, indicado mediante una punta de flecha, mostrando la línea de acción en la que actúa el vector.

Unity proporciona las clases Vector2 y Vector3 para representar vectores en dos y tres dimensiones (existe también una clase Vector4 que no usaremos). Podéis encontrar información adicional sobre estas clases y sobre aritmética de vectores en la documentación oficial de Unity.

B. MOVIMIENTO CINEMÁTICO

Como sabemos, todo GameObject, aunque sea un GameObject vacío, dispone de un componente de tipo Transform que contiene su posición, rotación y escalado.

El movimiento cinemático consiste en modificar, dentro de nuestros scripts, estas propiedades, para lo cual necesitaremos hacer uso de objetos de tipo Vector2 en 2D y Vector3 en 3D. Para los ejemplos siguientes trabajaremos con Vector3, pero todo será también aplicable para Vector2, aunque con una dimensión menos.

La clase Vector3 representa un vector en tres dimensiones, que puede indicar una posición, una dirección, etc. Estas dimensiones se representarán en las tres componentes del vector como números en coma flotante, correspondientes a los ejes X, Y y Z.

Por ejemplo, para establecer la posición de un objeto deberemos modificar el campo position del componente transform.

```
// Posicionamiento del objeto en (3, 3, 3)
transform.position = new Vector3 (3, 3, 3);

// Posición del objeto en (0 ,0, 0)
transform.position = new (Vector3.zero); //o new Vector3(0,0,0);
```

Como podemos ver, desde un script tenemos acceso directo a su componente Transform. Observad que para modificar la posición asignamos un nuevo vector, y no podemos modificar directamente las componentes de este.

Por otro lado, la clase Vector3 ofrece algunos vectores ya predefinidos que pueden ser de utilidad para indicar direcciones:

Eje	Dirección	Vector predefinido	Valores de Vector3
Z	Adelante	Vector3.forward	Vector3[0, 0, 1]
-Z	Atrás	Vector3.back	Vector3[0, 0, -1]
Y	Arriba	Vector3.up	Vector3[0, 1, 0]
-Y	Abajo	Vector3.down	Vector3[0, -1, 0]
X	Derecha	Vector3.right	Vector3[1, 0, 0]
-X	Izquierda	Vector3.left	Vector3[-1, 0, 0]

- **Desplazamientos.** Para añadir el desplazamiento a un objeto, podemos hacerlo de dos formas: bien mediante la suma de vectores, bien mediante el método Translate.

- Suma de vectores para desplazar un objeto 1 m en el eje X:

```
// Especificando valores en Vector3
transform.position=transform.position + new Vector3(1, 0, 0);
// Usando vectores predefinidos:
transform.position=transform.position + Vector3.right;
```

- Uso del método Translate para trasladar un objeto 1 m en el eje X:

```
// Especificando valores en Vector3
transform.Translate(new Vector(1,0,0));
// Usando vectores predefinidos:
transform.Translate(Vector3.right);
```

Generalmente realizaremos estas operaciones dentro del método Update() de nuestros scripts, para que se ejecuten en cada iteración del bucle del juego. Por ejemplo, el script siguiente, asociado a un GameObject, nos permite moverlo en horizontal y en vertical:

```
public class CuboCinematico : MonoBehaviour
{
    public float velocidad;
    void Start()
    { velocidad = 1f; }

    void Update()
    {
        if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow))
```

```
{  
    transform.Translate(Vector3.left * velocidad * Time.deltaTime);  
}  
  
if (Input.GetKey(KeyCode.D) || Input.GetKey(KeyCode.RightArrow))  
{  
    transform.Translate(Vector3.right * velocidad * Time.deltaTime);  
}  
  
if (Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))  
{  
    transform.Translate(Vector3.up * velocidad * Time.deltaTime);  
}  
  
if (Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow))  
{  
    transform.Translate(Vector3.down*velocidad*Time.deltaTime);  
}  
}  
}
```

IMPORTANTE

Este código es un fragmento de los scripts que puedes encontrar en la escena MovimientoCinematico del proyecto de ejemplo (script CuboCinematico.cs). En este script, además, podrás gestionar el movimiento del cubo mediante diferentes métodos de entrada, y podrás alternar entre movimiento por teclado, por ejes y por ratón.

Como podemos comprobar, se ha definido una propiedad para la velocidad, la cual podremos modificar desde la interfaz de Unity. Para hacer los cálculos del desplazamiento, vemos que se utiliza el vector unitario en la dirección deseada, multiplicado por la velocidad y por el DeltaTime.

¿Qué es el DeltaTime?

El método Update en el que estamos trabajando se invoca una vez por cada frame. El framerate, como sabemos, puede variar entre diferentes dispositivos y, además, es posible que ni siquiera sea un valor constante en el mismo dispositivo, ya que puede depender de la carga de este.

El DeltaTime es un valor que contiene el tiempo, en segundos, que tardó en completarse el último frame. Si multiplicamos los desplazamientos por este valor, lo que realmente hacemos es dividir los valores entre el framerate en cada momento. Con ello conseguimos que el movimiento de los objetos sea independiente de dicho framerate, y, por tanto, sean uniformes.

- **Escalados.** Para modificar la escala (Scale) de un objeto en el sistema de coordenadas local, deberemos modificar el localScale del componente Transform. Por ejemplo, para incrementar la altura de un objeto, podemos hacer:

```
transform.localScale = new Vector3 ( 1f , 2.0f , 1.0f);
```

- **Rotaciones.** Para modificar la componente de rotación, lo haremos con sus propiedades eulerAngles o rotation.

Unity representa las rotaciones de forma interna mediante cuaterniones (quaternions), una representación de ángulos muy eficiente y basada en números complejos. No obstante, generalmente utilizaremos vectores o rotaciones existentes para especificar nuevas rotaciones; para ello Unity proporciona métodos específicos. Así pues:

- Para especificar una rotación en grados con un vector utilizamos el método eulerAngles del componente Transform:

```
objeto.transform.eulerAngles = new Vector3 ( 90 , 45 , 0 );
```

- Para especificar una rotación en cuaterniones utilizamos directamente el componente rotation, pero convirtiendo previamente a cuaterniones los grados, mediante el método Quaternion.Euler:

```
objeto.transform.rotation = Quaternion.Euler( 90 , 45 , 0 );
```

En el proyecto de ejemplo dispones de la escena Rotaciones, con varios ejemplos y métodos interesantes.

Propiedad de McGraw-Hill®

5. Mecánicas: Movimiento físico y detección de colisiones

5.1. Movimiento físico

Decimos que un movimiento es físico cuando es provocado por causas físicas, como la gravedad u otras fuerzas. Hasta ahora, cuando añadimos objetos a una escena, estos quedan suspendidos en el aire, sin que ninguna fuerza actúe sobre ellos, y provocamos su movimiento mediante transformaciones directas de su posición.

Cuando incorporamos propiedades físicas a un objeto, este se ve influenciado por fuerzas como la gravedad (cae al suelo), y además podemos moverlo mediante la aplicación de diferentes tipos de fuerza. Para incorporar a un objeto estas propiedades físicas, necesitaremos añadirle un componente de tipo Rigidbody.

A. RIGIDBODY

El componente Rigidbody aporta propiedades físicas a los objetos, como puedan ser:

- **Mass** (masa). Indica el peso del objeto (en kg).
- **Drag** (resistencia). Resistencia que afecta al movimiento del objeto.
- **Angular Drag** (resistencia angular). Resistencia que afecta al giro del objeto.

Además de estas propiedades físicas, el Rigidbody permite controlar otros parámetros como:

- **Use Gravity** (uso de la gravedad). Indica si al objeto le afecta la gravedad.
- **Is Kinematic** (es cinemático). Cuando se marca esta opción, se desactiva el motor de físicas para el objeto, de forma que solo se puede manipular mediante el Transform.
- **Interpolate** (tipo de interpolación). Para indicar el uso de la interpolación. Suele recomendarse su uso cuando los movimientos resultan bruscos, especialmente cuando se trata de movimientos de una cámara que sigue a un objeto, donde el movimiento de la cámara es constante pero el movimiento del objeto es físico.
- **Collision Detection** (detección de colisiones). Permite indicar el tipo de detección de colisiones. Si este es discreto, la detección de colisiones tendrá lugar en ciertas posiciones del objeto, lo que puede provocar que objetos que vayan muy rápido atraviesen otros objetos. En estos casos, se recomienda utilizar detección de colisiones continua, que es más costosa computacionalmente pero más eficiente.
- **Constraints** (restricciones): Permiten congelar tanto la posición como la rotación de un objeto, de modo que no se apliquen las físicas en los ejes indicados. Esto nos será de gran utilidad en algunas ocasiones, como veremos más adelante.



Rigidbody2D

Cuando trabajamos en un proyecto 2D, el componente que debemos utilizar es el Rigidbody2D, que se encarga de aplicar propiedades físicas en dos dimensiones.

Recordad que los componentes Rigidbody y Rigidbody2D son incompatibles, por lo que no pueden asociarse simultáneamente a un mismo objeto.

B. APLICANDO FUERZAS

La aplicación de fuerzas sobre un objeto hará que este tenga un comportamiento más natural, y evitará situaciones no deseadas como, por ejemplo, que los objetos puedan traspasarse.

Para aplicar una fuerza sobre un objeto deberemos utilizar el método AddForce sobre el componente Rigidbody del objeto. Para ello, en primer lugar, obtendremos el componente mediante el método GetComponent de la clase MonoBehaviour.

Veámoslo con un fragmento de código del script cuboFisico.cs de la escena Movimiento Físico del proyecto de ejemplo de la unidad:

```
public class cuboFisico : MonoBehaviour {  
    private Rigidbody rb; // Referencia al componente Rigidbody  
  
    void Start() {  
        // Guardamos en rb el componente Rigidbody del objeto  
        rb = GetComponent<Rigidbody>();  
    }  
  
    void FixedUpdate() {  
        if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.LeftArrow))  
        { // Movimiento a la izquierda  
            if (rb != null) {  
                rb.AddForce(new Vector3(20.0f, 0.0f, 0.0f));  
            }  
        }  
        // ...  
    }  
}
```

Ahora veamos algunos detalles de este código:

- En primer lugar, hemos definido en la clase una propiedad rb de tipo Rigidbody.
- Esta propiedad se inicializa en el método Start, mediante GetComponent<TipoComponente>, con el que obtenemos una referencia a un componente concreto del objeto al que está asociado el script, en este caso, el Rigidbody.
- En el método FixedUpdate, como respuesta a la pulsación de una tecla, se aplica una fuerza con el método AddForce del componente Rigidbody, referenciado por rb.
- Observamos que la fuerza aplicada se expresa en forma de vector.

IMPORTANTE

Sobre Update y FixedUpdate

Nos fijamos en que estamos utilizando el método FixedUpdate, en lugar de Update, para el movimiento físico. La principal diferencia es que, así como el método Update se invoca para cada frame (y, por tanto, depende del framerate), FixedUpdate se ejecuta cada cierto tiempo de forma fija e independiente del framerate. Este tiempo se puede ajustar a las preferencias y, por defecto, tiene un valor de 0.02 segundos.

Por ejemplo, si tenemos un framerate de 30 fps, significa que en un segundo el método Update se invoca 30 veces, mientras que el método FixedUpdate se invocará 50 veces ($1/0.02$). En cambio, si el framerate es de 60 fps, el Update se invocará 60 veces, pero el FixedUpdate lo seguirá haciendo 50 veces.

Como comentamos, en la escena Movimiento Físico del proyecto de ejemplo de la unidad tienes disponible los scripts (carpeta *MovimientoFisico*) con los movimientos en todos los ejes.

Otra alternativa a este movimiento podría ser utilizando los ejes (Axis), de modo que el vector de fuerza se obtuviese a partir de ellos. En la escena Movimiento Físico puedes activar también este tipo de movimiento y consultar su código completo. Básicamente, lo que hacemos es:

1. Obtener el movimiento del eje, en horizontal y vertical, y crear el vector de movimiento:

```
moveHorizontal = Input.GetAxis("Horizontal");  
moveVertical = Input.GetAxis("Vertical");  
vectorMovimiento = new Vector3(moveHorizontal, 0.0f, moveVertical);
```

2. Aplicar un vector de fuerza, según el vector de movimiento obtenido y la velocidad:

```
rb.AddForce(movement * speed);
```

C. TIPOS DE FUERZAS

El método AddForce es uno de los métodos que ofrece el componente Rigidbody para aplicar fuerzas al objeto, pero este soporta también otros métodos para aplicar distintas fuerzas a los objetos. Los métodos que soporta un Rigidbody para aplicarle fuerzas son los siguientes:

Método	Descripción
AddForce	Aplica una fuerza indicada en coordenadas absolutas al mundo
AddRelativeForce	Aplica una fuerza en coordenadas relativas al objeto
AddTorque	Aplica una fuerza de torsión o giro al objeto en coordenadas del mundo
AddRelativeTorque	Aplica una fuerza de torsión o giro al objeto en coordenadas relativas al objeto
AddForceAtPosition	Aplica una fuerza en una posición determinada del objeto
AddExplosionForce	Simula sobre un objeto la fuerza aplicada por una explosión producida en un determinado punto y con cierto radio de acción. Podemos consultar un ejemplo completo en la documentación de Unity.

En la escena Movimiento Físico del proyecto de ejemplo puedes probar todos estos tipos de fuerza parametrizando el script desde el propio Inspector.

D. MODO DE FUERZA (FORCEMODE)

Estos métodos, además, admiten un parámetro opcional mode, que indica el modo en que se aplica la fuerza.

Este parámetro sirve para indicar el tipo de fuerza, que puede ser:

Modo	Descripción
ForceMode.Force	Añade una fuerza continua al Rigidbody, usando su masa.
ForceMode.Acceleration	Añade una aceleración continua al Rigidbody, ignorando su masa.
ForceMode.Impulse	Añade un impulso o fuerza instantánea al Rigidbody, utilizando su masa.
ForceMode.VelocityChange	Añade un cambio de velocidad instantáneo al Rigidbody, ignorando su masa.

Podemos ver el uso de estos métodos en la escena Movimiento Físico del proyecto de ejemplo de la unidad, donde podemos parametrizar el salto del objeto.

5.2. Detección de colisiones

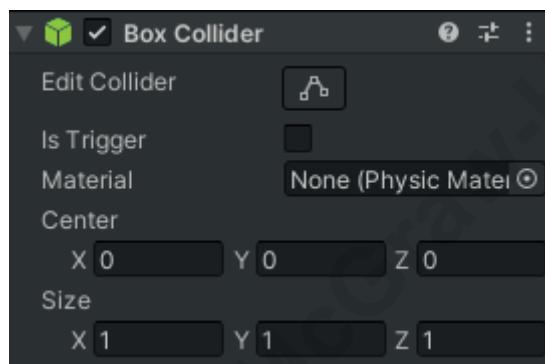
La detección de colisiones es una de las principales funciones del motor de videojuegos, y consiste en determinar cuándo dos objetos chocan entre sí. Una colisión implica, pues, dos objetos.

En Unity se distinguen dos tipos de colisiones:

- Colisiones por Collider, que detectan cuándo dos objetos chocan.
- Colisiones por disparador (Trigger), usadas para determinar cuándo un GameObject accede a una determinada zona y asociadas, por lo general, a GameObjects vacíos que solamente tienen el componente Transform y el Collider.

Para que un objeto pueda detectar colisiones necesita un componente de tipo Collider. Los colliders son invisibles y definen la forma del objeto que se usará para la detección de colisiones. Esta forma no tiene por qué ser la misma que la malla que define el objeto, ya que debe ser lo más simple posible para facilitar los cálculos al motor.

Veamos en el Inspector qué parámetros tiene, por ejemplo, un Box Collider, que es el collider asociado a un cubo:



Observamos que, aparte de la posición relativa al objeto y el tamaño, dispone de un flag denominado isTriggered. Esto sirve para indicar que la colisión es de tipo Trigger, es decir, no se producirá una colisión como tal, sino que se disparará un evento al que otros objetos puedan responder. Esto es de utilidad para saber, por ejemplo, cuándo el jugador pasa por un determinado sitio, para lanzarle algún objeto.

Generalmente la detección de colisiones va acompañada del uso de etiquetas para determinar el tipo de objeto sobre el que se está colisionando, o bien se utiliza la malla de colisiones de las preferencias del proyecto para determinar qué capas pueden colisionar con otras.

Es importante tener en cuenta que, del mismo modo que tenemos un componente Rigidbody específico para 2D o 3D, los colliders también serán específicos para 2D y 3D, para detectar colisiones en dos o en tres dimensiones.

CUÁNDO SE DETECTAN LAS COLISIONES

Las colisiones pueden detectarse en tres momentos diferentes: cuando se producen por primera vez (enter), cuando se mantienen (stay) o cuando dejan de producirse (exit).

Para gestionarlas, los GameObjects ofrecen los métodos siguientes:

Método	Descripción
void OnCollisionEnter (Collision collision)	Se detecta que el objeto entra en colisión con otro objeto/collider.
void OnTriggerEnter (Collider other)	
void OnCollisionStay (Collision collision)	El objeto mantiene la colisión con otro objeto/collider.
void OnTriggerStay (Collider other)	
void OnCollisionExit (Collision collision)	El objeto sale de la colisión con otro objeto/collider.
void OnTriggerExit (Collider other)	
void OnCollisionEnter2D (Collision collision)	Se detecta que el objeto entra en colisión con otro objeto/collider en 2D.
void OnTriggerEnter2D (Collider other)	

void OnCollisionStay2D (Collision colision)	El objeto mantiene la colisión con otro objeto/collider en 2D.
void OnTriggerStay2D (Collider other)	
void OnCollisionExit2D (Collision colision)	El objeto sale de la colisión con otro objeto/collider en 2D.
void OnTriggerExit2D (Collider other)	

Observamos que los métodos para detectar colisiones con objetos reciben un objeto de tipo Collision, mientras que los métodos de detección de colisiones por trigger reciben un objeto de tipo Collider. La clase Collision contiene, además de una referencia al objeto con el que ha colisionado, información acerca de la colisión en sí, como los puntos de contacto, la velocidad del impacto, etc.

En el script cuboFisico.cs de la escena Movimiento Físico del proyecto de ejemplo de la unidad utilizamos las colisiones para controlar los saltos. Si únicamente detectásemos la pulsación de la tecla de salto para aplicar el impulso hacia arriba, podríamos saltar infinitas veces. Para controlar esto se añade una propiedad jumping que inicializamos en «false» y establecemos en «true» cuando estamos saltando. Ahora bien, ¿cómo reestablecemos a «false» esta propiedad? Lo hacemos detectando la colisión con el suelo, mediante etiquetas:

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.CompareTag("Superficie"))
    {
        jumping = false;
    }
}
```

Por otro lado, en este script también se utiliza la detección de colisiones por trigger. Bajo la superficie donde reposa el cubo existe un GameObject vacío, al que hemos llamado Destructor, que contiene un collider marcado como IsTriggered. De este modo, podemos eliminar el objeto:

```
void OnTriggerEnter (Collider other) {
    if (other.name=="Destructor") {
        Destroy(gameObject);
    }
}
```

En este caso no utilizamos etiquetas, sino que directamente accedemos al atributo name. Este atributo es una referencia en el componente Collider al nombre del GameObject al que está asociado. Cuando el cubo que posee este script detecta una colisión por trigger, comprueba que el objeto con el que colisiona sea el Destructor, y, en ese caso, destruye el GameObject al que está asociado para que no caiga infinitamente, mediante el método Destroy.

IMPORTANTE

¿Destroy(GameObject) o Destroy(this)?

Un error habitual es intentar utilizar la referencia this como si fuese una referencia al GameObject actual. Debemos tener en cuenta que la clase que estemos utilizando, derivada de MonoBehaviour, es un componente más, un Script del GameObject, pero no es el GameObject en sí. Si utilizásemos Destroy(this) eliminaríramos únicamente el componente, es decir, el comportamiento que le aporta el script, pero no el GameObject.



5.3. Creación y destrucción de objetos

A. DESTRUCCIÓN DE OBJETOS

Como hemos visto en el ejemplo anterior, para eliminar un objeto utilizamos el método Destroy, al que podemos proporcionar directamente el objeto a destruir. Este método admite un segundo parámetro, que indica, mediante un float, una demora en la destrucción:

```
// Destruye el gameObject al segundo
Destroy(gameObject, 1);
```

Con esto podemos dar un tiempo adicional para que el objeto se destruya. Normalmente este es un proceso computacionalmente más costoso, y suele desactivarse primero, para que no sea visible ni interactúe con nada, y luego destruirse:

```
Destroy(gameObject, 1);           // programamos la destrucción al segundo
gameObject.SetActive(false);    // inhabilitamos el objeto
```

Como hemos comentado, podemos destruir también componentes o lanzar el mensaje a otros objetos que no sean el propio GameObject. Por ejemplo, para eliminar el GameObject con el que acabamos de colisionar, podríamos hacer:

```
private void OnCollisionEnter(Collision collision) {
    Destroy(collision.gameObject);
}
```

Ya que la colisión posee, entre otra información, una referencia al objeto con el que colisionamos.

B. CREACIÓN DE OBJETOS

Para crear un GameObject nuevo hacemos uso del método Instantiate, al que le proporcionaremos un GameObject [ya sea un prefab o un objeto de la escena] y retorna una instancia de ese prefab. Esto puede utilizarse para generar, de forma dinámica, cualquier GameObject: disparos, enemigos, etc.

El método Instantiate de MonoBehaviour admite varias formas de instanciación. De forma resumida, su sintaxis sería la siguiente:

```
public static Object Instantiate (Object original);
public static Object Instantiate (Object original, Transform parent);
public static Object Instantiate (Object original, Transform parent,
                                bool InstantiateInWorldSpace);
public static Object Instantiate (Object original, Vector3 position,
                                Quaternion rotation);
public static Object Instantiate (Object original, Vector3 position,
                                Quaternion rotation, Transform parent);
```

En la escena Colisiones del proyecto de ejemplo hay un fragmento de código en el script Destructor, asociado al gameObject vacío situado bajo la superficie, en el que se puede contemplar esta instanciación.

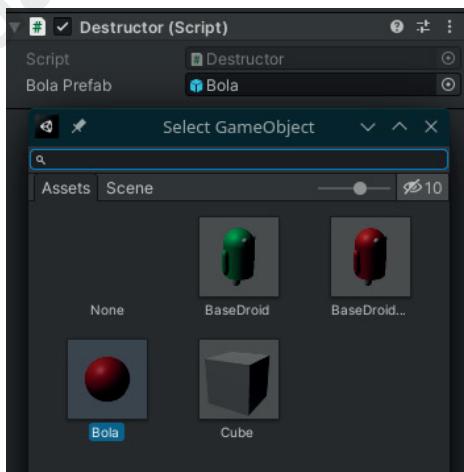
En primer lugar, se crea una propiedad de tipo GameObject en el Script:

```
public class Destructor : MonoBehaviour {  
    public GameObject BolaPrefab;  
    ...  
}
```

Posteriormente, cuando se detecta una colisión por trigger en este Destructor, si es un GameObject etiquetado como rojo, se crea un nuevo GameObject antes de destruirlo y se etiqueta también como rojo.

```
void OnTriggerEnter(Collider other) {  
    if (other.gameObject.CompareTag("rojo"))  
    {  
        // Instanciación de un nuevo GameObject  
        float x = Random.Range(-4.0f, 4.0f);  
        float z = Random.Range(-4.0f, 4.0f);  
        Vector3 position = new Vector3(x, 3.0f, z);  
        GameObject nuevo=GameObject.Instantiate(BolaPrefab, position,  
                                                Quaternion.identity);  
        nuevo.tag = "rojo";  
    }  
  
    // Destruimos el gameObject  
    Destroy(other.gameObject, 1.0f);  
    other.gameObject.SetActive(false);  
}
```

Hay que tener en cuenta que este objeto público BolaPrefab debe ser instanciado desde la propia interfaz, mediante el Inspector, arrastrando el prefab Bola hasta esta propiedad.



5.4. Referencias y comunicación entre objetos y componentes

Desde un script podemos acceder a los GameObjects y a sus componentes de varias formas:

- **Acceso implícito** al propio GameObject al que está asociado el script, mediante propiedades accesibles desde la propia clase MonoBehaviour, como name, gameObject, tag, etc.
- Acceso mediante **referencias públicas**, definiendo referencias a GameObjects como propiedades públicas y asignándolas dinámicamente desde el propio editor, de modo que la referencia queda asociada a este objeto, como hemos hecho en el epígrafe anterior con el prefab Bola.
- Buscando objetos de forma **explícita**, para lo cual tenemos diferentes métodos, como podemos ver en la tabla siguiente:

Método	Descripción
<code>GameObject Object.Find(String nombre)</code>	Obtiene una referencia al GameObject con el nombre indicado.
<code>GameObject GameObject.FindGameObjectWithTag(String tag);</code> <code>GameObject[] GameObject.FindGameObjectsWithTag(String Tag.)</code>	Obtiene un vector con las referencias de todos los GameObjects activos etiquetados con el tag indicado.
<code>GameObject GameObject.FindObjectOfType<type>();</code> <code>GameObject.FindObjectsOfType<type>();</code>	Obtiene una referencia o un vector de referencias al objeto (u objetos) del tipo (type) indicado, como por ejemplo una cámara o una luz.
<code>transform.Find("item")</code>	Permite buscar elementos padres, hermanos o hijos a partir del componente transform del objeto actual. Disponéis de más información a la documentación oficial de Unity.

Por otro lado, para acceder a los componentes de un GameObject, como, por ejemplo, su Rigidbody para aplicarle una fuerza, o cualquier otro componente (incluyendo los scripts), hacemos uso del método GetComponent<componente>.

Veamos algunos ejemplos:

```
// Obtiene una referencia al componente Rigidbody del propio objeto
Rigidbody rb = GetComponent<Rigidbody>();
rb.AddForce(new Vector3(0.0f, 3.0f, 0.0f), ForceMode.Impulse);

// Crea un GameObject nuevo y le aplica una fuerza a su Rigidbody
GameObject nuevo=GameObject.Instantiate(BolaPrefab, position, Quaternion.identity);
nuevo.GetComponent<Rigidbody>().AddForce(new Vector3(0.0f, 3.0f, 0.0f), ForceMode.Impulse);

// Busca un objeto llamado BolaVerde y aplica una fuerza a su Rigidbody
GameObject obj=GameObject.Find("BolaVerde");
obj.GetComponent<Rigidbody>().AddForce(new Vector3(0.0f, 3.0f, 0.0f), ForceMode.Impulse);
```



Unidad 6.

Desarrollo de juegos 2D y 3D

A large, semi-transparent watermark logo is centered on the page. It features a tan-colored bomb with a white base and a white fuse that is lit and curved upwards. The words "Propiedad de McGraw-Hill®" are written diagonally across the bomb in a light gray, sans-serif font.

1. El desarrollo de videojuegos

El desarrollo de videojuegos es un proceso complejo, que comprende desde su concepción hasta su distribución y mantenimiento, y en el que están implicados diferentes equipos de trabajo.

1.1. Etapas en el desarrollo de videojuegos

Como todo proceso de desarrollo de software, el desarrollo de videojuegos se divide en varias etapas interrelacionadas entre sí. Aunque cada empresa posee una forma de trabajar, existen etapas más o menos comunes en este proceso de desarrollo.

Podemos, pues, establecer tres grandes etapas en el desarrollo de un videojuego: la preproducción, la producción y la posproducción.

A. PREPRODUCCIÓN

La preproducción es la primera fase en el desarrollo de un videojuego, y tiene como objetivo principal asentar las bases de lo que será este. Esta fase suele dividirse en dos o tres más: la conceptualización, el diseño y la planificación.

- **Conceptualización o concepción.** Todo videojuego empieza con una idea a la que iremos dando forma y haciendo evolucionar durante el desarrollo del juego. En esta primera fase, esta idea empieza a materializarse, y se definen aspectos como el género, el estilo de juego, los personajes, el ambiente o la música, etc. El resultado de esta fase es un **storyboard** o guion gráfico que plasma todas estas ideas.

Un **Storyboard** es una secuencia de dibujos a partir de una plantilla, acompañada de textos que definen cómo se estructura una historia. Tiene su origen en el cine de animación, cuando Walt Disney lo empezó a utilizar para sus producciones en la década de los treinta, y se popularizó rápidamente. En la actualidad, es una herramienta que se utiliza para sintetizar la idea de una historia en sectores como la publicidad, el cine o la televisión. Cuando se aplica a videojuegos, deberemos tener en cuenta que estos pueden evolucionar de forma diferente según la interacción del usuario, de forma que se pueden establecer distintas líneas de acción

- **Diseño.** Partiendo de las ideas extraídas en la fase de conceptualización, se pone en marcha el diseño de una propuesta inicial del proyecto, que se plasmará en el documento de diseño del juego (Game Design Document o GDD) y será elaborado por un pequeño grupo multidisciplinar.

En primer lugar, la historia se desarrolla mediante un guión en el que se establecen las fases, los objetivos o los personajes del juego. A partir de esta historia, un equipo artístico empieza a elaborar los primeros esbozos de personajes, escenarios u objetos. En este punto también se deberá describir todo el aspecto relativo al sonido, como la música o los efectos.

De forma paralela, se irá desarrollando el gameplay, el funcionamiento y la interacción entre las diferentes entidades que componen el juego, según el género de este.

El equipo de desarrolladores, por su parte, planteará cómo abordar la programación del juego, y establecerá las tecnologías y metodologías que se utilizarán. En esta fase es interesante la creación de un pequeño prototipo, aunque sin considerar todavía aspectos gráficos.

Definición del **GDD** según GamerDic: Documento que se elabora durante la fase de preproducción de un videojuego —y que se va actualizando conforme avanza su desarrollo—, que contiene información detallada sobre los objetos, las reglas, los entornos, el contexto, la estructura narrativa, las condiciones de victoria/derrota y la estética del juego, sirviendo como guía a los diferentes grupos de trabajo durante el proceso de desarrollo.

Definición de documento de diseño [en línea]. GamerDic, diccionario on-line de términos sobre videojuegos y cultura gamer, 2013 [fecha de consulta: 27 de marzo del 2022].

- **Planificación.** Como última tarea de la preproducción, se planifican las diferentes tareas y la distribución del trabajo estableciendo un calendario con los diferentes plazos de entrega y equipos.

Esta planificación puede integrarse en la fase de diseño, y la documentación relativa a la planificación elaborada se integra en el documento de diseño del juego, junto con todo el trabajo artístico y de programación realizado.

B. PRODUCCIÓN

En esta segunda fase entran en acción todos los equipos implicados en el desarrollo del videojuego, y es la que más tiempo suele abarcar, entre unos meses y varios años.

- **Versión inicial o First Playable.** Tomando como base las especificaciones del documento de diseño del juego, se empieza con el desarrollo de una versión inicial de este (First Playable o FP) en que se generen todos los recursos: scripts, texturas, interfaces, modelos 3D, animaciones, sonido, etc. Si en la fase de diseño se realizó un prototipo, podemos arrancar esta segunda fase partiendo de él.

El proceso de desarrollo suele seguir un ciclo de vida iterativo e incremental, de manera que el producto se va mejorando en las diferentes iteraciones. Al finalizar la primera iteración, y una vez la mayoría de recursos están listos, o al menos en una versión inicial, empiezan a ensamblarse y ajustarse al resto de recursos dentro del motor de juegos, dando forma a esta primera versión jugable del juego, que se irá mejorando en diferentes revisiones.

- **Pruebas: versiones alfa y beta.** Como en todo producto software, hay que realizar pruebas para detectar y corregir fallos. Si bien durante el desarrollo ya se habrán detectado y corregido algunos errores de programación y de integración de recursos, en esta fase de pruebas se busca más la detección de errores de concepto y la mejora de la jugabilidad.

Las pruebas suelen dividirse en dos tipos: alfa y beta:

- **Pruebas alfa.** Se realizan por un grupo pequeño de personas, generalmente del mismo equipo de desarrollo, y tienen por objetivo corregir defectos graves y mejorar aspectos que no se han tenido en cuenta en el GDD. La versión del juego que se prueba en esta fase recibe el nombre de **versión alfa**, y se integra dentro del propio ciclo de desarrollo del juego. Esta versión alfa implementa todas las funcionalidades esenciales respecto a la jugabilidad, y contiene la mayor parte de recursos gráficos y de sonido del juego, pero no es todavía una versión definitiva.
- **Pruebas beta.** Se llevan a cabo por un equipo externo de jugadores, generalmente más numeroso, y con ellas se pretende buscar errores menores y mejorar la experiencia del usuario y el rendimiento del juego. La versión que se prueba recibe el nombre de **versión beta**, y cuenta con el juego finalizado en cuanto a funcionalidad, jugabilidad y recursos, y se supone ya libre de errores mayores.

Una vez se han finalizado las pruebas beta, se han corregido los bugs y se han incorporado todas las optimizaciones propuestas, se dispone de una versión Release Candidate. Esta versión se considera estable y preparada para su lanzamiento, aunque antes deberá pasar por una fase de validación por parte de los publishers e inversionistas, así como de certificación de la plataforma.

Cuando el producto está validado y certificado, se obtiene la Gold Master, la copia maestra del juego a partir de la que se va a realizar toda la distribución.

C. POSPRODUCCIÓN

Esta última fase se encarga del marketing, la distribución y el mantenimiento del videojuego, tareas que suelen realizarse de forma paralela.

- **Marketing y distribución.** El marketing se encarga de dar a conocer el juego, y suele empezar bastante antes de su publicación. En cuanto a la distribución, esta consiste en la generación de copias y en cómo estas llegan a las tiendas, ya sean físicas o digitales, para su venta o publicación.
- **Mantenimiento.** En cuanto a la parte de mantenimiento, esta comprende diversas tareas. Por un lado, se corrigen los errores que se van detectando y se incorporan algunas mejoras, tales como optimizaciones y balanceo de reglas, soporte, infraestructura o servicios en línea. Estas mejoras se distribuirán en forma de parches o actualizaciones del juego, que suelen descargarse de forma automática.

Por otro lado, en este momento también se pueden añadir nuevas extensiones que aprovechan la base del juego para añadir nuevas pantallas, funcionalidades, objetos o mejoras en los personajes. Todo este contenido descargable (Downloadable Content o DLC) puede ser gratuito o formar parte de la monetización del juego, generalmente a través de microtransacciones.

1.2. Equipos en el desarrollo de un videojuego

Los equipos de desarrollo de videojuegos son equipos multidisciplinares, están formados por profesionales especializados en diferentes disciplinas, tanto técnicas como artísticas, así como de gestión y publicidad. En empresas dedicadas al sector, el tamaño de los equipos puede ir de veinte a más de cien empleados, según los proyectos, y en ellos podemos identificar los roles siguientes:

A. EQUIPO DE DISEÑO

Es el encargado del diseño del juego y de su narrativa, desde su concepción hasta su desarrollo y conclusión. En equipos compuestos por múltiples desarrolladores suele existir el papel de diseñador líder, que se encarga de la coordinación entre los diferentes diseñadores. Entre los roles que podemos encontrar en este equipo, destacan:

- **Diseñador del juego.** Responsable del concepto general, de las mecánicas y del gameplay, y también elabora el documento de diseño del juego.
- **Diseñador de niveles o LevelDesigner.** Responsable de definir los diferentes niveles de los que se compone el juego, que es un proceso tanto artístico como técnico. Entre sus funciones principales están las siguientes:
 - La creación del mapa completo del juego en general y de los mapas de cada nivel en particular, estableciendo sus detalles.
 - Ayudar al diseñador o la diseñadora del juego a establecer la dificultad de los diferentes niveles.
 - Determinar qué mecánicas deberán aplicarse en determinados puntos del juego para poder avanzar.

En definitiva, este rol será el encargado de que el diseño del juego resulte entretenido para el jugador, y que al mismo tiempo le desafíe para completar los diferentes niveles y mantenga su interés por avanzar en el juego.

- **Guionista.** Encargado de crear la historia de fondo del videojuego, componiendo los textos y diálogos, y el diseño de las escenas de introducción y de transición. Asimismo, este rol se encargará de crear los manuales del juego y los libros de ayuda.
- **Diseñador artístico.** Se encarga del diseño del arte del videojuego (ya sea 2D o 3D), con diferentes propuestas visuales para personajes o escenarios, así como de toda la iconografía de este: elementos del HUD, menús, etc.
- **Diseñador del sonido.** Encargado del diseño tanto la música del juego como de los diferentes sonidos o efectos especiales, las voces en off, etc.

B. EQUIPO DE DESARROLLO

El equipo de desarrollo se encarga de transformar la idea y la historia del juego, junto con todo el desarrollo artístico en un producto software. Implementa la funcionalidad especificada por el equipo de diseño teniendo en cuenta las diferentes tecnologías necesarias para los distintos contenidos.

Entre los roles más destacados de este equipo se encuentran:

- **Desarrollador de físicas.** Encargado de la simulación de las físicas en el motor del juego, así como de la detección de colisiones, los movimientos, etc.
- **Desarrollador de IA.** Se encarga de desarrollar agentes inteligentes mediante secuencias de comandos, planificación, toma de decisiones basadas en reglas, etc.
- **Desarrolladores gráficos.** Responsables del uso de recursos del sistema de forma eficiente por parte de los recursos gráficos del juego, optimizando texturas, modelos, etc.
- **Desarrollador de audio.** Responsable de que los diferentes sonidos y la música se reproduzcan en el momento y lugar adecuados.

- **Desarrollador de jugabilidad [gameplay].** Responsable de la implementación del código para las interacciones, y de trabajar juntamente con los diseñadores de niveles para que todo funcione según lo planeado. Este rol también participará en el testeo de los prototipos y las funciones del juego.
- **Desarrolladores de scripting.** Se encargan de desarrollar y mejorar el motor de scripting del juego, con el que se desarrollan el resto de componentes.
- **Desarrollador de la interfaz de usuario.** Encargado del desarrollo de toda la interfaz, desde el HUD (Heads-Up Display) donde aparece toda la información relativa al juego hasta los diferentes menús del juego.

La mayoría de estas funciones ya se encuentran implementadas en los motores de videojuegos actuales. Por ejemplo, Unity ya cuenta con un motor de físicas, una representación gráfica, sistemas de audio, un motor de scripts, una interfaz o agentes de IA, por lo que la tarea de los desarrolladores se centraría más en conocer las herramientas ofrecidas por el motor que en su propio desarrollo. Aparte de todas estas tareas, existen otros roles a tener en cuenta, como la gestión de la interacción con el usuario o las comunicaciones de red, si estas son necesarias.

C. EQUIPO DE PRUEBAS

Si bien las pruebas alfa son llevadas a cabo, por norma general, por el propio equipo de desarrollo, las pruebas beta son realizadas por un equipo específico de beta-testers.

El papel de este equipo de pruebas consiste básicamente en jugar al videojuego en cuestión, pero de una forma metódica y minuciosa, fijándose en todos los detalles y examinando todas las posibilidades que ofrece el juego. Al final, este elabora un informe con todos los errores detectados para su posterior revisión y corrección. Este equipo también transmitirá a los desarrolladores su opinión y experiencia durante el juego, con la finalidad de añadir mejoras a este.

D. EQUIPO DE MARKETING

El equipo de marketing es el encargado de la comercialización y la promoción de los videojuegos. Las empresas de desarrollo de videojuegos grandes tendrán un equipo dedicado al marketing; sin embargo, las pequeñas compañías independientes suelen recurrir a agencias de juegos y otras agencias de marketing digital para la elaboración de estrategias de comercialización de juegos.

2. Desarrollo de juegos 2D

2.1. Características de los juegos 2D

La principal característica de los videojuegos 2D es obvia: trabajan únicamente con dos ejes de movimiento, y esto hace que resulten juegos planos y sin profundidad, en los que estos movimientos están limitados a ambos lados y hacia arriba o abajo.

Comparados con los juegos en 3D, suelen ser juegos más sencillos, al tener limitados los movimientos a estos dos ejes, y tener controles más simples y una menor interacción con otros objetos.

Algunos conceptos clave relacionados con los juegos 2D son los siguientes:

- **Sprites.** El componente fundamental de un juego 2D son los sprites: mapas de bits que representan de forma gráfica personajes, objetos o partes de ellos, y crean efectos de movimiento o cambios de estado mediante animaciones. Estos sprites forman parte de una imagen más grande, y poseen unas coordenadas X e Y que indican su ubicación dentro de esta. Además, se utilizan máscaras o claves de color para hacer transparentes ciertas partes del sprite para que se integren con el fondo. En Unity, los sprites son GameObjects en 2D a los que podemos asociarles imágenes, y que serían equiparables a las texturas de los polígonos en 3D.
- **Cámara y perspectiva.** La cámara en un juego 2D también se simplifica, pasando de una cámara en perspectiva, que ofrece profundidad en los juegos 3D, a una cámara ortográfica, donde se pierde esta profundidad de las imágenes.

En los primeros videojuegos la cámara era fija, y los diferentes niveles del juego eran literalmente pantallas. Posteriormente, apareció la posibilidad de que esta cámara se desplazase realizando un scroll y siguiendo al personaje, lo que posibilita que los niveles sean más extensos. Algunos juegos utilizan también lo que se conoce como efecto parallax, consistente en desplazar el fondo a una velocidad diferente de la del primer plano con el movimiento de la cámara, lo que crea una ilusión de profundidad.

- **Géneros.** Todas estas características de los videojuegos 2D afectan a la jugabilidad y son decisivas a la hora de determinar el género de un videojuego. Los géneros más usuales para videojuegos 2D son los juegos de plataformas (Super Mario Bros, Sonic the Hedgehog), los juegos de lucha (Street Fighter, Mortal Kombat) o los rompecabezas (Tetris).

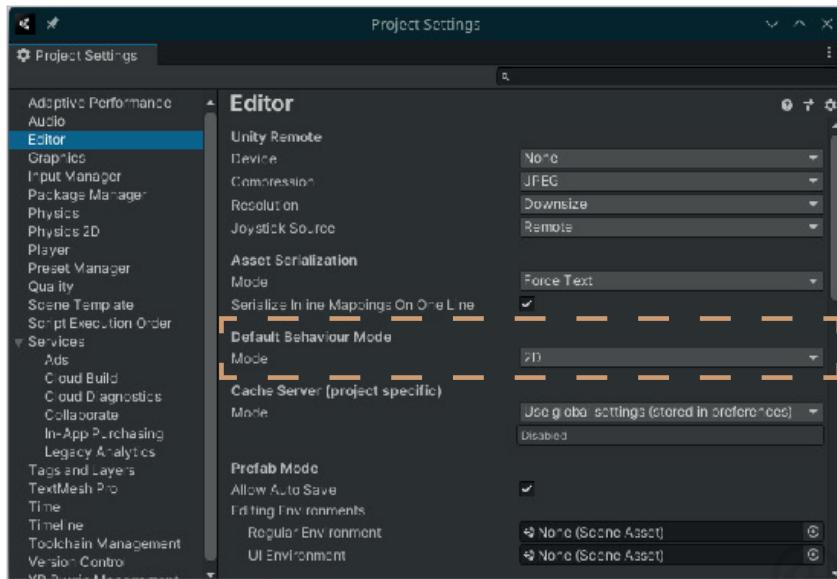
DESARROLLO 2D EN UNITY

El desarrollo de videojuegos 2D en Unity es posible desde su versión 4.3, en 2013, que incorporó soporte nativo al editor para 2D.

Cuando se inicia un proyecto nuevo, Unity Hub ofrece la posibilidad de escoger, entre otros, un proyecto 2D o 3D, el cual determinará el modo en que trabaje el editor.

La principal diferencia entre ambos modos (2D y 3D) es la posición y proyección de la cámara (perspectiva u ortográfica) y el modo en que se importan los recursos (Assets) de tipo imagen, que se importan como sprites en lugar de como texturas. Además, las escenas no disponen de Skybox, y la iluminación también se simplifica. Podéis encontrar más información sobre este modo en el documento *2D and 3D mode settings* de la documentación de Unity.

No obstante, es posible cambiar entre ambos modos durante el desarrollo del juego, mediante las preferencias del proyecto (Menú principal: *Edit > Project Settings*), en la opción *Editor > Default Behaviour Mode*.



2.2. Creación de niveles 2D

Básicamente, existen dos aproximaciones para la creación de niveles o mapas en videojuegos 2D: basados en sprites o basados en mosaicos.

A. DISEÑO DE MAPAS BASADOS EN SPRITES (SPRITE BASED)

El diseño de mapas basados en sprites no tiene un gran control sobre todo aquello que aparece en pantalla, lo que nos deja total libertad para ubicar cualquier imagen dentro de la escena.

Este flujo de trabajo ofrece niveles con mucho más nivel de detalle y riqueza visual, aunque su coste, tanto en dedicación como en consumo de recursos, es más elevado.

B. DISEÑO DE MAPAS BASADOS EN MOSAICO (TILE BASED)

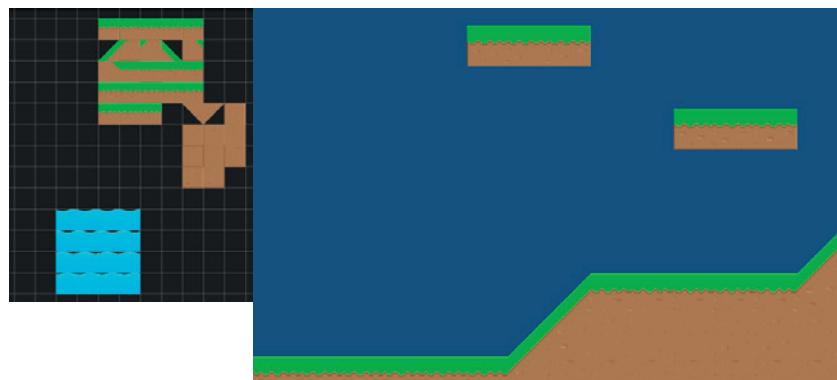
Se trata de una técnica muy usada en el desarrollo de juegos 2D y consiste en construir el mapa del juego a partir de pequeñas imágenes sobre una cuadrícula, generalmente en forma rectangular, hexagonal o isométrica. Estas piezas se conocen como *tiles*, que podríamos traducir como ‘teselas’, ‘losetas’ o ‘azulejos’.

Esta aproximación supone, por una parte, mejoras en el rendimiento y en el consumo de recursos, ya que los archivos de imágenes son relativamente pequeños y, además, se repiten varias veces. Asimismo, implica un incremento en la productividad, puesto que se ahorra tiempo en el desarrollo artístico.

Tilesets y Tilemaps

El tileset o atlas de sprites es el conjunto de tiles o losetas que pueden componer el escenario de un videojuego, y que se encuentran en una misma imagen. En dicha imagen se pueden distinguir los diferentes objetos o partes de objetos que formarán el escenario: paredes, plataformas, suelos, etc.

Por su parte, el tilemap es el escenario en sí, donde las diferentes losetas se encuentran ubicadas en la cuadrícula de forma coherente para formar lo que comúnmente conocemos como **mapa del nivel**. Como podemos ver, el mismo tileset sirve para crear multitud de mapas.



En Unity, en los proyectos 2D podemos generar la cuadrícula para un mapa mediante el GameObject de tipo 2D tilemap. Cuando lo hacemos, se crea un nodo de tipo grid y, dentro de él, otro de tipo tilemap. Esto permite utilizar varios tilemaps dentro del mismo grid, y con diferentes propiedades, como, por ejemplo, que unos posean collider e interactúen con otros GameObjects.

Cuando generamos una cuadrícula, el tamaño de esta se define en unidades, que por defecto serán de 100 píxeles cada una. Para ajustar este tamaño, por ejemplo, si vamos a utilizar tiles de 256x256 píxeles, debemos ajustar el tamaño de la unidad a este número de píxeles.

En el Caso práctico 2 de la unidad abordaremos cómo crear un proyecto 2D en Unity e importar un conjunto de Assets para el desarrollo de juegos 2D. En él crearemos nuestra propia paleta de tiles y crearemos un tilemap para nuestro primer nivel del juego.

2.3. Mecánicas 2D

Las mecánicas en un videojuego 2D vendrán determinadas por el género de este. Veamos algunos ejemplos:

- **Juegos de plataformas** (ej. *Mario Bros*), donde las principales mecánicas serán acciones como correr, saltar, disparar o recoger objetos para avanzar generalmente de forma lineal de un punto a otro.
- **Juegos de tipo Shoot'em up** (ej. *1942*), donde generalmente controlamos un vehículo que dispara proyectiles. La vista puede ser tanto lateral como desde arriba, y el personaje suele moverse en ambos ejes.
- **Laberintos** (ej. *Pac-Man*), donde el jugador puede desplazarse en el eje vertical y en el horizontal para recoger elementos y huir de los enemigos.
- **Rompebloques** (ej. *Breakout*), donde manejamos una paleta en el eje horizontal para golpear una pelota y romper bloques.
- **Juegos de lucha** (ej. *Street Fighter*), donde manejamos un personaje, generalmente en el eje horizontal, que admite saltos y varios tipos de golpes.
- **Beat'em up** (ej. *Double Dragon*), similar a los juegos de lucha, aunque suele permitirse también el movimiento en vertical, y las mecánicas de los golpes son más sencillas.

Como vemos, básicamente las mecánicas consistirán en desplazamientos de los personajes u objetos de un lugar a otro, y en la interacción de estos con otros objetos, ya sea recogiéndolos, golpeándolos o disparándolos.

Para los juegos 2D, utilizaremos la clase Vector2, en lugar de la clase Vector, puesto que los movimientos se restringirán únicamente a dos dimensiones. Así pues, la clase Vector2 tendrá, por ejemplo, los vectores predefinidos siguientes:

Eje	Dirección	Vector predefinido	Valores de Vector2
Y	Arriba	Vector2.up	Vector2[0, 1]
-Y	Abajo	Vector2.down	Vector2[0, -1]
X	Derecha	Vector2.right	Vector2[1, 0]
-X	Izquierda	Vector2.left	Vector2[-1, 0]
-	-	Vector2.zero	Vector2[0,0]

Disponemos de más información acerca de esta clase en la documentación oficial de Unity.

En los casos prácticos extendidos 1 y 2 de la unidad trabajaremos algunas mecánicas para mover nuestro personaje y los enemigos por la escena.

2.4. Manejando la cámara: scroll y efecto parallax

En los primeros videojuegos 2D el contenido que se mostraba en la pantalla correspondía, por norma general, al nivel completo, de modo que el avance por el juego era mediante el paso de una pantalla a otra.

El scroll apareció como una técnica para mejorar la sensación de movimiento, y para que el personaje pudiese avanzar. Esto abrió la posibilidad de crear niveles con mucha más extensión.

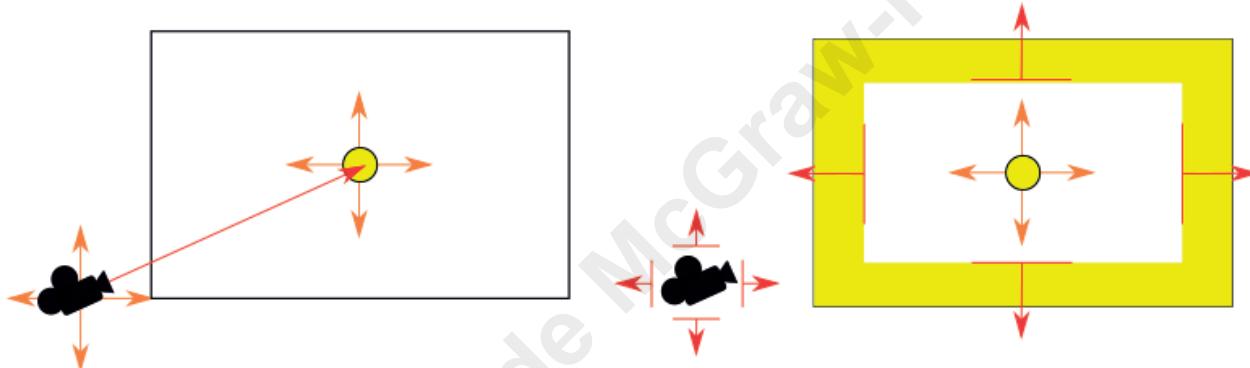
El scroll puede ser horizontal, cuando el escenario avanza en el eje X, vertical, cuando avanza en el eje Y, o una combinación de ambos.

La cámara que utilizamos para los videojuegos 2D es una cámara de proyección ortográfica, en la que no se aprecia la profundidad de los objetos. Esta cámara, por defecto, se encuentra fija y ubicada en la posición (0, 0, -10).

El scroll en Unity para los videojuegos 2D se consigue de forma natural, moviendo la cámara en los ejes X e Y.

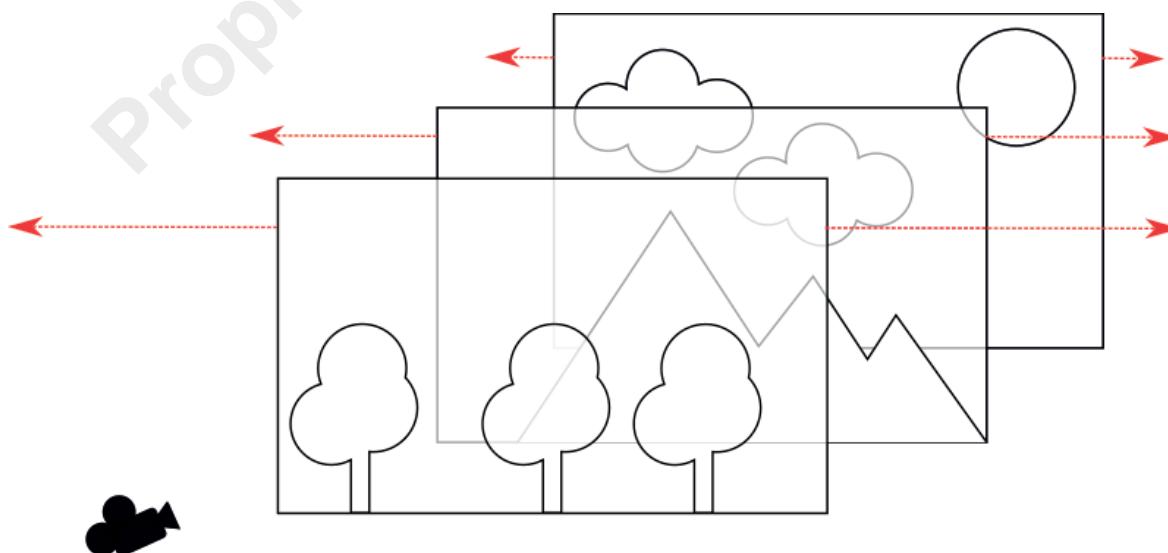
Podríamos decir que hay un par de aproximaciones para realizar esto:

- Que la cámara siga al jugador, de modo que este siempre esté en la posición central de la pantalla y lo que aparentemente se mueva sea todo el entorno.
- Que el jugador se mueva libremente en una región determinada y, cuando sobrepase esta, la cámara de desplace para seguirle.



A. EFECTO PARALLAX

El efecto parallax o de paralaje consiste en conseguir un efecto de profundidad en entornos 2D mediante el movimiento del fondo o de diferentes fondos a diferentes velocidades, de modo que aquellos planos que se encuentren más lejanos se muevan a menor velocidad que los más cercanos.



En el Caso práctico extendido 3 veremos cómo controlar la cámara para que esta siga al personaje, incorporaremos un fondo y añadiremos un efecto de parallax para dar sensación de profundidad.

2.5. Animación

Unity 5 incorpora el sistema de animación Mecanim, un completo sistema que ofrece un flujo de trabajo para realizar de forma sencilla animaciones de básicamente cualquier elemento de Unity, como objetos, personajes o propiedades.

Entre sus principales características, podemos citar:

- Soporte para clips de animación (Animation Clips), que describen la animación y que pueden ser importados desde herramientas externas o creadas en el propio Unity.
- Soporte para animaciones genéricas o humanoides, que permite animar diferentes partes de un cuerpo por separado.
- Previsualización de animaciones, transiciones e interacciones.
- Soporte para varias capas y máscaras.

Los conceptos fundamentales de la animación en Unity son los **clips de animación (Animation Clips)** que ya hemos comentado, que contienen la información sobre cómo se animan diferentes objetos o propiedades; y los **controladores de animación (Animation Controllers)**, que representan una máquina de estados, que controlan qué clips de animación se reproducen en función del estado en que se encuentre el GameObject.

Para agregar una animación a un GameObject, deberemos incorporarle un componente de tipo Animation, el cual tendrá una referencia al Animation Controller, a los Animation Clips y al Avatar, un sistema para representar personajes humanoides.

En el Caso práctico extendido 4 veremos cómo crear una animación de los sprites para los personajes de nuestro juego.

2.6. Creación del HUD y la interfaz de usuario

Unity dispone de un conjunto de GameObjects especialmente pensado para el diseño de interfaces de usuario, que servirán tanto para mostrar las opciones de los diferentes menús y pantallas de presentación del juego y los niveles como para mostrar información sobre la partida (puntuaciones, vidas, tiempo restante, etc.), en la misma vista del juego, a través del HUD (Head-Up Display).

A.CANVAS Y OBJETOS DE LA INTERFAZ

Para añadir elementos de interfaz de usuario a una escena, lo hacemos dentro del contexto de un Canvas, un tipo de GameObject que sirve como marco en el que poder incorporar los diferentes elementos de interfaz dentro de su jerarquía.

Así pues, siempre que añadamos un objeto de interfaz de usuario en una escena, se creará automáticamente el GameObject Canvas y el objeto de interfaz que hemos añadido como hijo, además de otro GameObject de tipo EventSystem que servirá para gestionar los eventos y que, generalmente, no necesitaremos modificar. Todos estos elementos de interfaz se ubicarán en una capa [layer] específica para ellos, la capa UI.

Cuando añadimos un Canvas, en la vista de escena este aparecerá como un rectángulo que representa el marco para poder posicionar los elementos de la interfaz. Este rectángulo se mostrará en la vista de escena con unas proporciones mucho mayores que las de la escena en que estemos trabajando, para poder trabajar de forma más cómoda y para que el diseño de la interfaz no interfiera con el de la escena. No obstante, cuando se muestren en modo Juego, la capa de UI se renderizará de forma superpuesta sobre el resto de la escena. Este renderizado del Canvas puede ser de tres tipos, que indicaremos en la propiedad Render Mode del Canvas en el Inspector:

- **Screen Space - Overlay.** Coloca el Canvas delante de todos los elementos de la escena, como si añadiésemos un papel de acetato transparente ante la cámara. Si el tamaño de la pantalla se modifica, el Canvas se redimensionará automáticamente para coincidir con la pantalla.
- **Screen Space - Camera.** Similar al modo Overlay, pero afectado por el tipo de perspectiva. Si la cámara está en perspectiva, la interfaz se verá también en perspectiva.
- **World Space.** El Canvas se comporta como cualquier otro objeto de la escena.

B. ELEMENTOS DE LA UI

Unity ofrece un conjunto interesante de GameObjects para crear elementos de interfaz. Para añadir uno de estos elementos, lo hacemos como cualquier otro objeto, bien desde el menú GameObject, bien desde el menú contextual en la vista jerárquica. En el submenú UI podemos encontrar los diferentes elementos de interfaz, como pueden ser los siguientes:

- Botones (Button), compuestos por el propio botón y un nodo hijo de tipo texto.
- Imágenes, bien a partir de un sprite o material (Image), bien a partir de una textura (RawImage).
- Botones de selección (Toggle).
- Sliders.
- Vistas y barras de scroll (ScrollView/Scrollbar).
- Selectores de opciones desplegables (Dropdown).
- Entrada de texto (InputText).
- Paneles.

Podemos encontrar mucha más información sobre la interfaz de usuario en las páginas del manual de Unity dedicadas a ello.

C. GESTIÓN DE ESCENAS: SCENEMANAGER

Generalmente, un juego está compuesto por múltiples escenas, que representan tanto los niveles como los diferentes menús y pantallas de transición entre ellas.

Los cambios entre estas escenas se realizan a través de una clase especial llamada SceneManager.

Debemos tener en cuenta que, cuando se produce un cambio de escena, todos los objetos de la escena donde nos encontramos se destruyen. Por este motivo es necesario disponer de mecanismos que se encarguen de mantener la información del juego que deseamos que persista en estos cambios de escena. Para ello, existen varias aproximaciones, desde las más sencillas, como utilizar una clase estática que gestione los datos, hasta frameworks de inyección de dependencias, pasando por utilizar las *PlayerPrefs*, que también sirven para guardar datos entre partidas o archivos.

A continuación, en el Caso práctico 2 veremos cómo añadir algunas escenas al juego para incorporar menús, así como diferentes elementos de interfaz que compongan el HUD.

3. Desarrollo de juegos 3D

Los videojuegos 3D entraron de lleno en el mercado en la década de los noventa del siglo pasado, con la quinta generación de consolas y las mejoras en el hardware gráfico de los equipos, de la mano de juegos como *Starfox*, *Wolfenstein 3D* o *Doom*.

3.1. Características de los juegos 3D

En los videojuegos 3D se incorpora una nueva dimensión, de modo que los movimientos pueden realizarse en un espacio tridimensional, igual que en el mundo real.

Se trata de juegos más complejos que los juegos 2D, en los que ya entra en juego la cámara con perspectiva, que puede moverse incluso de forma independiente al personaje y permite observar el entorno del juego desde diferentes ángulos.

A nivel de representación, en lugar de sprites se utilizan modelos tridimensionales donde los mapas de bits se utilizan para aportar texturas a esos modelos.

Por otra parte, la animación de los personajes también es más compleja, ya que, en lugar de sprites, debemos animar por completo todo un modelo, y estos deben reaccionar a otros elementos del entorno.

Un proceso muy importante en los videojuegos 3D es el del renderizado o render: la conversión de una imagen 3D a su correspondiente representación en una pantalla 2D, donde deberemos determinar el color de cada píxel en función de los diferentes objetos, colores, texturas e iluminación.

Todas estas características ofrecieron nuevas posibilidades y dieron pie a nuevos géneros, como los First Person Shooters (*Half-Life*, *Call of Duty*) o la adaptación de géneros clásicos, como las plataformas (*Super Mario 64*) o los juegos de carreras (*Need for Speed*).

3.2. Objetos en 3D

Los GameObjects que utilizaremos para juegos en 3D poseen una forma tridimensional, definida en su componente Mesh Filter, que puede tomar la forma de cubos, cápsulas, cilindros, planos, esferas, quads o cualquier otra malla que podamos importar.

Además de la forma, estos objetos poseen otro componente importante sin el cual tampoco se visualizaría el objeto: el Mesh Renderer, que define cómo se va a representar el objeto determinando tanto su material como sus propiedades relacionadas con la iluminación, como si reciben o proyectan sombras o contribuyen a la iluminación global de la escena.

También será habitual que posean un collider, que generalmente será una forma más simple que la malla en sí del objeto, para simplificar los cálculos en la detección de colisiones. En personajes humanoides podemos utilizar, por ejemplo, cápsulas que contengan al personaje.

A. MODELOS 3D

A pesar de que podemos utilizar los GameObjects 3D que ofrece Unity y realizar cualquier combinación con ellos, lo más habitual será importar modelos creados con herramientas de modelado como Blender, 3D Studio Max o Maya. Aunque el formato que utiliza Unity para representar objetos 3D es el .fbx, podemos utilizar directamente formatos nativos de estas herramientas de modelado y dejar que Unity realice la conversión en el proceso de importación.

B. MATERIALES

El material es un componente esencial para la representación del modelo y contiene la información acerca de su apariencia: las texturas, el color (tint) o los shaders.

Las texturas, para las que podemos utilizar cualquier tipo de imagen 2D, recubren la malla agregando los detalles del objeto. Si agregamos también un color, este afectará también al color de la textura.

Por otra parte, los shaders van a determinar cómo se muestran los GameObjects en pantalla.

Renderización y shaders

La renderización o el render es el proceso por el cual obtenemos una imagen 2D a partir de una escena en 3D. Este proceso es necesario, puesto que los dispositivos en los que se va a reproducir un juego o cualquier entorno 3D son pantallas en 2D.

Para ello, necesitamos determinar qué color tendrá cada píxel de la imagen 2D en función de todos los objetos de la escena, su color, transparencia, material o brillo, así como de otros factores externos, como todas las fuentes de iluminación. Con todo esto, podemos imaginar el coste computacional que tendría renderizar una imagen, por ejemplo, de 1920x1080 píxeles a 60 fps.

Aquí es donde entran en juego los shaders. El concepto de shader fue introducido en 1984 por Robert Cook, durante su trabajo en Lucasfilm (Pixar). Básicamente, un shader es un procedimiento que se puede invocar por un programa más general para calcular un valor o un conjunto de valores durante este proceso de renderizado a partir de ciertos parámetros. La traducción más literal de *shader* sería 'sombreador', aunque, como podemos intuir, hace más cosas aparte de determinar las sombras de los objetos.

Lo más interesante es que los shaders ofrecen una API a los desarrolladores, y pueden implementarse a distintos niveles: en software, incluidos en librerías gráficas como DirectX u OpenGL, o directamente embebidos en el hardware de la GPU.

C. DISEÑO DE NIVELES 3D

Del mismo modo que las diferentes mallas, podemos generar el entorno de nuestro juego, bien dentro del entorno de Unity, bien mediante herramientas externas e importarlo.

Dentro de Unity, podemos utilizar cualquier GameObject en la escena e ir ajustando sus propiedades. Uno de los elementos más interesantes es el generador de terrenos, que permite crear en forma de mosaico varios terrenos y ajustar tanto la altura como la apariencia del paisaje añadiendo elementos naturales como árboles o césped.

Para la creación de terrenos, Unity ofrece el GameObject Terrain, con una superficie de 1000x1000 metros en la que podemos esculpir cualquier terreno, como veremos en el Caso práctico 3.

D. ANIMACIÓN DE OBJETOS

Unity permite importar clips de animaciones realizadas con herramientas externas, cuando importa un modelo que las posea, o generarlas directamente sobre el editor de animaciones modificando su tamaño, posición o rotación.

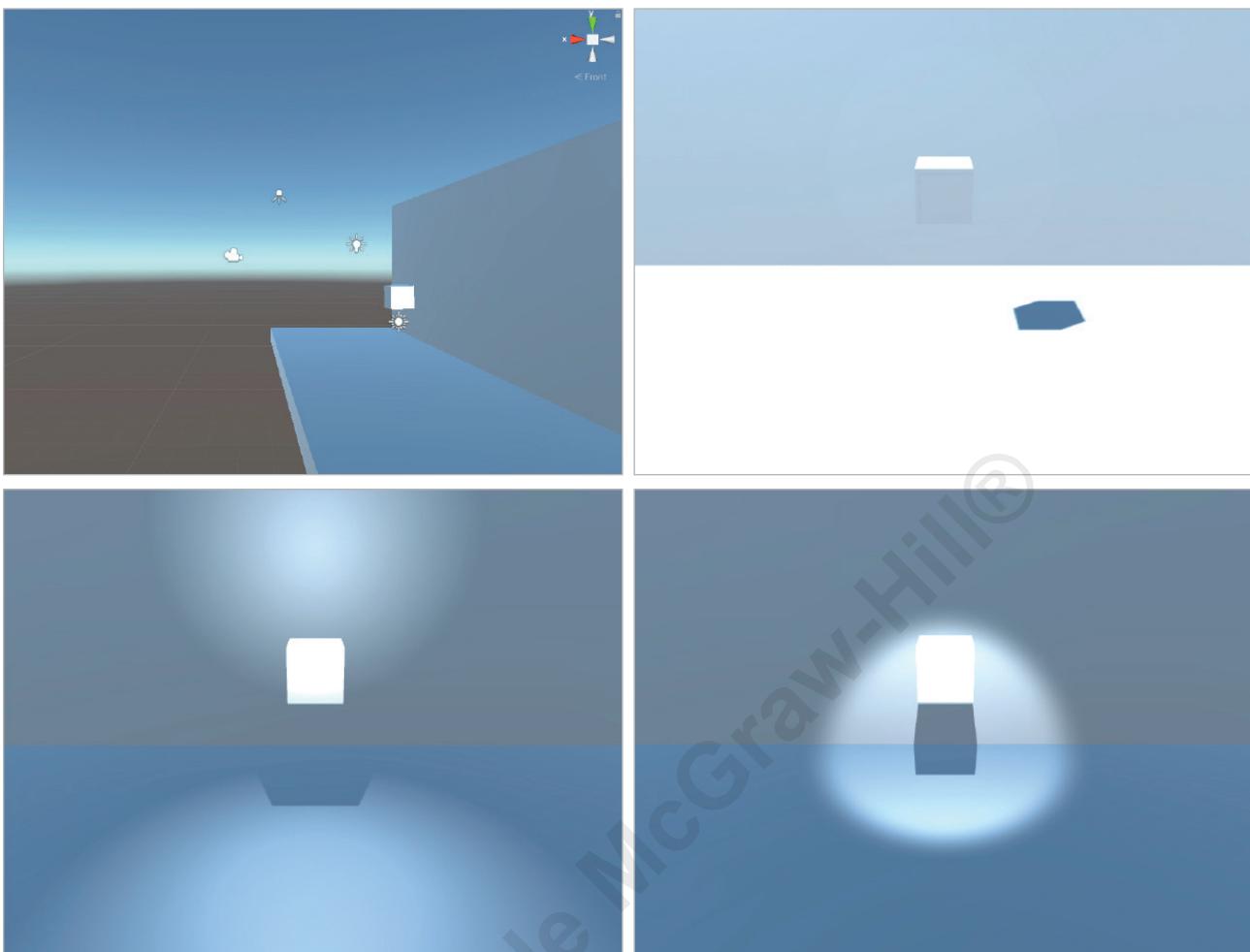
3.3. Iluminación

La iluminación desempeña un papel clave en las escenas 3D, proporcionando el ambiente deseado para cada una de ellas.

Las luces se agregan a la jerarquía de la escena como cualquier otro GameObject, y básicamente pueden ser de tres tipos:

- **Luces direccionales (Directional Light).** Se trata de fuentes de luz que se encuentran a gran distancia, de modo que sus rayos son paralelos en una dirección y su intensidad no disminuye. Se utilizan principalmente para emular la luz del sol y aportar, en general, un tipo de iluminación global.
- **Luces puntuales (Point Light).** Se trata de fuentes de luz ubicadas en cierto punto del espacio y que emiten luz en todas las direcciones, cuya intensidad va decreciendo a medida que nos alejamos del punto de luz. Su comportamiento sería el más parecido a las luces en el mundo real.
- **Focos (Spotlight).** Son fuentes de luz ubicadas en un punto del espacio y emiten luz en una dirección concreta. Suelen utilizarse para luces artificiales como linternas o faros.

En la imagen siguiente vemos una escena con los tres tipos de luz y un renderizado con el efecto conseguido. (Por orden: luz direccional, puntual y de foco):



3.4. Mecánicas en 3D

Como hemos comentado, la introducción de las 3D en los videojuegos dio pie a nuevas mecánicas y nuevos géneros, como los First y Third Person Shooters, y también a la adaptación de géneros clásicos a las 3D: videojuegos deportivos, de simulación, de acción y aventuras, etc.

Así como en los videojuegos 2D podemos utilizar tanto movimiento físico como cinemático, en los videojuegos 3D suele utilizarse generalmente el movimiento físico, el cual da mayor realismo al juego y ayuda a la inmersión en este.

Como ya vimos en la unidad anterior, para la aplicación del motor de físicas necesitamos de un componente RigidBody en los GameObjects, y trabajaremos con la clase Vector3 para especificar movimientos, posiciones, velocidades o fuerzas.

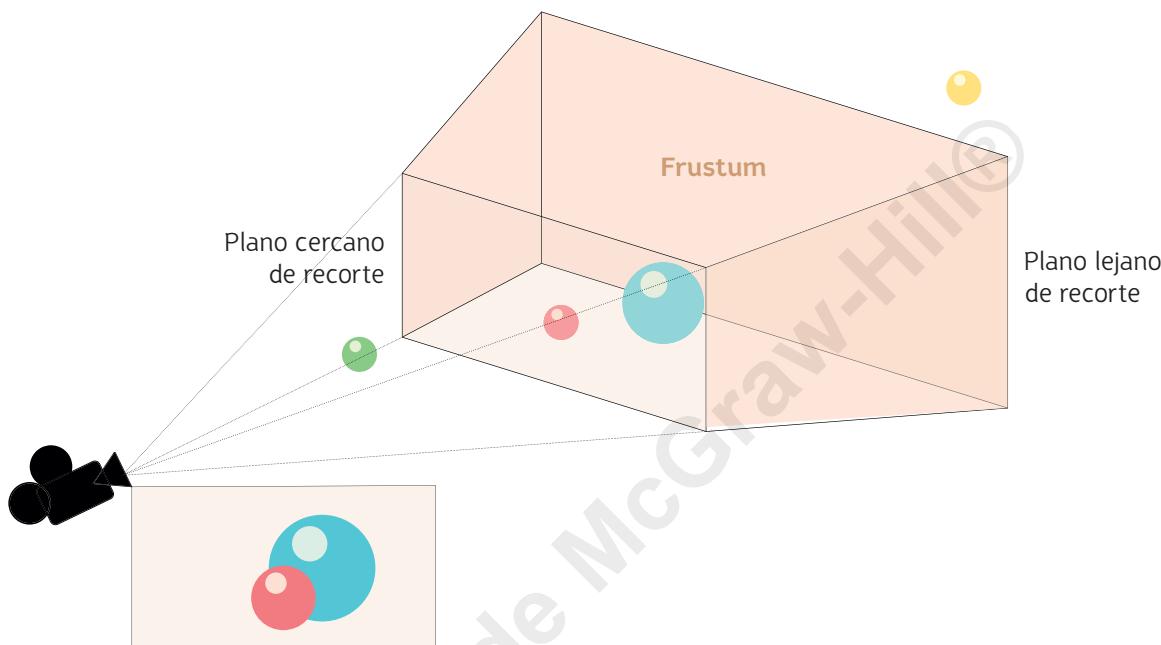
3.5. La cámara en juegos 3D

En los videojuegos 3D, la cámara tiene un papel mucho más importante que en los juegos 2D. Recordemos que, así como en los juegos 2D la proyección de la cámara era ortográfica, en los 3D ofrece una proyección perspectiva.

Vamos a ver algunos de los conceptos más importantes de la cámara cuando trabajamos con perspectiva.

A. VIEW FRUSTUM Y ASPECT

En geometría, un frustum o tronco es una porción de un sólido, generalmente un cono o una pirámide cortado por dos planos paralelos. Aplicado a una cámara de perspectiva, se refiere a la región de la escena 3D que se va a mostrar.



Como vemos, la cámara se situaría en el ápice de la pirámide (o centro de la perspectiva), y los planos que definen el tronco se denominan planos de recorte cercano (el más cercano a la cámara) y lejano (el más lejano). Todos aquellos objetos que no estén dentro del tronco no se renderizarán. En la imagen vemos que solamente se muestran la esfera azul y la esfera roja, mientras que la verde y la amarilla no, ya que una está demasiado cerca y la otra, demasiado lejos. El hecho de utilizar un plano lejano es obvio, ya que los objetos demasiado lejanos no suelen ser significativos. Respecto al plano de recorte cercano, es necesario para que objetos que estén demasiado cerca de la cámara no ocupen todo el campo de visión.

El ángulo de la cámara que define el tamaño del tronco se conoce como campo de visión (field of view) o FOV, de forma abreviada. Unity permite ajustar, entre otras cosas, tanto el campo de visión, a través del atributo Field of View, como la distancia a la que se sitúan los planos cercano y lejano, que definen el campo de dibujado. Por defecto, estos planos se encuentran a 0.3 y a 1000 metros, respectivamente.

B. SISTEMAS DE CÁMARAS. JUEGOS EN PRIMERA Y TERCERA PERSONA

En los videojuegos 3D vemos la evolución de la historia a través de la cámara, que puede ser en primera o en tercera persona.

En los videojuegos en **primera persona** la cámara se sitúa exactamente en la perspectiva del personaje. En estos juegos el control de la cámara es relativamente sencillo, pues su visión es la misma que la que tendría el personaje, y para explorar el entorno debemos mover el personaje.

Generalmente, en este tipo de videojuegos como mucho vemos las manos o las armas del personaje, pero no sabemos nada más de su forma.

Por otra parte, en los videojuegos en **tercera persona** disponemos de un sistema de cámaras que permiten ver al personaje, bien desde atrás, bien desde otros ángulos. Generalmente, la cámara seguirá al jugador, de manera que permite ampliar la vista que tenemos del entorno y la ubicación del personaje en este.

En los videojuegos en primera persona solamente tenemos un tipo de cámara, pero en los videojuegos en tercera persona existen diferentes tipos de sistemas de cámaras. Los más habituales son los siguientes:

- Sistemas de cámara fija, donde la cámara se ubica en una posición predeterminada y no puede ser controlada por el jugador. Se utilizó en videojuegos como Grim Fandango y los primeros Resident Evil.
- Sistemas de cámaras de detección o de seguimiento, que siguen al jugador mientras este se mueve, pero no son lo suficientemente flexibles para entornos abiertos. Ejemplos de este tipo de cámara son Crash Bandicoot o Tomb Raider.
- Sistemas de cámaras interactivas, los más comunes actualmente, que siguen al personaje, pero permiten el control de la cámara mediante algún sistema de entrada como sticks analógicos o el ratón. Un ejemplo de uso de este tipo de cámaras es Super Mario Sunshine.

3.6. Juegos 2.5D

A medio camino entre los videojuegos 2D y los 3D se encuentran los videojuegos 2.5D. Podríamos decir que se trata de videojuegos con mecánicas 2D que utilizan gráficos 3D. Algunos videojuegos que siguen este esquema serían *Donkey Kong Country: Tropical Freeze* o *Street Fighter V*, juegos que usan modelos 3D para los personajes, el entorno y los objetos, pero que siguen con las mecánicas y la jugabilidad de los juegos clásicos en 2D.