

Curso Completo de Android

Autor: Esteban

lucasegaacademy.com

*Guía completa con explicaciones extensas por unidad,
prácticas y examen final.*

Curso Completo de Android

Autor: Esteban

lucasegaacademy.com

Publicado por Colibrí Editorial EGA

© 2026 Esteban. Todos los derechos reservados.

Queda prohibida la reproducción total o parcial de esta obra
sin autorización escrita del autor.

ÍNDICE

PRÓLOGO	4
UNIDAD 1: Introducción a Android y Android Studio	10
UNIDAD 2: Fundamentos de Kotlin	42
UNIDAD 3: Actividades y su ciclo de vida	78
UNIDAD 4: Layouts XML y vistas	113
UNIDAD 5: Intents y navegación entre pantallas	154
UNIDAD 6: RecyclerView y listas dinámicas	185
UNIDAD 7: Persistencia de datos	216
UNIDAD 8: Consumo de APIs y trabajo en red	256
UNIDAD 9: Permisos y funciones del dispositivo	297
UNIDAD 10: Proyecto final y publicación en Google Play	342
EXAMEN FINAL	382

PRÓLOGO

Bienvenido al "Curso Completo de Desarrollo Android" de Lucas EGA Academy. Si abriste este libro, probablemente ya pasó por tu cabeza la misma idea que pasa por la cabeza de miles de personas cada semana: "quiero crear mi propia aplicación para celular". Tal vez tienes una idea de negocio, tal vez quieres conseguir un mejor empleo, tal vez simplemente te da curiosidad saber cómo se construye eso que usas todos los días con el pulgar. Cualquiera que sea tu motivo, este libro está escrito para llevarte desde cero hasta el punto en que puedas construir, probar y publicar aplicaciones reales para Android.

Primero lo más importante: no necesitas experiencia previa creando apps. Este libro no asume que ya programaste antes, aunque si alguna vez escribiste unas líneas de código en cualquier lenguaje, vas a avanzar más rápido. Lo que sí asumo es que sabes usar una computadora con normalidad: instalar programas, crear carpetas, copiar archivos, navegar por menús. Si eso lo dominas, tienes todo lo necesario para empezar. El resto lo construyes aquí, paso a paso, con la paciencia de un profesor que ha visto los mismos errores cientos de veces y sabe exactamente dónde se atora la gente.

Android es el sistema operativo móvil más usado del planeta. Más de siete de cada diez teléfonos en el mundo corren Android, y ese número incluye una enorme variedad de dispositivos: teléfonos baratos de gama de entrada, teléfonos insignia de gama alta, tabletas, televisiones, relojes inteligentes y hasta sistemas de automóviles. Aprender a desarrollar para Android no es aprender a hacer una app para un solo tipo de teléfono: es aprender a crear software para un ecosistema gigantesco, con millones de usuarios potenciales. Por eso las empresas contratan desarrolladores Android, por eso los freelancers consiguen proyectos, y por eso publicar tu propia app en Google Play puede convertirse en un negocio real.

Pero hay algo que debo decirte con honestidad desde la primera página: aprender desarrollo Android no es un atajo. No existe el curso milagroso que te convierte en programador profesional en una semana. Lo que sí existe es un camino probado, ordenado y sin rodeos, y ese es el camino que sigue este libro. Cada unidad se construye sobre la anterior. No vas a saltar a temas avanzados antes de dominar lo básico, porque la experiencia demuestra que esa prisa termina en frustración. Vas a ir lento al principio, y esa lentitud es intencional: los cimientos que pongas en las primeras unidades son los que van a sostener todo lo demás.

A quién va dirigido este libro

Este libro fue escrito pensando en tres tipos de lector, y probablemente eres al menos uno de ellos.

El primero es el principiante total. Nunca ha creado una aplicación, tal vez programó un poco en la escuela o tomó un curso de lógica de programación y se quedó a la mitad. Para ti, este libro empieza desde lo más básico: qué es Android, cómo se instala la herramienta de trabajo, cómo se escribe la primera línea de código. No se da nada por sabido. Cada término nuevo se explica la primera vez que aparece, y cada concepto se ilustra con analogías de la vida real antes de mostrar el código.

El segundo es el programador de otros lenguajes. Ya sabes programar en Python, JavaScript, Java o C#, pero nunca tocaste el desarrollo móvil. Para ti, el libro respeta tu experiencia: no te trata como si no supieras qué es una variable, pero sí te explica con cuidado todo lo que es exclusivo del mundo Android, como el ciclo de vida de las pantallas, los permisos del sistema y la forma en que Kotlin se integra con las herramientas de Google. Las unidades de Kotlin están diseñadas para que puedas avanzar rápido si ya dominas los fundamentos, y detenerte en los detalles que sí son nuevos para ti.

El tercero es el entusiasta que ya intentó aprender Android por su cuenta y se rindió. Viste tutoriales sueltos en internet, copiaste código que no entendías, tu app se atoró con un error rojo que nadie te explicó, y abandonaste. Para ti, este libro ofrece lo que los tutoriales sueltos no dan: orden, continuidad y explicación de los errores. Aquí no solo vas a ver qué código escribir, sino por qué se escribe así, qué pasa si lo escribes mal y cómo arreglarlo cuando algo sale mal. Porque algo va a salir mal, te lo garantizo: programar es resolver problemas, y aprender a leer errores es una habilidad tan importante como aprender a escribir código.

Si no eres ninguno de estos tres, pero tienes curiosidad genuina y estás dispuesto a practicar, este libro también es para ti. La curiosidad con disciplina vale más que el talento sin constancia.

Qué aprenderás

Al terminar este libro sabrás construir aplicaciones Android completas y funcionales. No demos ni maquetas: aplicaciones de verdad, que se instalan en un teléfono real y hacen cosas útiles. El recorrido está organizado en unidades que siguen una progresión lógica.

Empezarás entendiendo qué es Android como plataforma: su historia, cómo está construido, por qué se programa cómo se programa y qué herramientas usan los profesionales. Después instalarás Android Studio, el entorno oficial de desarrollo, y crearás tu primera aplicación "Hola Mundo", que aunque parezca un programa tonto, te enseña el ciclo completo de trabajo: escribir código, compilarlo, instalarlo en un dispositivo y ver el resultado.

Luego dominarás Kotlin, el lenguaje oficial y moderno para Android. Aprenderás variables, tipos de datos, la diferencia entre valores que cambian y valores que no, el famoso sistema de seguridad contra nulos, condicionales, ciclos, funciones y las expresiones lambda. Kotlin es el corazón de todo lo que harás después, así que estas unidades merecen tu máxima atención: cada hora que inviertas aquí te ahorra tres horas de confusión más adelante.

Más adelante en el libro (en las partes que siguen a esta) entrarás al desarrollo de interfaces: cómo se dibujan las pantallas, cómo responden a los toques del usuario, cómo se navega entre pantallas, cómo se guardan datos en el teléfono, cómo se consume información de internet y cómo se prepara una app para publicarla en Google Play. Cada tema nuevo se presenta con la misma estructura que verás en esta primera parte: explicación detallada, ejemplos paso a paso, práctica con soluciones y resumen.

Al final del libro habrás construido un proyecto final completo: una aplicación funcional que integra todo lo aprendido. Ese proyecto no es un ejercicio más: es tu carta de presentación. Es lo que le vas a mostrar a un empleador, a un cliente o a tus amigos cuando te pregunten "¿y tú qué sabes hacer?".

Cómo estudiar este libro

Este libro no se lee como una novela. Se trabaja como un taller. Cada unidad está diseñada para estudiarse en sesiones de una a dos horas, y te recomiendo un ritmo de una unidad por semana si estudias a tiempo parcial, o una unidad cada dos o tres días si le dedicas tiempo completo. No intentes devorar tres unidades en una noche: la programación se aprende con la práctica espaciada, no con maratones.

Cada unidad sigue la misma estructura, y conviene que la aproveches en orden. Primero lee la introducción de la unidad para entender el mapa: qué vas a ver y por qué importa. Después estudia la explicación detallada sin prisa, prestando atención a las analogías y a los errores comunes: esos errores están ahí porque alguien los cometió antes que tú, probablemente cientos de personas, y conocerlos de antemano te ahorra horas de frustración.

Luego viene la parte más importante: los ejemplos paso a paso. No los leas pasivamente. Abre Android Studio y escribe el código tú mismo, línea por línea, mientras lees la explicación de

cada línea. Escribir el código con tus propias manos activa un tipo de aprendizaje que la simple lectura no logra. Vas a cometer errores de dedo, vas a ver mensajes de error, y vas a aprender a corregirlos: eso es exactamente lo que le pasa a un programador profesional todos los días.

Después de los ejemplos, resuelve los ejercicios de la sección de práctica. Intenta cada uno por tu cuenta antes de mirar la solución. Si te atorras más de veinte minutos, mira la solución, entiéndela, ciérrala y vuelve a resolver el ejercicio sin mirar. La meta no es copiar la respuesta correcta: es entrenar tu cerebro para llegar solo a la respuesta correcta.

Al final de cada unidad, el resumen te sirve como repaso rápido. Si algún punto del resumen no te queda claro, regresa a la sección correspondiente antes de avanzar. Las unidades se encadenan: lo que no entiendas en la unidad 2 te va a perseguir en la unidad 5.

Un consejo sobre el ritmo: es normal sentir que avanzas lento. Programar es una habilidad motora e intelectual a la vez, como aprender a tocar un instrumento. Nadie se frustra porque no puede tocar una sinfonía en la primera semana de piano; date el mismo permiso con el código. La constancia diaria de una hora vale más que una sesión heroica de ocho horas una vez al mes.

Otro consejo: guarda todo lo que hagas. Crea una carpeta en tu computadora para este curso y guarda cada proyecto, cada ejercicio, cada experimento. Dentro de unos meses vas a querer volver a ver tu primera app y te vas a reír de lo simple que era; esa risa es la medida de tu progreso.

Requisitos

Vamos a lo práctico. Para trabajar con este libro necesitas tres cosas: una computadora adecuada, el software correcto y una base mínima de conocimientos.

La computadora: necesitas una PC o laptop con al menos 8 GB de memoria RAM, aunque 16 GB hacen la experiencia mucho más cómoda. Android Studio es un programa pesado: abre el editor, un emulador de teléfono virtual y varias herramientas al mismo tiempo, y todo eso consume memoria. Con 8 GB funciona, pero vas a notar que todo va un poco lento; con 16 GB el trabajo fluye. En cuanto al procesador, cualquier procesador moderno de los últimos años sirve, de Intel o AMD. Necesitas al menos 8 GB de espacio libre en el disco duro, de preferencia en un disco de estado sólido (SSD), porque el emulador y las herramientas cargan mucho más rápido desde un SSD. El sistema operativo puede ser Windows 10 u 11 de 64 bits, macOS reciente o una distribución de Linux de 64 bits como Ubuntu. Este libro usa ejemplos y capturas pensados para funcionar igual en los tres sistemas, y cuando haya una diferencia importante entre sistemas, se señala en el texto.

El software: la herramienta principal es Android Studio, el entorno de desarrollo oficial de Google para Android. Es gratuito y lo descargarás desde el sitio oficial de desarrolladores de Android. Dentro de Android Studio viene incluido el SDK de Android (el kit de herramientas del sistema), el emulador para probar tus apps sin necesidad de un teléfono físico, y todo lo necesario para programar en Kotlin. En la unidad 1 te guío paso a paso en la instalación, así que no necesitas instalar nada antes de empezar a leer. Además de Android Studio, te conviene tener una cuenta de Google (la misma que usas para Gmail te sirve) porque la necesitarás más adelante para publicar apps y usar algunos servicios. Y aunque no es obligatorio, tener un teléfono o tableta Android físico para probar tus apps es una gran ventaja: nada se compara con ver tu creación corriendo en un dispositivo real, en tu mano.

Los conocimientos: como dije al principio, no necesitas saber programar. Pero sí necesitas tres habilidades básicas. Primera: manejar tu computadora con soltura, incluyendo instalar programas y administrar archivos. Segunda: leer y escribir en inglés a nivel básico, porque los lenguajes de programación, los mensajes de error y la documentación oficial están en inglés; no necesitas hablarlo con fluidez, pero sí entender palabras técnicas comunes. Tercera: pensamiento lógico básico, es decir, la capacidad de seguir una secuencia de instrucciones y de razonar sobre causas y efectos ("si hago esto, pasa aquello"). Si alguna vez armaste un mueble siguiendo un instructivo, ya tienes esa habilidad.

Si cumples con eso, estás listo. No necesitas saber matemáticas avanzadas, no necesitas ser "bueno para las computadoras" en ningún sentido misterioso, y definitivamente no necesitas ningún talento especial. La programación es una habilidad que se entrena, no un don con el que se nace. Miles de personas con tu mismo punto de partida han recorrido este camino antes que tú, y muchas de ellas hoy viven de esto.

Una última cosa antes de empezar. Vas a cometer errores. Muchos. Vas a ver la pantalla llena de texto rojo, tu app se va a cerrar sola sin explicación aparente, y va a haber días en que

sientas que no entiendes nada. Eso no significa que no sirvas para esto: significa que estás aprendiendo de verdad. Cada programador profesional, sin excepción, pasa por eso todos los días de su vida laboral. La diferencia entre quien aprende y quien abandona no es la inteligencia: es la tolerancia a la frustración y la costumbre de buscar la solución en lugar de rendirse. Este libro está diseñado para acompañarte en esos momentos: los errores más comunes están documentados, las soluciones están explicadas, y el camino está trazado.

Así que respira hondo, abre tu computadora y empecemos. Tu primera app está a una unidad de distancia.

Bienvenido a bordo. Vamos a construir.

UNIDAD 1: Introducción a Android y Android Studio

Introducción

Esta primera unidad es la puerta de entrada a todo lo demás. Aquí no vas a escribir programas complicados ni a memorizar sintaxis rara: vas a entender el terreno donde vas a trabajar durante todo el libro. Piensa en esto como cuando alguien quiere aprender a cocinar: antes de encender la estufa, necesita conocer la cocina, saber dónde están los utensilios, qué hace cada aparato y cuáles son las reglas básicas de seguridad. Esta unidad es tu recorrido por la cocina.

Vamos a ver tres clases. En la clase 1 vas a descubrir qué es Android en realidad: no solo "el sistema de mi teléfono", sino una plataforma completa con una historia fascinante, una arquitectura por capas y un ecosistema de miles de millones de dispositivos. Entender esto te da contexto profesional: cuando alguien te pregunte en una entrevista de trabajo por qué elegiste Android, vas a tener una respuesta con fundamento, no solo "porque me gusta mi celular".

En la clase 2 vas a instalar Android Studio, el taller donde vas a construir todas tus aplicaciones. Instalarlo bien, desde la primera vez, te evita la mitad de los dolores de cabeza que sufren los principiantes. Te voy a guiar clic por clic, te voy a explicar qué es el SDK (el kit de desarrollo) y qué componentes necesitas, y te voy a advertir sobre los errores de instalación más comunes para que no los cometas.

En la clase 3 vas a crear y ejecutar tu primera aplicación: el clásico "Hola Mundo". Parece un programa insignificante, un texto en la pantalla que dice "Hola Mundo", pero detrás de ese texto hay un logro enorme: significa que tu entorno funciona, que sabes crear un proyecto, que entiendes la estructura básica de una app y que lograste instalarla en un dispositivo y verla correr. Todo programador profesional recuerda su primer "Hola Mundo" porque marca el momento en que dejó de ser espectador y se convirtió en creador.

Por qué importa esta unidad

Es tentador saltarse la teoría e ir directo al código. Muchos principiantes lo hacen, y casi todos se arrepienten. La razón es simple: sin contexto, el código parece magia arbitraria. ¿Por qué el proyecto tiene tantas carpetas? ¿Qué es ese archivo raro llamado `AndroidManifest.xml`? ¿Por

qué hay que pedir permisos para usar la cámara? Si no entiendes la plataforma, cada una de estas cosas se convierte en un obstáculo misterioso. Con el contexto de esta unidad, esas mismas cosas se vuelven lógicas y hasta obvias.

Además, esta unidad te enseña el ciclo de trabajo que vas a repetir cientos de veces: crear proyecto, escribir código, compilar, probar, corregir. Dominar ese ciclo desde el día uno hace que todo lo demás fluya. Es como aprender a afinar la guitarra antes de aprender canciones: aburrido por diez minutos, útil para siempre.

Qué sabrás al terminar

Cuando termines esta unidad vas a poder explicar con tus propias palabras qué es Android, cómo está organizado por dentro y qué tipos de dispositivos lo usan. Vas a tener Android Studio instalado y funcionando en tu computadora, con el SDK configurado y un emulador listo para probar tus apps. Vas a saber crear un proyecto nuevo desde cero, identificar cada parte importante de su estructura de carpetas y archivos, y ejecutar la app para verla funcionando en pantalla. También vas a conocer los errores más comunes de esta etapa y cómo resolverlos sin entrar en pánico. En resumen: vas a tener el taller montado y tu primera pieza construida. Todo lo que sigue en el libro se construye sobre esta base, así que tómate tu tiempo y asegúrate de que cada pieza quede bien colocada.

Explicación detallada

Clase 1: Qué es Android, su historia y su ecosistema

Empecemos con la pregunta más básica y la más importante: ¿qué es Android? La respuesta corta es que Android es un sistema operativo para dispositivos móviles, creado por Google, basado en el núcleo de Linux. La respuesta larga, la que te convierte en profesional, requiere entender cada parte de esa definición.

Un sistema operativo es el programa maestro que controla todo lo demás en un dispositivo. Cuando tocas la pantalla de tu teléfono, abres una app, te conectas al wifi o tomas una foto, hay un sistema operativo coordinando el hardware (la pantalla, el procesador, la memoria, la cámara) con el software (las aplicaciones). Sin sistema operativo, tu teléfono sería un pedazo caro de vidrio y metal sin nada que hacer. Android es ese coordinador en miles de millones de dispositivos.

Android está basado en Linux, que es un sistema operativo famoso por ser libre, estable y seguro, usado en servidores de todo el mundo. Esto no es un dato curioso sin importancia:

significa que Android heredó de Linux cosas como la gestión de memoria, la seguridad por permisos y la capacidad de correr en muchos tipos de hardware. Cuando más adelante pidas permisos en tu app o manejes archivos, estarás tocando conceptos que vienen directamente de esa herencia.

Ahora, la historia, porque entender de dónde viene Android explica muchas de sus decisiones de diseño. Android no nació en Google: nació en 2003 como una pequeña empresa emergente llamada Android Inc., fundada por Andy Rubin y su equipo, con la idea original de crear un sistema operativo para cámaras digitales. Sí, cámaras, no teléfonos. Cuando se dieron cuenta de que el mercado de las cámaras no era tan prometedor, giraron hacia los teléfonos móviles. En 2005, Google compró Android Inc., viendo el potencial de tener presencia en el mundo móvil. En 2007, Google anunció la Open Handset Alliance, una alianza de empresas de tecnología comprometidas con un estándar abierto para móviles, y en 2008 salió al mercado el primer teléfono con Android: el HTC Dream, también conocido como T-Mobile G1. Tenía teclado físico deslizante, una pantallita pequeña para los estándares de hoy, y nadie imaginaba que ese aparato torpe era el inicio de una revolución.

Desde entonces, cada versión importante de Android llevó nombre de postre en orden alfabético durante años: Cupcake, Donut, Eclair, Froyo, Gingerbread, Honeycomb, Ice Cream Sandwich, Jelly Bean, KitKat, Lollipop, Marshmallow, Nougat, Oreo, Pie... Los desarrolladores veteranos todavía hablan de esas versiones con nostalgia. A partir de Android 10, Google cambió a números simples (Android 10, 11, 12, 13, 14...), pero los postres quedaron en la cultura de la comunidad.

¿Por qué te cuento todo esto? Porque la historia explica el presente. Android es de código abierto (el proyecto se llama AOSP, Android Open Source Project), lo que significa que cualquier fabricante puede usarlo y adaptarlo. Esa apertura es la razón por la que existen miles de modelos de teléfonos Android de cientos de marcas y precios: Samsung, Xiaomi, Motorola, Huawei, Google Pixel y muchísimas más. También es la razón por la que Android domina el mercado mundial con más del 70 por ciento de los teléfonos. Y también es la razón de uno de sus desafíos: la fragmentación. Como hay tantos dispositivos con distintas versiones de Android, tamaños de pantalla y capacidades, los desarrolladores debemos crear apps que funcionen bien en esa variedad. No te preocupes: las herramientas modernas de Android manejan gran parte de ese problema por ti, y en este libro aprenderás las buenas prácticas para que tus apps se vean bien en cualquier teléfono.

Hablemos de la arquitectura, es decir, cómo está organizado Android por dentro. Imagina un pastel de capas, donde cada capa se apoya en la de abajo y ofrece servicios a la de arriba. En la base está el núcleo Linux, que habla directamente con el hardware. Encima hay una capa de bibliotecas nativas y el Android Runtime (ART), que es la máquina que ejecuta tus

aplicaciones. Luego viene el framework de aplicaciones, que es un conjunto de herramientas y servicios que Google te ofrece para que no tengas que reinventar la rueda: gestión de pantallas, notificaciones, acceso a sensores, y mucho más. En la cima están las aplicaciones mismas: las que vienen instaladas y las que tú vas a crear. Esta organización por capas es una gran noticia para ti como desarrollador: significa que no necesitas entender el hardware a fondo para crear una app, porque el framework te da todo resuelto en un nivel cómodo de abstracción.

El ecosistema Android va mucho más allá de los teléfonos. La misma plataforma corre en tabletas, relojes inteligentes (Wear OS), televisiones (Android TV y Google TV), automóviles (Android Auto y Android Automotive) e incluso en dispositivos del internet de las cosas. Aprender desarrollo Android te abre la puerta a todos esos mundos con la misma base de conocimientos. Además, el ecosistema incluye Google Play, la tienda donde publicarás tus apps y donde millones de usuarios pueden descubrirlas, y un conjunto enorme de servicios de Google (mapas, autenticación, nube, publicidad) que puedes integrar en tus aplicaciones.

Un concepto clave del ecosistema es la cuenta de desarrollador de Google Play. Publicar una app no es gratis ni instantáneo: necesitas registrarte como desarrollador, pagar una cuota única (alrededor de 25 dólares, aunque el precio puede variar) y pasar por un proceso de revisión. No necesitas hacer esto ahora; lo veremos con detalle en la parte del libro dedicada a la publicación. Lo menciono para que tengas el mapa completo desde el inicio.

Finalmente, hablemos del lenguaje. Históricamente, Android se programaba en Java. Hoy, el lenguaje oficial y recomendado por Google es Kotlin: más moderno, más conciso, más seguro y mucho más agradable de escribir. Todo este libro usa Kotlin. Si alguna vez ves tutoriales viejos en Java, no te confundas: los conceptos son los mismos, pero nosotros trabajamos con la herramienta actual. Google ha declarado que el desarrollo de Android es "Kotlin primero", lo que significa que las nuevas herramientas, ejemplos y documentación priorizan Kotlin. Estás aprendiendo el estándar vigente, no una moda pasajera.

Analogía para recordar esta clase: piensa en Android como una ciudad enorme. El núcleo Linux son los cimientos y las tuberías subterráneas que nadie ve pero que todo lo sostienen. El Android Runtime son las calles por donde circulan los autos. El framework son los servicios públicos: agua, luz, policía, que los ciudadanos (las apps) usan sin tener que construirlos. Y Google Play es el centro comercial donde los ciudadanos venden sus productos. Tú estás por convertirte en constructor en esta ciudad: necesitas conocer sus reglas, sus servicios y sus calles antes de levantar tu primer edificio.

Clase 2: Instalación de Android Studio y el SDK

Android Studio es el entorno de desarrollo integrado (IDE) oficial para crear aplicaciones Android. La palabra "entorno" aquí significa que no es un simple editor de texto: es un taller completo que incluye el editor de código con ayuda inteligente, el compilador que convierte tu código en una app instalable, el depurador para encontrar errores, el emulador para probar sin teléfono físico, y las herramientas para publicar en Google Play. Todo en un solo programa, todo diseñado para trabajar en conjunto. Es gratuito, lo mantiene Google junto con JetBrains (la empresa creadora de Kotlin), y es lo que usan los profesionales en empresas de todo el mundo. Aprender en la herramienta real, desde el día uno, te pone en el camino profesional correcto.

Antes de instalar, verifica los requisitos de tu computadora, porque instalar Android Studio en una máquina muy limitada es una receta para la frustración. Necesitas un sistema operativo de 64 bits: Windows 10 u 11, macOS 12 o posterior, o Linux de 64 bits como Ubuntu 20.04 o posterior. En memoria RAM, el mínimo oficial es 8 GB, pero te recomiendo ampliamente 16 GB si puedes: con 8 GB el emulador va lento y el sistema se arrastra cuando tienes el navegador abierto al mismo tiempo. El procesador puede ser cualquiera razonablemente moderno, Intel o AMD, aunque en procesadores muy viejos la instalación puede tardar mucho. Necesitas al menos 8 GB de espacio libre en disco, y de preferencia un disco SSD, porque Android Studio lee y escribe miles de archivos pequeños y en un disco mecánico viejo todo se vuelve eterno. Si tu computadora cumple con esto, adelante. Si está justo en el límite, considera usar un teléfono físico en lugar del emulador para las pruebas: ahorra mucha memoria.

Vamos con la instalación paso a paso. Estos pasos son para la versión actual (la familia Hedgehog e Iguana, y las que vengan después siguen la misma lógica).

Paso 1: descarga Android Studio desde el sitio oficial. Abre tu navegador y busca "Android Studio download" o ve directo al sitio de desarrolladores de Android (developer.android.com/studio). Descarga siempre desde el sitio oficial, nunca desde sitios de terceros: las versiones de otros sitios pueden estar desactualizadas o, peor, modificadas. El archivo pesa alrededor de 1 GB, así que la descarga puede tardar varios minutos dependiendo de tu conexión. Ten paciencia.

Paso 2: ejecuta el instalador. En Windows, abre el archivo .exe descargado y sigue el asistente: acepta los términos, deja las opciones por defecto (que incluyen instalar el SDK de Android, muy importante) y elige la carpeta de instalación si quieres cambiarla. En macOS, abre el archivo .dmg y arrastra Android Studio a la carpeta Aplicaciones, como con cualquier programa de Mac. En Linux, descomprime el archivo .tar.gz en la carpeta que prefieras (por

ejemplo, en tu carpeta personal o en /opt si tienes permisos) y ejecuta el script studio.sh que está dentro de la carpeta bin. En todos los sistemas, el instalador te va a preguntar si quieres importar configuraciones de una versión anterior: si es tu primera vez, elige no importar nada.

Paso 3: el asistente de configuración inicial. La primera vez que abres Android Studio, aparece un asistente (setup wizard) que te guía por la configuración. Elige la instalación "Standard" (estándar) a menos que sepas exactamente qué componentes personalizados necesitas: la opción estándar instala lo esencial, que es justo lo que queremos. El asistente va a descargar varios componentes: el SDK de Android, las herramientas de plataforma, el emulador y una imagen del sistema para el emulador. Esta descarga puede pesar varios gigabytes y tardar un buen rato. Es normal. Aprovecha para tomar un café.

Hablemos del SDK, porque es un concepto que vas a ver mencionado constantemente. SDK significa "Software Development Kit", kit de desarrollo de software. En el mundo Android, el SDK es el conjunto de herramientas, bibliotecas y documentación que te permite crear apps: incluye las APIs (las funciones del sistema que tu app puede usar), las herramientas de compilación (que convierten tu código en una app), las herramientas de plataforma (como adb, que comunica tu computadora con el teléfono) y las imágenes del sistema para el emulador. Cuando instalas Android Studio con la opción estándar, el SDK se instala automáticamente en una carpeta de tu sistema. Cada versión de Android tiene su nivel de API (por ejemplo, Android 14 es API 34): con el SDK Manager, que está dentro de Android Studio, puedes instalar los niveles de API que necesites. Para este libro, con tener instalado uno de los niveles recientes basta; Android Studio suele encargarse de esto solo.

El SDK Manager merece que lo conozcas desde ahora. Lo encuentras en Android Studio en el menú de configuración (en Windows y Linux está en File > Settings > Appearance and Behavior > System Settings > Android SDK; en macOS está en Android Studio > Settings con la misma ruta). Tiene tres pestañas: SDK Platforms (las versiones de Android para las que puedes desarrollar), SDK Tools (las herramientas como el emulador, las build tools y las platform tools) y SDK Update Sites (de dónde se descargan las actualizaciones). No necesitas tocar mucho aquí al principio, pero sí debes saber que existe, porque cuando algo falle más adelante ("me falta tal componente"), la solución casi siempre está en esta pantalla.

Otro componente vital es el emulador, oficialmente llamado Android Virtual Device (AVD). Es un teléfono virtual que corre dentro de tu computadora: puedes instalar tus apps en él, tocar la pantalla con el mouse, simular llamadas, mensajes, ubicación GPS y hasta simular que el teléfono se queda sin batería. Para crearlo, en Android Studio abres el Device Manager (el icono de teléfono en la barra lateral) y sigues el asistente: eliges un modelo de dispositivo (por ejemplo, un Pixel), eliges la versión de Android (la imagen del sistema que se descargó en la instalación), y le pones un nombre. La primera vez que arranca el emulador tarda un par de

minutos; las siguientes veces es más rápido. Si tu computadora tiene poca memoria, el emulador puede ir lento: en ese caso, prueba con una imagen del sistema más ligera o usa un teléfono físico conectándolo por USB con la depuración USB activada.

Ahora, los errores comunes de instalación, porque más vale prevenir. Error 1: "no tengo suficiente espacio en disco". El instalador necesita espacio no solo para descargarse sino para descomprimirse e instalarse; asegúrate de tener al menos 10 GB libres antes de empezar. Error 2: en Windows, el antivirus bloquea partes de la instalación. Si el instalador se congela o marca errores extraños, pausa temporalmente tu antivirus durante la instalación (y vuelve a activarlo después). Error 3: "HAXM no se pudo instalar" o problemas con la aceleración del emulador. En procesadores Intel, el emulador usa una tecnología de aceleración; en Windows necesitas tener activada la virtualización en la BIOS (busca opciones como "Intel VT-x" o "AMD-V" y actívalas). Si no sabes entrar a la BIOS, busca el modelo de tu computadora más "activar virtualización en BIOS" en internet: hay guías para cada marca. Error 4: descargar Android Studio de un sitio no oficial y obtener una versión vieja o corrupta. Repito: solo desde el sitio oficial. Error 5: en Linux, faltan bibliotecas del sistema de 32 bits o permisos: si algo falla al arrancar, revisa los requisitos listados en la página oficial para tu distribución.

Buenas prácticas desde la instalación: primero, mantén Android Studio actualizado, pero no actualices en medio de un proyecto importante: las actualizaciones a veces cambian comportamientos y es mejor hacerlas entre proyectos. Segundo, no instales todos los niveles de API "por si acaso": cada uno pesa gigabytes; instala el que uses y agrega más solo si un proyecto lo pide. Tercero, crea tu primer emulador con una versión de Android reciente y un dispositivo común (un Pixel): es la configuración que menos problemas da. Cuarto, si vas a usar un teléfono físico, activa las opciones de desarrollador en el teléfono (tocando siete veces el número de compilación en Ajustes > Acerca del teléfono) y luego activa la depuración USB: es el ritual de todo desarrollador Android y lo harás tantas veces que te lo aprenderás de memoria.

Analogía para esta clase: instalar Android Studio es como montar tu taller de carpintería. El IDE es el taller mismo, con sus mesas de trabajo. El SDK son tus herramientas: serruchos, martillos, taladros, cada uno para una tarea. El emulador es tu banco de pruebas, donde ensamblas el mueble antes de entregarlo. Y el SDK Manager es tu caja de herramientas: ahí guardas, ordenas y actualizas cada herramienta. Un carpintero con el taller desordenado pierde tiempo buscando el martillo; un desarrollador con el entorno mal instalado pierde tiempo peleando con errores que no son de su código. Monta bien tu taller una vez, y te sirve para todo el libro.

Clase 3: Crear y ejecutar tu primera app "Hola Mundo"

Llegó el momento. Con el taller montado, vamos a construir la primera pieza. En el mundo de la programación existe una tradición de más de cincuenta años: el primer programa que escribe alguien en un lenguaje o plataforma nueva muestra en pantalla las palabras "Hola Mundo" (o "Hello, World!" en inglés). No es una regla oficial, es un ritual: significa "ya estoy aquí, ya puedo crear cosas en este lugar". Tu primera app Android va a seguir esa tradición.

Vamos a crear el proyecto paso a paso. Abre Android Studio. En la pantalla de bienvenida verás un botón que dice "New Project" (proyecto nuevo). Haz clic ahí. Aparece una ventana con plantillas: son puntos de partida prefabricados para distintos tipos de app. Elige la plantilla "Empty Views Activity" (actividad vacía): es la más simple y la ideal para aprender, porque no trae código extra que te distraiga. Haz clic en "Next".

Ahora viene la pantalla de configuración del proyecto, y cada campo importa, así que vamos uno por uno. "Name" es el nombre de tu app, el que verán los usuarios: escribe "Hola Mundo". "Package name" es el identificador único de tu app en el mundo: algo como com.ejemplo.holamundo. Piensa en el package name como la CURP de tu app: no puede haber dos apps con el mismo package name en Google Play. Usa siempre minúsculas, sin espacios ni caracteres especiales, con el formato de dominio invertido (primero com, luego tu nombre o empresa, luego el nombre de la app). "Save location" es la carpeta de tu computadora donde se guardará el proyecto: elige o crea una carpeta dedicada a este curso, por ejemplo una carpeta llamada "ProyectosAndroid" en tus documentos. "Language" debe decir Kotlin: verifica que así sea, porque es el lenguaje de este libro. "Minimum SDK" es la versión mínima de Android en la que tu app podrá instalarse: Android Studio sugiere un valor razonable (como API 24, Android 7.0); déjalo como está por ahora. Este campo es una decisión de negocio en la vida real (mientras más bajo el mínimo, más teléfonos compatibles, pero menos funciones modernas disponibles), pero para aprender, el valor sugerido está perfecto. "Build configuration language" déjalo en Kotlin DSL, que es lo moderno. Haz clic en "Finish".

Android Studio va a trabajar unos momentos: crea las carpetas, genera los archivos base, descarga dependencias y sincroniza el proyecto con Gradle (el sistema de construcción que compila tu app). La primera sincronización puede tardar varios minutos porque descarga cosas de internet. Cuando termine, verás el proyecto abierto en el editor. Felicidades: ya tienes una app, aunque todavía no la hayas visto correr.

Antes de ejecutarla, exploremos la estructura del proyecto, porque entenderla ahora te ahorra confusión después. En el panel izquierdo está la vista del proyecto (en modo "Android", que es el que nos interesa). Las carpetas importantes son: "app > manifests", que contiene el archivo

AndroidManifest.xml, la "carta de presentación" de tu app ante el sistema: ahí se declara el nombre de la app, sus permisos, sus pantallas y cuál es la pantalla principal. "app > java > com.ejemplo.holamundo", que contiene tu código Kotlin: ahí está el archivo MainActivity.kt, la pantalla principal de tu app. "app > res", que contiene los recursos: imágenes, textos, colores y diseños de pantalla. Dentro de res está la carpeta "layout", con el archivo activity_main.xml, que describe cómo se ve la pantalla (qué texto, qué botones, dónde va cada cosa). Y "Gradle Scripts", con los archivos de configuración de construcción, como build.gradle.kts: ahí se declaran las dependencias (bibliotecas externas que tu app usa) y la configuración de compilación. No necesitas memorizar todo esto hoy; solo ubica dónde vive cada cosa.

Una distinción fundamental que debes entender desde ahora: en Android, la apariencia (lo que se ve) y la lógica (lo que hace) viven en archivos separados. El archivo activity_main.xml describe la pantalla con etiquetas, como un plano arquitectónico. El archivo MainActivity.kt contiene el código Kotlin que le da vida a esa pantalla, como el electricista que conecta la instalación del plano. Esta separación es una buena práctica profesional: permite que un diseñador trabaje en la apariencia mientras el programador trabaja en la lógica, sin estorbarse.

Ahora sí, vamos a ejecutar la app. Tienes dos opciones: el emulador o un teléfono físico. Para el emulador: abre el Device Manager, crea un dispositivo virtual si aún no lo hiciste (elige un Pixel con una versión reciente de Android), y luego en la barra superior de Android Studio selecciona ese dispositivo en el menú desplegable y presiona el botón verde de "Run" (el triángulo de reproducción). Android Studio va a compilar tu app (convertir el código en un paquete instalable), la va a instalar en el emulador y la va a abrir automáticamente. La primera compilación tarda un poco; las siguientes son más rápidas. Para un teléfono físico: conéctalo por USB, acepta la autorización de depuración USB que aparece en el teléfono, selecciónalo en el menú de dispositivos y presiona Run igual que con el emulador.

Cuando la app arranque, verás una pantalla con el texto "Hello World!". Ese texto viene de la plantilla: en el archivo activity_main.xml hay un TextView (un elemento de texto) con ese contenido. Vamos a personalizarlo para hacerlo tuyo, porque modificar algo y ver el cambio es la mejor forma de aprender. Abre app > res > layout > activity_main.xml. Verás algo como esto (la versión exacta puede variar un poco según la versión de Android Studio, pero la idea es la misma):

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

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

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

Cambia el valor de `android:text` de "Hello World!" a "Hola Mundo desde mi primera app". Vuelve a presionar Run. Verás tu texto en la pantalla. Acabas de modificar una app real y ver el resultado: ese ciclo (cambiar, ejecutar, observar) es el latido del desarrollo de software.

Y el archivo Kotlin, `MainActivity.kt`, se ve así:

```

package com.ejemplo.holamundo

import android.os.Bundle
import androidx.activity.enableEdgeToEdge
import androidx.appcompat.app.AppCompatActivity
import androidx.core.view.ViewCompat
import androidx.core.view.WindowInsetsCompat

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v,
insets ->
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
            v.setPadding(systemBars.left, systemBars.top, systemBars.right,
systemBars.bottom)
            insets
        }
    }
}

```

No te preocupes si no entiendes cada línea todavía: en las próximas unidades desmenuzaremos Kotlin pieza por pieza. Por ahora, quédate con lo esencial: la línea `setContentView(R.layout.activity_main)` es la que conecta este código con el archivo de diseño `activity_main.xml`. Es el puente entre el plano (XML) y la electricidad (Kotlin). Todo lo demás son detalles de configuración de la ventana que la plantilla incluye por ti.

Errores comunes en esta etapa, para que no te asusten. Error 1: "Gradle sync failed" o la sincronización se queda atorada. Casi siempre es un problema de conexión a internet (Gradle descarga dependencias la primera vez) o de que se interrumpió la descarga. Solución: verifica tu conexión y usa el menú File > Sync Project with Gradle Files para reintentar. Error 2: el emulador no arranca o va lentísimo. Revisa la aceleración de virtualización (clase 2) o prueba con un teléfono físico. Error 3: "no se detecta el dispositivo" con teléfono físico. Revisa que la depuración USB esté activada, que el cable transmita datos (algunos cables solo cargan) y que hayas aceptado el diálogo de autorización en el teléfono. Error 4: cambiaste el texto pero la app muestra lo viejo. Asegúrate de haber guardado el archivo (Ctrl+S o Cmd+S) y de haber presionado Run de nuevo: los cambios no se aplican solos. Error 5: pánico ante el texto rojo en la parte inferior (la ventana de "Build"). El texto rojo solo significa que la compilación encontró algo que no le gustó; lee el mensaje con calma, casi siempre dice exactamente qué archivo y qué línea tienen el problema.

Buenas prácticas que empiezan hoy: ponle nombres claros a tus proyectos y guárdalos siempre en la misma carpeta del curso. Acostúmbrate a leer los mensajes de error completos en lugar de cerrarlos con frustración: el mensaje de error es tu mejor maestro en esta etapa. Y ejecuta tu app con frecuencia, después de cada cambio pequeño, en lugar de escribir mucho código y probarlo todo al final: los cambios pequeños son fáciles de revisar si algo sale mal.

Analogía de cierre: crear tu primera app es como hornear tu primer pan siguiendo una receta. La plantilla es la receta probada: ya trae las cantidades correctas. Tú solo seguiste los pasos, metiste el pan al horno (presionaste Run) y salió algo comestible. Todavía no inventas tus propias recetas, pero ya sabes encender el horno, ya conoces la cocina y ya probaste el sabor del éxito. Las próximas unidades te enseñan los ingredientes uno por uno, hasta que puedas crear tus propios platillos.

Ejemplos paso a paso

En esta sección vamos a trabajar con código real, línea por línea. Abre tu proyecto "Hola Mundo" en Android Studio y sigue cada ejemplo. La meta no es memorizar, sino entender qué hace cada pieza.

Ejemplo 1: El layout XML de la primera app, explicado línea por línea

Abre el archivo app > res > layout > activity_main.xml. Vamos a desarmarlo como un mecánico desarma un motor: pieza por pieza.

```
<?xml version="1.0" encoding="utf-8"?>
```

Línea 1: es la declaración del documento XML. Le dice al sistema "este archivo está escrito en XML versión 1.0 y usa codificación UTF-8 (que incluye acentos y eñes)". Todos los archivos XML la llevan al inicio. No la modifiques.

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

Líneas 2 a 9: el contenedor principal de la pantalla. `ConstraintLayout` es un tipo de contenedor que permite posicionar los elementos con "restricciones" (constraints), es decir, reglas como "este texto va centrado" o "este botón va pegado abajo". Las líneas que empiezan con `xmlns` definen los prefijos de atributos: `android:` para atributos del sistema, `app:` para atributos de bibliotecas externas, `tools:` para atributos que solo sirven en el editor. `android:id="@+id/main"` le pone un nombre identificador a este contenedor ("main"), para poder encontrarlo desde el código Kotlin si hace falta. `android:layout_width="match_parent"` y `android:layout_height="match_parent"` significan que el contenedor ocupa todo el ancho y todo el alto de la pantalla (`match_parent` = "igual que el papá"). `tools:context=".MainActivity"` le dice al editor que este diseño pertenece a la pantalla `MainActivity`, para mostrarte una vista previa correcta.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hola Mundo desde mi primera app"
    android:textSize="24sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Líneas 10 a 18: el texto visible. `TextView` es el elemento que muestra texto en pantalla. `android:layout_width="wrap_content"` y `android:layout_height="wrap_content"` significan que el cuadro del texto se ajusta justo al tamaño del contenido (`wrap_content` = "envuelve el

contenido"): ni más grande ni más chico. `android:text` es el texto que se muestra: aquí va tu mensaje. `android:textSize="24sp"` define el tamaño de la letra: 24sp es un tamaño grande y legible (sp es la unidad de tamaño de texto en Android, que respeta la configuración de accesibilidad del usuario). Las cuatro líneas `app:layout_constraint...` son las restricciones que centran el texto: le dicen "tu borde de abajo va pegado al borde de abajo del papá, tu borde derecho al borde derecho del papá", y así con los cuatro lados. Cuando un elemento está restringido a los cuatro lados del contenedor, queda perfectamente centrado. Esa es la magia de `ConstraintLayout`.

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

Línea 19: cierra el contenedor. En XML, toda etiqueta que se abre se cierra: la diagonal al inicio (`</...>`) indica el cierre.

Haz este experimento: cambia `android:textSize` a "48sp", ejecuta la app y observa. Luego cambia una restricción, por ejemplo borra `app:layout_constraintTop_toTopOf="parent"`, y observa cómo el texto se mueve. Jugar así, cambiando una cosa a la vez y observando, es la forma más rápida de entender los layouts.

Ejemplo 2: Agregar un segundo texto debajo del primero

Vamos a agregar otro `TextView` que muestre tu nombre, colocado debajo del texto principal. Dentro del `ConstraintLayout`, después del primer `TextView`, agrega:

```
<TextView
    android:id="@+id/txtNombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Mi nombre es Alex"
    android:textSize="18sp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/txtSaludo" />
```

Explicación línea por línea. `android:id="@+id/txtNombre"` le da un identificador único a este nuevo texto ("txtNombre"): el símbolo `@+id/` significa "crea un identificador nuevo con este nombre". Los otros atributos ya los conoces: tamaño ajustado al contenido, tu texto, tamaño de letra 18sp (más chico que el título, jerarquía visual: lo importante va más grande). Las restricciones: izquierda y derecha pegadas a los bordes del papá (centrado horizontal), y arriba pegado a la parte de abajo de otro elemento (`@id/txtSaludo`). Para que esto funcione, el primer

TextView necesita tener el identificador txtSaludo: agrégale la línea `android:id="@+id/txtSaludo"` a su etiqueta. Así se encadenan elementos en ConstraintLayout: uno se posiciona respecto al otro.

Este ejemplo te enseña tres ideas profesionales: los identificadores permiten que los elementos se referencien entre sí, la jerarquía visual (títulos grandes, detalles chicos) guía el ojo del usuario, y los layouts se construyen por relaciones ("debajo de", "al lado de") en lugar de coordenadas fijas, lo que hace que se adapten a cualquier tamaño de pantalla.

Ejemplo 3: El código Kotlin mínimo, explicado

Ahora abre `app > java > com.ejemplo.holamundo > MainActivity.kt`. Vamos a entender cada línea del archivo que la plantilla generó:

```
package com.ejemplo.holamundo
```

Línea 1: declara a qué paquete pertenece este archivo. El paquete es como la dirección de la casa del archivo dentro del proyecto: coincide con el package name que elegiste al crear el proyecto. Todos los archivos Kotlin de tu app viven en algún paquete.

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

Líneas 2 a 4: importaciones. Le dicen a Kotlin "voy a usar estas clases que viven en otros paquetes, así que tráelas". Es como decirle al cocinero qué ingredientes saque de la alacena antes de empezar. Bundle, AppCompatActivity y las demás son clases del framework de Android.

```
class MainActivity : AppCompatActivity() {
```

Línea 5: declara la clase MainActivity, que representa tu pantalla. Los dos puntos (:) significan "hereda de": MainActivity hereda de AppCompatActivity, que es la clase base que Android ofrece para las pantallas. Heredar significa recibir gratis todo el comportamiento estándar de una pantalla (como saber mostrarse, pausarse y cerrarse) y solo agregar lo tuyo.

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


Líneas 6 y 7: la función `onCreate` es lo primero que se ejecuta cuando la pantalla nace. `override` significa "estoy reescribiendo un comportamiento que heredé": `AppCompatActivity` ya tiene un `onCreate`, pero nosotros lo personalizamos. La primera línea dentro siempre es `super.onCreate(...)`, que ejecuta el comportamiento original del papá antes de lo nuestro: es una regla de oro, nunca la omitas.

```
setContentView(R.layout.activity_main)
```

Línea 8: la línea más importante para ti hoy. Conecta este código con el diseño XML: le dice a la pantalla "tu apariencia está descrita en el archivo `activity_main`". `R.layout.activity_main` es la forma en que Kotlin referencia recursos: `R` es la clase de recursos generada automáticamente, `layout` es la carpeta, `activity_main` es el archivo. Sin esta línea, la pantalla nacería vacía.

```
enableEdgeToEdge()  
    ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v,  
insets ->  
        val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())  
        v.setPadding(systemBars.left, systemBars.top, systemBars.right,  
systemBars.bottom)  
        insets  
    }  
}
```

Líneas 9 a 15: código de la plantilla que ajusta el contenido para que no quede tapado por la barra de estado ni la barra de navegación del teléfono (el modo "edge to edge", de borde a borde). No necesitas entenderlo a fondo ahora: en las próximas unidades verás funciones, lambdas y variables con calma, y este bloque te servirá como ejemplo real de código profesional. Por hoy, solo reconoce su propósito general.

Ejemplo 4: Cambiar el texto desde Kotlin

Para cerrar, un ejemplo que une las dos mitades: vamos a cambiar el texto del saludo desde el código Kotlin en lugar del XML. Primero asegúrate de que el primer `TextView` tenga `android:id="@+id/txtSaludo"` en el XML. Luego, en `MainActivity.kt`, después de la línea `setContentView`, agrega:

```
val txtSaludo = findViewById<TextView>(R.id.txtSaludo)  
txtSaludo.text = "Hola Mundo desde Kotlin"
```

Explicación: `findViewById` busca en la pantalla el elemento con el identificador `txtSaludo` y te lo entrega como un objeto `TextView` que puedes manipular. La primera línea lo guarda en una variable llamada `txtSaludo`. La segunda línea cambia su propiedad `text` por el nuevo mensaje. Ejecuta la app: verás el nuevo texto, puesto por código, no por el XML. Acabas de ver la relación fundamental de Android: el XML dibuja, Kotlin controla. En las próximas unidades harás esto cientos de veces: encontrar elementos por su id y cambiarlos desde código en respuesta a lo que haga el usuario.

Ejemplo 5: Tu primer botón que responde al toque

Hasta ahora tus apps solo muestran texto. Vamos a agregar interactividad real: un botón que, al tocarlo, cambia el mensaje en pantalla. Este ejemplo introduce dos ideas que usarás en cada app de tu vida: los botones y la respuesta a eventos.

Primero, agrega el botón al XML, debajo del `TextView` del nombre:

```
<Button
    android:id="@+id/btnSaludar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Tócame"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/txtNombre"
    android:layout_marginTop="24dp" />
```

Explicación: `Button` es el elemento de botón estándar de Android. Lleva su propio id (`btnSaludar`) para encontrarlo desde Kotlin. El texto visible es "Tócame". Las restricciones lo centran horizontalmente y lo colocan debajo del texto del nombre. Lo nuevo aquí es `android:layout_marginTop="24dp"`: un margen superior de 24 dp (density-independent pixels, la unidad de distancia en Android) que deja un espacio de aire entre el texto y el botón. Sin ese margen, el botón quedaría pegado al texto y se vería apretado. Los márgenes son el "espacio para respirar" del diseño: úsalos siempre entre elementos.

Ahora, la magia en Kotlin. En `MainActivity.kt`, dentro de `onCreate` y después de `setContentView`, agrega:

```
val btnSaludar = findViewById<Button>(R.id.btnSaludar)
val txtSaludo = findViewById<TextView>(R.id.txtSaludo)

btnSaludar.setOnClickListener {
    txtSaludo.text = "Me tocaste, felicidades"
}
```

Línea por línea: la primera línea encuentra el botón por su id y lo guarda en la variable btnSaludar. La segunda hace lo mismo con el texto del saludo. La tercera es la estrella del ejemplo: setOnClickListener le dice al botón "cuando alguien te toque, ejecuta este bloque de código". Lo que está entre las llaves { } es una expresión lambda, un bloque de código que se guarda para ejecutarse después, cuando ocurra el toque. Dentro de la lambda, cambiamos el texto del saludo. Ejecuta la app, toca el botón y observa el cambio. Acabas de programar tu primera interacción usuario-app: el patrón "encuentra el elemento, escucha el evento, reacciona" que repetirás miles de veces en tu carrera.

Un detalle profesional: nota que declaramos las variables con val, no con var. En Kotlin, val significa "valor que no cambia" (como una constante) y var significa "variable que sí puede cambiar". Como btnSaludar y txtSaludo siempre apuntan al mismo elemento, val es lo correcto. Esta distinción la veremos a fondo en la unidad 2, pero empieza a acostumbrarte desde hoy: usa val por defecto y var solo cuando de verdad necesites reasignar.

Ejemplo 6: El AndroidManifest.xml, explicado línea por línea

El archivo AndroidManifest.xml es la carta de presentación de tu app ante el sistema Android. Todo lo que tu app es y necesita se declara aquí: su nombre, su icono, sus pantallas, sus permisos. Ábrelo en app > manifests > AndroidManifest.xml:

```

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.HolaMundo"
        tools:targetApi="34">
        <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>
    </application>
</manifest>

```

Vamos por partes. La etiqueta manifest envuelve todo el archivo. Dentro, la etiqueta application describe tu app como un todo: android:icon="@mipmap/ic_launcher" define el icono que se ve en el cajón de aplicaciones (apunta a un recurso de imagen); android:label="@string/app_name" define el nombre visible de la app (apunta a un recurso de texto, no está escrito directamente: buena práctica, porque así el nombre se puede traducir a otros idiomas sin tocar código); android:theme="@style/Theme.HolaMundo" define el estilo visual general (colores base, apariencia de los elementos). Observa el patrón: casi todo apunta a recursos con @tipo/nombre en lugar de valores escritos a mano. Esa es la filosofía Android: los valores viven en recursos centralizados.

Dentro de application está la etiqueta activity, que declara cada pantalla de tu app. android:name=".MainActivity" dice que esta pantalla es la clase MainActivity (el punto inicial es abreviatura del paquete). android:exported="true" indica que otros componentes del sistema pueden abrir esta pantalla. Y el intent-filter con action MAIN y category LAUNCHER es lo que le dice a Android "esta es la pantalla principal: ponle un icono en el cajón de aplicaciones y ábrela cuando el usuario toque ese icono". Si borras ese intent-filter, tu app se instalaría pero no tendría forma de abrirse: un error clásico de principiantes que mueven cosas sin entenderlas.

Experimento seguro: cambia android:label a un texto directo como android:label="Mi App Genial", ejecuta y mira el nombre bajo el icono en el emulador. Luego reviértelo a @string/app_name. Así compruebas que el manifest manda sobre cómo el sistema ve tu app.

Mini-proyecto de la unidad: tarjeta de presentación

Vamos a juntar todo lo aprendido en un mini-proyecto completo: una app de "tarjeta de presentación" con tu nombre, tu ocupación soñada ("Desarrollador Android") y un botón que muestre un mensaje motivacional al tocarlo. Crea un proyecto nuevo llamado "TarjetaPresentacion" y reemplaza el contenido de activity_main.xml con:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="#1B5E20"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/txtNombre"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Tu Nombre"
        android:textColor="#FFFFFF"
        android:textSize="32sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toTopOf="@id/txtOcupacion"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_chainStyle="packed" />

    <TextView
        android:id="@+id/txtOcupacion"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Desarrollador Android en formación"
        android:textColor="#C8E6C9"
        android:textSize="18sp"
        app:layout_constraintBottom_toTopOf="@id/btnMotivacion"
        app:layout_constraintEnd_toEndOf="parent"
```

```

        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/txtNombre" />

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

<TextView
    android:id="@+id/txtMensaje"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text=""
    android:textColor="#FFFFFF"
    android:textSize="20sp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/btnMotivacion"
    android:layout_marginTop="24dp" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

Novedades en este código: `android:background="#1B5E20"` pinta el fondo de verde oscuro (los colores en Android se escriben en hexadecimal: #RRGGBB). `android:textColor` cambia el color del texto a blanco o verde claro para que contraste con el fondo. `android:textStyle="bold"` pone el nombre en negritas. Los tres elementos principales forman una "cadena vertical" (vertical chain): cada uno está restringido al siguiente, y el atributo `app:layout_constraintVertical_chainStyle="packed"` en el primero los junta como un paquete centrado verticalmente. Es la técnica profesional para centrar grupos de elementos.

Y en `MainActivity.kt`, dentro de `onCreate` después de `setContent`:

```

val btnMotivacion = findViewById<Button>(R.id.btnMotivacion)
val txtMensaje = findViewById<TextView>(R.id.txtMensaje)

val mensajes = listOf(
    "Un día a la vez, una línea de código a la vez.",
    "Todo experto fue una vez un principiante.",
    "Los errores son maestros disfrazados.",
    "Tu primera app ya es un logro enorme."
)

btnMotivacion.setOnClickListener {
    txtMensaje.text = mensajes.random()
}

```

Aquí aparece `listOf`, que crea una lista de textos, y `mensajes.random()`, que elige uno al azar. Cada vez que tocas el botón, aparece un mensaje motivacional distinto. Ejecuta la app y tócalo varias veces. Este mini-proyecto usa todo lo de la unidad: proyecto nuevo, layout XML con restricciones y estilos, manifest implícito, Kotlin que encuentra elementos y reacciona a eventos. Guárdalo: es tu segunda app y la primera que hiciste casi solo.

Cómo leer los mensajes de error sin miedo

Merece una sección propia porque es la habilidad que más tiempo te va a ahorrar. Cuando algo falla, Android Studio muestra mensajes en la ventana "Build" (abajo) o en "Logcat". La mayoría de los principiantes ve el texto rojo y entra en pánico; los profesionales lo leen como una carta. Casi todos los mensajes de error de compilación siguen este formato: primero dicen QUÉ pasó ("error: cannot find symbol", "unresolved reference"), luego dicen DÓNDE (el nombre del archivo y el número de línea, como "MainActivity.kt:12"). Tu trabajo es: 1) leer el "qué", 2) ir al "dónde", 3) mirar esa línea con ojos frescos.

Los cinco mensajes que más vas a ver en esta etapa: "Unresolved reference: txtSaludo" significa que el id que buscas en Kotlin no existe en el XML (revisa que el `android:id` esté escrito idéntico, incluyendo mayúsculas). "Expecting member declaration" o errores de llaves suelen ser una llave `{` o `}` de más o de menos: cuenta las llaves de apertura y cierre. "Cannot resolve symbol R" indica que hay un error en algún archivo XML que impide generar la clase R: revisa tus layouts, casi siempre es una etiqueta mal cerrada. "Gradle sync failed" es problema de descarga o conexión: reintenta la sincronización. "Installation failed" al ejecutar en el emulador suele ser el emulador congelado: ciérralo y vuelve a abrirlo.

Regla de oro: lee el primer error de la lista, no el último. Cuando la compilación falla, el primer error suele ser la causa y los siguientes son consecuencias en cascada. Arregla el primero,

vuelve a compilar, y muchas veces los demás desaparecen solos. Y cuando busques el mensaje en internet (algo que hacen todos los programadores, sin excepción), copia el texto exacto del error entre comillas: así encuentras a alguien que ya tuvo tu mismo problema.

Práctica

Resuelve cada ejercicio por tu cuenta antes de leer la solución. Si te atorras más de veinte minutos, lee la solución, entiéndela y vuelve a intentarlo sin mirar.

Ejercicio 1: El historiador

Sin mirar tus apuntes, escribe con tus propias palabras: ¿qué es Android, quién lo creó originalmente, quién lo compró y en qué año salió el primer teléfono con Android? Además, explica qué significa que Android sea de código abierto y menciona una ventaja y un desafío que eso provoca.

Solución: Android es un sistema operativo para dispositivos móviles, basado en el núcleo Linux y desarrollado por Google. Lo creó originalmente la empresa Android Inc., fundada por Andy Rubin en 2003; Google la compró en 2005. El primer teléfono con Android fue el HTC Dream (T-Mobile G1), lanzado en 2008. Que sea de código abierto significa que su código fuente está disponible públicamente a través del proyecto AOSP, y cualquier fabricante puede usarlo y adaptarlo. Ventaja: hay miles de modelos de teléfonos de muchas marcas y precios, y Android domina el mercado mundial con más del 70 por ciento. Desafío: la fragmentación, es decir, la enorme variedad de dispositivos, versiones y tamaños de pantalla que el desarrollador debe tomar en cuenta al crear sus apps.

Ejercicio 2: El arquitecto

Dibuja en una hoja (a mano está bien) las cuatro capas de la arquitectura de Android, de abajo hacia arriba, y escribe una frase explicando qué hace cada capa y un ejemplo de qué vive en ella.

Solución: De abajo hacia arriba: 1) Núcleo Linux: habla directamente con el hardware; ejemplo: el controlador de la pantalla táctil o la gestión de memoria. 2) Bibliotecas nativas y Android Runtime (ART): ejecutan las aplicaciones y ofrecen funciones base en C/C++; ejemplo: la máquina virtual que corre tu app. 3) Framework de aplicaciones: servicios listos para usar que ofrece Android a los desarrolladores; ejemplo: el sistema de notificaciones o el acceso a la cámara. 4) Aplicaciones: los programas que usa el usuario; ejemplo: tu app "Hola Mundo" o WhatsApp. Cada capa se apoya en la de abajo y ofrece servicios a la de arriba, lo que permite programar sin conocer el hardware a fondo.

Ejercicio 3: El instalador

Enumera los pasos de instalación de Android Studio en orden, e indica qué es el SDK, qué contiene y dónde se configura dentro de Android Studio. Menciona además dos errores comunes de instalación y cómo se resuelve cada uno.

Solución: Pasos: 1) Descargar Android Studio solo desde el sitio oficial (developer.android.com/studio). 2) Ejecutar el instalador (.exe en Windows, .dmg en macOS, .tar.gz con studio.sh en Linux) con las opciones por defecto. 3) Completar el asistente de configuración inicial con la opción Standard, que descarga el SDK, las herramientas y la imagen del emulador. El SDK (Software Development Kit) es el kit de desarrollo: contiene las APIs de Android, las herramientas de compilación, las platform tools (como adb) y las imágenes del sistema para el emulador. Se configura en el SDK Manager (Settings > Appearance and Behavior > System Settings > Android SDK), con pestañas para plataformas, herramientas y sitios de actualización. Errores comunes: a) falta de espacio en disco: liberar al menos 10 GB antes de instalar; b) el emulador no arranca por virtualización desactivada: activar Intel VT-x o AMD-V en la BIOS; c) antivirus que bloquea la instalación en Windows: pausarlo temporalmente durante la instalación.

Ejercicio 4: El explorador del proyecto

Crea un proyecto nuevo llamado "Mi Primera App" con package name com.tunombre.miprimeraapp (usa tu nombre real). Luego, sin copiar de ningún lado, escribe qué contiene cada una de estas ubicaciones: app > manifests, app > java, app > res > layout, y Gradle Scripts. Ejecuta la app en el emulador y confirma que arranca.

Solución: app > manifests contiene el AndroidManifest.xml, la carta de presentación de la app: nombre, permisos, pantallas declaradas y pantalla principal. app > java contiene el código fuente Kotlin, incluyendo MainActivity.kt, la pantalla principal. app > res > layout contiene los archivos XML de diseño de pantallas, como activity_main.xml. Gradle Scripts contiene la

configuración de construcción, como build.gradle.kts, con dependencias y ajustes de compilación. La app debe compilar, instalarse en el emulador y mostrar el texto "Hello World!" (o el texto modificado si ya lo cambiaste).

Ejercicio 5: El personalizador

Modifica tu app "Mi Primera App" para que muestre dos textos centrados: arriba uno grande que diga "Bienvenido a mi primera app" y debajo uno más chico con tu nombre completo. Hazlo solo con XML. Escribe el código del segundo TextView y explica cada atributo que usaste.

Solución: El primer TextView lleva android:id="@+id/txtTitulo", android:text="Bienvenido a mi primera app", android:textSize="28sp" y las cuatro restricciones al parent para centrarlo. El segundo TextView es:

```
<TextView
    android:id="@+id/txtNombre"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Tu Nombre Completo"
    android:textSize="18sp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/txtTitulo" />
```

android:id crea el identificador txtNombre; wrap_content ajusta el tamaño al contenido; android:text es el texto visible; textSize 18sp lo hace más chico que el título para crear jerarquía visual; las restricciones de inicio y fin al parent lo centran horizontalmente; y Top_toBottomOf lo coloca debajo del título referenciando su id. Al ejecutar, se ven los dos textos centrados, uno debajo del otro.

Ejercicio 6: El programador

En el mismo proyecto del ejercicio 5, cambia el texto del título desde el código Kotlin (no desde el XML) para que diga "App creada con Kotlin". Escribe las líneas que agregaste en MainActivity.kt, indica dónde las colocaste exactamente y explica qué hace cada una.

Solución: Las líneas se agregan dentro de la función onCreate, después de setContentView(R.layout.activity_main), porque los elementos de la pantalla solo existen después de esa línea:

```
val txtTitulo = findViewById<TextView>(R.id.txtTitulo)
txtTitulo.text = "App creada con Kotlin"
```

La primera línea busca en la pantalla el elemento con id txtTitulo, lo trata como TextView y lo guarda en la variable txtTitulo. Es necesario importar android.widget.TextView si Android Studio lo pide (el IDE suele agregarlo solo). La segunda línea asigna un nuevo valor a la propiedad text del elemento. Al ejecutar, el título muestra "App creada con Kotlin" aunque en el XML diga otra cosa, lo que demuestra que el código puede controlar la apariencia en tiempo de ejecución.

Ejercicio 7: El detective de errores

Provoca a propósito tres errores en tu proyecto y documenta qué mensaje muestra Android Studio en cada caso: a) borra el android:id de un TextView que buscas con findViewById; b) quita una llave de cierre } en MainActivity.kt; c) escribe mal el nombre de un recurso en el manifest (por ejemplo, cambia @string/app_name por @string/nombre_que_no_existe). Para cada uno, anota el mensaje exacto, explica qué significa y cómo lo corrigiste.

Solución: a) Al compilar aparece "Unresolved reference": Kotlin no encuentra el identificador porque el XML ya no lo define. Significa que el id del XML y el del código no coinciden. Se corrige devolviendo el android:id al XML o actualizando el id en el código para que ambos sean idénticos. b) Aparecen errores de sintaxis como "Expecting member declaration" en cascada: el compilador se pierde porque los bloques no cierran donde deben. Se corrige contando llaves de apertura y cierre y devolviendo la faltante; Android Studio resalta en rojo la zona aproximada. c) Error de recurso no encontrado al compilar, del tipo "resource string/nombre_que_no_existe not found": el manifest referencia un recurso que no existe en res/values/strings.xml. Se corrige creando el recurso o devolviendo la referencia original. La lección: cada mensaje apunta a un archivo y una causa concreta; leerlo con calma es más rápido que adivinar.

Buenas prácticas del desarrollador principiante

Cierra esta unidad adoptando cinco hábitos que distinguen al profesional del aficionado. Primero, guarda y ejecuta con frecuencia: cada cambio pequeño probado de inmediato es un error pequeño fácil de encontrar. Segundo, pon nombres descriptivos a todo: txtNombre dice más que textView2, y dentro de seis meses te lo vas a agradecer. Tercero, no borres código que no entiendes: si la plantilla lo generó, déjalo ahí hasta que una unidad futura te explique qué hace; borrar a ciegas rompe cosas. Cuarto, comenta tu código cuando hagas algo poco obvio: en Kotlin los comentarios se escriben con // al inicio de la línea, y tu yo del futuro te lo

agradecerá. Quinto, respalda tus proyectos: copia la carpeta del proyecto a un USB, a la nube o usa un sistema de control de versiones como Git (lo verás más adelante en el libro). Perder una semana de trabajo por no respaldar es una lección que nadie quiere aprender a la mala.

Profundización: qué pasa cuando presionas Run (Gradle y la compilación)

Cuando presionas el botón verde de Run ocurren muchas cosas en pocos segundos, y entenderlas te convierte en un desarrollador que diagnostica en lugar de adivinar. El protagonista de este proceso es Gradle, el sistema de automatización de construcción de Android. Piensa en Gradle como el gerente de obra: tú le entregas los planos (tu código y recursos) y él coordina a todos los trabajadores para levantar el edificio (tu app instalada).

El proceso tiene estas etapas. Primero, Gradle lee los archivos de configuración (`build.gradle.kts` del proyecto y del módulo `app`): ahí se declara con qué versión del SDK compilar, cuáles son las dependencias (bibliotecas externas) y los ajustes de la app. Segundo, compila tu código Kotlin: traduce tus archivos `.kt` a un formato intermedio llamado `bytecode`. Tercero, procesa los recursos: convierte tus XML de layouts, tus textos de `strings.xml` y tus imágenes en un formato optimizado y genera la clase `R`, que es el puente que permite referenciar cada recurso desde el código (`R.layout.activity_main`, `R.id.txtSaludo`). Cuarto, empaqueta todo en un archivo APK (Android Package): el instalable de Android, que contiene el `bytecode` convertido al formato que entiende el Android Runtime, los recursos procesados y el `manifest`. Quinto, firma el APK con una llave de depuración (en modo desarrollo es automática) y lo instala en el dispositivo seleccionado mediante `adb`. Sexto, lanza la actividad principal. Todo eso, en segundos, cada vez que presionas Run.

¿Por qué te importa? Porque cada etapa tiene sus errores típicos y ahora sabrás dónde mirar. Si falla la lectura de configuración, el problema está en tus archivos `.gradle` (casi siempre una dependencia mal escrita). Si falla la compilación de Kotlin, el problema está en tu código (el mensaje dice archivo y línea). Si falla el procesamiento de recursos, el problema está en un XML (etiqueta mal cerrada, recurso referenciado que no existe: el famoso "cannot resolve symbol R" nace aquí). Si falla la instalación, el problema es el dispositivo o el emulador (desconéctalo, reconéctalo o reinícialo). Con este mapa, un error deja de ser "no funciona" y se convierte en "falla en la etapa 3, reviso mis XML".

Dos archivos de Gradle que debes ubicar desde hoy: `build.gradle.kts` a nivel de proyecto (configuración global, versiones de plugins) y `app/build.gradle.kts` a nivel de módulo (aquí vive lo interesante: `applicationId` que debe coincidir con tu package name, `minSdk` y `targetSdk` que definen compatibilidad, y el bloque `dependencies` donde se agregan bibliotecas). No necesitas editarlos en esta unidad, pero cuando en unidades futuras agreguemos una biblioteca, sabrás exactamente dónde y por qué.

Profundización: los recursos y el archivo strings.xml

En el ejemplo del manifest viste `android:label="@string/app_name"`: el nombre de la app no está escrito directamente, sino que apunta a un recurso de texto. Los recursos son todos los elementos no-código de tu app (textos, colores, dimensiones, imágenes) centralizados en la carpeta `res`. La filosofía es: los valores viven en un solo lugar y el código y los layouts los referencian. ¿La ventaja? Si quieres cambiar el nombre de tu app, lo cambias una vez en `strings.xml` y se actualiza en todas partes. Si quieres traducir tu app al inglés, agregas un archivo de textos en inglés y Android elige solo el correcto según el idioma del teléfono.

Abre `app > res > values > strings.xml`:

```
<resources>
    <string name="app_name">Hola Mundo</string>
</resources>
```

Cada string tiene un `name` (el identificador) y el texto. Para agregar el tuyo:

```
<resources>
    <string name="app_name">Hola Mundo</string>
    <string name="mensaje_bienvenida">Bienvenido a mi primera app</string>
    <string name="nombre_desarrollador">Tu Nombre</string>
</resources>
```

Y en el layout lo usas así: `android:text="@string/mensaje_bienvenida"`. En Kotlin lo usas así: `getString(R.string.mensaje_bienvenida)`. Los colores funcionan igual en `res/values/colors.xml`:

```
<resources>
    <color name="verde_fondo">#1B5E20</color>
    <color name="blanco">#FFFFFF</color>
</resources>
```

Y en el layout: `android:background="@color/verde_fondo"`. Buena práctica profesional desde el día uno: nunca escribas textos visibles directamente en el XML si el proyecto va a crecer; usa recursos de string. Parece más trabajo al inicio, pero cuando tu app tenga 20 pantallas y quieras cambiar una frase, lo agradecerás. Lo mismo con colores y tamaños: centralízalos y tu app tendrá una identidad visual consistente sin esfuerzo.

Glosario de la unidad

- Actividad (Activity): cada pantalla de una app Android; `MainActivity` es la pantalla principal.

- ADB (Android Debug Bridge): herramienta que comunica tu computadora con el dispositivo para instalar y depurar apps.
- APK: el paquete instalable de Android que se genera al compilar tu proyecto.
- AVD (Android Virtual Device): el emulador, un teléfono virtual dentro de tu computadora.
- ConstraintLayout: contenedor que posiciona elementos con restricciones en lugar de coordenadas fijas.
- Depuración USB: modo del teléfono que permite instalar y probar apps desde la computadora.
- Gradle: sistema que compila, empaqueta e instala tu app automáticamente.
- Intent-filter: declaración en el manifest que indica cómo el sistema puede abrir una pantalla; el de MAIN/LAUNCHER marca la pantalla inicial.
- Layout: archivo XML que describe la apariencia de una pantalla.
- Manifest (AndroidManifest.xml): carta de presentación de la app ante el sistema: nombre, icono, pantallas, permisos.
- Package name: identificador único de tu app, en formato de dominio invertido y minúsculas.
- Recurso (res): textos, colores, imágenes y dimensiones centralizados que el código referencia con R.
- SDK: kit de desarrollo con APIs, herramientas de compilación y platform tools.
- SDK Manager: pantalla de Android Studio para instalar y actualizar componentes del SDK.
- TextView / Button: elementos de interfaz para mostrar texto y botones tocables.
- Vista previa (preview): panel del editor que muestra cómo se verá el layout sin ejecutar la app.

Autoevaluación

Responde estas preguntas sin mirar el texto. Si fallas alguna, regresa a la sección indicada entre paréntesis.

1. ¿Quién creó originalmente Android y quién lo compró? (Clase 1)
2. ¿Qué contiene el SDK y dónde se gestiona dentro de Android Studio? (Clase 2)

3. ¿Qué hace la línea `setContentView(R.layout.activity_main)` y por qué es indispensable? (Clase 3)
4. En un `ConstraintLayout`, ¿qué hacen las cuatro restricciones `app:layout_constraint..._to...Of="parent"` aplicadas a un `TextView`? (Ejemplos)
5. ¿Qué declara el intent-filter con action `MAIN` y category `LAUNCHER` en el `AndroidManifest.xml`? (Ejemplo 6)
6. Si cambias un texto en el XML pero al ejecutar la app ves el texto viejo, ¿cuáles son las dos causas más probables? (Clase 3, errores comunes)

Respuestas: 1) Lo creó Android Inc. (Andy Rubin, 2003); Google lo compró en 2005. 2) El SDK contiene las APIs de Android, las herramientas de compilación, las platform tools (adb) y las imágenes del emulador; se gestiona en el SDK Manager (Settings > Appearance and Behavior > System Settings > Android SDK). 3) Conecta el código Kotlin de la pantalla con su diseño XML; sin ella la actividad nacería vacía porque el sistema no sabría qué layout mostrar. 4) Centran el elemento: al restringir sus cuatro bordes a los cuatro bordes del contenedor padre, el `TextView` queda perfectamente centrado en la pantalla. 5) Declara que esa actividad es la pantalla principal de la app: Android le pone un icono en el cajón de aplicaciones y la abre cuando el usuario lo toca. 6) Que no guardaste el archivo antes de ejecutar, o que no presionaste Run de nuevo después del cambio (los cambios no se aplican solos).

Si respondiste bien al menos 5 de 6, estás listo para la unidad 2. Si no, repasa: esta base te va a acompañar todo el libro.

Preguntas frecuentes de la unidad

¿Debo aprender Java antes de Kotlin? No. Kotlin es el lenguaje oficial y recomendado, y todo este libro lo usa. Saber Java ayuda a leer tutoriales viejos, pero no es un prerrequisito: empezar directo con Kotlin es el camino moderno y el que Google recomienda.

¿Android Studio es gratis, incluso para vender apps? Sí. Android Studio es gratuito sin restricciones de uso comercial. Lo que tiene costo es la cuenta de desarrollador de Google Play (pago único) cuando decidas publicar, tema que veremos en la parte de publicación del libro.

¿Necesito un teléfono Android físico o basta el emulador? El emulador basta para todo el libro. El teléfono físico es un lujo agradable (pruebas más rápidas y realistas, sensores reales), pero ningún ejercicio lo exige. Si tu computadora tiene poca memoria, el teléfono físico puede incluso ser mejor opción que el emulador.

Mi computadora es vieja y todo va lento, ¿qué hago? Tres trucos: usa un teléfono físico en lugar del emulador para ahorrar memoria, cierra el navegador y otros programas mientras trabajas en Android Studio, y evita actualizar a la última versión el día que sale (las versiones .1 suelen ser más estables). Con 8 GB de RAM se puede trabajar; con 16 GB se trabaja cómodo.

¿Cuánto tarda la primera compilación y es normal que tarde tanto? La primera compilación de un proyecto puede tardar varios minutos porque Gradle descarga dependencias de internet. Es completamente normal. Las siguientes compilaciones reutilizan lo descargado y tardan segundos. Si vives con internet lento, la primera sincronización es el momento de tener paciencia.

¿Puedo seguir este libro en macOS o Linux si los ejemplos mencionan Windows? Sí. El 99 por ciento del contenido es idéntico en los tres sistemas; las diferencias (instaladores, atajos de teclado, rutas de carpetas) se señalan en el texto cuando importan. Los proyectos, el código y el emulador funcionan igual en todas partes.

Resumen de la unidad

- Android es un sistema operativo móvil basado en Linux, desarrollado por Google, de código abierto a través del proyecto AOSP.
- Nació en Android Inc. (2003, Andy Rubin), Google lo compró en 2005 y el primer teléfono con Android (HTC Dream) salió en 2008.
- Su apertura explica su dominio del mercado (más del 70 por ciento) y su desafío principal: la fragmentación de dispositivos y versiones.
- La arquitectura se organiza en capas: núcleo Linux, bibliotecas nativas y Android Runtime (ART), framework de aplicaciones y aplicaciones.
- Kotlin es el lenguaje oficial y recomendado por Google para Android ("Kotlin primero"); Java fue el lenguaje histórico.
- Android Studio es el IDE oficial y gratuito: incluye editor, compilador, depurador, emulador y herramientas de publicación.
- El SDK (Software Development Kit) contiene las APIs, herramientas de compilación, platform tools e imágenes del emulador; se gestiona en el SDK Manager.
- El emulador (AVD) permite probar apps sin teléfono físico; un teléfono real con depuración USB activada también sirve.

- Un proyecto Android separa la apariencia (XML en res/layout) de la lógica (Kotlin en java): el plano y la electricidad.
- Archivos clave: `AndroidManifest.xml` (carta de presentación), `MainActivity.kt` (pantalla principal), `activity_main.xml` (diseño), `build.gradle.kts` (configuración de construcción).
- La línea `setContentView(R.layout.activity_main)` conecta el código Kotlin con el diseño XML.
- `ConstraintLayout` posiciona elementos con restricciones (constraints) en lugar de coordenadas fijas, lo que adapta la pantalla a cualquier tamaño.
- El ciclo de trabajo del desarrollador es: crear, escribir, compilar, probar, corregir; probar con frecuencia después de cambios pequeños evita errores grandes.
- Los errores de compilación en rojo no son el fin del mundo: casi siempre indican el archivo y la línea exactos del problema.

UNIDAD 2: Fundamentos de Kotlin

Introducción

Si la unidad 1 fue conocer la cocina y montar el taller, esta unidad es aprender a usar los cuchillos: la herramienta con la que vas a trabajar todos los días. Kotlin es el lenguaje de programación oficial y moderno para el desarrollo Android, y dominarlo no es opcional: es el cimiento sobre el que se construye absolutamente todo lo demás en este libro. Cada pantalla que diseñes, cada botón que programes, cada dato que guardes pasará por código Kotlin. La buena noticia es que Kotlin fue diseñado para ser amable con los principiantes sin dejar de ser potente para los expertos.

En esta unidad veremos tres clases. La clase 4 te presenta las variables y los tipos de datos: cómo guardar información en la memoria, la diferencia crucial entre `val` (valor fijo) y `var` (valor que cambia), y el famoso sistema de null safety, la característica más celebrada de Kotlin, que elimina de raíz una de las fuentes de errores más comunes en la historia de la programación. La clase 5 cubre las decisiones y las repeticiones: los condicionales (`if`, `when`) que permiten que tu programa tome decisiones, y los ciclos (`for`, `while`) que permiten repetir tareas sin escribir el mismo código cien veces. La clase 6 entra a las funciones: cómo empaquetar código reutilizable con parámetros y valores de retorno, y las expresiones lambda, esos bloques de código compactos que ya viste en acción con `setOnClickListener` en la unidad 1 y que ahora entenderás a fondo.

Por qué importa esta unidad

Hay una frase entre programadores: "los lenguajes se olvidan, los fundamentos no". La sintaxis exacta de un lenguaje la puedes buscar en internet en diez segundos; lo que no puedes buscar es la forma de pensar. Esta unidad te entrena en esa forma de pensar: cómo representar información con variables, cómo modelar decisiones con condicionales, cómo automatizar repeticiones con ciclos y cómo organizar el código con funciones. Esos cuatro pilares existen en todos los lenguajes de programación del mundo. Si los dominas en Kotlin, después podrás aprender cualquier otro lenguaje en una fracción del tiempo.

Además, hay una razón práctica e inmediata: sin Kotlin no puedes avanzar en el libro. La unidad 3 en adelante asume que sabes declarar variables, escribir un `if`, recorrer una lista con un `for` y crear una función. Si intentas saltarte esta unidad, cada ejemplo posterior te parecerá escrito en jeroglíficos. En cambio, si la trabajas con calma, los ejemplos futuros se leerán casi solos, como leer español con algunas palabras técnicas nuevas.

Qué sabrás al terminar

Al terminar esta unidad vas a poder declarar variables con `val` y `var` eligiendo correctamente cuál usar en cada caso, y explicar por qué esa elección importa. Vas a conocer los tipos de datos básicos (números enteros, decimales, textos, booleanos) y sabrás cuándo usar cada uno. Vas a entender el concepto de `null`, por qué es peligroso y cómo Kotlin te protege con la `null safety`: los tipos anulables, el operador de llamada segura y el operador Elvis. Vas a escribir condicionales con `if-else` y `when` para que tus programas tomen decisiones, incluyendo el `when` como expresión. Vas a usar ciclos `for` y `while` (y `do-while`) para repetir tareas, recorrer rangos y listas, y sabrás cuándo preferir cada uno. Vas a definir funciones con parámetros y valores de retorno, usar parámetros con valores por defecto y argumentos nombrados, y vas a entender las lambdas: qué son, cómo se escriben y por qué `setOnClickListener` funciona cómo funciona. También conocerás los errores más comunes de cada tema y las buenas prácticas que usan los profesionales. En resumen: vas a pensar como programador y a escribir Kotlin con confianza.

Una nota sobre cómo practicar esta unidad: los ejemplos de Kotlin puro (sin Android) los puedes probar de dos formas. La más fácil es usar un "Kotlin Playground" en línea: buscas "Kotlin playground" en tu navegador y escribes el código ahí, sin instalar nada. La segunda es crear un proyecto de Kotlin en Android Studio o usar la herramienta de "scratch" del IDE. En este libro presentaré los ejemplos dentro de una función `main`, que es el punto de entrada de un programa Kotlin sencillo: todo lo que esté dentro de `fun main()` se ejecuta de arriba hacia abajo. Cuando veas un ejemplo, cópialo dentro de un `main` en el playground y ejecútalo para ver el resultado con tus propios ojos.

Explicación detallada

Clase 4: Variables, tipos de datos, `val` vs `var` y `null safety`

Empecemos por lo más básico de lo básico: una variable es un espacio con nombre en la memoria de la computadora donde guardas un dato para usarlo después. La analogía clásica es la de las cajas etiquetadas: imagina una repisa con cajas, cada caja tiene una etiqueta con un nombre ("`edad`", "`nombre`", "`precio`") y dentro hay un valor. Cuando tu programa necesita el dato, busca la caja por su etiqueta. Cuando el dato cambia, abres la caja y reemplazas el contenido. Programar, en su esencia más simple, es guardar datos en cajas, transformarlos y mostrar resultados.

En Kotlin, crear una variable (declararla) se hace con dos palabras clave: `val` y `var`. Esta es la primera gran decisión que tomarás cientos de veces al día, así que entiéndela bien. `val` declara

una variable de solo lectura: una vez que le asignas un valor, ese valor no puede cambiar. Es como escribir con pluma en la etiqueta de la caja y sellarla: lo que guardaste ahí se queda. var declara una variable mutable: puedes cambiar su valor cuantas veces quieras. Es como escribir con lápiz y tener un borrador a la mano.

```
val nombre = "Ana"  
var edad = 25  
edad = 26
```

En este ejemplo, nombre es val: siempre será "Ana". Si intentas hacer nombre = "María" después, el compilador te marcará un error. edad es var: empieza en 25 y luego cambia a 26 sin problema. La regla de oro de Kotlin, y de la programación moderna en general, es: usa val por defecto, siempre, y cambia a var solo cuando tengas una razón concreta para que el valor cambie. ¿Por qué? Porque los valores que no cambian son más fáciles de razonar: si ves un val, sabes que su valor es el mismo en todo el programa, y eso elimina una familia entera de errores difíciles de rastrear ("¿en qué momento cambió esta variable?"). Los programadores experimentados leen val como una promesa de estabilidad. Acostúmbrate desde hoy y tu código será más robusto desde el primer día.

Ahora, los tipos de datos. Cada caja no solo tiene etiqueta y contenido: también tiene un "tipo" que dice qué clase de dato puede guardar. Kotlin tiene varios tipos básicos, y aunque el compilador muchas veces los adivina solo (inferencia de tipos), debes conocerlos:

Los números enteros: Int es el más común, guarda números enteros de -2 mil millones a 2 mil millones aproximadamente. Ejemplos: edad, cantidad de productos, número de intentos. Long es su hermano mayor, para enteros gigantescos. En Android usarás Int el 99 por ciento de las veces.

Los números decimales: Double guarda números con punto decimal, como 19.99 o 3.14159. Ejemplo clásico: precios, promedios, coordenadas. Existe también Float, una versión más ligera, pero Double es el estándar en Kotlin y el que debes usar por defecto.

Los textos: String guarda cadenas de texto entre comillas dobles: "Hola Mundo", "Ana", "Calle Falsa 123". Los textos son probablemente el tipo que más usarás en apps reales: nombres de usuario, mensajes, direcciones.

Los booleanos: Boolean solo puede valer true (verdadero) o false (falso). Parece simple, pero es el tipo que controla todas las decisiones de tu programa: "¿el usuario está registrado?", "¿la contraseña es correcta?", "¿quedan intentos?". Cada if de tu vida evalúa un Boolean.

Los caracteres individuales: Char guarda un solo carácter entre comillas simples, como 'A' o '7'. Se usa poco en apps comunes, pero existe.

Puedes declarar el tipo explícitamente con dos puntos después del nombre, o dejar que Kotlin lo infiera del valor:

```
val edad: Int = 25
val precio = 19.99
val nombre: String = "Ana"
val activo = true
```

Ambas formas son correctas. La inferencia (no escribir el tipo) hace el código más limpio y es lo común en Kotlin moderno. El tipo explícito se usa cuando quieres dejar clara tu intención o cuando el compilador no puede adivinar el tipo solo. Buena práctica: confía en la inferencia para casos obvios y escribe el tipo cuando aporte claridad.

Un error común de principiantes: intentar guardar un tipo en una variable de otro tipo. En Kotlin eso es un error de compilación, no una advertencia silenciosa:

```
var edad = 25
edad = "veinticinco"
```

Esto no compila: edad es Int y "veinticinco" es String. Kotlin es de tipado fuerte: las cajas aceptan un solo tipo de contenido. Otros lenguajes permiten estas mezclas y generan errores raros en tiempo de ejecución; Kotlin te protege en tiempo de compilación, cuando el error es barato de corregir. Si necesitas convertir, hay funciones explícitas: "25".toInt() convierte texto a entero, 25.toString() convierte entero a texto, 19.99.toInt() convierte decimal a entero (truncando los decimales, cuidado).

Ahora llegamos a la estrella de la clase: la null safety, o seguridad contra nulos. Primero, ¿qué es null? Es un valor especial que significa "ausencia de valor", "nada", "vacío". La caja existe, tiene etiqueta, pero está vacía. Ejemplo de la vida real: preguntas el segundo apellido de un usuario que no tiene segundo apellido; preguntas la foto de perfil de alguien que nunca subió una. En esos casos, el dato no es un texto vacío: es la ausencia del dato, y eso se representa con null.

El problema histórico: en muchos lenguajes (incluido Java, el antecesor de Kotlin en Android), cualquier variable podía valer null en cualquier momento, y si intentabas usar una variable nula (por ejemplo, pedirle su longitud a un texto que resultó ser null), el programa se estrellaba con el infamous NullPointerException. Tony Hoare, el científico que inventó la referencia nula en los años sesenta, la llamó después "mi error del billón de dólares", por la cantidad

inconmensurable de errores, caídas y vulnerabilidades que ha causado. Kotlin se diseñó para que ese error sea casi imposible.

¿Cómo? Con una regla simple y genial: en Kotlin, las variables NO pueden valer null a menos que tú lo declares explícitamente. Por defecto, todo es no-nulo:

```
var nombre: String = "Ana"
nombre = null
```

Esto no compila. String significa "texto garantizado, nunca null". Si quieres permitir la ausencia de valor, agregas el signo de interrogación al tipo:

```
var apodo: String? = "Anita"
apodo = null
```

String? (se lee "String anulable") significa "texto o null". El ? es tu declaración explícita de "acepto el riesgo de que esto esté vacío, y me haré responsable". Esta distinción, que parece un detalle, cambia todo: el compilador ahora SABE qué variables pueden ser nulas y te obliga a manejarlas con cuidado antes de usarlas.

¿Y cómo se usan las variables anulables de forma segura? Kotlin te da herramientas elegantes. La primera es la llamada segura, con el operador ?. (signo de interrogación seguido de punto):

```
val apodo: String? = null
println(apodo?.length)
```

apodo?.length significa: "si apodo no es null, dame su longitud; si es null, no hagas nada y el resultado es null". Sin el ?, el compilador te impediría llamar a .length directamente, porque sería peligroso. La llamada segura es como tocar la puerta antes de entrar: verificas que haya alguien antes de actuar. Puedes encadenarlas: usuario?.perfil?.foto?.url navega por una cadena de objetos donde cualquiera podría ser null, sin estrellarse.

La segunda herramienta es el operador Elvis, escrito como ?: (parece una carita con el pelo peinado hacia atrás, de ahí el nombre). Proporciona un valor por defecto cuando algo es null:

```
val apodo: String? = null
val apodoSeguro: String = apodo ?: "Sin apodo"
println(apodoSeguro)
```

Se lee: "usa apodo, pero si es null, usa 'Sin apodo' en su lugar". El resultado nunca es null: apodoSeguro es un String garantizado. El operador Elvis es perfectísimo para la vida real de las apps: mostrar "Sin foto" cuando no hay foto, usar 0 cuando no hay calificación, poner "Invitado" cuando no hay nombre de usuario.

También existe la aserción no nula !!, que le dice al compilador "confía en mí, esto no es null". Úsala casi nunca: si te equivocas, el programa se estrella igual que en Java. Los profesionales la evitan; si sientes la tentación de usar !!, casi siempre hay una forma mejor con ?. o ?!.

Errores comunes de esta clase: primero, declarar todo como var por costumbre y luego no saber qué cambió ni dónde; solución: val por defecto. Segundo, pelear con el compilador por la null safety en lugar de agradecerle: cuando Kotlin te impide usar una variable anulable sin protección, no es un estorbo, es un guardián evitándote un crash en el teléfono del usuario. Tercero, usar !! para "callar" al compilador rápido: funciona hoy, explota mañana. Cuarto, confundir null con texto vacío o cero: null es ausencia de valor, "" es un texto que existe pero no tiene letras, 0 es un número que existe y vale cero; son tres cosas distintas y tratarlas igual genera bugs lógicos.

Analogía de cierre: piensa en las variables como cajas etiquetadas en una bodega. val son cajas selladas con cinta adhesiva: lo que guardaste se queda para siempre. var son cajas abiertas: puedes cambiar el contenido. Los tipos son las formas de las cajas: no puedes meter una pelota de fútbol (String) en una caja diseñada para huevos (Int). Y la null safety es el sistema de inventario de la bodega: cada caja tiene una marca que dice "siempre tiene algo dentro" o "puede estar vacía", y el encargado (el compilador) no te deja abrir una caja marcada como "puede estar vacía" sin guantes de protección (?. o ?!). Una bodega así nunca pierde mercancía ni sufre accidentes.

Clase 5: Condicionales y ciclos

Los programas interesantes toman decisiones y repiten tareas. Sin decisiones, una app haría exactamente lo mismo para todos los usuarios en todas las situaciones: inútil. Sin repeticiones, tendrías que escribir el mismo código cientos de veces: inmanejable. Esta clase te da las dos herramientas: condicionales para decidir, ciclos para repetir.

Empecemos con el condicional if, la estructura de decisión más fundamental de la programación. Su lógica es idéntica a la del lenguaje cotidiano: "Si pasa esto, HAZ aquello; SI NO, haz esto otro".

```

val edad = 20

if (edad >= 18) {
    println("Eres mayor de edad")
} else {
    println("Eres menor de edad")
}

```

Desmenucemos: if evalúa la condición entre paréntesis (`edad >= 18`). Esa condición produce un Boolean: `true` o `false`. Si es `true`, se ejecuta el bloque entre las primeras llaves; si es `false`, se ejecuta el bloque del `else`. Los operadores de comparación que puedes usar: `==` (igual que), `!=` (distinto de), `>` (mayor que), `<` (menor que), `>=` (mayor o igual), `<=` (menor o igual). Ojo con el detalle clásico: para comparar igualdad se usan DOS signos de igual (`==`); un solo `=` es asignación y te dará un error o un comportamiento inesperado. Este es probablemente el error más cometido por principiantes en toda la historia de la programación: confúndelo una vez, aprende la lección, no lo repitas.

Puedes encadenar varias condiciones con `else if`:

```

val calificacion = 85

if (calificacion >= 90) {
    println("Excelente")
} else if (calificacion >= 80) {
    println("Muy bien")
} else if (calificacion >= 70) {
    println("Aprobado")
} else {
    println("Reprobado")
}

```

Kotlin evalúa de arriba hacia abajo y ejecuta solo el primer bloque cuya condición sea `true`. Con `calificacion = 85`, imprime "Muy bien" y salta todo lo demás. El orden importa: si pusieras la condición `>= 70` primero, un 95 caería en "Aprobado" por error, porque 95 también es `>= 70`. Regla práctica: en cadenas de `else if` con rangos, ordena de la condición más estricta a la más general.

Para combinar condiciones existen los operadores lógicos: `&&` significa "Y" (ambas deben ser `true`), `||` significa "O" (basta que una sea `true`), `!` significa "NO" (invierte el valor):


```
val tieneBoleto = true
val esSocio = false

if (tieneBoleto && esSocio) {
    println("Entrada VIP")
} else if (tieneBoleto || esSocio) {
    println("Entrada general")
} else {
    println("Sin entrada")
}
```

Con `tieneBoleto = true` y `esSocio = false`: la primera condición es `false` (porque `esSocio` es `false`), la segunda es `true` (porque `tieneBoleto` es `true`), así que imprime "Entrada general". Los operadores lógicos son el pegamento de las decisiones complejas: "si el usuario está registrado Y tiene saldo suficiente", "si es administrador O es el dueño del dato".

Una característica elegante de Kotlin: `if` es una expresión, es decir, produce un valor que puedes guardar en una variable:

```
val edad = 20
val categoria = if (edad >= 18) "adulto" else "menor"
println(categoria)
```

Esto asigna "adulto" o "menor" a categoría según la condición, en una sola línea. Cuando ambas ramas son simples, esta forma es más limpia que el `if` de varias líneas. Nota que aquí el `else` es obligatorio: como la expresión debe producir un valor sí o sí, Kotlin exige cubrir el caso contrario.

Cuando tienes muchas ramas basadas en el mismo valor, el `if-else` se vuelve torpe y aparece `when`, la joya de Kotlin para decisiones múltiples. Piensa en `when` como una versión elegante y potente del "switch" de otros lenguajes:

```

val dia = 3
val nombreDia = when (dia) {
    1 -> "Lunes"
    2 -> "Martes"
    3 -> "Miércoles"
    4 -> "Jueves"
    5 -> "Viernes"
    6, 7 -> "Fin de semana"
    else -> "Día inválido"
}
println(nombreDia)

```

Cada rama tiene un valor (o varios separados por coma, como 6, 7), la flecha -> y el resultado. when evalúa día, busca la rama que coincida y produce su resultado. La rama else cubre todo lo demás y, cuando when se usa como expresión (guardando el resultado), es obligatoria. Pero when es mucho más que un comparador de igualdad: puede evaluar rangos, tipos y condiciones arbitrarias:

```

val edad = 25
val etapa = when {
    edad < 13 -> "Niñez"
    edad < 20 -> "Adolescencia"
    edad < 60 -> "Adultez"
    else -> "Adultez mayor"
}

```

En esta forma, when no lleva argumento entre paréntesis: cada rama es una condición Boolean y se ejecuta la primera que sea true. Es un reemplazo hermoso para las cadenas largas de else if: más legible, menos propenso a errores de orden. Buena práctica profesional: si tu cadena de if-else if supera tres ramas sobre el mismo tema, considera reescribirla con when; tu código se vuelve autodocumentado.

Pasemos a los ciclos, las estructuras de repetición. El ciclo for recorre una secuencia de valores, ejecutando el bloque una vez por cada valor. La forma más común es con rangos:

```

for (i in 1..5) {
    println("Vuelta número $i")
}

```

1..5 crea un rango del 1 al 5 (ambos incluidos). La variable i toma cada valor del rango en cada vuelta: primero 1, luego 2, y así hasta 5. Dentro del bloque, "\$i" dentro del texto es una plantilla

de string (string template): Kotlin reemplaza \$i por el valor actual de i. Las plantillas evitan concatenar textos con + y hacen el código mucho más legible: "Hola, \$nombre, tienes \$edad años" se lee casi como español.

El for también recorre colecciones, y esta es su forma más útil en apps reales:

```
val frutas = listOf("manzana", "banana", "naranja")
for (fruta in frutas) {
    println("Me gusta la $fruta")
}
```

listOf crea una lista inmutable (una colección ordenada de elementos). El for toma cada elemento, lo guarda en fruta y ejecuta el bloque. Este patrón (for-elemento-in-colección) lo usarás constantemente: recorrer la lista de productos para mostrarlos, la lista de mensajes para dibujarlos, la lista de resultados para procesarlos.

Otras variantes útiles del for con rangos: 1..10 step 2 recorre de 2 en 2 (1, 3, 5, 7, 9); 10 downTo 1 recorre hacia atrás (10, 9, 8... 1); 1 until 10 recorre del 1 al 9 (excluye el final). until es perfecto para índices de listas, porque los índices empiezan en 0: for (i in 0 until frutas.size) recorre todas las posiciones válidas.

El ciclo while repite mientras una condición sea true, sin un número predeterminado de vueltas:

```
var intentos = 3
while (intentos > 0) {
    println("Te quedan $intentos intentos")
    intentos--
}
```

intentos-- resta 1 a la variable en cada vuelta. El ciclo se detiene cuando intentos llega a 0. La diferencia clave con for: while se usa cuando NO sabes cuántas vueltas necesitarás de antemano ("repite hasta que el usuario acierte", "repite mientras haya datos por leer"). Su primo do-while ejecuta el bloque al menos una vez y luego verifica la condición:

```
var numero = 0
do {
    println("El número es $numero")
    numero++
} while (numero < 3)
```

Aquí el bloque se ejecuta primero y la condición se revisa después, garantizando al menos una ejecución. Es útil para menús ("muestra el menú, y repite mientras el usuario no elija salir").

Dos palabras de control para ciclos: `break` sale del ciclo inmediatamente, `continue` salta a la siguiente vuelta sin terminar la actual:

```
for (i in 1..10) {  
    if (i == 3) continue  
    if (i == 7) break  
    println(i)  
}
```

Esto imprime 1, 2, 4, 5, 6: salta el 3 (`continue`) y se detiene antes del 7 (`break`). Úsalos con moderación: un ciclo con muchos `break`s y `continue`s se vuelve difícil de seguir.

El error más temido de los ciclos es el ciclo infinito: un `while` cuya condición nunca se vuelve `false`. El programa se queda dando vueltas para siempre y hay que detenerlo a la fuerza:

```
var x = 1  
while (x > 0) {  
    println(x)  
    x++  
}
```

Como `x` siempre crece, `x > 0` siempre es `true`: ciclo infinito. La vacuna es simple: asegúrate de que algo dentro del ciclo modifique la condición hacia su fin. Cada vez que escribas un `while`, pregúntate: "¿qué hace que esto termine?". Si no tienes una respuesta clara, reescríbelo.

Errores comunes de la clase: confundir `=` con `==` en las condiciones; olvidar el `else` en un `if` usado como expresión; ordenar mal las ramas de un `else if` encadenado; en `when` como expresión, olvidar la rama `else` (el compilador lo exige); off-by-one en rangos: usar `1..10` cuando querías 10 elementos empezando en 0 (usa `0` until `10` o `0..9`); y el clásico ciclo infinito por olvidar modificar la variable de control dentro del `while`. Buenas prácticas: prefiere `when` sobre cadenas largas de `else if`; usa plantillas de string (`$variable`) en lugar de concatenar con `+`; nombra la variable del `for` con algo significativo (`for (fruta in frutas)` en lugar de `for (x in frutas)`); y mantén los bloques de los ciclos cortos: si el interior de un `for` crece mucho, es señal de que ese código merece ser una función (tema de la próxima clase).

Analogía de cierre: los condicionales son como los cruces de caminos con señalamientos: "si vas al norte, toma la izquierda; si no, sigue derecho". Tu programa llega al `if`, lee el señalamiento (la condición) y elige el camino. Los ciclos son como una caminadora: el `for` es

una caminadora programada ("corre 5 kilómetros y para"), el while es una caminadora con sensor ("corre hasta que tu pulso baje de 120"). Y programar bien es saber cuándo necesitas un señalamiento y cuándo una caminadora.

Clase 6: Funciones, parámetros y lambdas básicas

Si las variables son las cajas donde guardas datos, las funciones son las máquinas de la fábrica: reciben materia prima (parámetros), hacen un trabajo y entregan un producto (valor de retorno). Sin funciones, tu código sería una lista interminable de instrucciones repetidas; con funciones, es un conjunto de piezas reutilizables, cada una con un propósito claro. Esta es la herramienta de organización más importante de la programación, y dominarla separa el código de aficionado del código profesional.

Una función en Kotlin se declara con la palabra `fun`, seguida del nombre, los parámetros entre paréntesis y, opcionalmente, el tipo de retorno después de dos puntos:

```
fun saludar(nombre: String): String {  
    return "Hola, $nombre"  
}  
  
fun main() {  
    val mensaje = saludar("Ana")  
    println(mensaje)  
}
```

Anatomía completa: `fun` anuncia "aquí empieza una función". `saludar` es el nombre: usa siempre nombres en minúscula que describan la acción (verbo), como `calcularTotal`, `mostrarMensaje`, `validarCorreo`. `(nombre: String)` declara un parámetro: la función recibe un texto y dentro de la función se llamará `nombre`. `: String` después del paréntesis declara el tipo de retorno: esta función devuelve un texto. El cuerpo entre llaves construye el saludo y `return` lo entrega a quien llamó a la función. En `main`, `saludar("Ana")` "llama" o "invoca" la función pasándole el argumento "Ana"; el valor retornado se guarda en `mensaje` y se imprime: "Hola, Ana".

La distinción parámetro vs argumento confunde a muchos principiantes, así que aclaremos: el parámetro es la variable declarada en la definición de la función (`nombre: String`), es el "molde". El argumento es el valor concreto que pasas al llamar (`saludar("Ana")`), es el "material" que viertes en el molde. Parámetro = en la definición; argumento = en la llamada. Con el tiempo lo dirás sin pensar.

Las funciones pueden recibir varios parámetros y de distintos tipos:

```

fun calcularTotal(precio: Double, cantidad: Int, tieneDescuento: Boolean): Double {
    val subtotal = precio * cantidad
    return if (tieneDescuento) subtotal * 0.9 else subtotal
}

fun main() {
    val total = calcularTotal(100.0, 3, true)
    println("Total a pagar: $total")
}

```

calcularTotal recibe precio, cantidad y tieneDescuento; multiplica precio por cantidad; si tieneDescuento es true aplica 10 por ciento de descuento (multiplica por 0.9). Al llamarla, los argumentos deben ir en el mismo orden que los parámetros: 100.0 es el precio, 3 la cantidad, true el descuento. Resultado: 270.0. Observa que dentro usamos el if como expresión para el return: Kotlin permite devolver directamente el resultado de un if, elegante y conciso.

No todas las funciones devuelven algo. Algunas solo hacen una acción (mostrar un mensaje, guardar un dato, pintar la pantalla). Esas se declaran sin tipo de retorno, o equivalentemente con el tipo Unit (que significa "no devuelve nada útil"):

```

fun mostrarBienvenida(nombre: String) {
    println("Bienvenido, $nombre")
    println("Esperamos que disfrutes la app")
}

```

Si no pones tipo de retorno, Kotlin asume Unit automáticamente. Estas funciones "hacen" en lugar de "calcular": su valor está en el efecto que producen, no en lo que devuelven.

Dos características de Kotlin que hacen las funciones más expresivas: los valores por defecto y los argumentos nombrados. Un parámetro puede tener un valor por defecto, que se usa cuando al llamar no pasas ese argumento:

```

fun saludar(nombre: String, saludo: String = "Hola"): String {
    return "$saludo, $nombre"
}

fun main() {
    println(saludar("Ana"))
    println(saludar("Luis", "Buenos días"))
}

```

La primera llamada usa el saludo por defecto ("Hola, Ana"); la segunda lo personaliza ("Buenos días, Luis"). Los valores por defecto reducen la necesidad de crear múltiples versiones de la misma función y hacen las llamadas comunes más cortas. Y los argumentos nombrados permiten especificar a qué parámetro corresponde cada valor, en cualquier orden:

```
fun crearUsuario(nombre: String, edad: Int, ciudad: String = "Desconocida") {  
    println("$nombre, $edad años, vive en $ciudad")  
}  
  
fun main() {  
    crearUsuario(edad = 30, nombre = "Ana")  
}
```

Al nombrar los argumentos (`edad = 30, nombre = "Ana"`), el orden ya no importa y el código se autodocumenta: cualquiera que lea la llamada sabe qué es cada valor sin ir a ver la definición. Combinados, valores por defecto y argumentos nombrados son una de las razones por las que el código Kotlin se lee tan bien.

Una función también puede ser de una sola expresión, usando el signo `=` en lugar de las llaves y el `return`:

```
fun doble(n: Int): Int = n * 2  
fun esMayorDeEdad(edad: Int): Boolean = edad >= 18
```

Cuando el cuerpo es una sola expresión, esta forma es más compacta y los profesionales la prefieren. `doble(5)` devuelve 10; `esMayorDeEdad(20)` devuelve `true`.

Hablemos del alcance (scope): las variables declaradas dentro de una función solo existen dentro de esa función. Nacen cuando la función empieza y mueren cuando termina:

```
fun ejemplo() {  
    val secreto = 42  
    println(secreto)  
}  
  
fun main() {  
    ejemplo()  
}
```

La variable `secreto` solo vive dentro de `ejemplo`; desde `main` no puedes acceder a ella. Esto es una característica, no una limitación: evita que las funciones se contaminen entre sí y es la base para razonar sobre programas grandes. Si necesitas que un dato salga de una función, lo

devuelves con return; si necesitas que entre, lo pasas como parámetro. Las funciones son cápsulas: entra materia prima por los parámetros, sale producto por el return, y lo de adentro no se mezcla con lo de afuera.

Ahora, las lambdas: el concepto que cierra el círculo con lo que viste en la unidad 1. Una lambda es una función sin nombre, un bloque de código que puedes guardar en una variable o pasar como argumento a otra función. Recuerda el botón:

```
btnSaludar.setOnClickListener {  
    txtSaludo.text = "Me tocaste, felicidades"  
}
```

Ese bloque entre llaves es una lambda: le estás pasando a `setOnClickListener` un pedazo de código que se ejecutará cuando el usuario toque el botón, no ahora. La sintaxis básica de una lambda es: parámetros entre llaves con flecha, luego el cuerpo:

```
val sumar: (Int, Int) -> Int = { a, b -> a + b }  
fun main() {  
    println(sumar(3, 4))  
}
```

Desmenucemos: `(Int, Int) -> Int` es el tipo de la lambda: "recibe dos `Int` y devuelve un `Int`". `{ a, b -> a + b }` es la lambda: `a` y `b` son sus parámetros, la flecha `->` los separa del cuerpo, y `a + b` es el resultado (la última expresión de una lambda es su valor de retorno, sin necesidad de `return`). `sumar(3, 4)` la invoca como si fuera una función normal e imprime 7. Las lambdas brillan cuando una función recibe comportamiento como parámetro: "ordena esta lista usando ESTE criterio", "cuando pase ESTO, haz ESTO OTRO". `setOnClickListener` es exactamente eso: recibe el comportamiento a ejecutar al hacer clic.

Kotlin tiene una convención elegante: si la lambda es el último parámetro de una función, puedes escribirla fuera de los paréntesis. Por eso `setOnClickListener { ... }` se ve así y no `setOnClickListener({ ... })`: es la misma llamada, con sintaxis trailing lambda. Cuando la lambda tiene un solo parámetro, puedes omitirlo y usar la palabra mágica `it`:

```
val numeros = listOf(1, 2, 3, 4, 5)  
for (n in numeros) {  
    println(n * 2)  
}
```

Ese `for` lo conoces. Su versión con lambda, que verás mucho en código profesional, usa `forEach`:


```
val numeros = listOf(1, 2, 3, 4, 5)
numeros.forEach {
    println(it * 2)
}
```

forEach recorre la lista y ejecuta la lambda una vez por elemento; it representa el elemento actual en cada vuelta. Al principio it parece magia oscura; con la práctica se vuelve natural y el código queda notablemente más compacto.

Errores comunes de la clase: olvidar el return en funciones con tipo de retorno declarado (el compilador lo exige en cuerpos con llaves); confundir parámetros con argumentos al leer mensajes de error; declarar el tipo de retorno mal (por ejemplo, : Int cuando devuelves un String); intentar usar fuera de la función una variable declarada dentro (problema de alcance); en lambdas, olvidar la flecha -> entre parámetros y cuerpo; y abusar de it en lambdas largas, donde un nombre explícito sería más claro. Buenas prácticas: una función, un propósito (si necesitas la palabra "y" para describir lo que hace tu función, probablemente deberían ser dos); nombres verbo + objeto (calcularTotal, no calculo ni funcion1); prefiere funciones de una sola expresión cuando el cuerpo sea trivial; usa valores por defecto para simplificar las llamadas comunes; y documenta con un comentario // las funciones cuyo propósito no sea obvio por el nombre.

Analogía de cierre: las funciones son como los electrodomésticos de tu cocina. La licuadora (función licuar) recibe fruta y leche (parámetros) y devuelve un licuado (retorno): no necesitas saber cómo gira el motor por dentro, solo qué le metes y qué te da. Puedes usar la misma licuadora cien veces con distintas frutas (reutilización). Y una lambda es como darle a alguien una receta escrita en un papelito: "cuando lleguen los invitados, haz ESTO". No la ejecuta ahora; la guarda y la ejecuta cuando llegue el momento. setOnClickListener guarda tu papelito y lo ejecuta cuando el usuario toca el botón.

Ejemplos paso a paso

Todos los ejemplos de esta sección son Kotlin puro: cópialos dentro de una función fun main() en el Kotlin Playground (busca "Kotlin playground" en tu navegador) y ejecútalos para ver los resultados. Cada ejemplo incluye explicación línea por línea.

Ejemplo 1: val, var y tipos de datos en acción

```
fun main() {  
    val nombre = "Carlos"  
    var puntos = 0  
    val precio: Double = 49.99  
    val esPremium: Boolean = true  
  
    println("Jugador: $nombre")  
    println("Puntos iniciales: $puntos")  
  
    puntos = 10  
    puntos = puntos + 5  
    println("Puntos finales: $puntos")  
    println("Precio premium: $precio, es premium: $esPremium")  
}
```

Línea por línea: `val nombre = "Carlos"` declara una constante de texto con inferencia de tipo (Kotlin deduce `String`). `var puntos = 0` declara una variable entera mutable que empieza en 0. `val precio: Double = 49.99` declara una constante decimal con tipo explícito. `val esPremium: Boolean = true` declara una constante booleana con tipo explícito. Los `println` muestran los valores usando plantillas de string (`$nombre` se reemplaza por el valor). Luego `puntos = 10` reasigna la variable (permitido por ser `var`), y `puntos = puntos + 5` lee el valor actual (10), le suma 5 y guarda 15. El último `println` muestra 15. Si intentaras `nombre = "Juan"`, el compilador marcaría error porque `nombre` es `val`. Ejecuta el programa y verifica que la salida sea: Jugador: Carlos, Puntos iniciales: 0, Puntos finales: 15, y la línea del precio.

Experimento: cambia `var puntos` por `val puntos` y observa el error de compilación en las líneas de reasignación. Luego revuértelo. Así interiorizas la diferencia más importante de Kotlin.

Ejemplo 2: null safety con un caso real

Imagina una app donde el usuario puede tener un apodo opcional. Vamos a manejar ese dato anulable de forma segura:

```

fun main() {
    val nombre: String = "María"
    var apodo: String? = null

    println("Nombre: $nombre")
    println("Apodo con llamada segura: ${apodo?.uppercase()}")
    println("Apodo con Elvis: ${apodo ?: "sin apodo"}")

    apodo = "Mari"
    println("Apodo con llamada segura: ${apodo?.uppercase()}")
    println("Apodo con Elvis: ${apodo ?: "sin apodo"}")
    println("Longitud del nombre: ${nombre.length}")
}

```

Línea por línea: nombre es un String garantizado (no anulable). apodo es String? (anulable) que empieza en null: el usuario aún no tiene apodo. apodo?.uppercase() usa la llamada segura: como apodo es null, no se ejecuta uppercase() y el resultado es null, que se imprime como "null" sin estrellar el programa. apodo ?: "sin apodo" usa el operador Elvis: como apodo es null, produce "sin apodo". Luego asignamos "Mari" a apodo (permitido: es var y String? acepta textos). Ahora apodo?.uppercase() sí se ejecuta y produce "MARI", y el Elvis produce "Mari" porque ya no es null. Finalmente, nombre.length funciona directo porque nombre es no-nulo: el compilador lo garantiza. Observa las llaves \${}: cuando la expresión dentro de la plantilla es más compleja que una simple variable (como apodo?.uppercase()), se envuelve entre \${ }.

La lección del ejemplo: el mismo código maneja con elegancia los dos casos (con apodo y sin apodo), sin un solo if. Esa es la belleza de la null safety bien usada: los casos borde se vuelven triviales.

Ejemplo 3: condicionales para una app de calificaciones

```

fun main() {
    val calificacion = 82

    val resultado = if (calificacion >= 70) {
        "Aprobado"
    } else {
        "Reprobado"
    }
    println("Resultado: $resultado")

    val mencion = when {
        calificacion >= 95 -> "con honores"
        calificacion >= 90 -> "excelente"
        calificacion >= 80 -> "muy bien"
        calificacion >= 70 -> "aprobado"
        else -> "reprobado"
    }
    println("Mención: $mencion")

    val dia = 6
    val tipoDia = when (dia) {
        1, 2, 3, 4, 5 -> "día laboral"
        6, 7 -> "fin de semana"
        else -> "día inválido"
    }
    println("El día $dia es $tipoDia")
}

```

Línea por línea: calificacion vale 82. El if como expresión asigna "Aprobado" a resultado porque $82 \geq 70$; observa que aquí el if usa llaves multilinea pero sigue siendo una expresión que produce un valor. El when sin argumento evalúa condiciones en orden: 82 no es ≥ 95 ni ≥ 90 , sí es ≥ 80 , así que mencion es "muy bien" y las demás ramas se saltan. El when con argumento compara día contra valores: 6 cae en la rama "6, 7" (varios valores separados por coma) y tipoDia es "fin de semana". La salida completa: "Resultado: Aprobado", "Mención: muy bien", "El día 6 es fin de semana". Cambia calificacion a 64 y a 97, ejecuta de nuevo y observa cómo cambian los tres mensajes: así compruebas que las ramas funcionan.

Ejemplo 4: ciclos para procesar una lista de compras

```

fun main() {
    val precios = listOf(120.0, 45.5, 300.0, 89.99)

    var total = 0.0
    for (precio in precios) {
        total += precio
    }
    println("Total con for: $total")

    var i = 0
    var totalWhile = 0.0
    while (i < precios.size) {
        totalWhile += precios[i]
        i++
    }
    println("Total con while: $totalWhile")

    println("Precios con IVA (16%):")
    for (j in precios.indices) {
        val conIva = precios[j] * 1.16
        println("Producto ${j + 1}: $conIva")
    }
}

```

Línea por línea: precios es una lista inmutable de cuatro decimales. total empieza en 0.0 (var porque se acumula). El for recorre cada precio y total += precio suma cada uno al acumulado (es abreviatura de total = total + precio). Resultado: 555.49. La versión con while hace lo mismo con índice manual: i recorre las posiciones de 0 a precios.size - 1; precios[i] accede al elemento en la posición i; i++ incrementa el índice. precios.indices es un rango con todos los índices válidos (0..3), ideal para cuando necesitas la posición además del valor. En el último for, j + 1 convierte el índice base 0 en número de producto base 1 para mostrarlo bonito, y cada precio se multiplica por 1.16 para agregar el IVA. Compara las dos versiones del total: el for es más corto y más difícil de romper (sin riesgo de ciclo infinito); el while te da control manual del índice. En código profesional verás el for (o forEach) el 90 por ciento de las veces.

Ejemplo 5: funciones que resuelven un problema real

Vamos a construir un pequeño "cotizador": calcula el total de una compra con descuento según el tipo de cliente.

```

fun calcularDescuento(tipoCliente: String): Double {
    return when (tipoCliente) {
        "premium" -> 0.20
        "frecuente" -> 0.10
        else -> 0.0
    }
}

fun calcularTotal(precio: Double, cantidad: Int, tipoCliente: String = "nuevo"): Double {
    val subtotal = precio * cantidad
    val descuento = calcularDescuento(tipoCliente)
    return subtotal * (1 - descuento)
}

fun mostrarTicket(producto: String, total: Double) {
    println("Producto: $producto")
    println("Total a pagar: $$total")
    println("-----")
}

fun main() {
    mostrarTicket("Laptop", calcularTotal(15000.0, 1, "premium"))
    mostrarTicket("Mouse", calcularTotal(500.0, 2))
    mostrarTicket("Teclado", calcularTotal(1200.0, 1, tipoCliente = "frecuente"))
}

```

Línea por línea: `calcularDescuento` recibe el tipo de cliente y devuelve el porcentaje de descuento como decimal usando `when` como expresión (premium 20%, frecuente 10%, los demás 0%). `calcularTotal` recibe precio, cantidad y `tipoCliente` con valor por defecto "nuevo": calcula el subtotal, pide el descuento a la otra función (reutilización: una función llamando a otra) y devuelve el subtotal menos el descuento. `mostrarTicket` no devuelve nada (Unit implícito): solo imprime el ticket formateado; observa `$$total`: el primer \$ imprime un signo de pesos literal y `$total` inserta el valor. En `main`: la primera llamada combina todo (cliente premium: $15000 * 0.8 = 12000.0$); la segunda usa el valor por defecto "nuevo" ($500 * 2 = 1000.0$, sin descuento); la tercera usa argumento nombrado `tipoCliente = "frecuente"` ($1200 * 0.9 = 1080.0$). Este ejemplo muestra el diseño profesional: cada función tiene un solo propósito, se combinan como piezas de Lego y `main` queda limpio y legible.

Ejemplo 6: lambdas y el patrón que usarás en Android

```
fun procesarNumeros(numeros: List<Int>, operacion: (Int) -> Int): List<Int> {
    val resultado = mutableListOf<Int>()
    for (n in numeros) {
        resultado.add(operacion(n))
    }
    return resultado
}

fun main() {
    val numeros = listOf(1, 2, 3, 4, 5)

    val dobles = procesarNumeros(numeros, { n -> n * 2 })
    println("Dobles: $dobles")

    val triples = procesarNumeros(numeros) { n -> n * 3 }
    println("Triples: $triples")

    val cuadrados = procesarNumeros(numeros) { it * it }
    println("Cuadrados: $cuadrados")

    numeros.forEach { println("Elemento: $it") }
}
```

Línea por línea: `procesarNumeros` es una función de orden superior: recibe una lista y una lambda (`operacion`) cuyo tipo es `(Int) -> Int` ("recibe un `Int`, devuelve un `Int`"). Dentro, crea una lista mutable vacía, recorre los números, aplica `operacion(n)` a cada uno y guarda el resultado. En `main`: la primera llamada pasa la lambda entre paréntesis con sintaxis completa `{ n -> n * 2 }`. La segunda usa la sintaxis trailing lambda (fuera de los paréntesis) porque la lambda es el último parámetro: `procesarNumeros(numeros) { n -> n * 3 }`. La tercera abrevia el parámetro único con `it`: `{ it * it }`. Las tres producen listas transformadas: `[2, 4, 6, 8, 10]`, `[3, 6, 9, 12, 15]`, `[1, 4, 9, 16, 25]`. El `forEach` final imprime cada elemento usando `it`. Ahora relea el ejemplo del botón de la unidad 1: `btnSaludar.setOnClickListener { txtSaludo.text = "..." }`. ¿Ves el patrón? `setOnClickListener` es una función que recibe una lambda (el comportamiento al hacer clic), escrita con sintaxis trailing lambda, y dentro de la lambda usas las variables de afuera. Ya no es magia: es exactamente lo que acabas de construir tú mismo.

Mini-proyecto de la unidad: juego de adivinanza en consola

Vamos a juntar condicionales, ciclos, variables, funciones y null safety en un juego completo: la computadora piensa un número del 1 al 20 y el jugador tiene 5 intentos para adivinarlo. (En el playground no hay entrada de teclado interactiva real en todos los casos, pero el código está escrito para consola estándar y funciona en cualquier entorno Kotlin con `readLine()`.)

```
fun generarNumeroSecreto(): Int {
    return (1..20).random()
}

fun evaluarIntento(secreto: Int, intento: Int): String {
    return when {
        intento == secreto -> "correcto"
        intento < secreto -> "bajo"
        else -> "alto"
    }
}

fun leerIntento(): Int? {
    print("Adivina el número (1-20): ")
    val entrada = readLine()
    return entrada?.toIntOrNull()
}

fun main() {
    val secreto = generarNumeroSecreto()
    var intentosRestantes = 5
    var gano = false

    println("Piensa un número del 1 al 20. Tienes 5 intentos.")

    while (intentosRestantes > 0 && !gano) {
        val intento = leerIntento()

        if (intento == null) {
            println("Eso no es un número válido. Intenta de nuevo.")
            continue
        }

        when (evaluarIntento(secreto, intento)) {
            "correcto" -> {
                println("Felicidades, adivinaste: era el $secreto")
                gano = true
            }
        }
    }
}
```



```

                "bajo" -> println("Muy bajo. Te quedan ${intentosRestantes - 1}
intentos.")
                "alto" -> println("Muy alto. Te quedan ${intentosRestantes - 1}
intentos.")
            }
            intentosRestantes--
        }

        if (!gano) {
            println("Se acabaron los intentos. El número era $secreto.")
        }
    }
}

```

Cómo funciona, por partes: `generarNumeroSecreto` usa `(1..20).random()` para elegir al azar. `evaluarIntento` compara con `when` y devuelve "correcto", "bajo" o "alto". `leerIntento` pide la entrada con `readLine()` (que devuelve `String?`, anulable porque el usuario podría no escribir nada): `entrada?.toIntOrNull()` usa llamada segura y convierte el texto a número, devolviendo `null` si no es un número válido. En `main`: el `while` sigue mientras queden intentos Y el jugador no haya ganado (operador `&&` y negación `!`). Si el intento es `null` (entrada inválida), `continue` salta a la siguiente vuelta sin restar intento: buena experiencia de usuario. El `when` reacciona al resultado: si es correcto, celebra y marca `gano = true`, lo que terminará el ciclo; si no, da la pista y descuenta un intento. Al salir del ciclo, si no ganó, revela el número.

Este mini-proyecto usa TODO lo de la unidad: `val` y `var` (`secreto` no cambia, `intentosRestantes` sí), tipos (`Int`, `String`, `Boolean`), `null safety` (`String?`, `?.`, `toIntOrNull`), `if` y `when`, `while` con `break` implícito por condición, `continue`, funciones con parámetros y retornos, y valores por defecto implícitos en el diseño. Si entiendes este programa completo, estás listo para la unidad 3. Juega con él: cambia el rango a `1..100`, cambia el número de intentos, agrega una pista de "muy cerca" cuando la diferencia sea de 2 o menos. Modificar un programa que funciona es la mejor práctica que existe.

Práctica

Resuelve cada ejercicio por tu cuenta antes de leer la solución. Escríbelos en el Kotlin Playground y ejecútalos.

Ejercicio 1: El contador de ahorros

Declara una variable `val` con tu nombre y una variable `var` con tu ahorro inicial (por ejemplo 500). Suma tres depósitos (200, 150 y 300) actualizando la variable, y al final imprime: "Ana ahorró un total de \$1150" (con tu nombre y tu total). Explica por qué el nombre es `val` y el ahorro es `var`.

Solución:

```
fun main() {  
    val nombre = "Ana"  
    var ahorro = 500  
    ahorro += 200  
    ahorro += 150  
    ahorro += 300  
    println("$nombre ahorró un total de $$ahorro")  
}
```

El nombre es `val` porque la identidad no cambia durante el programa; el ahorro es `var` porque cada depósito modifica el valor. La salida es "Ana ahorró un total de \$1150". Nota el `$$ahorro`: el primer `$` es el signo de pesos literal y `$ahorro` inserta el valor. El operador `+=` es abreviatura de `ahorro = ahorro + 200`.

Ejercicio 2: El perfil con datos opcionales

Crea variables para el nombre (no nulo), el segundo nombre (anulable, empieza en `null`) y la edad (entero). Imprime el nombre completo usando el operador Elvis para el segundo nombre (si es `null`, muestra solo el primer nombre) y la longitud del segundo nombre usando llamada segura. Luego asigna un segundo nombre y vuelve a imprimir. Explica qué imprime la longitud cuando el segundo nombre es `null`.

Solución:

```

fun main() {
    val primerNombre = "Luis"
    var segundoNombre: String? = null
    val edad = 28

    val nombreCompleto = "$primerNombre ${segundoNombre ?: ""}.trim()"
    println("Nombre: $nombreCompleto, edad: $edad")
    println("Longitud del segundo nombre: ${segundoNombre?.length}")

    segundoNombre = "Alberto"
    println("Nombre: $primerNombre ${segundoNombre ?: ""}")
    println("Longitud del segundo nombre: ${segundoNombre?.length}")
}

```

Cuando `segundoNombre` es `null`, `segundoNombre?.length` no ejecuta `length` y produce `null`, que se imprime como `"null"` sin error. El Elvis `${segundoNombre ?: ""}` sustituye el `null` por texto vacío para armar el nombre completo. Después de asignar `"Alberto"`, la longitud es 7. La lección: con `?.` y `?:` el código maneja el caso `null` sin un solo `if`.

Ejercicio 3: El semáforo

Escribe un programa que reciba el color del semáforo ("rojo", "amarillo", "verde") en una variable y, usando `when`, imprima la acción: "Detente", "Prepárate", "Avanza". Si el color no es válido, imprime "Color desconocido". Hazlo una vez con `when` como expresión guardando el resultado en una variable. Prueba con los cuatro casos.

Solución:

```

fun main() {
    val color = "amarillo"

    val accion = when (color) {
        "rojo" -> "Detente"
        "amarillo" -> "Prepárate"
        "verde" -> "Avanza"
        else -> "Color desconocido"
    }
    println("Semáforo en $color: $accion")
}

```

Con `"amarillo"` imprime `"Semáforo en amarillo: Prepárate"`. Al ser una expresión que produce un valor, la rama `else` es obligatoria: cubre cualquier texto inesperado. Cambia color a `"verde"`,

"rojo" y "morado" para verificar las cuatro ramas. Este patrón (when como expresión con else) es el estándar profesional para mapear valores a resultados.

Ejercicio 4: La tabla de multiplicar

Usa un ciclo for para imprimir la tabla de multiplicar del 7, del 1 al 10, con el formato "7 x 3 = 21". Luego haz lo mismo con un while. Explica en qué se diferencian las dos versiones y cuál prefieres y por qué.

Solución:

```
fun main() {
    println("Con for:")
    for (i in 1..10) {
        println("7 x $i = ${7 * i}")
    }

    println("Con while:")
    var j = 1
    while (j <= 10) {
        println("7 x $j = ${7 * j}")
        j++
    }
}
```

La versión for declara el rango 1..10 y la variable i se actualiza sola: imposible olvidar el incremento. La versión while requiere declarar j, verificar j <= 10 y no olvidar j++ (si lo olvidas, ciclo infinito). Para un número fijo de repeticiones conocido de antemano, el for es superior: más corto, más claro y más seguro. El while se reserva para cuando no sabes cuántas vueltas necesitarás.

Ejercicio 5: La calculadora con funciones

Crea cuatro funciones: sumar, restar, multiplicar y dividir, cada una recibe dos Double y devuelve Double. La función dividir debe manejar la división entre cero devolviendo 0.0 y mostrando un mensaje de advertencia. Crea una función mostrarResultado que reciba el nombre de la operación y el resultado y lo imprima bonito. En main, prueba las cuatro operaciones, incluyendo una división entre cero.

Solución:

```

fun sumar(a: Double, b: Double): Double = a + b
fun restar(a: Double, b: Double): Double = a - b
fun multiplicar(a: Double, b: Double): Double = a * b

fun dividir(a: Double, b: Double): Double {
    if (b == 0.0) {
        println("Advertencia: no se puede dividir entre cero")
        return 0.0
    }
    return a / b
}

fun mostrarResultado(operacion: String, resultado: Double) {
    println("$operacion = $resultado")
}

fun main() {
    mostrarResultado("10 + 5", sumar(10.0, 5.0))
    mostrarResultado("10 - 5", restar(10.0, 5.0))
    mostrarResultado("10 * 5", multiplicar(10.0, 5.0))
    mostrarResultado("10 / 5", dividir(10.0, 5.0))
    mostrarResultado("10 / 0", dividir(10.0, 0.0))
}

```

Las tres primeras usan sintaxis de una sola expresión (= en lugar de llaves y return). dividir usa cuerpo con llaves porque tiene lógica de validación: si el divisor es cero, avisa y devuelve 0.0 en lugar de estrellarse o devolver infinito. mostrarResultado es Unit (no devuelve nada). La salida muestra los cuatro resultados correctos más la advertencia en la división entre cero. Lección: las funciones encapsulan reglas del negocio (como "no dividir entre cero") en un solo lugar.

Ejercicio 6: El transformador con lambdas

Crea una función aplicarATodos que reciba una lista de Int y una lambda (Int) -> Int, y devuelva la lista transformada. Úsala tres veces con trailing lambda e it: una para duplicar, una para sumar 100, y una para quedarte solo con números pares convertidos a su mitad (pista: dentro de la lambda puedes usar if como expresión). Imprime los tres resultados.

Solución:

```

fun aplicarATodos(numeros: List<Int>, transformacion: (Int) -> Int): List<Int> {
    val resultado = mutableListOf<Int>()
    for (n in numeros) {
        resultado.add(transformacion(n))
    }
    return resultado
}

fun main() {
    val numeros = listOf(10, 21, 30, 45, 50)

    val duplicados = aplicarATodos(numeros) { it * 2 }
    println("Duplicados: $duplicados")

    val masCien = aplicarATodos(numeros) { it + 100 }
    println("Más cien: $masCien")

    val mitadesDePares = aplicarATodos(numeros) { if (it % 2 == 0) it / 2 else it }
    println("Mitades de pares: $mitadesDePares")
}

```

La función es genérica: no sabe qué transformación hará, solo aplica la lambda que recibe. Las llamadas usan trailing lambda con `it` para el parámetro único. La tercera lambda usa `if` como expresión dentro: si el número es par (`it % 2 == 0`, el operador `%` da el residuo de la división), devuelve su mitad; si no, lo deja igual. Resultados: `[20, 42, 60, 90, 100]`, `[110, 121, 130, 145, 150]`, `[5, 21, 15, 45, 25]`. Este es el poder de las funciones de orden superior: una sola función, comportamientos infinitos.

Profundización: colecciones, las cajas con compartimentos

En los ejemplos usaste `listOf` varias veces sin detenernos a estudiarlo, y merece su sección porque las colecciones son el pan de cada día en las apps: la lista de contactos, el catálogo de productos, los mensajes del chat, todo son colecciones. Kotlin ofrece tres tipos principales, y elegir el correcto es una decisión de diseño.

La lista (`List`) es una colección ordenada donde cada elemento tiene una posición (índice, empezando en 0) y puede haber duplicados. `listOf` crea una lista inmutable (no puedes agregar ni quitar elementos); `mutableListOf` crea una mutable:

```
fun main() {
    val frutas = listOf("manzana", "banana", "naranja")
    println(frutas[0])
    println(frutas.size)

    val compras = mutableListOf("pan", "leche")
    compras.add("huevos")
    compras.remove("pan")
    println(compras)
}
```

`frutas[0]` accede al primer elemento ("manzana"); `frutas.size` dice cuántos hay (3). En la lista mutable, `add` agrega al final y `remove` quita el elemento. Intenta `compras.add` en una lista de `listOf` y el compilador te lo impedirá: esa es la gracia de la inmutabilidad, proteger datos que no deben cambiar. Regla práctica: usa `listOf` por defecto y `mutableListOf` solo cuando necesites modificar la colección.

El conjunto (Set) es una colección sin orden garantizado y sin duplicados: si agregas un elemento repetido, simplemente no se duplica. Es perfecto para casos como "la lista de etiquetas de un producto" o "los IDs de usuarios conectados", donde lo importante es la pertenencia, no el orden:

```
fun main() {
    val etiquetas = mutableSetOf("oferta", "nuevo")
    etiquetas.add("oferta")
    etiquetas.add("popular")
    println(etiquetas)
    println("Tiene 'nuevo': ${"nuevo" in etiquetas}")
}
```

Aunque agregamos "oferta" dos veces, el set contiene cada valor una sola vez. El operador `in` verifica pertenencia de forma legible: "nuevo" `in` etiquetas se lee casi como español.

El mapa (Map) guarda pares clave-valor: cada valor se encuentra por su clave, como un diccionario (la palabra es la clave, la definición es el valor). Es la estructura ideal para configuraciones, perfiles de usuario o cachés:

```

fun main() {
    val capitales = mapOf("México" to "Ciudad de México", "España" to "Madrid")
    println(capitales["México"])

    val puntuaciones = mutableMapOf("Ana" to 100)
    puntuaciones["Luis"] = 85
    println(puntuaciones["Luis"])
    println("Jugadores: ${puntuaciones.keys}")
}

```

"clave" to "valor" crea cada par. capitales["México"] busca por clave y devuelve "Ciudad de México". Si la clave no existe, el resultado es null (¡otra aparición de la null safety en la vida real!): por eso capitales["Francia"] ?: "desconocida" es el patrón correcto. En el mapa mutable, puntuaciones["Luis"] = 85 agrega o actualiza. puntuaciones.keys te da todas las claves.

Las colecciones se combinan de maravilla con lo que ya sabes: for recorre listas, when clasifica elementos, las lambdas las transforman. De hecho, Kotlin ofrece funciones de colección con lambdas que reemplazan ciclos enteros, y las verás en unidades futuras (filter, map, sortedBy). Por ahora quédate con la base sólida: crear listas, mapas y sets, acceder a sus elementos y elegir mutable solo cuando lo necesites.

Depurando tu lógica: cuando el programa corre pero se equivoca

Hay dos tipos de errores: los de compilación (el programa no arranca y el compilador te dice dónde) y los lógicos (el programa arranca pero hace algo incorrecto). Los segundos son más traicioneros porque nadie te avisa: solo notas que el resultado está mal. La técnica más simple y efectiva para cazarlos es el println estratégico: imprime los valores de tus variables en los puntos clave y compara lo que esperabas con lo que realmente pasa.

```

fun calcularPromedio(a: Double, b: Double, c: Double): Double {
    println("Entradas: a=$a, b=$b, c=$c")
    val suma = a + b + c
    println("Suma: $suma")
    val promedio = suma / 3
    println("Promedio: $promedio")
    return promedio
}

```

Si el promedio sale mal, estos println te dicen exactamente en qué paso se desvió el cálculo. Los profesionales usan el depurador de Android Studio (puntos de interrupción que pausan la ejecución para inspeccionar variables), que verás en unidades futuras; mientras tanto, el

println es tu mejor amigo y no tiene nada de vergonzoso: hasta los programadores senior lo usan para diagnósticos rápidos.

Los sospechosos habituales de los errores lógicos en esta etapa: condiciones invertidas (usaste > cuando era <), el off-by-one en ciclos (1..10 vs 0 until 10), variables que no se reinician entre vueltas, el == vs = en comparaciones, y asumir que una variable anulable tiene valor sin verificarlo. Ante un resultado extraño, no cambies código al azar: formula una hipótesis ("creo que el problema está en el cálculo del descuento"), agrega un println para comprobarla, y solo entonces corrige. Depurar es el método científico aplicado al código: observa, hipotetiza, experimenta, concluye.

Glosario de la unidad

- Alcance (scope): la región del código donde una variable existe y puede usarse; lo de adentro de una función no se ve desde afuera.
- Argumento: el valor concreto que pasas al llamar a una función.
- Booleano (Boolean): tipo de dato con solo dos valores, true y false; controla todas las decisiones.
- Bucle/Ciclo: estructura que repite un bloque de código; for (repeticiones conocidas), while (condición).
- Ciclo infinito: while cuya condición nunca se vuelve false; el programa se queda dando vueltas para siempre.
- Condicional: estructura que elige qué código ejecutar según una condición (if, when).
- Expresión: fragmento de código que produce un valor (if y when pueden usarse como expresiones).
- Función: bloque de código reutilizable con nombre, parámetros y opcionalmente un valor de retorno; se declara con fun.
- Función de orden superior: función que recibe otra función (lambda) como parámetro.
- Inferencia de tipos: el compilador deduce el tipo de la variable a partir del valor asignado.
- Lambda: función sin nombre escrita entre llaves, que se puede guardar o pasar como argumento; it es el parámetro implícito cuando hay uno solo.
- Lista/Mapa/Set: colecciones de datos; lista ordenada con duplicados, mapa de pares clave-valor, set sin duplicados.
- Null: valor especial que significa ausencia de valor; distinto de "" y de 0.

- Null safety: sistema de Kotlin que impide usar valores null sin protección explícita (? y ?:).
- Operador Elvis (?:): devuelve el valor de la izquierda, o el de la derecha si el de la izquierda es null.
- Parámetro: la variable declarada en la definición de la función que recibe los argumentos.
- Plantilla de string: texto con \$variable o \${expresion} que Kotlin reemplaza por sus valores.
- Rango: secuencia de valores como 1..5, 1 until 10 o 10 downTo 1, usada con for.
- Retorno: el valor que una función entrega a quien la llamó, con la palabra return.
- Trailing lambda: convención de escribir la lambda fuera de los paréntesis cuando es el último parámetro.
- val/var: palabras clave para declarar variables de solo lectura (val) o mutables (var).

Autoevaluación

Responde sin mirar el texto. Si fallas alguna, regresa a la sección indicada.

1. ¿Cuándo usas val y cuándo var? ¿Cuál es la regla de oro? (Clase 4)
2. ¿Qué significa String? y en qué se diferencia de String? (Clase 4)
3. Escribe una línea que muestre el nombre de usuario o "Invitado" si la variable es null. (Clase 4)
4. ¿Por qué un when usado como expresión exige la rama else? (Clase 5)
5. ¿Qué imprime este código y por qué? `for (i in 1..5) { if (i == 3) continue; println(i) }` (Clase 5)
6. ¿Qué es una lambda y cómo se relaciona con `setOnClickListener`? (Clase 6)

Respuestas: 1) val para valores que no cambian, var para los que sí; la regla de oro es usar val por defecto y var solo con una razón concreta, porque los valores inmutables son más fáciles de razonar y generan menos errores. 2) String? es el tipo anulable: acepta un texto o null; String (sin ?) garantiza que nunca es null y el compilador te impide asignarle null. 3) `println(usuario ?: "Invitado")` o `val nombre = usuario ?: "Invitado"`: el operador Elvis usa el valor de la izquierda salvo que sea null, en cuyo caso usa el de la derecha. 4) Porque como expresión debe producir un valor en todos los casos posibles; sin else habría entradas sin resultado definido y el compilador lo rechaza. 5) Imprime 1, 2, 4, 5: el continue salta la vuelta cuando i vale 3, así que el 3 nunca se imprime. 6) Una lambda es una función sin nombre que se pasa como argumento; `setOnClickListener` recibe una lambda (el código a ejecutar al tocar)

escrita con sintaxis trailing lambda, por eso se ve como `setOnClickListener { ... }` en lugar de llevar paréntesis.

Si respondiste bien al menos 5 de 6, dominas los fundamentos y estás listo para construir interfaces reales en la siguiente parte del libro. Si no, repasa: cada concepto de esta unidad aparecerá decenas de veces más adelante.

Preguntas frecuentes de la unidad

¿Siempre debo usar `val`? ¿Nunca `var`? Casi siempre `val`. Usa `var` solo para lo que genuinamente cambia con el tiempo: contadores, acumuladores, estados que evolucionan (como `intentosRestantes` en el juego). Si dudas, empieza con `val`: el compilador te avisará si intentas reasignarla y ahí decides con información, no por costumbre.

¿De verdad nunca debo usar `!!`? Casi nunca. La aserción no nula existe para casos raros de interoperabilidad con Java donde tú sabes algo que el compilador no puede saber. En tu código normal, siempre hay una alternativa mejor: `?.` para operar con seguridad, `?:` para valores por defecto, o un `if` que verifique explícitamente. Cada `!!` es una apuesta; con el tiempo, alguna se pierde.

¿Cuándo uso `if` y cuándo `when`? Usa `if` para una o dos condiciones simples. Usa `when` cuando tengas tres o más ramas sobre el mismo valor o tema: el código queda más legible y es más difícil equivocarse en el orden de las condiciones. Si tu cadena de `else if` ya ocupa media pantalla, es hora de un `when`.

¿`for` o `forEach` con lambda? Para esta etapa, domina el `for` clásico: es explícito y fácil de depurar. El `forEach` con `it` es el estilo profesional compacto y lo verás mucho en código real, así que acostúmbrate a leerlo desde ahora, pero no te obligues a escribirlo todo con lambdas todavía. La legibilidad manda sobre la elegancia cuando estás aprendiendo.

No termino de entender las lambdas, ¿es grave? No, es normal. Las lambdas son el concepto más abstracto de la unidad. Lo importante hoy es reconocer el patrón: "una función recibe un bloque de código para ejecutarlo después", como `setOnClickListener`. Con cada unidad que uses botones, listas y eventos, las lambdas se volverán familiares por repetición. Vuelve a releer la clase 6 después de la unidad 3 y verás cuánto más claro se ve.

¿Cómo práctico más allá de los ejercicios? Tres ideas: reescribe cada ejemplo con tus propias variaciones (cambia valores, agrega ramas, rompe cosas a propósito y arréglalas); inventa mini-programas de consola (una calculadora de propinas, un conversor de monedas, un generador de contraseñas); y explica cada concepto en voz alta como si enseñaras a alguien: lo que no puedas explicar con claridad es lo que necesitas repasar.

Resumen de la unidad

- Una variable es un espacio con nombre en memoria para guardar datos; las analogías de cajas etiquetadas ayudan a razonar sobre ellas.
- val declara variables de solo lectura (no cambian); var declara variables mutables (sí cambian); usa val por defecto y var solo con razón.
- Tipos básicos: Int (enteros), Double (decimales), String (textos), Boolean (true/false), Char (un carácter); Kotlin infiere el tipo cuando es obvio.
- Kotlin es de tipado fuerte: no puedes guardar un String en un Int; usa conversiones explícitas como toInt() y toString().
- null significa ausencia de valor; es distinto de "" (texto vacío) y de 0 (número cero).
- Por defecto las variables no aceptan null; el tipo anulable se marca con ? (String? acepta texto o null).
- La llamada segura ?. ejecuta la operación solo si el valor no es null; el operador Elvis ?: provee un valor por defecto ante null; evita !! salvo casos excepcionales.
- if evalúa una condición Boolean y ejecuta un bloque u otro; == compara igualdad, = asigna (no los confundas).
- Los operadores lógicos && (Y), || (O) y ! (NO) combinan condiciones; if también funciona como expresión que produce un valor.
- when maneja decisiones múltiples con elegancia: por valor, por rangos, por condiciones; como expresión requiere la rama else.
- for recorre rangos (1..5, 1 until 10, 10 downTo 1) y colecciones (for (fruta in frutas)); while repite mientras una condición sea true; do-while garantiza al menos una ejecución.
- Las plantillas de string (\$variable y \${expresion}) insertan valores en textos de forma legible.
- break sale del ciclo, continue salta a la siguiente vuelta; verifica siempre qué hace terminar a un while para evitar ciclos infinitos.
- Las funciones se declaran con fun, reciben parámetros, devuelven valores con return; parámetro es en la definición, argumento en la llamada.
- Los parámetros pueden tener valores por defecto y los argumentos pueden nombrarse (orden libre, código autodocumentado).
- Las funciones de una sola expresión usan = en lugar de llaves y return; las variables internas solo existen dentro de su función (alcance).

- Una lambda es una función sin nombre que se pasa como argumento; la sintaxis trailing lambda e it la hacen compacta; `setOnClickListener` es el ejemplo que ya usabas.
- Buenas prácticas: una función un propósito, nombres verbo + objeto, val por defecto, `when` sobre `else-if` largos, `for` sobre `while` para repeticiones fijas.

Lo que sigue en el libro

Con esta primera parte completaste los cimientos: entiendes la plataforma Android, tienes tu taller montado con Android Studio, creaste tus primeras apps y dominas los fundamentos de Kotlin. Todo lo que viene se construye directamente sobre esto, así que si algo de estas dos unidades te quedó tambaleando, este es el momento de repasarlo, no después.

En la siguiente parte del libro entrarás al territorio donde las apps cobran vida de verdad: el diseño de interfaces con layouts modernos, los componentes visuales que usan las apps profesionales (listas, tarjetas, menús, diálogos), la navegación entre pantallas y el ciclo de vida de las actividades. Ahí vas a descubrir que el XML y el Kotlin que aprendiste aquí eran solo el vocabulario: ahora empezarás a escribir frases completas.

Un consejo para el camino: conserva todos tus proyectos de esta parte. Cuando dentro de unas semanas construyas pantallas complejas, volver a tu "Hola Mundo" y a tu tarjeta de presentación te recordará cuánto avanzaste. Y cuando algo no funcione (porque algo no funcionará, te lo garantizo), regresa a estas páginas: la mayoría de los errores "misteriosos" de los niveles avanzados tienen su raíz en un fundamento de estas dos unidades que se entendió a medias.

Nos vemos en la siguiente parte. Sigue construyendo.

UNIDAD 3: Actividades y su ciclo de vida

Bienvenido a la unidad donde tu aplicación cobra vida de verdad. En la unidad anterior montaste el taller: instalaste Android Studio, creaste tu primer proyecto y viste cómo se ve un proyecto por dentro. Ahora vas a conocer al actor principal de toda aplicación Android: la Activity. Si tu app fuera una obra de teatro, las actividades serían los actores que salen al escenario; si fuera una casa, cada actividad sería un cuarto diferente por el que camina el usuario.

En esta unidad vas a recorrer las clases 7, 8 y 9 del curso. La clase 7 responde la pregunta más básica y a la vez más importante: qué es una Activity y cómo se crea. Vas a entender que una Activity no es otra cosa que una pantalla con la que el usuario interactúa, pero que por dentro es una clase de Kotlin con reglas muy específicas impuestas por el sistema operativo. Aprenderás a declararla en el manifiesto, a conectarla con su layout XML y a lanzarla desde código. Al terminar esta clase sabrás crear una pantalla nueva desde cero sin depender de plantillas.

La clase 8 es el corazón de la unidad: el ciclo de vida de la Activity. Aquí está la diferencia entre alguien que "hace apps que funcionan" y alguien que hace apps que funcionan bien. Android no deja que tu actividad viva para siempre haciendo lo que quiera; el sistema la prende, la pausa, la detiene y la destruye según lo que el usuario haga y según los recursos del teléfono. Cada uno de esos momentos dispara un método: `onCreate`, `onStart`, `onResume`, `onPause`, `onStop` y `onDestroy`. Si no entiendes este ciclo, tu app perderá datos cuando el usuario rote el teléfono, seguirá gastando batería en segundo plano o se cerrará de forma extraña. Te voy a explicar cada método con una analogía que no se te va a olvidar: la de un restaurante, donde tu actividad es el cocinero y Android es el gerente que le dice cuando empezar, cuando hacer pausa y cuando apagar la estufa.

La clase 9 cierra la unidad con dos habilidades prácticas que usarás en el 90 por ciento de tus apps: pasar datos entre actividades con `Intent` y extras, y guardar el estado de la pantalla con `onSaveInstanceState`. Pasar datos es como mandarle una nota a la siguiente pantalla: "oye, el usuario se llama María y pidió dos tacos". Guardar el estado es como tomarle una foto a tu escritorio antes de que alguien lo desordene, para poder dejarlo igual después. Al terminar sabrás construir flujos de varias pantallas que se comunican entre sí y que sobreviven a rotaciones de pantalla sin perder lo que el usuario escribió.

Por que importa tanto esta unidad. Piénsalo así: toda aplicación que uses todos los días, WhatsApp, YouTube, tu banco, está hecha de actividades que aparecen, se pausan y desaparecen siguiendo exactamente estas reglas. Los errores más comunes de los

programadores principiantes en Android nacen aquí: poner código pesado en el método equivocado, olvidar guardar el estado, lanzar actividades sin declararlas en el manifiesto o pasar datos de la forma incorrecta. Cada uno de esos errores se siente como una app "bugueada", lenta o que pierde información. Dominar esta unidad te pone por encima de la mayoría de los tutoriales rápidos de internet, porque aquí vas a entender el por qué, no solo el cómo.

Que sabrás al terminar. Serás capaz de crear actividades nuevas y declararlas correctamente, explicar con tus palabras qué hace cada método del ciclo de vida y en qué orden se llaman, lanzar una segunda pantalla pasándole datos con un Intent, recuperar esos datos del otro lado, guardar y restaurar el estado de la interfaz cuando el sistema destruye y recrea tu actividad, y detectar los errores típicos del ciclo de vida antes de que se conviertan en bugs. En resumen: tus pantallas dejarán de ser dibujos estáticos y se convertirán en piezas vivas que responden al usuario y al sistema.

Una nota sobre el código de esta unidad. Todo el código Kotlin está escrito en Kotlin moderno (1.9 en adelante) y pensado para que compile en concepto dentro de un proyecto estándar de Android Studio. Cuando necesitemos tocar vistas desde código te mostraré las dos formas clásicas: findViewById, que es la base que todo desarrollador debe conocer, y View Binding, que es la forma moderna y recomendada. No necesitas librerías externas ni configuraciones raras: solo Android Studio y lo que ya instalaste en la unidad anterior.

Mi recomendación para estudiar esta unidad: no la leas como novela. Lee una sección, abre Android Studio y prueba el código. Rota el emulador con Ctrl+F11 y observa que métodos se llaman. Pon mensajes en el Logcat y mira el ciclo de vida pasar frente a tus ojos. La programación Android se aprende con las manos en el teclado, y esta unidad es el lugar perfecto para empezar ese hábito. Vamos paso a paso, como en clase, sin prisa pero sin pausa.

Explicación detallada

Qué es una Activity

Una Activity es una pantalla de tu aplicación con la que el usuario puede interactuar. Eso es todo, y a la vez es muchísimo. Cada vez que abres una app y ves una pantalla con botones, textos o imágenes, casi seguro estás viendo una Activity. La pantalla de login es una Activity. La lista de contactos es otra Activity. La pantalla de detalle de un contacto es una tercera. Una app típica tiene entre cinco y cincuenta actividades, y todas siguen las mismas reglas que vas a aprender aquí.

Técnicamente, una Activity es una clase de Kotlin que hereda de la clase Activity de Android (en la práctica casi siempre de AppCompatActivity, que es una versión mejorada y compatible con teléfonos viejos). Heredar significa que tu clase recibe gratis todo el comportamiento que Android ya programó: saber dibujarse en pantalla, recibir toques del usuario, pausarse cuando llega una llamada, etc. Tú solo agregas lo que hace única a tu pantalla.

Analogía de la vida real: piensa en una Activity como un cuarto de una casa. La casa es tu aplicación. Cada cuarto tiene una función: la cocina sirve para cocinar, la recámara para dormir. No puedes estar en dos cuartos al mismo tiempo; estás en uno, y cuando caminas a otro, el primero queda atrás. Igual pasa con las actividades: el usuario ve una a la vez, y cuando navega a otra, la anterior queda en pausa o detenida en segundo plano. Y así como cada cuarto necesita estar registrado en los planos de la casa para que exista legalmente, cada Activity debe estar declarada en el archivo AndroidManifest.xml para que el sistema la reconozca.

El manifiesto es como el acta de nacimiento de tus pantallas. Si creas una Activity en código pero olvidas declararla en el manifiesto, la app se cerrará con un error en cuanto intentes abrirla. Este es el error número uno de los principiantes, y Android Studio normalmente te ayuda declarándola automáticamente cuando creas la Activity con el asistente, pero si la creas a mano debes acordarte tú. Más adelante verás exactamente cómo se ve esa declaración.

La Activity y su layout: dos mitades de lo mismo

Toda Activity tiene dos partes que trabajan juntas: la lógica en Kotlin y el diseño en XML. El archivo XML describe cómo se ve la pantalla (qué botones hay, dónde están los textos, de qué color), y la clase Kotlin describe qué hace (qué pasa cuando tocan el botón, qué texto se muestra). La conexión entre ambas se hace en el método onCreate con una línea que verás mil veces en tu carrera:

```
setContentView(R.layout.activity_main)
```

Esta línea le dice a Android: "dibuja en esta pantalla el diseño que está descrito en el archivo activity_main.xml". Es como colgar un plano en la pared del cuarto: el cuarto existe, pero el plano dice dónde va cada mueble.

Para tocar las vistas desde Kotlin necesitas obtener una referencia a cada elemento del layout. La forma clásica es `findViewById`: le das el id que le pusiste en el XML y te devuelve el objeto para manipularlo. La forma moderna es View Binding, que genera automáticamente una clase con todas tus vistas listas para usarse, sin búsquedas manuales y sin riesgo de equivocarte de id. View Binding es la recomendación actual de Google y es lo que usaremos en los ejemplos principales, pero te explicaré `findViewById` porque lo verás en muchísimo código existente y en exámenes.

Error común de principiante número dos: llamar a `findViewById` antes de `setContentView`. Es como querer agarrar un mueble de un cuarto que todavía no has construido: la app se cae con un `NullPointerException` porque las vistas aún no existen. La regla de oro es simple: primero `setContentView`, después busca tus vistas. Siempre en ese orden.

Quién manda: tú o el sistema

Aquí viene una idea que al principio cuesta trabajo aceptar: en Android, tú no controlas cuándo se crea o se destruye tu Activity. La controla el sistema operativo. Tú escribes los métodos del ciclo de vida, pero es Android quien decide cuándo llamarlos. Esto se llama inversión de control, y es como la diferencia entre manejar tu propio coche y subirte a un taxi: en el taxi tú dices a dónde quieres ir (escribes qué debe pasar en cada método), pero el taxista decide la ruta y los tiempos (el sistema decide cuándo llamar cada método).

¿Por qué Android hace esto? Porque un teléfono tiene recursos limitados: batería, memoria RAM, procesador. Si cada app hiciera lo que quisiera, la batería duraría una hora. Entonces Android actúa como un gerente estricto: cuando tu actividad ya no se ve, la pausa; si necesita memoria, la destruye; cuando el usuario regresa, la recrea. Tu trabajo es escribir código que se porte bien en cada una de esas situaciones. Eso es exactamente lo que te enseña el ciclo de vida.

El ciclo de vida: la analogía del restaurante

Imagina que tu Activity es el cocinero de un restaurante y Android es el gerente. El gerente no cocina, pero decide todo: cuándo entra el cocinero a la cocina, cuándo puede empezar a cocinar, cuándo debe hacer una pausa porque llegó una inspección, cuándo puede volver y cuándo debe apagar la estufa e irse a su casa. El cocinero no discute: simplemente hace lo correcto en cada momento. Los métodos del ciclo de vida son las órdenes del gerente, y tu código es lo que el cocinero hace al recibir cada orden.

Vamos método por método, en el orden en que normalmente se llaman cuando el usuario abre tu app por primera vez.

onCreate: la inauguración del restaurante

onCreate se llama una sola vez en la vida de la Activity, cuando se crea. Es el momento de la inauguración: el cocinero entra por primera vez a la cocina, acomoda sus cuchillos, enciende la estufa y cuelga el menú en la pared. Aquí haces la configuración inicial que solo necesita hacerse una vez: llamar a setContentView para dibujar el layout, obtener referencias a las vistas con View Binding o findViewById, configurar listeners de botones, inicializar datos y leer los extras que te mandó la actividad anterior.

Recibe un parámetro llamado savedInstanceState de tipo Bundle. Si es la primera vez que se crea la actividad, este parámetro es nulo. Si la actividad se está recreando después de que el sistema la destruyó (por ejemplo, al rotar el teléfono), aquí viene guardado el estado anterior. En la clase 9 verás cómo usarlo.

Error común: poner trabajo pesado en onCreate, como descargar archivos de internet o leer una base de datos enorme en el hilo principal. Es como si el cocinero, en plena inauguración con clientes esperando, se pusiera a ir al mercado a comprar verduras. La app se congela y Android puede mostrar el temido mensaje "la aplicación no responde". La regla es: onCreate debe ser rápido; el trabajo pesado va en segundo plano.

onStart: abrir las puertas al público

Después de onCreate viene onStart. El restaurante ya está listo y ahora se abren las puertas: los clientes pueden ver el local, aunque el cocinero todavía no está cocinando. En términos técnicos, la Activity ya es visible en pantalla pero aún no tiene el foco, es decir, el usuario todavía no puede interactuar con ella. Aquí se hacen preparativos finales de visualización: por ejemplo, registrar observadores que actualizan la interfaz.

onStart también se llama cuando el usuario regresa a una actividad que estaba detenida (en onStop). Es como reabrir el restaurante después de haberlo cerrado un rato: hay que volver a prender las luces.

onResume: a cocinar, el restaurante está lleno

onResume significa que la Activity está al frente, visible y lista para interactuar con el usuario. El cocinero está cocinando a todo vapor y los meseros toman órdenes. Este es el estado donde tu actividad pasa la mayor parte de su vida útil. Aquí debes iniciar lo que solo tiene sentido cuando el usuario te está viendo: animaciones, reproducción de video, sensores, actualizaciones en tiempo real.

Buena práctica: todo lo que inicies en `onResume` deberías detenerlo en `onPause`. Son pareja: se abren y se cierran juntos. Si prendes la cámara en `onResume` y no la apagas en `onPause`, seguirás gastando batería aunque el usuario ya esté en otra app. El gerente (Android) no te va a regañar de inmediato, pero el usuario verá cómo su batería se drena y desinstalará tu app.

`onPause`: llegó la inspección, pausa todo

`onPause` se llama cuando la actividad deja de estar al frente pero todavía se ve parcialmente. El caso típico: llega una llamada o aparece un diálogo encima de tu pantalla. El cocinero debe soltar el cuchillo de inmediato: pausar el video, guardar datos no guardados, apagar sensores. `onPause` debe ser rapidísimo, porque el sistema no muestra la siguiente actividad hasta que tu `onPause` termina. Si tardas aquí, la transición entre pantallas se siente lenta y trabada.

Ojo con un detalle importante: `onPause` no significa que la actividad vaya a destruirse. Muchas veces solo es una pausa temporal y luego viene `onResume` de nuevo. Por eso aquí no liberas recursos grandes de forma definitiva; solo haces pausas ligeras. Es como ponerle pausa a la licuadora, no como desarmarla.

`onStop`: se cierra el restaurante por hoy

En `onStop` la actividad ya no es visible para nada. El usuario abrió otra app o regresó a la pantalla anterior. El restaurante cerró sus puertas: ya no hay clientes adentro. Aquí detienes lo que consume recursos y no necesita seguir corriendo: animaciones, actualizaciones de ubicación, conexiones. También es un buen lugar para guardar información en la base de datos o en preferencias, porque después de `onStop` el sistema puede destruir tu actividad en cualquier momento sin avisar.

Analogía para no olvidar: `onPause` es "baja el cuchillo un momento", `onStop` es "apaga la estufa y vete". En `onPause` la actividad todavía se ve un poco; en `onStop` ya no se ve nada.

`onDestroy`: demolición del local

`onDestroy` es el final: la actividad se destruye. Puede ser porque el usuario la cerró definitivamente (con el botón atrás desde la primera pantalla) o porque el sistema necesitó memoria. Aquí liberas los últimos recursos: cierras conexiones, cancelas tareas. Después de `onDestroy`, la actividad ya no existe.

Pero atención: `onDestroy` no siempre se llama. Si el sistema está muy apurado de memoria, puede matar tu proceso sin pasar por `onDestroy`. Por eso nunca pongas aquí código crítico del que dependa la integridad de los datos del usuario. Lo crítico (guardar un borrador, por

ejemplo) debe ir en `onPause` u `onStop`, que sí tienen garantía de llamarse en el camino normal.

onRestart: la reapertura

Hay un método más que completa el cuadro: `onRestart`. Se llama cuando una actividad que estaba en `onStop` va a volver a mostrarse. Es la reapertura del restaurante: antes de abrir las puertas (`onStart`) hay que avisar que se va a reabrir. En la práctica casi nunca necesitas sobrescribirlo, pero debes saber que existe para entender los diagramas del ciclo de vida.

El orden completo, de memoria

Apertura normal: `onCreate`, `onStart`, `onResume`. El usuario toca el botón de inicio o abre otra app: `onPause`, `onStop`. El usuario regresa: `onRestart`, `onStart`, `onResume`. El usuario gira el teléfono: `onPause`, `onStop`, `onDestroy`, y luego todo de nuevo con `onCreate`, `onStart`, `onResume`, porque el sistema destruye y recrea la actividad. El usuario presiona atrás para salir: `onPause`, `onStop`, `onDestroy`, y no hay recreación.

Memoriza esa secuencia de la rotación, porque es la causa del bug más famoso de Android: al girar el teléfono, todo lo que el usuario escribió se borra si no guardaste el estado. En la clase 9 lo arreglamos con `onSaveInstanceState`.

Buena práctica general: usa el Logcat para ver el ciclo en acción. Agrega un `Log.d` en cada método con el nombre del método, corre la app, navega, rota el emulador y mira el orden real. Verlo una vez vale más que leerlo diez veces.

Pasar datos entre actividades: el Intent mensajero

Las actividades no viven aisladas: forman flujos. La pantalla de login le pasa el nombre de usuario a la pantalla principal; la lista de productos le pasa el producto elegido a la pantalla de detalle. El mensajero que lleva datos de una actividad a otra se llama `Intent`.

Un `Intent` es un objeto que describe una intención: "quiero abrir esta otra pantalla". Hay dos tipos. El `Intent` explícito dice exactamente qué actividad quieres abrir, por su nombre de clase; es el que usarás el 99 por ciento del tiempo dentro de tu propia app. El `Intent` implícito describe una acción general ("quiero compartir texto", "quiero abrir una página web") y el sistema busca qué app puede hacerlo; así es como tu app puede abrir el navegador o compartir en WhatsApp sin programar esas funciones.

Analogía: el Intent explícito es como mandarle un recado a tu hermano por su nombre: "Juan, ven a la cocina". El Intent implícito es como gritar en la casa "alguien que sepa manejar, necesito ir al centro": no te importa quién responda, solo que alguien pueda hacerlo.

Para mandar datos junto con el Intent usas los extras: pares de clave y valor que viajan pegados al Intent, como notitas adhesivas en el recado. Del lado que envía usas `putExtra` con una clave de texto y el valor; del lado que recibe usas `getStringExtra`, `getIntExtra` y métodos similares con la misma clave. La regla de oro: la clave debe ser exactamente igual en ambos lados. Define tus claves como constantes para no equivocarte con una letra; un error de dedo en la clave significa que el dato "se pierde" y recibirás un nulo sin ningún mensaje de error que te avise.

Qué puedes mandar en un extra: textos, números enteros, decimales, booleanos, y también objetos más complejos si implementan `Parcelable` (la forma moderna y rápida) o `Serializable` (la forma vieja y lenta). Para este curso nos quedamos con los tipos simples, que cubren la enorme mayoría de los casos reales.

Guardar el estado: `onSaveInstanceState`

Ya sabes que al rotar el teléfono el sistema destruye y recrea tu actividad. Sin protección, todo lo que el usuario escribió en los campos se pierde y aparece la pantalla como recién nacida. La solución es `onSaveInstanceState`: un método que el sistema llama justo antes de destruir tu actividad, dándote un `Bundle` (una bolsita de pares clave-valor) para que guardes lo importante. Cuando la actividad se recrea, ese `Bundle` te llega de vuelta en `onCreate` y en `onRestoreInstanceState`, y ahí restauras los valores.

Analogía: es como cuando en la escuela sonaba la alarma de sismo a media clase. La maestra decía "anoten en qué página del libro iban y qué estaban escribiendo". Al regresar, cada quienabría su cuaderno y continuaba donde se quedó. `onSaveInstanceState` es ese momento de anotar; `onCreate` con el `Bundle` es el regreso al salón.

Detalle importante que confunde a muchos: las vistas con id asignado (como un `EditText` con `android:id`) guardan y restauran su texto automáticamente, sin que escribas nada. Android hace eso por ti. `onSaveInstanceState` es para lo que Android no puede adivinar: variables de tu lógica, el puntaje de un juego, la posición de una lista, el paso actual de un formulario. Si tu app pierde datos al rotar, primero revisa si tus vistas tienen id; si lo tienen y aun así se pierde algo, entonces sí necesitas `onSaveInstanceState` manual.

Otro detalle: `onSaveInstanceState` no se llama cuando el usuario cierra la actividad con el botón atrás a propósito. En ese caso el usuario quiso irse y no hay nada que restaurar. Solo se llama cuando el sistema destruye la actividad por su cuenta (rotación, falta de memoria, cambio de idioma).

Errores comunes de principiantes y cómo evitarlos

Error uno: olvidar declarar la Activity en el `AndroidManifest.xml`. Síntoma: la app se cierra al abrir la segunda pantalla con un error que menciona `ActivityNotFoundException`. Solución: declara cada actividad en el manifiesto. Si usas el asistente de Android Studio para crearla, se declara sola; si la creas a mano, agrécala tú.

Error dos: llamar `findViewById` antes de `setContentView`. Síntoma: `NullPointerException` al iniciar. Solución: primero `setContentView`, luego las vistas. Con View Binding el equivalente es inflar el binding antes de usarlo.

Error tres: escribir mal la clave de un extra. Síntoma: el dato llega nulo y no hay error visible. Solución: usa constantes para las claves, por ejemplo companion object con `const val NOMBRE_USUARIO = "nombre_usuario"`.

Error cuatro: hacer trabajo pesado en `onCreate` o en `onPause`. Síntoma: la app se congela o las transiciones se sienten lentas. Solución: trabajo pesado en corrutinas o hilos de fondo; `onCreate` y `onPause` deben terminar rápido.

Error cinco: no guardar el estado y perder datos al rotar. Síntoma: el usuario gira el teléfono y su texto desaparece. Solución: ids en las vistas para lo automático y `onSaveInstanceState` para tu lógica.

Error seis: iniciar en `onResume` algo que nunca se detiene en `onPause`. Síntoma: batería drenada, cámara o GPS activos en segundo plano. Solución: pareja obligatoria, lo que se abre en `onResume` se cierra en `onPause`; lo que se abre en `onStart` se cierra en `onStop`.

Error siete: confiar en `onDestroy` para guardar datos críticos. Síntoma: datos perdidos cuando el sistema mata la app. Solución: lo crítico se guarda en `onPause` u `onStop`.

Buenas prácticas de la unidad

Primera: una Activity, una responsabilidad. Si tu Activity hace login, lista productos, reproduce música y calcula impuestos, es momento de dividirla. Las actividades gordas son difíciles de probar y de mantener.

Segunda: no guardes referencias a la Activity en objetos que vivan más que ella (variables estáticas, tareas de fondo sin cancelar). Eso causa fugas de memoria: la actividad ya murió pero algo la sigue abrazando y el sistema no puede liberarla.

Tercera: usa View Binding en proyectos nuevos. Elimina una categoría entera de errores (ids mal escritos, vistas nulas) y el código queda más limpio.

Cuarta: pon logs del ciclo de vida mientras aprendes y quítalos (o bájalos de nivel) antes de publicar. Ver el ciclo en el Logcat es la mejor escuela.

Quinta: prueba siempre la rotación. Es la prueba de fuego de esta unidad: si tu pantalla sobrevive a girar el teléfono sin perder datos ni crashearse, entendiste el ciclo de vida.

Ejemplos paso a paso

Ejemplo 1: crear una segunda Activity y declararla en el manifiesto

Vamos a construir una mini app de dos pantallas: la primera pide tu nombre y la segunda te saluda. Paso uno: crea la segunda Activity. En Android Studio, clic derecho en tu paquete, New, Activity, Empty Views Activity, y nómbrala SaludoActivity. Android Studio genera dos archivos: SaludoActivity.kt y activity_saludo.xml, y además la declara sola en el manifiesto. Verifica que el manifiesto haya quedado así:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:allowBackup="true"
        android:label="Mi Primera App"
        android:theme="@style/Theme.MiPrimeraApp">

        <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>

        <activity
            android:name=".SaludoActivity"
            android:exported="false" />

    </application>

</manifest>

```

Explicación línea por línea. La etiqueta `application` envuelve todo lo que pertenece a tu app. Cada `activity` declara una pantalla con su nombre de clase. `MainActivity` tiene `exported="true"` y un `intent-filter` que dice `MAIN` y `LAUNCHER`: eso la marca como la puerta de entrada, la pantalla que se abre cuando tocas el icono de la app. `SaludoActivity` lleva `exported="false"` porque no queremos que otras apps la abran directamente; solo la usaremos desde dentro. Si olvidas esta declaración, verás el famoso `ActivityNotFoundException`.

Ejemplo 2: la primera pantalla envía el nombre con un Intent

Este es el layout de la pantalla principal, `activity_main.xml`. Tiene un `EditText` para escribir el nombre y un `Button` para continuar:


```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="24dp"
    android:gravity="center">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Como te llamas?"
        android:textSize="22sp" />

    <EditText
        android:id="@+id/etNombre"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Escribe tu nombre"
        android:layout_marginTop="16dp" />

    <Button
        android:id="@+id/btnSaludar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Saludarme"
        android:layout_marginTop="16dp" />

</LinearLayout>

```

Ahora la lógica en MainActivity.kt usando View Binding. Para usar View Binding, actívalo en el build.gradle del módulo con `viewBinding { isEnabled = true }` dentro del bloque `android`, y sincroniza el proyecto:

```

package com.lucasega.miprimeraapp

import android.content.Intent
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

    companion object {
        const val EXTRA_NOMBRE = "com.lucasega.miprimeraapp.NOMBRE"
    }

    private lateinit var binding: ActivityMainBinding

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

        binding.btnSaludar.setOnClickListener {
            val nombre = binding.etNombre.text.toString()

            val intent = Intent(this, SaludoActivity::class.java)
            intent.putExtra(EXTRA_NOMBRE, nombre)
            startActivity(intent)
        }
    }
}

```

Explicación línea por línea. El companion object guarda la constante EXTRA_NOMBRE con la clave del extra; usar el nombre del paquete como prefijo es la convención para que las claves no choquen. lateinit var binding declara el objeto de View Binding que se inicializará después. En onCreate, primero super.onCreate como manda la regla, luego se infla el binding con el layoutInflater, que convierte el XML en objetos reales, y setContentView(binding.root) dibuja esa vista. Después se configura el listener del botón: al hacer clic se lee el texto del EditText, se crea un Intent explícito diciendo "quiero abrir SaludoActivity", se pega el extra con putExtra usando la clave constante, y startActivity lanza la segunda pantalla. Fíjate que this dentro del listener se refiere a la Activity porque el listener está dentro de ella; en casos ambiguos se usa this@MainActivity.

Ejemplo 3: la segunda pantalla recibe el dato

El layout activity_saludo.xml es sencillo, solo un TextView grande para el saludo:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="24dp">

    <TextView
        android:id="@+id/tvSaludo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hola!"
        android:textSize="28sp" />

</LinearLayout>
```

Y SaludoActivity.kt lo recibe así:

```

package com.lucasega.miprimeraapp

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivitySaludoBinding

class SaludoActivity : AppCompatActivity() {

    private lateinit var binding: ActivitySaludoBinding

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

        val nombre = intent.getStringExtra(MainActivity.EXTRA_NOMBRE)

        if (nombre.isNullOrEmpty()) {
            binding.tvSaludo.text = "Hola, visitante!"
        } else {
            binding.tvSaludo.text = "Hola, $nombre! Bienvenido a Lucas EGA Academy"
        }
    }
}

```

Explicación. Se infla el binding de activity_saludo igual que antes. La línea clave es `intent.getStringExtra(MainActivity.EXTRA_NOMBRE)`: cada Activity tiene una propiedad llamada `intent` que contiene el Intent con el que fue lanzada; de ahí se extrae el valor usando exactamente la misma clave constante definida en MainActivity. Como el extra podría no venir (si alguien abre esta pantalla sin mandar el dato), se valida con `isNullOrEmpty` y se muestra un saludo genérico. Esta validación es buena práctica: nunca confíes ciegamente en que el extra existe. Observa también el uso de plantillas de string con `$nombre`, la forma idiomática de Kotlin para interpolar variables.

Ejemplo 4: ver el ciclo de vida en el Logcat

Crea una actividad de prueba o agrega estos métodos a MainActivity para observar el ciclo de vida en acción:

```

package com.lucasega.miprimeraapp

import android.os.Bundle
import android.util.Log
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityMainBinding

class CicloActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
        Log.d("CICLO", "onCreate llamado")
    }

    override fun onStart() {
        super.onStart()
        Log.d("CICLO", "onStart llamado")
    }

    override fun onResume() {
        super.onResume()
        Log.d("CICLO", "onResume llamado")
    }

    override fun onPause() {
        super.onPause()
        Log.d("CICLO", "onPause llamado")
    }

    override fun onStop() {
        super.onStop()
        Log.d("CICLO", "onStop llamado")
    }

    override fun onDestroy() {
        super.onDestroy()
        Log.d("CICLO", "onDestroy llamado")
    }
}

```

Explicación. Cada método sobrescribe al original y lo primero que hace es llamar a super, que ejecuta el comportamiento base de Android; si olvidas el super, la app se cierra con un error `SuperNotCalledException`, así que memorízalo como reflejo. Log.d escribe en el Logcat con la etiqueta "CICLO". Para verlo, abre la pestaña Logcat en Android Studio y filtra por "CICLO". Prueba esto: abre la app y verás `onCreate`, `onStart`, `onResume` en orden. Presiona el botón de inicio del emulador y verás `onPause`, `onStop`. Regresa a la app y verás `onRestart` (si lo agregas), `onStart`, `onResume`. Gira el emulador con `Ctrl+F11` y verás la secuencia completa de destrucción y recreación: `onPause`, `onStop`, `onDestroy`, `onCreate`, `onStart`, `onResume`. Este experimento de cinco minutos te enseña más que cualquier diagrama.

Ejemplo 5: guardar el estado con `onSaveInstanceState`

Imagina un contador de puntos en un juego. Cada vez que tocas un botón sumas un punto. Si giras el teléfono sin guardar el estado, el contador regresa a cero y el jugador se enoja. Vamos a protegerlo:

```

package com.lucasega.miprimeraapp

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityContadorBinding

class ContadorActivity : AppCompatActivity() {

    private lateinit var binding: ActivityContadorBinding
    private var puntos = 0

    companion object {
        const val CLAVE_PUNTOS = "puntos"
    }

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

        if (savedInstanceState != null) {
            puntos = savedInstanceState.getInt(CLAVE_PUNTOS, 0)
        }
        actualizarTexto()

        binding.btnSumar.setOnClickListener {
            puntos++
            actualizarTexto()
        }
    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
        outState.putInt(CLAVE_PUNTOS, puntos)
    }

    private fun actualizarTexto() {
        binding.tvPuntos.text = "Puntos: $puntos"
    }
}

```

Explicación línea por línea. La variable puntos vive en la actividad y se reinicia a cero cada vez que la actividad se crea de nuevo; ese es el problema. La constante CLAVE_PUNTOS evita errores de dedo. En onCreate se revisa si savedInstanceState es nulo: si lo es, es la primera

creación y puntos se queda en cero; si no es nulo, significa que el sistema está recreando la actividad y ahí viene el valor guardado, que se recupera con `getInt`. El segundo parámetro de `getInt` es el valor por defecto por si la clave no existiera. `onSaveInstanceState` se sobrescribe y guarda los puntos en el `Bundle` con `putInt` antes de que el sistema destruya la actividad. Prueba: suma varios puntos, gira el emulador y verás que el contador conserva su valor. Sin esas dos piezas, volvería a cero. El layout de este ejemplo necesita un `TextView` con id `tvPuntos` y un `Button` con id `btnSumar`; el `View Binding` genera `ActivityContadorBinding` a partir de `activity_contador.xml`.

Práctica

Resuelve cada ejercicio en Android Studio antes de leer la solución. La única forma de que estos conceptos se queden grabados es ensuciándote las manos con código.

Ejercicio 1: tu segunda pantalla

Crea un proyecto nuevo con una Empty Views Activity llamada `MainActivity`. Agrégale un botón con el texto "Ir a detalle". Crea una segunda Activity llamada `DetalleActivity` con un `TextView` que diga "Estás en el detalle". Haz que el botón abra la segunda pantalla. Declara todo lo necesario.

Solución: primero crea `DetalleActivity` con el asistente (New, Activity, Empty Views Activity), lo que genera `DetalleActivity.kt` y `activity_detalle.xml` y la declara en el manifiesto automáticamente. En `activity_detalle.xml` pon un `LinearLayout` con un `TextView` centrado con el texto "Estás en el detalle". En `activity_main.xml` pon el botón con id `btnDetalle`. En `MainActivity.kt`, dentro de `onCreate` y después de `setContentView`, configura el listener:

```
binding.btnDetalle.setOnClickListener {
    val intent = Intent(this, DetalleActivity::class.java)
    startActivity(intent)
}
```

No olvides el import de `android.content.Intent`. Si la app se cierra al tocar el botón, lo primero que debes revisar es el `AndroidManifest.xml`: `DetalleActivity` debe aparecer declarada dentro de la etiqueta `application`. Este ejercicio te entrena en el flujo completo que repetirás cientos de veces: crear, declarar, lanzar.

Ejercicio 2: el mensajero con dos datos

Modifica el ejercicio anterior: en la pantalla principal agrega dos EditText, uno para el nombre y otro para la edad. Al tocar el botón, manda ambos datos a DetalleActivity y muestra el texto "Hola [nombre], tienes [edad] años". Usa constantes para las claves y valida que los datos no vengan vacíos.

Solución: en MainActivity define en el companion object dos constantes, por ejemplo EXTRA_NOMBRE y EXTRA_EDAD. Al hacer clic, lee ambos EditText y crea el Intent:

```
val intent = Intent(this, DetalleActivity::class.java)
intent.putExtra(EXTRA_NOMBRE, binding.etNombre.text.toString())
intent.putExtra(EXTRA_EDAD, binding.etEdad.text.toString())
startActivity(intent)
```

En DetalleActivity recupera con getStringExtra usando las constantes de MainActivity y valida:

```
val nombre = intent.getStringExtra(MainActivity.EXTRA_NOMBRE)
val edad = intent.getStringExtra(MainActivity.EXTRA_EDAD)

binding.tvResultado.text = if (nombre.isNullOrBlank() || edad.isNullOrBlank()) {
    "Faltan datos, regresa y completa el formulario"
} else {
    "Hola $nombre, tienes $edad años"
}
```

La lección aquí es doble: las claves como constantes evitan el error silencioso del dato nulo, y la validación hace tu pantalla robusta aunque la abran sin datos. En apps reales, las pantallas siempre deben defenderse de datos faltantes.

Ejercicio 3: detective del ciclo de vida

Agrega logs a todos los métodos del ciclo de vida en MainActivity, igual que en el ejemplo 4. Luego responde por escrito, mirando el Logcat: a) qué métodos se llaman al abrir la app por primera vez y en qué orden; b) qué métodos se llaman al presionar el botón de inicio del emulador; c) qué métodos se llaman al girar la pantalla; d) qué métodos se llaman al presionar el botón atrás para salir de la app.

Solución: a) Al abrir: onCreate, onStart, onResume, en ese orden exacto. Es la secuencia de nacimiento de toda actividad. b) Al ir al inicio: onPause y luego onStop. La actividad ya no se ve pero sigue viva en memoria, lista para regresar. c) Al girar: onPause, onStop, onDestroy, y después onCreate, onStart, onResume. Fíjate que sí pasa por onDestroy: el sistema mata la

actividad y la vuelve a crear desde cero con la nueva orientación. Por eso se pierde todo lo no guardado. d) Al presionar atrás: onPause, onStop, onDestroy, y ahí termina; no hay recreación porque el usuario decidió salir. Si tus respuestas del Logcat coinciden con esto, ya tienes el ciclo de vida en la cabeza. Si algo no coincidió, repite el experimento: la discrepancia suele revelar un malentendido valioso.

Ejercicio 4: el contador que no olvida

Crea la ContadorActivity del ejemplo 5: un TextView que muestra los puntos y un botón "Sumar punto". Implementa onSaveInstanceState para que los puntos sobrevivan a la rotación. Luego haz la prueba de fuego: suma 7 puntos, gira el emulador y verifica que siga diciendo 7. Después, como reto extra, agrega un segundo botón "Reiniciar" que ponga el contador en cero.

Solución: el código base es el del ejemplo 5. La clave está en las dos piezas inseparables: guardar en onSaveInstanceState con `outState.putInt(CLAVE_PUNTOS, puntos)` y restaurar en onCreate revisando si `savedInstanceState` es nulo. Si olvidas cualquiera de las dos, el contador se reinicia al girar. Para el botón reiniciar:

```
binding.btnReiniciar.setOnClickListener {  
    puntos = 0  
    actualizarTexto()  
}
```

Nota que reiniciar no necesita tocar el Bundle: cuando gires después de reiniciar, onSaveInstanceState guardará el cero, que es el valor correcto. Este ejercicio te enseña el patrón mental: el Bundle es una foto del estado justo antes de la destrucción; todo lo que cambie después de la foto ya es otro momento.

Ejercicio 5: encuentra el bug

Este código tiene tres errores de los que vimos en la unidad. Encuéntralos y corrígelos:

```

class RotaActivity : AppCompatActivity() {
    private lateinit var binding: ActivityRotaBinding
    private var mensaje = ""

    override fun onCreate(savedInstanceState: Bundle?) {
        binding.tvMensaje.text = "Hola"
        binding = ActivityRotaBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }

    override fun onPause() {
        descargarArchivoGrande()
    }
}

```

Solución: error uno, se usa binding antes de inicializarlo y antes de setContentView; el orden correcto es inflar el binding, luego setContentView y al final tocar las vistas. Error dos, falta la llamada a super.onCreate(savedInstanceState) al inicio de onCreate; sin ella la app se cierra con SuperNotCalledException. Error tres, se hace trabajo pesado (descargarArchivoGrande) dentro de onPause; onPause debe ser rapidísimo porque bloquea la transición a la siguiente pantalla, así que esa descarga debe ir a una corrutina o a un servicio en segundo plano. Además, como observación extra, la variable mensaje nunca se guarda en onSaveInstanceState, así que se perdería al rotar. Este ejercicio simula exactamente lo que harás en entrevistas de trabajo: leer código ajeno y detectar problemas del ciclo de vida.

Ejercicio 6: mini-proyecto, app de registro en dos pasos

Construye una app con dos pantallas que simule un registro. Pantalla 1: pide nombre y correo, con un botón "Continuar". Valida que ninguno esté vacío; si falta algo, muestra un mensaje en la misma pantalla sin avanzar. Pantalla 2: muestra "Registro exitoso" con los datos recibidos y un botón "Corregir" que regresa a la pantalla 1. Además, lo que el usuario escribió en la pantalla 1 no debe perderse si gira el teléfono antes de continuar.

Solución: en la pantalla 1, al tocar Continuar, lee los campos y valida:

```
val nombre = binding.etNombre.text.toString().trim()
val correo = binding.etCorreo.text.toString().trim()

if (nombre.isEmpty() || correo.isEmpty()) {
    binding.tvError.text = "Completa todos los campos"
} else {
    val intent = Intent(this, ConfirmacionActivity::class.java)
    intent.putExtra(EXTRA_NOMBRE, nombre)
    intent.putExtra(EXTRA_CORREO, correo)
    startActivity(intent)
}
```

Usa trim para que un campo con puros espacios cuente como vacío. En la pantalla 2 recupera los extras con las constantes, los muestra y el botón Corregir simplemente llama a finish(), que cierra la actividad actual y regresa a la anterior. Para que el texto sobreviva a la rotación en la pantalla 1, asegúrate de que los EditText tengan android:id; con eso Android restaura su contenido automáticamente sin código extra. Si quieres ir más allá, guarda también el mensaje de error con onSaveInstanceState. Este mini-proyecto junta todo lo de la unidad: navegación, extras, validación, finish y estado. Si lo terminaste sin copiar, la unidad está dominada.

Para llevar más lejos

findViewById a fondo: la base que debes dominar

View Binding es moderno y recomendado, pero findViewById es la base sobre la que está construido todo. Lo verás en tutoriales viejos, en código de empresas y en exámenes, así que vamos a dominarlo. La idea es simple: cada vista de tu XML puede tener un android:id, y findViewById busca en el layout ya dibujado la vista con ese id y te la devuelve como objeto.

Mira el mismo ejemplo del saludo pero con findViewById, sin View Binding:

```

package com.lucasega.miprimeraapp

import android.content.Intent
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    companion object {
        const val EXTRA_NOMBRE = "com.lucasega.miprimeraapp.NOMBRE"
    }

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

        val etNombre = findViewById<EditText>(R.id.etNombre)
        val btnSaludar = findViewById<Button>(R.id.btnSaludar)

        btnSaludar.setOnClickListener {
            val nombre = etNombre.text.toString()
            val intent = Intent(this, SaludoActivity::class.java)
            intent.putExtra(EXTRA_NOMBRE, nombre)
            startActivity(intent)
        }
    }
}

```

Explicación. Después de `setContentView`, se buscan las vistas: `findViewById<EditText>(R.id.etNombre)` significa "búscame la vista con id `etNombre` y trámela como `EditText`". El tipo entre ángulos (`<EditText>`) es importante: le dice a Kotlin qué clase de vista esperas; si pides un `Button` pero el id es de un `TextView`, la app se cae con un `ClassCastException`. Por eso los ids deben estar bien puestos en el XML.

Ventajas de `findViewById`: no requiere configuración en el `build.gradle`, funciona en cualquier proyecto y es lo que todo desarrollador Android reconoce al instante. Desventajas: si escribes mal el id, el compilador no te avisa y la app se cae en ejecución con un nulo; además es más verboso. `View Binding` elimina esos riesgos porque genera una clase con todas tus vistas y si renombas un id en el XML el código que lo usa deja de compilar, avisándote de inmediato.

Mi recomendación práctica: en tus proyectos nuevos usa View Binding; cuando leas código ajeno o mantengas apps viejas, lee findViewById con fluidez. Un buen ejercicio es tomar uno de los ejemplos de esta unidad escrito con View Binding y reescribirlo con findViewById, y viceversa. Ese ir y venir te da la flexibilidad que piden los trabajos reales.

Error clásico con findViewById: buscar una vista que está en otro layout. Por ejemplo, llamar findViewById(R.id.tvSaludo) en MainActivity cuando tvSaludo vive en activity_saludo.xml. El resultado es nulo y la app se cae al usarlo. Cada Activity solo puede buscar vistas en el layout que ella misma dibujó con setContentView. Con View Binding este error es imposible porque cada binding solo conoce su propio layout.

Intents implícitos: pedirle ayuda a otras apps

Hasta ahora usamos Intents explícitos: "abre esta pantalla mía". Los Intents implícitos son más ambiciosos: describen una acción y dejan que el sistema encuentre quién puede hacerla. Es como la diferencia entre decirle a tu hermano "tráeme agua" y gritar en la calle "alguien vende agua?": en el segundo caso no te importa quién responda.

Los tres usos más comunes: compartir texto, abrir una página web y marcar un número. Mira cómo se comparten los logros de tu app en WhatsApp, Telegram o donde el usuario quiera:

```
val texto = "Llevo $puntos puntos en mi app de Lucas EGA Academy!"
val intent = Intent(Intent.ACTION_SEND).apply {
    type = "text/plain"
    putExtra(Intent.EXTRA_TEXT, texto)
}
val chooser = Intent.createChooser(intent, "Compartir con...")
startActivity(chooser)
```

Explicación. Intent.ACTION_SEND es la acción estándar de "enviar algo". El apply es una función de Kotlin que configura el objeto recién creado: le pone el tipo MIME "text/plain" (texto plano) y le pega el texto en el extra especial EXTRA_TEXT. createChooser muestra el diálogo del sistema con todas las apps que pueden compartir texto, para que el usuario elija. Sin el chooser, si el usuario ya tiene una app predeterminada, se abriría directamente sin preguntar.

Para abrir una página web:

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("https://lucasegaacademy.com"))
startActivity(intent)
```

ACTION_VIEW con una Uri le dice al sistema "alguien que pueda mostrar esto"; el navegador la abre. Necesitas importar android.net.Uri. Para marcar un número (ojo: ACTION_DIAL abre el marcador sin llamar directamente; ACTION_CALL sí llama pero requiere un permiso especial peligroso, así que en este curso usamos DIAL):

```
val intent = Intent(Intent.ACTION_DIAL, Uri.parse("tel:5551234567"))
startActivity(intent)
```

Buena práctica con intents implícitos: verifica que haya al menos una app que pueda resolverlo antes de lanzarlo, porque en un emulador sin navegador tu app se cerraría. La forma moderna es usar intent.resolveActivity(packageManager) y solo llamar a startActivity si no es nulo. En un teléfono real casi siempre hay navegador, pero en emuladores de prueba esta validación te salva de un crash vergonzoso en una demo.

Analogía final: el Intent explícito es un taxi privado que lleva tu recado a una dirección exacta; el Intent implícito es pegar un anuncio en el periódico: describes lo que necesitas y quien pueda hacerlo responde. Ambos son la misma clase Intent; lo único que cambia es si dices el destino o describes la acción.

La pila de actividades, finish() y el botón atrás

Cuando navegas de una pantalla a otra, Android no destruye la anterior: la guarda en una pila, como un montón de platos. Abres MainActivity (plato uno), luego DetalleActivity (plato dos encima). Cuando el usuario presiona el botón atrás, Android quita el plato de arriba (destruye DetalleActivity) y muestra el que quedó debajo (MainActivity regresa con onRestart, onStart, onResume). Esta pila se llama back stack y es la razón por la que el botón atrás "regresa" de forma natural sin que programes nada.

A veces quieres cerrar una pantalla desde código, sin esperar al botón atrás. Para eso existe finish(): le dice a la actividad actual "termina tu trabajo y sácate de la pila". Es lo que usamos en el ejercicio 6 para el botón "Corregir": cerrar la pantalla de confirmación y volver al formulario. Otro uso típico es la pantalla de login: una vez que el usuario entra, no quieres que el botón atrás lo regrese al login, así que después de abrir la pantalla principal llamas a finish() en el login.

También puedes controlar el botón atrás sobrescribiendo onBackPressed, aunque en las versiones modernas se usa el onBackPressedDispatcher. Un caso clásico: pedir confirmación antes de salir si el usuario tiene cambios sin guardar. Pero cuidado: abusar de esto frustra al usuario; el botón atrás debe comportarse como espera todo el mundo salvo que tengas una razón de peso.

Detalle fino que confunde: cuando regresas con el botón atrás a una actividad que estaba en `onStop`, esa actividad no pasa por `onCreate` de nuevo; pasa por `onRestart`, `onStart` y `onResume`. Sus variables siguen intactas porque nunca se destruyó. En cambio, si llegas a ella con un `Intent` nuevo desde otra pantalla, se crea una instancia nueva encima en la pila, y ahora tienes dos copias de la misma pantalla apiladas. Esto es normal, pero si lo haces en un ciclo (A abre B, B abre A, A abre B...) la pila crece sin control. Para esos casos existen banderas del `Intent` como `FLAG_ACTIVITY_CLEAR_TOP`, que en lugar de apilar una copia nueva, limpia todo lo que esté encima de la actividad destino si ya existe en la pila.

Ejercicio 7: la pila en acción

Crea tres actividades: A, B y C, cada una con un botón que abre la siguiente (A abre B, B abre C). En cada una agrega logs del ciclo de vida. Navega A, B, C y luego presiona atrás dos veces. Escribe qué ves en el Logcat y explica por qué. Como reto extra, en C agrega un botón "Volver a A limpiando" que use `FLAG_ACTIVITY_CLEAR_TOP`.

Solución: al navegar A, B, C verás en A: `onPause`, `onStop` (queda viva debajo); en B lo mismo cuando abres C. Al presionar atrás la primera vez, C muestra `onPause`, `onStop`, `onDestroy` (muere de verdad) y B muestra `onRestart`, `onStart`, `onResume` (regresa sin pasar por `onCreate`). Al presionar atrás de nuevo, B muere y A regresa igual. La lección: atrás destruye la de arriba y resucita la de abajo sin recrearla. Para el reto extra:

```
binding.btnVolverA.setOnClickListener {
    val intent = Intent(this, AActivity::class.java)
    intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TOP
    startActivity(intent)
}
```

Con esta bandera, si A ya está en la pila, Android destruye B y C y te deja en la A original, sin crear una copia nueva. Es la técnica estándar para botones tipo "volver al inicio" o "cerrar sesión".

Caso real: el formulario que sobrevive a todo

Juntemos todo lo de la unidad en un caso realista: una pantalla de pedido de tacos. El usuario escribe su nombre, elige cuántos tacos quiere con botones de más y menos, y al confirmar pasa a la pantalla de resumen. Si gira el teléfono a la mitad, nada se pierde. Este ejemplo combina extras, estado y ciclo de vida en un solo flujo, como los que construirás en tu trabajo.

El layout activity_pedido.xml lleva un EditText para el nombre, un TextView con la cantidad, dos botones para subir y bajar, y un botón confirmar:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="24dp"
    android:gravity="center">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Pedido de tacos"
        android:textSize="24sp" />

    <EditText
        android:id="@+id/etCliente"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="Tu nombre"
        android:layout_marginTop="16dp" />

    <TextView
        android:id="@+id/tvCantidad"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Tacos: 1"
        android:textSize="20sp"
        android:layout_marginTop="16dp" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_marginTop="8dp">

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

        <Button
```

```

        android:id="@+id/btnMas"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="+"
        android:layout_marginStart="16dp" />

</LinearLayout>

<Button
    android:id="@+id/btnConfirmar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Confirmar pedido"
    android:layout_marginTop="24dp" />

</LinearLayout>

```

Y la actividad completa:

```

package com.lucasega.miprimeraapp

import android.content.Intent
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityPedidoBinding

class PedidoActivity : AppCompatActivity() {

    private lateinit var binding: ActivityPedidoBinding
    private var cantidad = 1

    companion object {
        const val EXTRA_CLIENTE = "com.lucasega.miprimeraapp.CLIENTE"
        const val EXTRA_CANTIDAD = "com.lucasega.miprimeraapp.CANTIDAD"
        const val CLAVE_CANTIDAD = "cantidad"
    }

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

        if (savedInstanceState != null) {
            cantidad = savedInstanceState.getInt(CLAVE_CANTIDAD, 1)
        }
        mostrarCantidad()
    }
}

```

```

binding.btnMas.setOnClickListener {
    cantidad++
    mostrarCantidad()
}

binding.btnMenos.setOnClickListener {
    if (cantidad > 1) {
        cantidad--
        mostrarCantidad()
    }
}

binding.btnConfirmar.setOnClickListener {
    val cliente = binding.etCliente.text.toString().trim()
    if (cliente.isEmpty()) {
        binding.etCliente.error = "Escribe tu nombre"
    } else {
        val intent = Intent(this, ResumenActivity::class.java)
        intent.putExtra(EXTRA_CLIENTE, cliente)
        intent.putExtra(EXTRA_CANTIDAD, cantidad)
        startActivity(intent)
    }
}

}

override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putInt(CLAVE_CANTIDAD, cantidad)
}

private fun mostrarCantidad() {
    binding.tvCantidad.text = "Tacos: $cantidad"
}

}

```

Fíjate en cada decisión. El EditText tiene id, así que el nombre sobrevive a la rotación automáticamente; la cantidad es una variable de lógica, así que se guarda a mano en onSaveInstanceState. El botón menos no deja bajar de 1, porque un pedido de cero tacos no tiene sentido: esa validación de rango es típica de apps reales. Al confirmar se valida el nombre con trim y se muestra el error directamente en el campo con la propiedad error, que pinta el mensaje en rojo debajo del EditText; es la forma estándar de validar formularios en Android. Solo si todo está bien se crea el Intent con los dos extras. La pantalla de resumen (ResumenActivity) los recibe igual que en el ejemplo 3 y los muestra. Si completaste este caso y sobrevivió a la rotación con 5 tacos y tu nombre intactos, ya programas como profesional.

Tres errores de ciclo de vida que rompieron apps reales

Te cuento tres historias para que veas que esto no es teoría. Primera: una app de música iniciaba la reproducción en onCreate y nunca la detenía. El usuario "cerraba" la app con el botón de inicio, pero la música seguía sonando y la batería se acababa. El error fue no parear: lo que se inicia cuando la actividad está visible debe detenerse en onPause u onStop. La regla de la pareja te habría salvado.

Segunda: una app bancaria guardaba el borrador de una transferencia solo en onDestroy. Un día el sistema mató la app por falta de memoria sin llamar a onDestroy y el usuario perdió los datos de una transferencia importante. Lo crítico siempre va en onPause u onStop, que sí se llaman en el camino normal hacia la destrucción.

Tercera: un juego guardaba el puntaje en una variable pero no implementaba onSaveInstanceState. Cada vez que alguien giraba el teléfono por accidente a mitad de partida, el puntaje volvía a cero y las reseñas de la app se llenaban de enojo. Dos métodos pequeños (guardar y restaurar) habrían evitado cientos de reseñas de una estrella.

La moraleja es la misma en los tres casos: el ciclo de vida no es un tema de examen, es el contrato entre tu app y el teléfono. Respétalo y tu app se sentirá sólida; ignóralo y se sentirá rota aunque tu lógica sea perfecta.

Guía rápida: qué código va en cada método

Cuando llesves unas semanas programando, esta guía te servirá de acordeón. Guárdala en tu memoria:

En onCreate pon la configuración que se hace una sola vez: setContentView o inflar el binding, buscar vistas, crear el adaptador de una lista, leer los extras del Intent, restaurar el estado desde savedInstanceState y configurar listeners. Nada que tarde más de unos milisegundos.

Si necesitas cargar datos de internet o de una base de datos, aquí solo lanzas la corrutina o la tarea; el trabajo pesado nunca bloquea este método.

En onStart registra lo que debe estar activo mientras la actividad es visible: observadores de datos, receptores de broadcasts, sensores de bajo consumo. Piensa en onStart como "el usuario ya me puede ver, prepárate".

En onResume inicia lo exclusivo de estar al frente: reproducir video o audio, iniciar animaciones, encender la cámara, pedir actualizaciones frecuentes de ubicación. Todo lo que aquí inicies debe detenerse en onPause, sin excepciones.

En onPause haz pausas ligeras y rápidas: pausar el video, guardar el borrador del formulario, apagar sensores. Nada de red, nada de disco pesado, nada que tarde. Este método es el cuello de botella de todas las transiciones.

En onStop detén lo que consume recursos en segundo plano y guarda datos importantes: cerrar conexiones, cancelar corrutinas ligadas a la interfaz, escribir en la base de datos o en preferencias. Después de aquí, el sistema puede destruirte sin avisar.

En onDestroy libera los últimos recursos: cancelar tareas largas, quitar listeners globales. Y recuerda: no pongas aquí nada crítico, porque no tienes garantía de que se llame.

En onSaveInstanceState guarda el estado de la interfaz que el sistema no restaura solo: contadores, pasos de un asistente, posición de scroll personalizada, selecciones. Usa claves constantes y tipos simples.

Un truco de profesionales: cuando dudes en qué método poner algo, pregúntate "qué pasa si el usuario recibe una llamada justo ahora". Si tu código debe pausarse con la llamada, va en onPause. Si debe seguir pero sin gastar batería visible, va en onStop. Si debe sobrevivir a que maten la app, va en onSaveInstanceState o en almacenamiento persistente. Esa sola pregunta resuelve el 90 por ciento de las dudas.

Ejercicio 8: reto final, pantalla que lo aguanta todo

Combina todo: crea una pantalla de "nota rápida" con un EditText grande, un contador de caracteres en un TextView que se actualiza mientras escribes, y un botón "Enviar" que abre una segunda pantalla mostrando la nota y su longitud. Requisitos: a) el contador se actualiza en tiempo real; b) al girar el teléfono no se pierde ni la nota ni el conteo; c) si la nota está vacía, el botón muestra un error y no avanza; d) agrega logs del ciclo de vida y verifica el orden al girar.

Solución: para el contador en tiempo real usa un TextWatcher en el EditText:

```
binding.etNota.addTextChangedListener(object : TextWatcher {
    override fun beforeTextChanged(s: CharSequence?, start: Int, count: Int, after:
Int) {}
    override fun onTextChanged(s: CharSequence?, start: Int, before: Int, count:
Int) {
        binding.tvContador.text = "Caracteres: ${s?.length ?: 0}"
    }
    override fun afterTextChanged(s: Editable?) {}
}))
```

Necesitas importar `android.text.TextWatcher` y `android.textEditable`. El `EditText` con `id` restaura su texto solo al girar, y como el contador se recalcula desde el texto, también queda correcto sin guardarlo a mano; aun así, guárdalo en `onSaveInstanceState` como práctica. El botón valida con `isBlank` y usa la propiedad `error` si está vacío; si hay texto, manda la nota con un extra a la segunda pantalla, que la muestra junto con su longitud usando `getStringExtra`. Los logs al girar deben mostrar `onPause`, `onStop`, `onDestroy`, `onCreate`, `onStart`, `onResume`. Si tu app cumple los cuatro requisitos, felicidades: dominas la unidad 3 de punta a punta y estás listo para construir interfaces de verdad en la unidad 4.

Preguntas frecuentes de estudiantes

Pregunta: si el sistema puede destruir mi actividad en cualquier momento después de `onStop`, ¿cómo sé que mis datos están a salvo? Respuesta: distingue dos tipos de datos. Los datos de la interfaz temporal (lo que el usuario escribió, el paso actual) se protegen con `onSaveInstanceState`, que el sistema llama antes de destruirte. Los datos importantes y permanentes (una nota guardada, el progreso de un curso) deben escribirse en almacenamiento persistente, como una base de datos o `SharedPreferences`, en `onPause` u `onStop`. La regla mental: si al usuario le dolería perderlo aunque cierre la app a propósito, va a almacenamiento persistente; si solo duele perderlo al girar la pantalla, basta el `Bundle`.

Pregunta: ¿por qué mi app se cierra al abrir la segunda actividad aunque el código se ve bien? Respuesta: en nueve de cada diez casos es el manifiesto. Revisa que la actividad esté declarada dentro de la etiqueta `application` y que el nombre de la clase coincida exactamente, incluyendo mayúsculas. El décimo caso suele ser un `findViewById` antes de `setContentView` o un `id` mal escrito. Lee el Logcat: la primera línea del error casi siempre dice la causa.

Pregunta: ¿puedo pasar un objeto completo, no solo textos y números? Respuesta: sí, si tu clase implementa `Parcelable`. En Kotlin moderno es muy fácil con la anotación `@Parcelize` del plugin `kotlin-parcelize`: marcas la clase y listo. Luego usas `putExtra` con el objeto y `getParcelableExtra` del otro lado. Para este curso nos quedamos con tipos simples, pero

cuando construyas apps reales, Parcelable será tu herramienta para pasar un Producto, un Usuario o un Pedido completos entre pantallas.

Pregunta: ¿qué diferencia hay entre onPause y onStop en la vida real? Respuesta: imagina que estás viendo un video en tu app y te llega un mensaje de WhatsApp que se asoma como notificación flotante. Tu actividad sigue visible parcialmente: solo se llama onPause. Si en cambio abres WhatsApp completo, tu actividad deja de verse: se llama onPause y luego onStop. Por eso el video se pausa en onPause (ya no estás al frente) pero la conexión de red pesada se cierra en onStop (ya nadie te ve).

Pregunta: ¿tengo que llamar a super en todos los métodos del ciclo de vida? Respuesta: sí, siempre, y siempre como primera línea. El super ejecuta la lógica interna de Android sin la cual tu actividad no funciona. Si lo olvidas, obtienes un SuperNotCalledException y la app se cierra. Hazlo un reflejo: al sobrescribir cualquier método del ciclo, escribe la llamada a super antes que cualquier otra cosa.

Pregunta: ¿por qué a veces onDestroy no se llama y mi limpieza no se ejecuta? Respuesta: porque el sistema puede matar tu proceso directamente cuando necesita memoria con urgencia, sin despedirse. Por eso la limpieza crítica (guardar datos) va en onPause u onStop, y onDestroy queda solo para liberar recursos no críticos. Diseña tu app suponiendo que onDestroy es un lujo, no una garantía.

Pregunta: ¿cuántas actividades debe tener una app? Respuesta: las que necesite, pero cada una con una sola responsabilidad clara. Una app de tacos puede tener: lista de platillos, detalle del platillo, carrito, confirmación y perfil. Cinco pantallas, cinco actividades, cada una experta en lo suyo. Si una actividad empieza a hacer de todo, es señal de dividirla. En el desarrollo moderno también existen los fragments para dividir pantallas, pero ese es tema de unidades avanzadas; por ahora, actividades bien separadas te llevan muy lejos.

Resumen de la unidad

- Una Activity es una pantalla interactiva de tu app: una clase Kotlin que hereda de AppCompatActivity y se conecta a un layout XML con setContentView.
- Toda Activity debe declararse en el AndroidManifest.xml; la principal lleva el intent-filter MAIN y LAUNCHER.
- El sistema controla el ciclo de vida, no tú: escribes qué pasa en cada método y Android decide cuándo llamarlo.

- Orden de apertura: onCreate, onStart, onResume. Orden al ocultarse: onPause, onStop. Al girar: se destruye (onPause, onStop, onDestroy) y se recrea desde cero.
- onCreate es para configuración inicial y debe ser rápido; onPause debe ser rapidísimo; lo que se inicia en onResume se detiene en onPause.
- Los Intent explícitos abren pantallas de tu app; los extras llevan datos con pares clave-valor y las claves deben ser constantes idénticas en ambos lados.
- onSaveInstanceState guarda el estado antes de una destrucción del sistema y el Bundle se recupera en onCreate; las vistas con id restauran su contenido solas.
- Nunca guardes datos críticos solo en onDestroy, porque no siempre se llama; usa onPause u onStop.
- View Binding es la forma moderna de acceder a las vistas; findViewById es la base que debes conocer para leer código existente.
- La prueba de fuego de esta unidad: tu pantalla debe sobrevivir a la rotación sin perder datos ni cerrarse.

UNIDAD 4: Layouts XML y vistas

Si la unidad 3 te enseñó a darle vida a las pantallas, esta unidad te enseña a darles cuerpo y cara. Un layout es el plano arquitectónico de tu pantalla: dice qué elementos hay, dónde va cada uno, de qué tamaño y de qué color. Las vistas son los ladrillos: textos, botones, campos de entrada e imágenes. Sin layouts bien contruidos, la mejor lógica del mundo se ve como una app de 1998; con layouts bien contruidos, hasta una app sencilla se siente profesional.

En esta unidad recorres las clases 10, 11 y 12. La clase 10 es sobre los contenedores: `LinearLayout`, `RelativeLayout` y `ConstraintLayout`. Son tres formas de organizar los elementos en pantalla, de la más sencilla a la más potente. Aprenderás cuándo usar cada uno, porque elegir el layout correcto es la mitad del trabajo: un formulario vertical pide `LinearLayout` a gritos, mientras que una tarjeta de perfil con elementos alineados por todos lados pide `ConstraintLayout`.

La clase 11 presenta a los actores: `TextView`, `Button`, `EditText` e `ImageView`. Son las cuatro vistas que usarás en casi todas las pantallas que construyas en tu vida. Verás sus atributos más importantes en XML: cómo poner texto, tamaños, colores, márgenes, hints, tipos de teclado y cómo cargar imágenes. Al terminar sabrás leer cualquier layout y construir los tuyos sin adivinar atributos.

La clase 12 conecta todo con la lógica: los eventos. Un botón que no hace nada es decoración; aquí aprenderás a darle vida con `onClick` en XML y con listeners en Kotlin, y a validar lo que el usuario escribe antes de aceptarlo. La validación es lo que separa una app de juguete de una app de verdad: nadie quiere una app que se caiga porque el usuario dejó un campo vacío o escribió letras donde iban números.

Por qué importa esta unidad. El usuario juzga tu app en tres segundos, y lo que juzga es la interfaz. No ve tu código elegante ni tu arquitectura; ve botones, textos y colores. Además, Android tiene miles de modelos de teléfono con pantallas de todos los tamaños: tu layout debe verse bien en un teléfono pequeño y en una tableta grande. Entender los layouts es entender cómo lograr eso sin volverte loco creando un diseño para cada pantalla.

Qué sabrás al terminar. Serás capaz de construir layouts con los tres contenedores principales, elegir el adecuado para cada pantalla, usar las cuatro vistas esenciales con sus atributos clave, conectar botones a código Kotlin con listeners, validar entradas del usuario mostrando errores claros, y evitar los errores típicos como vistas encimadas, textos cortados o layouts que se rompen en pantallas pequeñas. En resumen: tus apps van a verse y sentirse como apps de verdad.

Cómo estudiar esta unidad. Cada ejemplo de layout debes probarlo en el editor visual de Android Studio y en el emulador. Cambia un atributo, mira qué pasa. Rompe el layout a propósito y arréglalo. El diseño de interfaces se aprende jugando con los atributos hasta que tu ojo reconoce qué hace cada uno. Y cuando algo se vea mal, el Layout Inspector de Android Studio es tu mejor amigo: te muestra el árbol de vistas real que se dibujó en pantalla.

Explicación detallada

Qué es un layout: el plano de tu pantalla

Un layout es un archivo XML que describe la interfaz de una pantalla: qué elementos contiene y cómo se organizan. No es código que se ejecuta; es una descripción que Android lee y convierte en vistas reales cuando la actividad llama a `setContentView`. Piensa en la diferencia entre el plano de una casa y la casa construida: el XML es el plano, y lo que ves en el teléfono es la casa.

Analogía: si tu pantalla fuera una página de un periódico, el layout sería el diseño editorial que decide dónde va el titular, dónde la foto y dónde el texto. El editor no escribe las noticias (eso es la lógica en Kotlin), pero sin su plano la página sería un caos.

Todo layout tiene una vista raíz, que es el contenedor principal, y dentro de él van las vistas hijas. Los contenedores se llaman `ViewGroup` y las piezas individuales se llaman `View`. Un `ViewGroup` puede contener otros `ViewGroup`, formando un árbol: la raíz, sus hijos, los hijos de sus hijos. Este árbol es exactamente lo que Android recorre para dibujar tu pantalla, de afuera hacia adentro.

Las medidas: `match_parent`, `wrap_content` y `dp`

Antes de ver los contenedores necesitas entender cómo se mide todo en Android. Cada vista tiene `layout_width` (ancho) y `layout_height` (alto), y hay tres formas de definirlos.

`match_parent` significa "ocupa todo el espacio que te dé tu papá". Si un botón tiene `layout_width="match_parent"` dentro de un contenedor que ocupa todo el ancho de la pantalla, el botón se estira de lado a lado. Es como decirle a un niño "ocupa todo el sillón".

`wrap_content` significa "ajústate a lo que mides": la vista será tan grande como necesite su contenido. Un `TextView` con texto corto será pequeño; con texto largo, crecerá. Es como ropa a la medida.

El tercer modo son medidas fijas con dp, como "48dp". El dp (density-independent pixel) es la unidad mágica de Android: mide igual en todos los teléfonos sin importar la densidad de su pantalla. Nunca uses pixeles crudos (px) para tamaños, porque en un teléfono de alta densidad se verían minúsculos. Para textos se usa sp (scale-independent pixel), que además respeta el tamaño de letra que el usuario configuró en su teléfono por accesibilidad. Regla de oro: dp para tamaños de vistas y márgenes, sp para tamaños de texto.

LinearLayout: la fila de la tortillería

LinearLayout es el contenedor más sencillo: acomoda a sus hijos en una sola dirección, uno tras otro, como la fila de una tortillería. Con `android:orientation="vertical"` los apila de arriba hacia abajo; con `"horizontal"` los pone de izquierda a derecha. Es perfecto para formularios, listas de opciones y cualquier pantalla que sea básicamente "una cosa debajo de otra".

Su truco especial es el peso: `layout_weight`. Si dos botones están en un LinearLayout horizontal y ambos tienen `layout_width="0dp"` con `layout_weight="1"`, el espacio se reparte en partes iguales entre ellos, sin importar el tamaño de la pantalla. Es como repartir una pizza en partes iguales: no importa qué tan grande sea la pizza, cada quien recibe lo justo. Los pesos son la forma más fácil de hacer diseños que se adaptan a cualquier pantalla.

Atributos clave de LinearLayout: `orientation` (vertical u horizontal), `gravity` (cómo se alinean los hijos dentro del contenedor) y, en cada hijo, `layout_gravity` (cómo se alinea ese hijo en particular) y `layout_weight` (cuánto espacio extra le toca). La diferencia entre `gravity` y `layout_gravity` confunde a todos al principio: `gravity` es la orden del papá ("todos mis hijos se centran"), `layout_gravity` es la petición del hijo ("yo me quiero ir a la derecha"). Memorízalo así y nunca se te olvida.

Limitación: LinearLayout solo sabe de filas y columnas. Si necesitas un diseño donde un elemento esté a la derecha de otro y debajo de un tercero al mismo tiempo, anidar LinearLayouts se vuelve un enredo de cajas dentro de cajas que es lento de dibujar y difícil de mantener. Para esos casos existen los siguientes dos contenedores.

RelativeLayout: el mapa de relaciones

RelativeLayout posiciona cada hijo en relación con el papá o con sus hermanos: "pon este botón debajo del título", "alineá esta imagen a la derecha del papá", "centra este texto". Usa atributos como `layout_below`, `layout_above`, `layout_toEndOf`, `layout_alignParentTop`, `layout_centerInParent`. Es como dar direcciones en un pueblo: "la tienda está al lado de la iglesia, frente al parque".

Fue durante años el contenedor favorito para pantallas medianamente complejas, porque con un solo nivel de anidamiento logras diseños que en LinearLayout requerirían tres o cuatro contenedores anidados. Su desventaja: cuando hay muchas relaciones cruzadas, el layout se vuelve difícil de leer ("este va debajo de aquel que está a la derecha del otro...") y medirlo dos veces puede hacerlo más lento. Hoy Google recomienda ConstraintLayout para casos complejos, pero RelativeLayout sigue apareciendo en código existente y es perfectamente válido para pantallas sencillas con pocas relaciones.

ConstraintLayout: el arquitecto profesional

ConstraintLayout es el contenedor moderno y recomendado por Google para pantallas complejas. Su idea es brillante: cada vista se posiciona con restricciones (constraints) que la conectan a los bordes del papá o a otras vistas. "El título va pegado arriba del papá, el botón va debajo del título con 16dp de separación y centrado horizontalmente". Con un solo ConstraintLayout plano, sin anidamientos, puedes construir casi cualquier diseño.

Analogía: si LinearLayout es una fila de tortillería y RelativeLayout es dar direcciones en un pueblo, ConstraintLayout es un arquitecto con un plano de coordenadas: cada muro sabe exactamente a qué otros muros está atado. El resultado es un diseño preciso, rápido de dibujar y fácil de ajustar.

Las restricciones se escriben con atributos como `layout_constraintTop_toTopOf`, `layout_constraintBottom_toBottomOf`, `layout_constraintStart_toStartOf`, `layout_constraintEnd_toEndOf`. Cada atributo tiene dos partes: de qué lado de mi vista (Top, Bottom, Start, End) y a qué lado de quién me ato (`toTopOf`, `toBottomOf`, del papá con "parent" o de otra vista con su id). Por ejemplo, `app:layout_constraintTop_toBottomOf="@id/tvTitulo"` significa "mi borde de arriba va pegado al borde de abajo de tvTitulo". Nota que estos atributos usan el prefijo `app:` en lugar de `android:`, porque vienen de una librería; por eso el XML necesita la declaración `xmlns:app="http://schemas.android.com/apk/res-auto"` en la raíz.

Regla de oro de ConstraintLayout: toda vista necesita al menos una restricción horizontal y una vertical. Si dejas una vista sin restricciones, se queda pegada en la esquina superior izquierda (posición 0,0) y el editor te muestra una advertencia. Es el error más común al empezar: arrastras un botón al editor visual, se ve bien ahí, pero al correr la app aparece en la esquina porque nunca le pusiste restricciones reales.

Otras joyas de ConstraintLayout: las cadenas (chains), que reparten un grupo de vistas con un estilo (`spread`, `packed`, `weighted`) como si fueran un LinearLayout pero sin anidar; las guías (guidelines), líneas invisibles a un porcentaje de la pantalla perfectas para diseños responsivos; y las barreras (barriers), que se mueven solas según el tamaño del contenido.

Con estas tres herramientas puedes hacer pantallas que se adaptan a teléfonos y tabletas sin duplicar layouts.

Cuándo usar cada contenedor. Usa `LinearLayout` para secuencias simples: formularios, filas de botones, listas verticales. Usa `RelativeLayout` para pantallas con pocas relaciones claras o cuando mantienes código viejo. Usa `ConstraintLayout` para todo lo demás, especialmente pantallas complejas o que deben verse bien en muchos tamaños. Y recuerda: anidar muchos `LinearLayout` uno dentro de otro es la receta para layouts lentos; si ves más de tres niveles de anidamiento, es momento de migrar a `ConstraintLayout`.

Las vistas protagonistas

`TextView` muestra texto. Parece lo más simple del mundo, pero tiene atributos poderosos: `text` (el contenido), `textSize` en sp, `textColor`, `textStyle` (bold, italic), `gravity` (alineación del texto dentro de la vista), `maxLines` y `ellipsize` (qué hacer cuando el texto es muy largo: cortarlo con "..."). Un detalle que salva vidas: si el texto puede ser largo, pon `android:maxLines="2"` y `android:ellipsize="end"` para que no rompa tu diseño.

`Button` es un botón que el usuario puede tocar. Hereda de `TextView`, así que todos los atributos de texto le sirven, más `background` para el color de fondo. En apps modernas se usa `MaterialButton` para estilos más bonitos, pero el `Button` clásico es perfecto para aprender. Su atributo más importante no está en el XML sino en el código: el listener que define qué pasa al tocarlo, tema de la clase 12.

`EditText` es un campo donde el usuario escribe. Sus atributos clave: `hint` (el texto gris de ayuda que desaparece al escribir, como "Escribe tu nombre"), `inputType` (qué teclado mostrar: text, `textEmailAddress`, `textPassword`, number, phone), `maxLength` y `lines`. El `inputType` es más importante de lo que parece: con `textPassword` el texto se oculta con puntitos, con number aparece el teclado numérico. Elegir el `inputType` correcto es una de esas cosas pequeñas que hacen que tu app se sienta profesional.

`ImageView` muestra imágenes. Sus atributos esenciales: `src` (la imagen a mostrar, normalmente un recurso en la carpeta `drawable` con `@drawable/mi_imagen`) y `scaleType` (cómo se ajusta la imagen al tamaño de la vista: `centerCrop` recorta para llenar, `fitCenter` ajusta completa sin recortar). Para imágenes de internet necesitarás una librería como Coil o Glide, pero para recursos locales `ImageView` solo es suficiente. Ojo: las imágenes pesadas en pantallas pequeñas desperdician memoria; usa imágenes del tamaño adecuado.

Márgenes y rellenos: el aire del diseño

Toda vista tiene dos formas de respirar: el padding (relleno interno) y el layout_margin (margen externo). El padding es el espacio entre el borde de la vista y su contenido: un botón con padding grande se ve gordito y tocable. El margen es el espacio entre la vista y sus vecinas: separa un botón de otro. Analogía: el padding es el aire dentro de una caja de regalo alrededor del regalo; el margen es la distancia entre una caja de regalo y la siguiente en la mesa.

Puedes ponerlos en los cuatro lados a la vez (`padding="16dp"`) o por lado (`paddingTop`, `paddingBottom`, `layout_marginStart`, `layout_marginEnd`). Usa `Start` y `End` en lugar de `Left` y `Right`: así tu app se adapta sola a idiomas que se leen de derecha a izquierda, como el árabe. Es un detalle profesional que cuesta cero trabajo.

El error más común de diseño en principiantes es no usar márgenes: todo pegado, todo apretado, la app se ve barata. La regla práctica: separa los elementos relacionados con 8dp y las secciones diferentes con 16dp o 24dp. El aire es gratis y hace que todo se vea caro.

Eventos: del dibujo a la acción

Un layout sin eventos es un cartel: se ve bonito pero no hace nada. Los eventos son la conexión entre lo que el usuario toca y tu código Kotlin. El evento más usado es el clic en un botón, y hay dos formas de escucharlo.

La forma uno es el atributo `android:onClick` en el XML: le pones el nombre de un método, y Android lo llama cuando tocan la vista. El método debe ser público, recibir una `View` y estar en la `Activity`. Es rápido de escribir pero tiene desventajas: si renombras el método en Kotlin y olvidas el XML, la app se cae en ejecución; además solo sirve para clics simples.

La forma dos, la profesional, es el listener en Kotlin con `setOnClickListener`. Le pasas una lambda, un bloque de código que se ejecuta al hacer clic. Ventajas: el compilador verifica todo, puedes usar la misma lógica para varias vistas, y es lo que verás en el 95 por ciento del código profesional. Es la que usamos en todos los ejemplos de este libro.

Además del clic hay otros eventos útiles: `OnLongClickListener` (mantener presionado), `TextWatcher` (escuchar cada letra que se escribe en un `EditText`, como viste en la unidad 3), `OnCheckedChangeListener` (para switches y checkboxes). Cada vista publica sus eventos y tú decides cuáles escuchar. La filosofía es siempre la misma: la vista avisa "pasó algo" y tu lambda decide qué hacer.

Validar la entrada del usuario: la portería

Validar significa revisar lo que el usuario escribió antes de aceptarlo, como un portero que revisa identificaciones. Las validaciones más comunes: campo vacío (`isBlank`), formato de correo (contiene `@` y punto, o mejor, `Patterns.EMAIL_ADDRESS`), longitud mínima de contraseña, que un número sea realmente un número, y que dos campos coincidan (contraseña y confirmación).

Dónde validar: justo cuando el usuario intenta continuar, típicamente en el clic del botón. Cómo avisar: la forma estándar es la propiedad `error` del `EditText`, que muestra el mensaje en rojo debajo del campo y pone un icono de alerta. Para errores generales que no pertenecen a un campo (como "sin conexión"), se usa un `Toast`, un mensajito flotante que aparece y desaparece solo.

Buena práctica: valida de lo más barato a lo más caro. Primero lo local (campos vacíos, formatos), que no cuesta nada; al final lo que requiere red o base de datos. Y nunca confíes solo en la validación visual: si el dato va a un servidor, el servidor debe validar también, porque un usuario avanzado puede saltarse tu app.

Errores comunes de principiantes y cómo evitarlos

Error uno: vistas sin restricciones en `ConstraintLayout`. Síntoma: todo aparece amontonado en la esquina superior izquierda al correr la app aunque en el editor se vea bien. Solución: cada vista necesita restricción horizontal y vertical; usa el botón de "infer constraints" del editor o ponlas a mano.

Error dos: confundir `gravity` con `layout_gravity`. Síntoma: el texto no se centra donde quieres. Solución: `gravity` alinea el contenido dentro de la vista; `layout_gravity` alinea la vista dentro de su papá.

Error tres: usar `px` o números sin unidad. Síntoma: en tu teléfono se ve bien pero en otro se ve gigante o minúsculo. Solución: `dp` para tamaños, `sp` para textos, siempre.

Error cuatro: anidar `LinearLayouts` como muñecas rusas. Síntoma: la pantalla tarda en dibujarse y el XML es ilegible. Solución: si pasas de tres niveles, migra a `ConstraintLayout`.

Error cinco: olvidar el `hint` o el `inputType` en los `EditText`. Síntoma: el usuario no sabe qué escribir y le aparece el teclado equivocado. Solución: `hint` descriptivo e `inputType` correcto siempre.

Error seis: poner android:onClick y luego renombrar el método en Kotlin. Síntoma: crash al tocar el botón con un error de método no encontrado. Solución: prefiere setOnClickListener en Kotlin.

Error siete: no validar y convertir texto a número directamente. Síntoma: NumberFormatException cuando el usuario escribe letras. Solución: usa toIntOrNull y maneja el nulo con un mensaje de error amable.

Buenas prácticas de la unidad

Primera: diseña primero en papel o en tu cabeza qué layout usarás antes de abrir el editor. Treinta segundos de planeación ahorran treinta minutos de arrastrar vistas.

Segunda: nombra los ids con convención clara: prefijo por tipo (tv para TextView, btn para Button, et para EditText, iv para ImageView) seguido de su función (tvNombre, btnGuardar). Tu yo del futuro te lo agradecerá.

Tercera: extrae los textos al archivo strings.xml en lugar de escribirlos directo en el layout. Así tu app se puede traducir y los textos se corrigen en un solo lugar.

Cuarta: prueba tu layout en al menos dos tamaños de pantalla: un teléfono pequeño y uno grande. El editor de Android Studio te deja previsualizar varios dispositivos a la vez.

Quinta: el aire es diseño. Márgenes consistentes de 8dp y 16dp hacen que cualquier pantalla se vea profesional sin esfuerzo.

Ejemplos paso a paso

Ejemplo 1: pantalla de login con LinearLayout

Vamos a construir la pantalla más clásica de todas: un login con logo, campo de correo, campo de contraseña y botón de entrar. Es un LinearLayout vertical porque todo va uno debajo de otro, como una fila de tortillería.

El archivo activity_login.xml completo:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center"
```



```

        android:padding="32dp">

        <ImageView
            android:id="@+id/ivLogo"
            android:layout_width="96dp"
            android:layout_height="96dp"
            android:src="@drawable/ic_logo"
            android:contentDescription="Logo de la academia" />

        <TextView
            android:id="@+id/tvTitulo"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Bienvenido de nuevo"
            android:textSize="24sp"
            android:textStyle="bold"
            android:layout_marginTop="16dp" />

        <TextView
            android:id="@+id/tvSubtitulo"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Inicia sesion para continuar"
            android:textSize="14sp"
            android:layout_marginTop="4dp" />

        <EditText
            android:id="@+id/etCorreo"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Correo electronico"
            android:inputType="textEmailAddress"
            android:layout_marginTop="24dp" />

        <EditText
            android:id="@+id/etPassword"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Contrasena"
            android:inputType="textPassword"
            android:layout_marginTop="12dp" />

        <Button
            android:id="@+id/btnEntrar"
            android:layout_width="match_parent"

```

```

        android:layout_height="wrap_content"
        android:text="Entrar"
        android:layout_marginTop="24dp" />

<TextView
    android:id="@+id/tvRegistro"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="No tienes cuenta? Registrate"
    android:textSize="14sp"
    android:layout_marginTop="16dp" />

</LinearLayout>

```

Explicación del XML. La raíz es un `LinearLayout` vertical que ocupa toda la pantalla, con `gravity="center"` para centrar a sus hijos horizontalmente y `padding` de 32dp para que nada toque los bordes. El `ImageView` del logo mide 96dp fijos y lleva `contentDescription`, que es el texto que leen los lectores de pantalla para personas con discapacidad visual: ponerlo es buena práctica de accesibilidad. Los `TextView` usan `sp` para el tamaño y uno lleva `textStyle` `bold`. Los `EditText` llevan `hint` e `inputType` correctos: `textEmailAddress` muestra un teclado con `@` a la mano, `textPassword` oculta lo escrito. Los márgenes crecen donde hay separación lógica: 24dp antes del primer campo y del botón, 12dp entre campos. El botón ocupa todo el ancho (`match_parent`) porque en un login el botón principal suele ser grande y fácil de tocar.

Ahora la lógica en `LoginActivity.kt`, con validación completa:

```

package com.lucasega.miprimeraapp

import android.os.Bundle
import android.util.Patterns
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityLoginBinding

class LoginActivity : AppCompatActivity() {

    private lateinit var binding: ActivityLoginBinding

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

        binding.btnEntrar.setOnClickListener {
            if (validarFormulario()) {

```

```

        val correo = binding.etCorreo.text.toString().trim()
        Toast.makeText(this, "Bienvenido, $correo", Toast.LENGTH_SHORT).show()
    }
}

private fun validarFormulario(): Boolean {
    val correo = binding.etCorreo.text.toString().trim()
    val password = binding.etPassword.text.toString()

    if (correo.isEmpty()) {
        binding.etCorreo.error = "Escribe tu correo"
        return false
    }

    if (!Patterns.EMAIL_ADDRESS.matcher(correo).matches()) {
        binding.etCorreo.error = "Ese correo no parece valido"
        return false
    }

    if (password.length < 6) {
        binding.etPassword.error = "Minimo 6 caracteres"
        return false
    }

    return true
}
}

```

Explicación línea por línea. En onCreate se infla el binding y se configura el listener del botón: al hacer clic se llama a validarFormulario y solo si regresa true se continúa (aquí mostramos un Toast de bienvenida; en una app real aquí irías a la pantalla principal con un Intent). La función validarFormulario revisa en orden de barato a caro: primero si el correo está vacío, luego si tiene formato válido usando Patterns.EMAIL_ADDRESS, una utilidad de Android que ya sabe cómo se ve un correo válido, y al final si la contraseña mide al menos 6 caracteres. Cada fallo marca el campo con error y regresa false de inmediato, así el usuario ve un problema a la vez, del más básico al más específico. Nota el trim en el correo para ignorar espacios accidentales, y que la contraseña no lleva trim porque los espacios podrían ser parte intencional de ella.

Ejemplo 2: tarjeta de perfil con ConstraintLayout

Ahora algo más elaborado: una tarjeta de perfil como las de redes sociales, con foto a la izquierda, nombre y descripción a la derecha, y un botón de seguir debajo. Con ConstraintLayout lo hacemos en un solo nivel, sin anidar nada.

El archivo activity_perfil.xml completo:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="24dp">

    <ImageView
        android:id="@+id/ivFoto"
        android:layout_width="96dp"
        android:layout_height="96dp"
        android:src="@drawable/ic_persona"
        android:contentDescription="Foto de perfil"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/tvNombre"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Esteban Rivera"
        android:textSize="22sp"
        android:textStyle="bold"
        app:layout_constraintStart_toEndOf="@id/ivFoto"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="@id/ivFoto"
        android:layout_marginStart="16dp" />

    <TextView
        android:id="@+id/tvDescripcion"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Estudiante de desarrollo Android en Lucas EGA Academy"
        android:textSize="14sp"
        android:maxLines="2"
        android:ellipsize="end"
        app:layout_constraintStart_toEndOf="@id/ivFoto"
        app:layout_constraintEnd_toEndOf="parent"
```

```

        app:layout_constraintTop_toBottomOf="@id/tvNombre"
        android:layout_marginStart="16dp"
        android:layout_marginTop="4dp" />

<Button
    android:id="@+id/btnSeguir"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="Seguir"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/ivFoto"
    android:layout_marginTop="24dp" />

<TextView
    android:id="@+id/tvSeguidores"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="0 seguidores"
    android:textSize="14sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/btnSeguir"
    android:layout_marginTop="8dp" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

Explicación del XML. La raíz declara los dos namespaces: android y app (este último obligatorio para las restricciones). La foto se ata con su inicio al inicio del papá y su arriba al arriba del papá: queda en la esquina superior izquierda. El nombre usa `layout_width="0dp"`, que en `ConstraintLayout` significa "estírate según tus restricciones": va del final de la foto (`Start_toEndOf`) al final del papá (`End_toEndOf`), y su arriba se alinea con el arriba de la foto, quedando a la misma altura. La descripción va debajo del nombre (`Top_toBottomOf`) con el mismo rango horizontal, y lleva `maxLines="2"` con `ellipsize="end"` para que un texto largo no rompa la tarjeta. El botón va debajo de la foto, estirado de lado a lado (`0dp` entre inicio y fin del papá), y el contador de seguidores va centrado horizontalmente debajo del botón (inicio y fin atados al papá con ancho `wrap_content`: al tener ambas restricciones, se centra solo). Cada vista tiene su restricción horizontal y vertical: ninguna queda huérfana en la esquina.

La lógica en `PerfilActivity.kt`: un botón que alterna entre Seguir y Siguiendo, con contador:

```

package com.lucasega.miprimeraapp

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityPerfilBinding

class PerfilActivity : AppCompatActivity() {

    private lateinit var binding: ActivityPerfilBinding
    private var siguiendo = false
    private var seguidores = 0

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

        binding.btnSeguir.setOnClickListener {
            siguiendo = !siguiendo
            if (siguiendo) {
                seguidores++
                binding.btnSeguir.text = "Siguiendo"
            } else {
                seguidores--
                binding.btnSeguir.text = "Seguir"
            }
            binding.tvSeguidores.text = "$seguidores seguidores"
        }
    }
}

```

Explicación. Dos variables guardan el estado: siguiendo (booleano) y seguidores (entero). Al hacer clic se invierte el booleano con el operador !, se ajusta el contador y se actualizan los dos textos. Es el patrón de "interruptor" que verás en botones de favorito, me gusta y seguir de todas las apps. Como reto, combina esto con la unidad 3: guarda siguiendo y seguidores en onSaveInstanceState para que sobrevivan a la rotación.

Ejemplo 3: RelativeLayout en acción

Para que veas el segundo contenedor, una fila de encabezado con RelativeLayout: un título a la izquierda y un botón de cerrar a la derecha, ambos centrados verticalmente:

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="16dp">

    <TextView
        android:id="@+id/tvEncabezado"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Mis cursos"
        android:textSize="20sp"
        android:textStyle="bold"
        android:layout_alignParentStart="true"
        android:layout_centerVertical="true" />

    <Button
        android:id="@+id/btnCerrar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Cerrar"
        android:layout_alignParentEnd="true"
        android:layout_centerVertical="true" />

</RelativeLayout>

```

Explicación. El título se alinea al inicio del papá y se centra verticalmente; el botón se alinea al final del papá y también se centra verticalmente. Con dos atributos por vista logramos una fila de encabezado típica, sin anidar nada. Si esto lo hicieras con LinearLayout necesitarías un horizontal con pesos o gravedad; con RelativeLayout es directo. Para pantallas con pocas relaciones como esta, RelativeLayout sigue siendo una opción válida y legible.

Ejemplo 4: repartir espacio con pesos y una calculadora que no se cae

Este ejemplo une dos temas: los pesos de LinearLayout y la validación numérica. Una mini calculadora que suma dos números, con los botones repartidos equitativamente usando `layout_weight`.

El layout `activity_calculadora.xml`:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"

```

```

        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="24dp"
        android:gravity="center">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Suma dos numeros"
            android:textSize="22sp"
            android:textStyle="bold" />

        <EditText
            android:id="@+id/etNumero1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Primer numero"
            android:inputType="number"
            android:layout_marginTop="24dp" />

        <EditText
            android:id="@+id/etNumero2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Segundo numero"
            android:inputType="number"
            android:layout_marginTop="12dp" />

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="horizontal"
            android:layout_marginTop="16dp">

            <Button
                android:id="@+id/btnSumar"
                android:layout_width="0dp"
                android:layout_height="wrap_content"
                android:layout_weight="1"
                android:text="Sumar" />

            <Button
                android:id="@+id/btnLimpiar"
                android:layout_width="0dp"
                android:layout_height="wrap_content"

```



```

        android:layout_weight="1"
        android:text="Limpiar"
        android:layout_marginStart="12dp" />

</LinearLayout>

<TextView
    android:id="@+id/tvResultado"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Resultado: - -"
    android:textSize="20sp"
    android:layout_marginTop="24dp" />

</LinearLayout>

```

Explicación del XML. Los dos botones viven en un `LinearLayout` horizontal interno. Ambos tienen `layout_width="0dp"` y `layout_weight="1"`: el ancho cero con peso significa "reparte todo el espacio disponible en proporción a mi peso"; como ambos pesan 1, cada uno ocupa exactamente la mitad. El botón Limpiar lleva un margen de 12dp para que no se peguen. Este es el patrón estándar para filas de botones gemelos y funciona en cualquier tamaño de pantalla: en un teléfono chico cada botón mide la mitad del chico, en una tableta la mitad de la tableta.

La lógica en `CalculadoraActivity.kt`, con validación numérica a prueba de usuarios creativos:

```

package com.lucasega.miprimeraapp

import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityCalculadoraBinding

class CalculadoraActivity : AppCompatActivity() {

    private lateinit var binding: ActivityCalculadoraBinding

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

        binding.btnSumar.setOnClickListener {
            val n1 = binding.etNumero1.text.toString().toIntOrNull()
            val n2 = binding.etNumero2.text.toString().toIntOrNull()

            if (n1 == null) {
                binding.etNumero1.error = "Escribe un numero valido"
                return@setOnClickListener
            }
            if (n2 == null) {
                binding.etNumero2.error = "Escribe un numero valido"
                return@setOnClickListener
            }

            val suma = n1 + n2
            binding.tvResultado.text = "Resultado: $suma"
        }

        binding.btnLimpiar.setOnClickListener {
            binding.etNumero1.text.clear()
            binding.etNumero2.text.clear()
            binding.tvResultado.text = "Resultado: --"
            binding.etNumero1.requestFocus()
        }
    }
}

```

Explicación línea por línea. La estrella es `toIntOrNull`: intenta convertir el texto a entero y si no puede (campo vacío, letras, símbolos) regresa nulo en lugar de lanzar una excepción. Es la

forma segura de leer números en Kotlin y elimina de un plumazo el temido `NumberFormatException`. Si `n1` es nulo se marca el error en el campo y se sale del listener con `return@setOnClickListener`, que es la forma de hacer "return" dentro de una lambda en Kotlin (un return solo saldría de la lambda, y necesitas especificar de cuál). Si ambos son válidos se suman y se muestra el resultado con interpolación. El botón Limpiar vacía los campos con `text.clear()`, restaura el texto del resultado y devuelve el foco al primer campo con `requestFocus` para que el usuario siga escribiendo sin tocar la pantalla. Este ejemplo es oro puro: `toIntOrNull` más validación por campo es el patrón que usarás en cada formulario numérico de tu carrera.

Práctica

Ejercicio 1: replica el login

Sin mirar el ejemplo 1, construye desde cero la pantalla de login: logo (puedes usar cualquier drawable), título, dos `EditText` con hint e `inputType` correctos, y un botón de entrar que ocupe todo el ancho. Luego compara tu XML con el del ejemplo y anota las diferencias.

Solución: lo esencial que debe tener tu layout es un `LinearLayout` vertical con padding generoso (24dp a 32dp), los `EditText` con `android:hint` y `android:inputType="textEmailAddress"` y `"textPassword"`, márgenes de 12dp a 24dp separando secciones, y el botón con `layout_width="match_parent"`. Si tu versión usa dp para tamaños, sp para textos y el botón no quedó del ancho de su contenido (`wrap_content`), vas bien. Las diferencias de redacción en los textos no importan; lo que importa es la estructura: contenedor, orden, medidas y atributos correctos. Si te faltó el `contentDescription` en la imagen, agrégalo: es el detalle que distingue al profesional.

Ejercicio 2: tres botones con pesos

Crea un `LinearLayout` horizontal con tres botones: "Rojo", "Verde" y "Azul", que ocupen cada uno un tercio exacto del ancho en cualquier pantalla. Al tocar cada botón, un `TextView` debajo debe mostrar el texto "Elegiste: [color]".

Solución: cada botón lleva `android:layout_width="0dp"` y `android:layout_weight="1"` dentro del `LinearLayout` horizontal. Los tres pesan lo mismo, así que el espacio se divide en tercios perfectos. En la actividad, un solo listener puede servir para los tres si comparas qué botón se tocó, pero lo más claro para empezar es un listener por botón:

```
binding.btnRojo.setOnClickListener {  
    binding.tvElegido.text = "Elegiste: Rojo"  
}
```

Repite para verde y azul. La lección del ejercicio es el peso: 0dp más peso igual es la fórmula mágica del reparto proporcional. Como variante avanzada, dale al botón verde `layout_weight="2"` y observa cómo ahora ocupa la mitad mientras los otros ocupan un cuarto cada uno: los pesos son proporciones, no porcentajes fijos.

Ejercicio 3: tarjeta con ConstraintLayout desde cero

Construye una tarjeta de "curso": una imagen cuadrada a la izquierda, el nombre del curso a la derecha de la imagen, la duración debajo del nombre, y un botón "Ver curso" estirado debajo de todo. Hazlo con un solo ConstraintLayout, sin anidar. Verifica en el emulador que nada aparezca en la esquina superior izquierda.

Solución: la imagen se ata al inicio y arriba del papá. El nombre va con `Start_toEndOf` de la imagen y `End_toEndOf` del papá, ancho 0dp, arriba alineado con la imagen. La duración va debajo del nombre con `Top_toBottomOf`, mismo rango horizontal. El botón va debajo de la imagen con `Top_toBottomOf`, estirado de lado a lado con ancho 0dp entre inicio y fin del papá. Si algo salió en la esquina, revisa que esa vista tenga sus dos restricciones; el 99 por ciento de las veces falta una. Este ejercicio es el que más se parece al trabajo real: las tarjetas son el pan de cada día en apps de cursos, tiendas y redes sociales.

Ejercicio 4: formulario con validación completa

Crea un formulario de registro con nombre, correo, contraseña y confirmación de contraseña. Valida: ningún campo vacío, correo con formato válido, contraseña de mínimo 8 caracteres, y que la confirmación coincida con la contraseña. Muestra el error en el campo correspondiente y solo avanza si todo pasa.

Solución: la función de validación encadena las revisiones:

```

private fun validar(): Boolean {
    val nombre = binding.etNombre.text.toString().trim()
    val correo = binding.etCorreo.text.toString().trim()
    val pass = binding.etPass.text.toString()
    val confirm = binding.etConfirm.text.toString()

    if (nombre.isEmpty()) {
        binding.etNombre.error = "Escribe tu nombre"
        return false
    }
    if (!Patterns.EMAIL_ADDRESS.matcher(correo).matches()) {
        binding.etCorreo.error = "Correo no valido"
        return false
    }
    if (pass.length < 8) {
        binding.etPass.error = "Minimo 8 caracteres"
        return false
    }
    if (pass != confirm) {
        binding.etConfirm.error = "No coincide con la contrasena"
        return false
    }
    return true
}

```

Fíjate en el orden: de lo más simple a lo más específico, y cada fallo regresa false inmediatamente para mostrar un error a la vez. La comparación `pass != confirm` es directa porque son strings. Necesitas importar `android.util.Patterns`. Este es exactamente el tipo de formulario que te pedirán en un examen o en tu primer trabajo.

Ejercicio 5: encuentra el bug en el layout

Este XML tiene cuatro errores. Encuéntralos:

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hola"
        android:textSize="20px" />

    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="textPassword" />

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Entrar"
        android:onClick="entrar" />

</LinearLayout>

```

Solución: error uno, el TextView usa textSize="20px": los pixeles crudos se ven distinto en cada pantalla; debe ser 20sp. Error dos, el EditText no tiene hint: el usuario no sabe qué escribir ahí; además le falta id, sin el cual no podrás leerlo desde Kotlin. Error tres, el botón usa android:onClick="entrar": funciona, pero si renombras el método en Kotlin la app se cae; lo profesional es setOnClickListener en código. Error cuatro, más sutil: el layout no tiene padding, así que todo quedará pegado a los bordes; agrega android:padding="16dp" o más a la raíz. Este ejercicio entrena tu ojo de revisor, una habilidad que vale oro en equipos de trabajo.

Ejercicio 6: mini-proyecto, pantalla de producto

Construye la pantalla de detalle de un producto para una tiendita: arriba una imagen grande del producto, debajo el nombre con estilo bold, debajo el precio en grande y de otro color, debajo una descripción de máximo 3 líneas con puntos suspensivos si se pasa, y al final dos botones gemelos con pesos: "Agregar al carrito" y "Comprar ahora". Al tocar cada botón muestra un Toast distinto. Elige tú el contenedor raíz y justifica tu elección por escrito.

Solución: la elección natural es un LinearLayout vertical, porque todo va uno debajo de otro en secuencia; no hay relaciones cruzadas que justifiquen ConstraintLayout. La imagen lleva

scaleType="centerCrop" y un tamaño fijo en dp o match_parent con altura fija. El precio usa textColor con un color de tu archivo colors.xml y textSize grande en sp. La descripción lleva android:maxLines="3" y android:ellipsize="end". Los botones van en un LinearLayout horizontal interno con 0dp y peso 1 cada uno, igual que en el ejemplo 4. Los Toast se muestran con Toast.makeText(this, "Agregado al carrito", Toast.LENGTH_SHORT).show(). Si justificaste el LinearLayout por la secuencia simple, entendiste el criterio más importante de la unidad: el layout se elige según la forma del diseño, no por moda.

Más layouts y herramientas profesionales

ScrollView: cuando la pantalla no alcanza

Hay pantallas con tanto contenido que no caben en un teléfono chico: un formulario largo, los términos y condiciones, una receta. La solución es ScrollView, un contenedor que hace su contenido desplazable verticalmente. Su regla es estricta: solo puede tener un hijo directo. Normalmente ese hijo es un LinearLayout vertical con todo el contenido adentro.

```

<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="24dp">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Términos y condiciones"
            android:textSize="22sp"
            android:textStyle="bold" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Aquí va el texto larguísimo..."
            android:textSize="14sp"
            android:layout_marginTop="16dp" />

        <Button
            android:id="@+id/btnAceptar"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Aceptar"
            android:layout_marginTop="24dp" />

    </LinearLayout>

</ScrollView>

```

Explicación. El `ScrollView` ocupa toda la pantalla y su único hijo, el `LinearLayout`, tiene `layout_height="wrap_content"`: esto es clave, porque si el hijo midiera `match_parent` nunca habría nada que desplazar. El contenido puede ser tan largo como quieras; el usuario lo recorre con el dedo. Errores típicos: poner dos hijos directos en el `ScrollView` (la app se cae), o ponerle `match_parent` al hijo (el scroll no funciona). Para desplazamiento horizontal existe `HorizontalScrollView`, útil para tiras de imágenes, pero úsalo con moderación porque el scroll horizontal confunde a muchos usuarios.

Buena práctica: si tu pantalla cabe justo en un teléfono grande pero no en uno chico, envuélvela en ScrollView desde el principio. Es más barato prevenir que arreglar después cuando un usuario con teléfono pequeño reporte que el botón de "Guardar" no se ve.

Material Design básico: colores, temas y estilos

Hasta ahora tus pantallas funcionan, pero se ven "de fábrica". Material Design es el lenguaje visual de Android: reglas de color, forma y movimiento que hacen que las apps se vean profesionales. No necesitas dominarlo todo hoy, pero sí lo esencial: colores, temas y estilos.

Los colores de tu app viven en `res/values/colors.xml`. Define ahí tu paleta: un color primario (tu marca), uno secundario (acentos) y los básicos. Luego úsalos en los layouts con `@color/mi_primario` en lugar de códigos hexadecimales regados por todos lados. Si mañana cambias de marca, editas un archivo y toda la app se actualiza. Lo mismo con los textos: `res/values/strings.xml` centraliza cada texto con `@string/bienvenida`. Esta es la práctica que separa el código de estudiante del código profesional.

El tema de tu app se define en `res/values/themes.xml` y se aplica en el manifiesto. El tema controla el color de la barra superior, el estilo de los botones y si tu app es clara u oscura. Cuando creas un proyecto, Android Studio genera un tema Material que ya se ve decente; tu trabajo es personalizar sus colores primarios.

Los estilos (`res/values/styles.xml` o dentro de `themes.xml`) agrupan atributos repetidos. Si tus diez botones principales son iguales, crea un estilo con el color, el tamaño de texto y los márgenes, y aplícalo con `style="@style/MiBoton"`. Cambias el estilo una vez y los diez botones se actualizan. Es como el CSS de la web: no repitas, reutiliza.

Un consejo visual que vale oro: elige dos colores y una sola familia de tamaños de texto (por ejemplo 14sp para cuerpo, 20sp para títulos, 28sp para encabezados grandes). La consistencia se ve profesional; el arcoíris se ve amateur. Y respeta el modo oscuro: prueba tu app con el tema oscuro del teléfono, porque muchos usuarios lo usan de noche y los textos negros sobre fondos que creías blancos pueden volverse invisibles.

Ejemplo 5: formulario largo con ScrollView y estilos

Vamos a juntar ScrollView, estilos y validación en un registro extendido. Primero el estilo en `res/values/styles.xml`:

```
<style name="CampoFormulario">
    <item name="android:layout_width">match_parent</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:layout_marginTop">12dp</item>
</style>
```

Ahora el layout activity_registro.xml: un ScrollView con un LinearLayout que usa el estilo en cada campo:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="24dp">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Crea tu cuenta"
            android:textSize="26sp"
            android:textStyle="bold" />

        <EditText
            android:id="@+id/etNombre"
            style="@style/CampoFormulario"
            android:hint="Nombre completo"
            android:inputType="textPersonName" />

        <EditText
            android:id="@+id/etCorreo"
            style="@style/CampoFormulario"
            android:hint="Correo electronico"
            android:inputType="textEmailAddress" />

        <EditText
            android:id="@+id/etTelefono"
            style="@style/CampoFormulario"
            android:hint="Telefono"
            android:inputType="phone" />

    </LinearLayout>

</ScrollView>
```

```

        <EditText
            android:id="@+id/etPass"
            style="@style/CampoFormulario"
            android:hint="Contraseña (mínimo 8)"
            android:inputType="textPassword" />

        <Button
            android:id="@+id/btnRegistrar"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Registrarme"
            android:layout_marginTop="24dp" />

    </LinearLayout>

</ScrollView>

```

Explicación. Cada EditText aplica `style="@style/CampoFormulario"` y ya no repite ancho, alto ni margen: si mañana decides que los campos lleven 16dp de separación, cambias el estilo y listo. Los `inputType` son variados a propósito: `textPersonName` capitaliza la primera letra, `phone` muestra el teclado telefónico. El `ScrollView` garantiza que en un teléfono pequeño el botón `Registrarme` siga siendo alcanzable con scroll. La actividad de este ejemplo reutiliza la lógica de validación del ejercicio 4, agregando la validación del teléfono (mínimo 7 dígitos, solo números). Este ejemplo representa cómo se construyen los formularios en el mundo real: estilos para no repetir, scroll para no romper, validación para no fallar.

Ejercicio 7: pantalla con scroll que valida

Toma el ejemplo 5 y complétalo: agrega un campo de dirección con `inputType textPostalAddress`, valida que el teléfono tenga al menos 7 dígitos usando una expresión regular o una revisión manual, y haz que al registrar con éxito se abra una segunda pantalla que muestre todos los datos recibidos con extras. Prueba en el emulador pequeño girando la pantalla.

Solución: el campo dirección lleva `android:id="@+id/etDireccion"` y el mismo estilo. Para el teléfono, una validación sencilla es quitar espacios y guiones y revisar que todo sean dígitos:

```
val tel = binding.etTelefono.text.toString().trim().replace(" ", "").replace("-", "")
if (tel.length < 7 || !tel.all { it.isDigit() }) {
    binding.etTelefono.error = "Telefono no valido"
    return false
}
```

La función `all` con `it.isDigit` revisa que cada carácter sea dígito: es Kotlin idiomático y legible. Al registrar, manda cada dato con `putExtra` y sus constantes, y la segunda pantalla los muestra cómo en la unidad 3. Al girar, los `EditText` con `id` conservan su texto automáticamente gracias al `ScrollView` y al `LinearLayout` con `wrap_content`. Este ejercicio une las unidades 3 y 4: navegación con datos más interfaces que sobreviven a la rotación.

Ejercicio 8: rediseña con `ConstraintLayout`

Toma tu pantalla de login del ejercicio 1 (hecha con `LinearLayout`) y reconstrúyela con un solo `ConstraintLayout`, sin anidar. Compara ambos XML: ¿cuál es más corto? ¿Cuál te resultó más fácil de ajustar?

Solución: el logo se centra arriba con `Start_toStartOf` y `End_toEndOf` del papá más `Top_toTopOf`. El título va debajo del logo centrado igual. Cada campo va debajo del anterior, estirado con `ancho 0dp` entre inicio y fin del papá. El botón igual, debajo del último campo. En líneas de código el `ConstraintLayout` suele ser más largo por los atributos, pero tiene cero anidamiento y cada elemento se mueve de forma independiente: cambiar la posición del logo no arrastra a los demás, cosa que en `LinearLayout` a veces sí pasa. No hay respuesta única sobre cuál es mejor: para secuencias simples `LinearLayout` es más rápido de escribir; para ajustes finos `ConstraintLayout` da más control. El profesional sabe usar ambos y elegir según el diseño.

Para llevar más lejos

Rendimiento: por qué importa el árbol de vistas

Cada vista de tu layout cuesta tiempo de medición y dibujado. Un árbol profundo (muchos contenedores anidados) tarda más en dibujarse que uno plano, y en teléfonos baratos eso se siente como una app lenta. La herramienta para diagnosticarlo es el `Layout Inspector` de `Android Studio`: con la app corriendo, abre `View`, `Tool Windows`, `Layout Inspector` y verás el árbol real de tu pantalla con los tiempos de cada vista. Si ves ramas de 5 o 6 niveles de `LinearLayouts` anidados, es momento de aplanarlos con `ConstraintLayout`.

Otra causa de lentitud: fondos innecesarios. Cada `android:background` que pones obliga a dibujar una capa extra. Si tu `LinearLayout` tiene fondo blanco y cada hijo también, estás pintando blanco sobre blanco varias veces (overdraw). El depurador de GPU de Android Studio pinta la pantalla con colores según el overdraw: si ves mucho rojo, simplifica fondos. No te obsesiones al principio, pero ten el concepto en mente: las apps rápidas cuidan su árbol de vistas.

strings.xml: tu app lista para traducirse

Acostúmbrate desde hoy: ningún texto directo en el layout. Cada texto va a `res/values/strings.xml` con un nombre claro:

```
<string name="bienvenida">Bienvenido de nuevo</string>
<string name="entrar">Entrar</string>
<string name="hint_correo">Correo electronico</string>
```

Y en el layout: `android:text="@string/bienvenida"`. Ventajas: corriges un texto en un solo lugar aunque aparezca en diez pantallas, y cuando quieras tu app en inglés creas `res/values-en/strings.xml` con las traducciones y Android elige solo según el idioma del teléfono. Las apps profesionales viven así; las de estudiante llevan textos quemados en el XML. Tú decides de qué lado estar desde tu primer proyecto serio.

El editor visual y el modo texto: usa los dos

Android Studio te deja diseñar layouts de dos formas: arrastrando vistas en el editor visual (modo Design) o escribiendo XML (modo Code), con un modo dividido que muestra ambos. El visual es rápido para prototipar; el texto te da control total. Mi recomendación: aprende en modo texto, como en este libro, porque así entiendes qué hace cada atributo. Cuando domines el XML, el editor visual se vuelve un acelerador, no una muleta. Y cuando el visual haga algo raro (una vista que salta sola), cambia a texto: ahí siempre está la verdad.

Preguntas frecuentes de estudiantes

Pregunta: ¿`LinearLayout` o `ConstraintLayout` para un formulario simple? Respuesta: `LinearLayout`. Es menos código, más legible y perfectamente rápido. Reserva `ConstraintLayout` para cuando el diseño tenga relaciones cruzadas o deba adaptarse a muchos tamaños.

Pregunta: ¿por qué mi texto se sale de la pantalla en un teléfono chico? Respuesta: casi siempre es un ancho fijo en dp muy grande o un texto sin maxLines. Usa Odp con restricciones o wrap_content con límites, y prueba en el emulador más pequeño que soportes.

Pregunta: ¿cómo centro una vista? Respuesta: depende del papá. En LinearLayout vertical, el hijo lleva android:layout_gravity="center_horizontal". En ConstraintLayout, ata su inicio y su fin al papá con ancho wrap_content. En el papá, android:gravity="center" centra a todos los hijos.

Pregunta: ¿qué hago si el teclado tapa mi botón? Respuesta: envuelve la pantalla en ScrollView para que el usuario pueda desplazarse, y en el manifiesto agrega android:windowSoftInputMode="adjustResize" a la actividad para que el layout se encoja cuando aparece el teclado.

Pregunta: ¿Button o MaterialButton? Respuesta: para aprender, Button clásico. Para apps que publiques, MaterialButton, que trae estilos modernos, esquinas redondeadas y estados de presionado sin esfuerzo extra. La lógica del listener es idéntica.

CheckBox, Switch y RadioButton: vistas de selección

Además de las cuatro vistas protagonistas hay tres que aparecen en casi todas las apps: CheckBox (casilla de verificación para aceptar términos o elegir varias opciones), Switch (interruptor de encendido/apagado para ajustes) y RadioButton (opciones excluyentes dentro de un RadioGroup, como elegir un plan: básico, pro o premium).

El CheckBox y el Switch comparten el atributo checked (empieza activado o no) y el evento setOnCheckedChangeListener, que te avisa cada vez que el usuario lo cambia:

```
binding.swNotificaciones.setOnCheckedChangeListener { _, activado ->
    if (activado) {
        Toast.makeText(this, "Notificaciones activadas", Toast.LENGTH_SHORT).show()
    } else {
        Toast.makeText(this, "Notificaciones desactivadas",
            Toast.LENGTH_SHORT).show()
    }
}
```

La lambda recibe dos parámetros: la vista y el nuevo estado booleano; el guion bajo indica que ignoramos el primero. Este patrón es el estándar para pantallas de ajustes.

Los RadioButton van dentro de un RadioGroup, que garantiza que solo uno esté seleccionado a la vez. Para saber cuál eligió el usuario, el grupo ofrece checkedRadioButtonId:

```

val seleccionado = binding.rgPlan.checkedRadioButtonId
val plan = when (seleccionado) {
    R.id.rbBasico -> "Básico"
    R.id.rbPro -> "Pro"
    R.id.rbPremium -> "Premium"
    else -> "Ninguno"
}

```

El when de Kotlin es perfecto aquí: limpio y legible. Si checkedRadioButtonId regresa -1 significa que no hay selección, y el else lo maneja. Error común: poner RadioButtons sueltos sin RadioGroup; entonces el usuario puede seleccionar varios a la vez y tu lógica de "elige un plan" se rompe.

Ejemplo 6: pantalla de ajustes completa

Juntemos las vistas de selección en una pantalla de ajustes real, con LinearLayout y un RadioGroup:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="24dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Ajustes"
        android:textSize="24sp"
        android:textStyle="bold" />

    <Switch
        android:id="@+id/swNotificaciones"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Notificaciones"
        android:layout_marginTop="24dp" />

    <Switch
        android:id="@+id/swModoOscuro"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Modo oscuro"

```

```

        android:layout_marginTop="12dp" />

<CheckBox
    android:id="@+id/cbTerminos"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Acepto los términos y condiciones"
    android:layout_marginTop="24dp" />

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Tu plan:"
    android:textSize="16sp"
    android:textStyle="bold"
    android:layout_marginTop="24dp" />

<RadioGroup
    android:id="@+id/rgPlan"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp">

    <RadioButton
        android:id="@+id/rbBasico"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Básico (gratis)" />

    <RadioButton
        android:id="@+id/rbPro"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Pro ($99/mes)" />

    <RadioButton
        android:id="@+id/rbPremium"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Premium ($199/mes)" />

</RadioGroup>

<Button
    android:id="@+id/btnGuardarAjustes"

```



```
android:layout_width="match_parent"  
android:layout_height="wrap_content"  
android:text="Guardar ajustes"  
android:layout_marginTop="24dp" />
```

```
</LinearLayout>
```

Y la actividad que lee todo al guardar:

```

package com.lucasega.miprimeraapp

import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityAjustesBinding

class AjustesActivity : AppCompatActivity() {

    private lateinit var binding: ActivityAjustesBinding

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

        binding.btnGuardarAjustes.setOnClickListener {
            if (!binding.cbTerminos.isChecked) {
                Toast.makeText(this, "Debes aceptar los términos",
                    Toast.LENGTH_SHORT).show()
                return@setOnClickListener
            }

            val plan = when (binding.rgPlan.checkedRadioButtonId) {
                R.id.rbBasico -> "Básico"
                R.id.rbPro -> "Pro"
                R.id.rbPremium -> "Premium"
                else -> "Ninguno"
            }

            val notif = if (binding.swNotificaciones.isChecked) "sí" else "no"
            val oscuro = if (binding.swModoOscuro.isChecked) "sí" else "no"

            val resumen = "Plan: $plan, notificaciones: $notif, modo oscuro:
$oscuro"
            Toast.makeText(this, resumen, Toast.LENGTH_LONG).show()
        }
    }
}

```

Explicación. Al guardar primero se valida el CheckBox de términos con isChecked: si no está marcado, se avisa y se sale del listener. Luego se lee el plan con el when sobre checkedRadioButtonId, y los switches con isChecked convertidos a texto con if de expresión. Todo se junta en un resumen que se muestra en un Toast largo. Este ejemplo cubre el 90 por

ciento de las pantallas de ajustes y formularios de opciones que verás en apps reales. Como práctica extra, guarda estas preferencias con `SharedPreferences` para que persistan aunque se cierre la app: ese es el siguiente paso natural y tema de unidades futuras.

Ejercicio 9: encuesta de una pregunta

Crea una pantalla con la pregunta "¿Qué curso quieres aprender primero?", cuatro `RadioButton` (Android, iOS, Web, Datos) dentro de un `RadioGroup`, y un botón "Votar". Si no eligió nada, muestra un `Toast` pidiendo que elija; si eligió, muestra un `Toast` con su elección y abre una segunda pantalla que muestre "Votaste por: [elección]".

Solución: valida que `checkedRadioButtonId` no sea `-1` antes de continuar. Mapea el id al nombre con `when`, como en el ejemplo 6. Manda la elección con un extra a la segunda pantalla (aquí se unen las unidades 3 y 4 otra vez: casi todo lo que construyas combinará navegación con interfaces). El `Toast` de error usa `LENGTH_SHORT` y el de confirmación puede ser `LENGTH_LONG`. La lección: los `RadioGroup` siempre se validan contra `-1`, porque el usuario puede tocar el botón sin elegir nada.

Ejercicio 10: interruptor maestro

Agrega a la pantalla de ajustes un `Switch` maestro llamado "Todas las notificaciones" que al apagarse desactive y apague los otros dos switches (notificaciones y modo oscuro quedan como ejemplo, o crea dos switches de notificaciones: sonido y vibración). Al encenderse, los reactiva.

Solución: la clave está en la propiedad `isEnabled`, que activa o desactiva una vista (desactivada se ve gris y no responde):

```
binding.swMaestro.setOnCheckedChangeListener { _, activado ->
    binding.swSonido.isEnabled = activado
    binding.swVibracion.isEnabled = activado
    if (!activado) {
        binding.swSonido.isChecked = false
        binding.swVibracion.isChecked = false
    }
}
```

Al apagar el maestro se desactivan y se apagan los dependientes; al encenderlo se reactivan (pero se respeta que el usuario los vuelva a prender a mano, no se fuerzan a `true`). Este patrón de "interruptor maestro" aparece en ajustes de notificaciones, privacidad y permisos de todas las apps grandes. Fíjate que cambiar `isChecked` desde código también dispara el

OnCheckedChangeListener de esos switches: si esos listeners hacen algo costoso, debes tenerlo en cuenta para no duplicar trabajo.

Mini-proyecto final: ficha de curso que une las dos unidades

Para cerrar, un proyecto que usa todo lo aprendido en las unidades 3 y 4: una app con dos pantallas. La primera muestra una ficha de curso (imagen, título, descripción, precio, botón "Inscribirme"); la segunda es la confirmación de inscripción que recibe los datos con un Intent. Además, la ficha debe sobrevivir a la rotación sin perder nada.

Pantalla 1, activity_curso.xml, con ConstraintLayout:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="24dp">

    <ImageView
        android:id="@+id/ivCurso"
        android:layout_width="0dp"
        android:layout_height="200dp"
        android:src="@drawable/ic_curso_android"
        android:scaleType="centerCrop"
        android:contentDescription="Portada del curso"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/tvTituloCurso"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Desarrollo Android desde cero"
        android:textSize="24sp"
        android:textStyle="bold"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@id/ivCurso"
        android:layout_marginTop="16dp" />

    <TextView
        android:id="@+id/tvDescripcionCurso"
```

```

        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:text="Aprende a crear apps profesionales con Kotlin, layouts modernos
y el ciclo de vida dominado."
        android:textSize="15sp"
        android:maxLines="3"
        android:ellipsize="end"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@id/tvTituloCurso"
        android:layout_marginTop="8dp" />

<TextView
    android:id="@+id/tvPrecio"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="$499 MXN"
    android:textSize="26sp"
    android:textStyle="bold"
    android:textColor="@color/mi_primario"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/tvDescripcionCurso"
    android:layout_marginTop="16dp" />

<CheckBox
    android:id="@+id/cbTerminos"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="Acepto los términos de la academia"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/tvPrecio"
    android:layout_marginTop="16dp" />

<Button
    android:id="@+id/btnInscribirme"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="Inscribirme"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/cbTerminos"
    android:layout_marginTop="24dp" />

</androidx.constraintlayout.widget.ConstraintLayout>

```

La actividad combina validación, extras y estado:

```

package com.lucasega.miprimeraapp

import android.content.Intent
import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.lucasega.miprimeraapp.databinding.ActivityCursoBinding

class CursoActivity : AppCompatActivity() {

    private lateinit var binding: ActivityCursoBinding

    companion object {
        const val EXTRA_CURSO = "com.lucasega.miprimeraapp.CURSO"
        const val CLAVE_TERMINOS = "terminos"
    }

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

        if (savedInstanceState != null) {
            binding.cbTerminos.isChecked =
savedInstanceState.getBoolean(CLAVE_TERMINOS, false)
        }

        binding.btnInscribirme.setOnClickListener {
            if (!binding.cbTerminos.isChecked) {
                Toast.makeText(this, "Acepta los términos para continuar",
Toast.LENGTH_SHORT).show()
                return@setOnClickListener
            }
            val intent = Intent(this, ConfirmacionActivity::class.java)
            intent.putExtra(EXTRA_CURSO, binding.tvTituloCurso.text.toString())
            startActivity(intent)
        }

    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
        outState.putBoolean(CLAVE_TERMINOS, binding.cbTerminos.isChecked)
    }
}

```

Nota: el CheckBox con id conserva su estado automáticamente al rotar, pero aquí lo guardamos a mano como práctica del patrón de la unidad 3: guardar en onSaveInstanceState y restaurar en onCreate. La pantalla de confirmación recibe el nombre del curso con getStringExtra y lo muestra con un mensaje de éxito. Si construyes este proyecto completo, con rotación probada y sin crashes, tienes el nivel para avanzar a la siguiente unidad del libro: ya sabes darle vida a las pantallas y vestirlas como profesional.

Checklist antes de entregar tu pantalla

Antes de dar por terminada cualquier pantalla, recorre esta lista como piloto antes de despegar. Uno: cada vista tiene su restricción horizontal y vertical si usas ConstraintLayout; nada debe aparecer en la esquina al correr la app. Dos: todos los textos están en sp y todos los tamaños en dp; ni un solo px en el layout. Tres: cada EditText tiene hint, inputType correcto e id. Cuatro: cada imagen tiene contentDescription. Cinco: los botones importantes usan match_parent o pesos, no quedan del tamaño de su texto perdidos en una esquina. Seis: probaste en un teléfono chico y en uno grande, y si algo no cabe, hay un ScrollView. Siete: giraste el emulador y nada se perdió ni se encimó. Ocho: cada botón hace algo y cada entrada del usuario se valida antes de aceptarse. Nueve: los textos viven en strings.xml, no quemados en el layout. Diez: el layout no tiene más de tres niveles de anidamiento. Si tu pantalla pasa estos diez puntos, está lista para producción. Imprímelo, pégalo junto a tu monitor y úsalo en cada pantalla que construyas de aquí en adelante; en unas semanas lo harás de memoria.

Resumen de la unidad

- Un layout es el plano XML de tu pantalla; los ViewGroup son contenedores y las View son los elementos; juntos forman un árbol que Android dibuja.
- match_parent ocupa todo el espacio del papá, wrap_content se ajusta al contenido; usa dp para tamaños y sp para textos, nunca px.
- LinearLayout acomoda hijos en fila o columna; layout_weight reparte el espacio en proporción y es la clave de los diseños adaptables.
- gravity alinea el contenido dentro de la vista; layout_gravity alinea la vista dentro de su papá; no los confundas.
- RelativeLayout posiciona por relaciones (debajo de, al lado de, alineado al papá); bueno para pantallas con pocas relaciones claras.
- ConstraintLayout posiciona con restricciones; cada vista necesita una restricción horizontal y una vertical; es el recomendado para pantallas complejas.

- TextView muestra texto, Button se toca, EditText recibe escritura (con hint e inputType correctos) e ImageView muestra imágenes (con src y scaleType).
- El padding es aire interno y el layout_margin es distancia externa; el aire consistente hace que todo se vea profesional.
- Los eventos conectan la interfaz con el código: prefiere setOnClickListener en Kotlin sobre android:onClick en XML.
- Valida siempre la entrada del usuario: campos vacíos, formatos, longitudes; usa la propiedad error en el campo y toIntOrNull para números.
- Nombra los ids con convención (tv, btn, et, iv), pon contentDescription en las imágenes y prueba en varios tamaños de pantalla.

UNIDAD 5: Intents y navegación entre pantallas

Introducción

Bienvenido a la Unidad 5. Hasta este momento del curso ya sabes construir pantallas con vistas y layouts, manejar el ciclo de vida de una Activity y responder a los eventos del usuario. Tienes, en resumen, las piezas de un lego sobre la mesa. El problema es que esas piezas todavía no están conectadas entre sí: cada pantalla vive sola, aislada, como una casa sin puertas ni caminos hacia las demás. Esta unidad resuelve exactamente eso. Aquí aprenderás a conectar las pantallas de tu app para que el usuario pueda moverse entre ellas de forma natural: tocar un botón y pasar a otra pantalla, enviar datos de una pantalla a otra, compartir contenido con otras aplicaciones, abrir una página web, marcar un número de teléfono y construir interfaces flexibles con Fragments que se adaptan a teléfonos y tablets.

Esta unidad cubre las clases 13, 14 y 15 del curso. La clase 13 está dedicada a los intents explícitos, que son el mecanismo con el que una app abre sus propias pantallas. Aprenderás qué es un Intent, cómo construirlo, cómo decirle a Android que inicie otra Activity, como pasarle información con extras y como recibir un resultado de vuelta cuando la segunda pantalla se cierra. Es el tipo de navegación que usas cuando tocas "Iniciar sesión" y la app te lleva al menú principal, o cuando eliges un producto de una lista y se abre su pantalla de detalle. La clase 14 trata los intents implícitos, que son una de las ideas más elegantes de Android: en lugar de decirle al sistema que abra una pantalla específica de tu app, le pides una acción, como compartir un texto, abrir una página web o llamar a un número, y Android encuentra la aplicación capaz de hacerlo. Verás como compartir contenido en redes sociales, como abrir el navegador con una URL, como marcar un teléfono con un toque y como asegurarte de que siempre haya una app disponible para atender tu petición. La clase 15 introduce los Fragments, que son como "mini pantallas" reutilizables que viven dentro de una Activity. Entenderás que son, cuando conviene usarlos en lugar de crear otra Activity, como el FragmentManager administra sus entradas y salidas, y como hacer transacciones básicas: agregar, reemplazar y quitar fragments con código.

Por qué importa tanto esta unidad. Piensa en cualquier aplicación que uses todos los días: WhatsApp, tu banco, la app de fotos, la tienda donde compras. Ninguna de ellas es una sola pantalla estática. Todas están hechas de decenas de pantallas conectadas por un flujo de navegación bien pensado. La diferencia entre una app que se siente profesional y una que se siente amateur muchas veces no está en lo bonita que sea cada pantalla, sino en cómo se conectan entre sí: si los datos viajan correctamente de una a otra, si el botón de regresar lleva

al lugar esperado, si compartir algo con un amigo funciona al primer intento. Los intents y los fragments son las herramientas que te dan ese control. Sin intents no hay navegacion. Sin fragments no hay interfaces que se adapten a pantallas grandes ni componentes de pantalla reutilizables. Dominar estos temas te convierte de alguien que sabe dibujar pantallas en alguien que sabe construir aplicaciones completas.

Al terminar esta unidad sabras abrir cualquier pantalla de tu app desde cualquier otra usando intents explicitos. Sabras enviar datos entre pantallas con extras: textos, números y objetos más complejos. Sabras iniciar una pantalla esperando su respuesta y leer ese resultado cuando se cierre. Sabras usar intents implicitos para compartir texto e imágenes con otras aplicaciones, para abrir paginas web en el navegador, para marcar números de teléfono y para pedirle al sistema que elija la mejor app disponible. Sabrás qué es un Fragment, en qué se diferencia de una Activity, cuando te conviene usar uno, y como agregar, reemplazar y remover fragments con el FragmentManager usando transacciones. También conocerás los errores más comunes que cometen los principiantes con intents y fragments, como la famosa excepcion `ActivityNotFoundException` o el error de agregar un fragment dos veces, y como evitarlos con buenas prácticas probadas.

Una nota sobre como esta organizada la unidad para que le saques el máximo provecho. Primero encontraras la sección de explicación detallada, donde desmenuzamos los conceptos con analogias de la vida real, porque entender la idea antes del codigo hace que todo lo demas sea más fácil. Después vienen los ejemplos paso a paso, con codigo Kotlin y XML real que puedes copiar a tu proyecto, explicado línea por línea para que no quede ningún misterio. Luego la sección de práctica, con ejercicios numerados y sus soluciones explicadas, para que compruebes por ti mismo que lo aprendido se te quedo. Y al final, un resumen con los puntos clave. Te recomiendo leer en orden, pero sobre todo te recomiendo escribir el codigo con tus manos: la navegacion entre pantallas es de esos temas que solo se entienden de verdad cuando los pruebas en un emulador o en tu teléfono y ves como la app salta de una pantalla a otra.

Un último consejo antes de empezar. A lo largo de esta unidad verás mucho codigo parecido entre si: crear un Intent, llamar a `startActivity`, pasar extras con `putExtra`. Esa repeticion es intencional y es buena noticia: significa que con unos pocos patrones bien aprendidos puedes resolver la mayoria de los casos de navegacion que vas a encontrar en proyectos reales. No memorices cada metodo de memoria; entiende el patron general y consulta la documentacion cuando necesites un detalle. Vamos a ello: tu app esta a punto de dejar de ser una coleccion de pantallas sueltas y convertirse en una aplicacion de verdad.

Explicación detallada

Clase 13: Intents explícitos, la puerta entre tus pantallas

Imagina que tu app es un edificio de oficinas con muchas salas. Cada sala es una Activity: la sala de bienvenida, la sala del perfil, la sala de configuracion. Un Intent explícito es como un mensajero con una orden escrita que dice "lleva al usuario de la sala A a la sala B". La palabra "explícito" es clave: tú le dices a Android exactamente que pantalla quieres abrir, nombrando su clase con todas sus letras. No hay ambigüedad, no hay eleccion del sistema, no hay sorpresa. Tu app dice "abre DetalleActivity" y Android abre DetalleActivity. Punto.

Técnicamente, un Intent es un objeto de la clase `android.content.Intent` que describe una operacion a realizar. Piénsalo como un sobre: en el exterior lleva la direccion (la Activity destino), y adentro puede llevar cartas (los datos extras). Cuando llamas a `startActivity(intent)`, le entregas ese sobre al sistema Android, que es como el servicio de mensajería del edificio: lee la direccion, encuentra la sala correspondiente, la abre y lleva al usuario hasta ahí. Lo interesante es que tú no abres la pantalla directamente con algo como `"DetalleActivity.abrir()"`. Siempre pasas por el sistema. Esto es así por diseño: Android necesita saber que pantallas están activas para manejar el boton de regresar, la memoria y el ciclo de vida. El Intent es tu único canal oficial para pedir navegacion, y por eso es tan importante dominarlo.

La forma más simple de crear un intent explícito usa el constructor que recibe dos cosas: el contexto actual (normalmente `this` dentro de una Activity) y la clase de la Activity destino escrita como `NombreActivity::class.java`. Ese `"::class.java"` suele confundir al principio. Recuerda que en Kotlin, para referirte a la clase de Java que representa a `DetalleActivity`, escribes `DetalleActivity::class.java`. Es simplemente la manera de decir "la clase `DetalleActivity`" en el lenguaje que Android entiende por dentro. Una vez creado el intent, lo pasas a `startActivity()` y listo: la nueva pantalla aparece con su animacion de entrada, y la anterior queda "debajo" en una pila que Android administra. Esa pila se llama pila de actividades o back stack, y es la razon por la que el boton de regresar del teléfono funciona automaticamente: al presionarlo, Android cierra la pantalla de arriba y muestra la que estaba debajo, sin que tu escribas una sola línea de código para eso.

Pero abrir pantallas vacías rara vez es suficiente. Casi siempre necesitas llevar datos contigo. Si tocas un producto en una lista, la pantalla de detalle necesita saber QUE producto mostrar. Para eso están los extras: pares de clave y valor que metes dentro del intent con `putExtra()`. La clave es un String que tu inventas, como `"nombre_producto"` o `"precio"`, y el valor puede ser un String, un Int, un Boolean, un Double y varios tipos más. En la Activity destino, recuperas esos valores con metodos como `getStringExtra("nombre_producto")` o `getIntExtra("precio", 0)`.

Piensa en los extras como las notas adhesivas que le pegas al sobre del mensajero: "oye, cuando llegues, entregale también este papelito con el nombre del producto". Es un mecanismo simple pero poderosísimo, y lo usarás en prácticamente todas las apps que construyas.

Hay un detalle importante con `getIntExtra` y sus hermanos numericos: reciben un segundo parametro que es el valor por defecto. Por ejemplo, `getIntExtra("edad", 0)` significa "dame la edad que venia en el intent, y si no venia ninguna, usa 0". Esto existe porque los extras son opcionales: nadie te garantiza que quien abrio tu pantalla haya mandado ese dato. Si olvidas el valor por defecto en los tipos numericos, el codigo ni siquiera compila, porque Kotlin te obliga a manejar el caso de "no llego nada". Con los String es distinto: `getStringExtra` devuelve un String? (anulable), es decir, puede ser null. Tendras que manejarlo con el operador Elvis (`?:`) o con un `let`. Este es uno de los primeros lugares donde el sistema de nulabilidad de Kotlin te protege de errores reales: si la pantalla se abre sin el dato esperado, tu decides que mostrar en lugar de que la app se caiga.

Otro patron fundamental es abrir una pantalla esperando su respuesta. El caso clásico es un formulario: la pantalla principal abre la pantalla de "nuevo contacto", el usuario llena los datos y al guardar, la pantalla se cierra y devuelve el contacto creado. En el Android moderno esto se hace con la API de Activity Result (`registerForActivityResult`), que reemplazo al viejo `startActivityForResult`. La idea es elegante: registras un "lanzador" que dice que vas a iniciar una actividad y que quieres escuchar su resultado; cuando el usuario termina en la segunda pantalla, tu codigo recibe un callback con el resultado. La segunda pantalla devuelve datos llamando a `setResult(RESULT_OK, intentConDatos)` antes de cerrarse con `finish()`. Es como mandar al mensajero con un sobre de ida y un sobre de vuelta ya preparado: va, hace su trabajo y regresa con la respuesta. Veremos el codigo completo en la sección de ejemplos, pero quedate con la idea: navegar no siempre es un viaje de ida; muchas veces necesitas el viaje redondo.

Un concepto que conviene entender desde ahora es el contexto (Context). Cuando creas el intent escribes `Intent(this, DetalleActivity::class.java)`, y ese "this" es el contexto de la Activity actual. El contexto es la conexion de tu codigo con el sistema Android: a traves de el pides recursos, accedes a servicios y, en este caso, le pides al sistema que inicie otra pantalla. Dentro de una Activity, "this" funciona porque Activity es un tipo de Context. Si algun día creas intents dentro de un Fragment o de un Adapter, tendras que obtener el contexto de otra forma (por ejemplo, `requireContext()`), pero la idea es la misma: el intent necesita saber desde dónde se está pidiendo la navegación.

Finalmente, un apunte sobre el archivo `AndroidManifest.xml`. Cada Activity de tu app debe estar declarada ahí; si no, Android no sabe que existe y la app se cae al intentar abrirla con un

error que dice que la actividad no fue declarada. Android Studio normalmente agrega la declaracion automaticamente cuando creas una Activity con el asistente, así que rara vez tendras que escribirla a mano, pero debes saber que existe y revisar el manifest cuando una navegacion falle con ese mensaje. Es como el directorio del edificio: si la sala no esta registrada en el directorio, el mensajero no puede llevarte hasta ella aunque la sala exista fisicamente.

Clase 14: Intents implícitos, pedirle un favor al sistema

Si el intent explicito es un mensajero con una direccion exacta, el intent implicito es un mensajero con una descripcion del trabajo: "necesito que alguien comparta este texto", "necesito que alguien abra esta página web", "necesito que alguien marque este número". Tú no dices QUE app debe hacerlo. Le describes al sistema la ACCION que quieres y el TIPO de datos involucrados, y Android busca entre todas las apps instaladas cuál puede atender tu petición. Si hay varias, le muestra al usuario un menú para elegir (el famoso "chooser" o selector). Esta es una de las ideas más brillantes de Android: tu app no necesita saber cómo compartir en redes sociales ni como abrir un PDF; simplemente pide la accion y el ecosistema responde.

La analogía perfecta es pedir un taxi por radio. Tú no llamas a un taxista en especifico por su nombre; dices "necesito un taxi para dos personas con equipaje en esta direccion". La central busca que unidad cumple los requisitos y te la manda. En Android, la "central" es el sistema, tu "pedido" es el intent implicito con su accion y sus datos, y los "taxis" son las apps instaladas que declararon que pueden manejar esa accion. La accion se expresa con constantes como `Intent.ACTION_SEND` (compartir), `Intent.ACTION_VIEW` (ver o abrir algo), `Intent.ACTION_DIAL` (marcar un número) e `Intent.ACTION_SENDTO` (enviar a un destinatario especifico, típico para SMS o correo). Los datos se expresan con `setData()` usando una Uri, o con `setType()` para indicar el tipo MIME, como "text/plain" para texto o "image/jpeg" para una foto.

El caso más común es compartir texto. Creas un intent con accion `ACTION_SEND`, le pones el texto con `putExtra(Intent.EXTRA_TEXT, "mira esta app increible")`, defines el tipo con `setType("text/plain")` y lo lanzas. Buena práctica: envolverlo en `Intent.createChooser(intent, "Compartir con...")`. El chooser garantiza que siempre aparezca el menú de seleccion con un titulo amable, incluso si el usuario ya configuro una app predeterminada. Sin chooser, Android podria abrir directamente la app predeterminada sin preguntar, y el usuario perderia la oportunidad de elegir. Es un detalle pequeño que mejora mucho la experiencia.

Abrir una página web es todavia más simple: un intent con `ACTION_VIEW` y como dato la Uri de la página, por ejemplo `Uri.parse("https://www.lucasegaacademy.com")`. Android encontrara

el navegador (o los navegadores) instalados y abra la página ahí. Para marcar un número se usa `ACTION_DIAL` con una Uri de tipo `"tel:5551234567"`. Ojo con la diferencia entre `ACTION_DIAL` y `ACTION_CALL`: `DIAL` abre la app de teléfono con el número ya escrito para que el usuario presione llamar; `CALL` marcaría directamente sin pasar por la pantalla de marcado, pero requiere un permiso especial (`CALL_PHONE`) que debes declarar y solicitar en tiempo de ejecución. Para la mayoría de los casos, `ACTION_DIAL` es lo correcto y lo seguro: no pide permisos y deja la decisión final en manos del usuario. También puedes abrir mapas con una Uri de tipo `"geo:latitud,longitud"` y `ACTION_VIEW`, o redactar un correo con `ACTION_SENDTO` y una Uri `"mailto:alguien@ejemplo.com"`. El patrón siempre es el mismo: acción + datos; solo cambian los valores.

Ahora, la pregunta del millón: ¿qué pasa si ninguna app instalada puede atender tu petición. Si lanzas el intent a ciegas, Android arroja una `ActivityNotFoundException` y tu app se cae. Es uno de los errores clásicos de principiantes y es fácil de evitar. La forma moderna y recomendada es usar `resolveActivity()`: antes de llamar a `startActivity`, preguntas `intent.resolveActivity(packageManager)` y si el resultado no es null, significa que hay al menos una app capaz de atenderte. Solo entonces lanzas el intent. En código se ve así de simple: un `if` que protege el `startActivity`. Alternativamente puedes envolver el lanzamiento en un `try-catch` para `ActivityNotFoundException` y mostrar un mensaje amable como un `Toast` que diga "No hay ninguna aplicación disponible para esta acción". Cualquiera de las dos técnicas funciona; lo importante es no lanzar intents implícitos sin protección. Piénsalo como mirar antes de cruzar la calle: cuesta una línea de código y evita un accidente.

Otro detalle fino: los intents implícitos también pueden llevar extras, igual que los explícitos. Por ejemplo, al compartir puedes agregar `Intent.EXTRA_SUBJECT` para el asunto si el destino es un correo. Y al abrir una web, no necesitas extras porque toda la información va en la Uri. La regla general es: los datos principales van en `setData()` o en el tipo MIME, y los detalles adicionales van en extras.

Hay una trampa conceptual que conviene aclarar: un intent implícito también puede abrir pantallas de TU propia app si esas pantallas declaran un "intent filter" en el manifest. Un intent filter es como un anuncio que pone una Activity diciendo "yo se manejar estas acciones con estos tipos de datos". Por ejemplo, tu app de notas podría declarar que maneja `ACTION_SEND` con texto, y entonces aparecería en el menú de compartir de otras apps. Es un tema avanzado que excede esta unidad, pero es bueno que sepas que existe: es la razón por la que tu app puede recibir contenido compartido desde otras apps. Por ahora, concentremos en el flujo de salida: tu app pidiendo acciones al sistema.

En resumen, los intents implícitos te conectan con todo el ecosistema Android sin escribir integraciones específicas. Compartir, navegar, llamar, ubicar en mapas, enviar correos: todo

se resuelve con el mismo patron de accion más datos, protegido con `resolveActivity`. Es código corto, pero detrás hay una arquitectura muy bien pensada que hace que las apps de Android se sientan como partes de un todo en lugar de islas separadas.

Clase 15: Fragments, las mini pantallas reutilizables

Un Fragment es un pedazo de interfaz con su propio ciclo de vida que vive DENTRO de una Activity. Si la Activity es una casa, el Fragment es una habitación modular que puedes meter, sacar o intercambiar sin demoler la casa. Tiene su propio layout XML, su propio código Kotlin y su propio ciclo de vida (`onCreate`, `onCreateView`, `onViewCreated`, `onDestroyView`, etc.), pero no puede existir solo: siempre necesita una Activity que lo hospede. Esa es la diferencia fundamental con una Activity. Una Activity es una pantalla completa e independiente que el sistema puede abrir; un Fragment es un componente de pantalla reutilizable que tu colocas donde quieras.

La analogía que mejor funciona es la de las piezas de un rompecabezas frente a cuadros completos. Una Activity es un cuadro ya enmarcado: ocupa toda la pared y es una sola pieza. Un Fragment es una pieza del rompecabezas: por si sola no es el cuadro completo, pero combinada con otras piezas forma la imagen, y puedes reutilizar la misma pieza en rompecabezas distintos. Por ejemplo, piensa en una app de noticias. En un teléfono, la lista de noticias y el detalle de la noticia son dos pantallas separadas (dos Activities, o una Activity que intercambia dos fragments). En una tablet, con más espacio, la lista y el detalle aparecen lado a lado EN LA MISMA pantalla. Si la lista y el detalle son fragments, reutilizas exactamente el mismo código en ambos casos: en el teléfono muestras un fragment a la vez, en la tablet muestras los dos juntos. Si hubieras hecho todo con Activities separadas, tendrías que duplicar lógica o hacer malabares. Esa es la razón de ser de los fragments: flexibilidad y reutilización.

Cuando conviene usar un Fragment en lugar de una Activity nueva. Usa un Fragment cuando quieras reutilizar un pedazo de interfaz en varias pantallas (una lista de productos que aparece en tres secciones distintas de la app), cuando quieras una interfaz adaptable que cambie según el tamaño de pantalla (teléfono vs tablet), cuando necesites cambiar solo una parte de la pantalla sin recargar todo lo demás (pestanas o tabs donde el encabezado se queda fijo y solo cambia el contenido), o cuando uses componentes de navegación modernos como el Navigation Component o un ViewPager, que están contruidos sobre fragments. Usa una Activity nueva cuando la pantalla sea conceptualmente independiente, tenga su propio flujo completo y no necesite compartir espacio con otras partes (una pantalla de ajustes, un formulario de registro). Regla práctica de principiante: si dudas, una Activity nueva casi siempre es la respuesta correcta y más simple; introduce fragments cuando aparezca una necesidad real de reutilización o de dividir la pantalla.

Todo Fragment necesita un "contenedor" en el layout de la Activity: normalmente un `FrameLayout` o, en proyectos modernos, un `FragmentContainerView`, que es un `FrameLayout` especializado para hospedar fragments. En el XML de la Activity escribes algo como un `FragmentContainerView` con un id, por ejemplo `@+id/contenedor_fragment`, y ese espacio vacío es donde tu código colocara el fragment en tiempo de ejecución. También puedes declarar un fragment directamente en el XML con la etiqueta `<fragment>`, pero esa forma está en desuso porque no permite reemplazarlo después; la forma moderna y flexible es el contenedor vacío más transacciones en código.

El `FragmentManager` es el encargado de administrar los fragments de una Activity, como un director de escena que mete y saca actores del escenario. Lo obtienes con `supportFragmentManager` dentro de una Activity (o con `parentFragmentManager` / `childFragmentManager` dentro de un Fragment, según el caso). Con él inicias "transacciones": operaciones que agregan, reemplazan o quitan fragments del contenedor. Una transacción se construye así: comienzas con `beginTransaction()`, encadenas operaciones como `replace(R.id.contenedor_fragment, MiFragment())` o `add(...)`, opcionalmente llamas a `addToBackStack(null)` para que el botón de regresar deshaga la transacción, y cierras con `commit()`. El `commit()` no ejecuta la transacción al instante de forma sincrónica; la programa para ejecutarse en el hilo principal lo antes posible. En la práctica se siente instantáneo, pero es bueno saberlo para no intentar usar el fragment "recién agregado" en la línea siguiente como si ya estuviera en pantalla.

Hablemos de las operaciones básicas. `add()` coloca un fragment en el contenedor conservando lo que ya había (los fragments se apilan visualmente uno sobre otro). `replace()` quita lo que había y pone el nuevo en su lugar: es la operación más usada para navegación entre secciones. `remove()` quita un fragment existente. `addToBackStack()` mete la transacción en la pila de retroceso de fragments, de modo que al presionar el botón atrás, Android revierte la transacción (por ejemplo, vuelve al fragment anterior) en lugar de cerrar la Activity. Sin `addToBackStack`, el botón atrás saldría de la Activity directamente aunque hubieras cambiado de fragment varias veces. Piensa en la pila de retroceso como el historial del navegador: cada transacción guardada es una página visitada a la que puedes volver con "atrás".

El ciclo de vida del Fragment merece un párrafo porque es fuente clásica de confusiones. Además de `onCreate` y `onDestroy`, tiene métodos propios relacionados con su vista: `onCreateView()` es donde "infla" el layout XML del fragment (ahí conviertes el XML en objetos Vista con `layoutInflater.inflate()`), y `onViewCreated()` se ejecuta justo después, cuando la vista ya existe: ese es el lugar ideal para buscar vistas con `findViewById` o configurar listeners. Cuando el fragment se destruye, `onDestroyView()` libera la vista pero el fragment como objeto puede seguir vivo (por ejemplo, si está en la pila de retroceso). El error típico es guardar referencias a

vistas en propiedades del fragment y usarlas después de `onDestroyView`, lo que causa fugas de memoria o crashes. Regla de oro: usa las vistas solo entre `onViewCreated` y `onDestroyView`, y nunca guardes vistas en variables que vivan más que la vista misma.

Cómo se comunican un Fragment y su Activity. Un fragment no debería hablar directamente con otro fragment; la comunicación pasa por la Activity anfitriona, como compañeros de cuarto que se dejan notas en la cocina en lugar de entrar al cuarto del otro. Las formas más simples: la Activity puede llamar a métodos públicos del fragment si tiene su referencia, y el fragment puede obtener su Activity con `requireActivity()` y llamarle métodos (con un cast a la clase concreta, o mejor, a través de una interfaz). Para pasar datos INICIALES a un fragment se usa el patrón de arguments: un Bundle que se asigna a la propiedad `arguments` del fragment ANTES de agregarlo. Es el equivalente a los extras del Intent pero para fragments. En código moderno también existen los ViewModels compartidos, pero eso pertenece a una unidad posterior; con arguments y métodos públicos resuelves la gran mayoría de los casos de principiante.

Para cerrar la explicación: Activity y Fragment no compiten, se complementan. La Activity es el marco, el escenario, la que el sistema abre y cierra; el Fragment es el contenido modular que se monta en el escenario. Dominar ambos te da el vocabulario completo de la navegación en Android: intents para moverte ENTRE escenarios, fragments para reorganizar el contenido DENTRO de un escenario.

Errores comunes de principiantes y cómo evitarlos

Error 1: olvidar declarar la Activity en el `AndroidManifest.xml`. Síntoma: la app se cierra al intentar navegar y en el Logcat aparece un mensaje diciendo que la actividad no fue declarada en el manifest. Causa: creaste la clase Kotlin pero no su entrada en el manifest. Como evitarlo: crea las Activities con el asistente de Android Studio (clic derecho en el paquete, New, Activity), que genera la declaración automáticamente. Si creas la clase a mano, agrega tu mismo la etiqueta `<activity android:name=".DetalleActivity" />` dentro de `<application>`. Consejo extra: cuando veas ese error, lo primero que debes revisar siempre es el manifest; el noventa por ciento de las veces ahí está el problema.

Error 2: `ActivityNotFoundException` al lanzar un intent implícito. Síntoma: crash al tocar el botón de compartir en un emulador sin apps que compartan, o al intentar abrir algo raro. Causa: lanzaste el intent sin verificar que exista una app capaz de atenderlo. Como evitarlo: protege SIEMPRE los intents implícitos con `resolveActivity(packageManager) != null` antes de `startActivity`, o envuelve el lanzamiento en try-catch. Hazlo un hábito: intent implícito sin verificación es código incompleto.

Error 3: `NullPointerException` (o comportamiento raro) al leer extras. Sintoma: la pantalla de detalle muestra textos vacíos o la app truena al abrirla desde cierto flujo. Causa: la Activity destino espera un extra con cierta clave, pero quien la abrió no lo mandó o usó una clave distinta ("nombre" vs "nombre_producto"). Como evitarlo: define las claves como constantes en un companion object de la Activity destino, por ejemplo `const val EXTRA_NOMBRE = "extra_nombre"`, y usalas en ambos lados. Así el compilador te avisa si escribes mal el nombre de la constante, cosa que nunca haría con un String suelto. Además, maneja siempre el caso de dato ausente con valores por defecto o el operador Elvis.

Error 4: agregar el mismo fragment dos veces y ver pantallas duplicadas o encimadas. Sintoma: después de girar el teléfono, el fragment aparece duplicado o la interfaz se ve rara. Causa: en `onCreate` de la Activity agregas el fragment inicial sin verificar si ya existe uno restaurado por el sistema. Cuando giras la pantalla, Android destruye y recrea la Activity pero RESTAURA los fragments automáticamente; si tu código agrega otro encima, se duplican. Como evitarlo: envuelve la transacción inicial en un `if (savedInstanceState == null)`. Ese parámetro es null solo la primera vez que se crea la Activity; en recreaciones (como la rotación) no es null y no debes agregar el fragment de nuevo. Este es probablemente el error de fragments más reportado por principiantes, y la solución es una sola línea.

Error 5: usar `getActivity()` o `requireActivity()` en el momento equivocado dentro de un Fragment. Sintoma: `IllegalStateException` diciendo que el fragment no está attached a una actividad. Causa: llamaste a `requireActivity()` en el constructor del fragment, en una propiedad inicializada muy temprano, o en un callback que llegó cuando el fragment ya se había despegado (por ejemplo, una respuesta de red tardía). Como evitarlo: accede a la actividad solo cuando el fragment esté en un estado seguro (`onViewCreated` en adelante) y, en callbacks asíncronos, verifica `isAdded` antes de tocar la actividad o las vistas.

Error 6: confundir `add()` con `replace()` y terminar con fragments encimados. Sintoma: ves el contenido de dos fragments superpuestos, uno tras otro. Causa: usaste `add()` pensando que reemplazaba. Como evitarlo: para navegación entre secciones usa `replace()`; reserva `add()` para casos donde realmente quieres apilar (diálogos o fragments superpuestos intencionalmente). Si ves pantallas "fantasma" detrás de la actual, casi seguro es un `add` donde debía ir un `replace`.

Buenas prácticas

1. Nombra tus Activities y fragments con intención clara: `DetalleProductoActivity` dice mucho más que `Pantalla2`. Tu yo del futuro te lo agradecerá.

2. Centraliza las claves de extras como constantes en companion objects. Evita los "strings magicos" regados por el codigo.
3. Usa Intent.createChooser para compartir. Le da al usuario el control y un titulo amable en el selector.
4. Prefiere ACTION_DIAL sobre ACTION_CALL salvo que tu app realmente necesite llamar sin intervencion del usuario (y entonces gestiona el permiso CALL_PHONE correctamente).
5. En fragments, configura vistas en onCreateView, no en onCreateView más alla del inflado. Y nunca conserves referencias a vistas más alla de onDestroyView.
6. Agrega el fragment inicial solo cuando savedInstanceState == null. Grabatelo como mantra.
7. Usa addToBackStack cuando el usuario esperaria que "atras" lo lleve al fragment anterior (navegacion por secciones). No lo uses para el fragment inicial ni para flujos que deben cerrarse de golpe.
8. Mantén los intents explicitos para navegar DENTRO de tu app y los implicitos para salir HACIA el ecosistema (compartir, web, teléfono, mapas). Esa division mental te ahorrara decisiones.
9. Prueba la navegacion con el boton atras del sistema en cada flujo nuevo que construyas. La mitad de los bugs de navegacion se descubren ahi.
10. No pases objetos gigantescos por extras ni por arguments. Para datos grandes usa un ViewModel compartido, una base de datos o un repositorio; los extras están pensados para identificadores y valores pequeños.

Ejemplos paso a paso

Ejemplo 1: Intent explícito con datos de ida (clase 13)

Escenario: una app de tienda. La pantalla principal (MainActivity) tiene un boton "Ver producto" que abre la pantalla de detalle (DetalleActivity) llevando el nombre y el precio del producto.

Primero, la Activity destino. Fijate como declara sus claves de extras como constantes en el companion object:

```

package com.lucasega.tienda

import android.os.Bundle
import android.widget.TextView
import androidx.appcompat.app.AppCompatActivity

class DetalleActivity : AppCompatActivity() {

    companion object {
        const val EXTRA_NOMBRE = "extra_nombre"
        const val EXTRA_PRECIO = "extra_precio"
    }

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

        val tvNombre = findViewById<TextView>(R.id.tvNombre)
        val tvPrecio = findViewById<TextView>(R.id.tvPrecio)

        val nombre = intent.getStringExtra(EXTRA_NOMBRE) ?: "Producto sin nombre"
        val precio = intent.getDoubleExtra(EXTRA_PRECIO, 0.0)

        tvNombre.text = nombre
        tvPrecio.text = "Precio: $$precio"
    }
}

```

Explicación línea por línea. El companion object con const val define las claves una sola vez; ambas Activities usaran estas constantes y así es imposible escribir mal la clave en un lado. En onCreate, después de setContentView, buscamos los dos TextView del layout. Luego viene la lectura de extras: intent es una propiedad que toda Activity tiene y que contiene el Intent con el que fue abierta. getStringExtra(EXTRA_NOMBRE) devuelve el texto o null si no venia; el operador Elvis (?:) pone "Producto sin nombre" como respaldo para que la pantalla nunca muestre un vacío misterioso. getDoubleExtra(EXTRA_PRECIO, 0.0) devuelve el precio o 0.0 si no venia ningún dato. Finalmente asignamos los textos a las vistas. Nota: "\$\$precio" en un string template de Kotlin escribe un signo de dolar literal seguido del valor de la variable precio.

Ahora la Activity origen, que construye el intent y lo lanza:

```

package com.lucasega.tienda

import android.content.Intent
import android.os.Bundle
import android.widget.Button
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

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

        val btnVerProducto = findViewById<Button>(R.id.btnVerProducto)

        btnVerProducto.setOnClickListener {
            val intent = Intent(this, DetalleActivity::class.java)
            intent.putExtra(DetalleActivity.EXTRA_NOMBRE, "Audifonos Bluetooth
X200")
            intent.putExtra(DetalleActivity.EXTRA_PRECIO, 899.99)
            startActivity(intent)
        }
    }
}

```

Explicación. Dentro del listener del boton creamos el Intent con this (el contexto de MainActivity) y DetalleActivity::class.java (la clase destino escrita en notacion de Java). Con putExtra metemos los dos datos usando LAS MISMAS constantes definidas en DetalleActivity: así, si mañana renombramos la clave, el compilador nos obliga a actualizar ambos lados. startActivity(intent) entrega el sobre al sistema y la pantalla de detalle aparece. El boton de regresar del teléfono volvera a MainActivity automaticamente gracias a la pila de actividades.

Y no olvides el manifest. Si creaste DetalleActivity con el asistente, Android Studio ya agrego esto dentro de <application>:

```
<activity android:name=".DetalleActivity" />
```

Sin esa línea, el ejemplo se cae con el error de actividad no declarada que vimos en la sección de errores.

Ejemplo 2: Abrir una pantalla esperando su respuesta (clase 13)

Escenario: MainActivity abre FormularioActivity para capturar el nombre de un nuevo cliente, y al guardar, el nombre regresa a MainActivity.

En MainActivity registramos el lanzador de resultados. Ojo: el registro se hace como propiedad de la clase, NO dentro de onCreate:

```
package com.lucasega.clientes

import android.app.Activity
import android.content.Intent
import android.os.Bundle
import android.widget.Button
import android.widget.TextView
import androidx.activity.result.contract.ActivityResultContracts
import androidx.appcompat.app.AppCompatActivity

class MainActivity : AppCompatActivity() {

    private val lanzadorFormulario =
        registerForActivityResult(ActivityResultContracts.StartActivityForResult())
    { resultado ->
        if (resultado.resultCode == Activity.RESULT_OK) {
            val nombre =
                resultado.data?.getStringExtra(FormularioActivity.EXTRA_RESULTADO_NOMBRE)
            findViewById<TextView>(R.id.tvCliente).text = "Cliente: ${nombre ?:
                "sin nombre"}"
        }
    }

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

        findViewById<Button>(R.id.btnNuevoCliente).setOnClickListener {
            val intent = Intent(this, FormularioActivity::class.java)
            lanzadorFormulario.launch(intent)
        }
    }
}
```

Explicación. registerForActivityResult recibe un "contrato" (StartActivityForResult es el contrato estandar para este caso) y un lambda que se ejecuta cuando la segunda pantalla se cierra. El

lambda recibe un objeto resultado con resultCode (RESULT_OK si todo salio bien, RESULT_CANCELED si el usuario salio con atras) y data (el intent de vuelta, que puede ser null, de ahi el ?.). Solo si el codigo es RESULT_OK leemos el extra con la clave definida en FormularioActivity y actualizamos el TextView. En el boton, en lugar de startActivity usamos lanzadorFormulario.launch(intent): el lanzador es quien abre la pantalla y quien escuchara su respuesta.

En FormularioActivity, al guardar devolvemos el dato:

```
package com.lucasega.clientes

import android.app.Activity
import android.content.Intent
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import androidx.appcompat.app.AppCompatActivity

class FormularioActivity : AppCompatActivity() {

    companion object {
        const val EXTRA_RESULTADO_NOMBRE = "extra_resultado_nombre"
    }

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

        val etNombre = findViewById<EditText>(R.id.etNombre)

        findViewById<Button>(R.id.btnGuardar).setOnClickListener {
            val datos = Intent()
            datos.putExtra(EXTRA_RESULTADO_NOMBRE, etNombre.text.toString())
            setResult(Activity.RESULT_OK, datos)
            finish()
        }
    }
}
```

Explicación. Al presionar guardar, creamos un Intent Vacío (sin destino, porque no va a abrir nada: solo transporta datos de vuelta), le metemos el nombre capturado, llamamos a setResult(RESULT_OK, datos) para decir "todo bien, aquí está la respuesta" y cerramos la pantalla con finish(). En ese momento, el lambda registrado en MainActivity se dispara con el

resultado. Si el usuario presiona atras sin guardar, la pantalla se cierra sola con `RESULT_CANCELED` y el lambda simplemente no hace nada útil: el flujo redondo queda completo y seguro.

Ejemplo 3: Intents implícitos, compartir, web y llamada (clase 14)

Escenario: una pantalla con tres botones: compartir la app con un amigo, abrir el sitio de la academia en el navegador y marcar al teléfono de soporte. Los tres usan intents implícitos protegidos con `resolveActivity`.

```
package com.lucasega.miapp

import android.content.ActivityNotFoundException
import android.content.Intent
import android.net.Uri
import android.os.Bundle
import android.widget.Button
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity

class AyudaActivity : AppCompatActivity() {

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

        findViewById<Button>(R.id.btnCompartir).setOnClickListener { compartirApp() }
        findViewById<Button>(R.id.btnAbrirWeb).setOnClickListener { abrirSitioweb() }
        findViewById<Button>(R.id.btnLlamar).setOnClickListener { marcarSoporte() }
    }

    private fun compartirApp() {
        val intent = Intent(Intent.ACTION_SEND).apply {
            type = "text/plain"
            putExtra(Intent.EXTRA_SUBJECT, "Te recomiendo esta academia")
            putExtra(Intent.EXTRA_TEXT, "Aprende programacion gratis en
https://www.lucasegaacademy.com")
        }
        val chooser = Intent.createChooser(intent, "Compartir con...")
        lanzarSeguro(chooser)
    }

    private fun abrirSitioweb() {
        val intent = Intent(Intent.ACTION_VIEW,
```

```

Uri.parse("https://www.lucasegaacademy.com"))
    lanzarSeguro(intent)
}

private fun marcarSoporte() {
    val intent = Intent(Intent.ACTION_DIAL, Uri.parse("tel:5551234567"))
    lanzarSeguro(intent)
}

private fun lanzarSeguro(intent: Intent) {
    try {
        if (intent.resolveActivity(packageManager) != null) {
            startActivity(intent)
        } else {
            avisarSinApp()
        }
    } catch (e: ActivityNotFoundException) {
        avisarSinApp()
    }
}

private fun avisarSinApp() {
    Toast.makeText(this, "No hay ninguna app disponible para esta accion",
Toast.LENGTH_LONG).show()
}
}

```

Explicación línea por línea. Cada boton delega en un metodo privado para que onCreate quede limpio y legible. En compartirApp() creamos el intent con la accion ACTION_SEND y usamos apply para configurarlo en bloque: type = "text/plain" dice que el contenido es texto plano, EXTRA_SUBJECT es el asunto (algunas apps lo usan) y EXTRA_TEXT es el mensaje. Luego lo envolvemos en createChooser con el titulo "Compartir con..." para que el usuario siempre elija la app destino. En abrirSitioWeb() usamos el constructor de Intent que recibe accion y Uri juntos: ACTION_VIEW más la Uri de la página. En marcarSoporte() usamos ACTION_DIAL con una Uri "tel:" más el número; fíjate que NO es ACTION_CALL, así no necesitamos permisos y el usuario confirma la llamada en su app de teléfono.

El metodo lanzarSeguro() es el guardian: primero pregunta con resolveActivity(packageManager) si existe al menos una app capaz de atender el intent; si la hay, lanza; si no, muestra el aviso. El try-catch es un segundo cinturón de seguridad por si algo cambia entre la verificación y el lanzamiento (raro, pero posible). El Toast informa al usuario en lugar de dejar que la app se caiga. Este patrón de "verificar antes de lanzar" debes aplicarlo a TODO intent implícito que escribas de ahora en adelante.

Ejemplo 4: Fragment con transacción básica (clase 15)

Escenario: una Activity con dos botones ("Inicio" y "Perfil") que intercambian dos fragments en un contenedor. Es el esqueleto de una app con secciones.

Primero el layout de la Activity (activity_principal.xml). Usa `FragmentManager` como contenedor:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <Button
            android:id="@+id/btnInicio"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Inicio" />

        <Button
            android:id="@+id/btnPerfil"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:text="Perfil" />

    </LinearLayout>

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/contenedorFragment"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />

</LinearLayout>
```

Explicación del XML. Un LinearLayout vertical divide la pantalla: arriba una barra horizontal con los dos botones (cada uno con `layout_weight="1"` para repartirse el ancho a la mitad), y debajo el `FragmentManager` que ocupa todo el espacio restante (`layout_weight="1"` con `height 0dp`). Ese contenedor empieza vacío; nuestro código pondrá ahí los fragments.

Ahora un fragment sencillo (`InicioFragment`). Cada fragment necesita su clase Kotlin y su layout:

```
package com.lucasega.miapp

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.fragment.app.Fragment

class InicioFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return inflater.inflate(R.layout.fragment_inicio, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        view.findViewById<TextView>(R.id.tvTituloInicio).text = "Bienvenido al inicio"
    }
}
```

Explicación. La clase hereda de `Fragment` (el de `androidx.fragment.app`, siempre el de `AndroidX`, nunca el viejo `android.app.Fragment` que está obsoleto). `onCreateView` recibe el `inflater` y "infla" el layout `fragment_inicio.xml`, convirtiéndolo en un objeto `View` que se devuelve; el tercer parámetro `false` significa "crea la vista pero no la agregues todavía al contenedor", porque eso lo hará el `FragmentManager`. `onViewCreated` se ejecuta cuando la vista ya existe: ahí buscamos el `TextView` USANDO `view.findViewById` (importante: se busca en la vista del fragment, no en la actividad) y le ponemos texto. `PerfilFragment` sería idéntico pero con su propio layout y texto.

Finalmente, la Activity que orquesta las transacciones:

```
package com.lucasega.miapp

import android.os.Bundle
import android.widget.Button
import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.Fragment

class PrincipalActivity : AppCompatActivity() {

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

        if (savedInstanceState == null) {
            mostrarFragment(InicioFragment(), false)
        }

        findViewById<Button>(R.id.btnInicio).setOnClickListener {
            mostrarFragment(InicioFragment(), true)
        }

        findViewById<Button>(R.id.btnPerfil).setOnClickListener {
            mostrarFragment(PerfilFragment(), true)
        }
    }

    private fun mostrarFragment(fragment: Fragment, agregarAlHistorial: Boolean) {
        val transaccion = supportFragmentManager.beginTransaction()
        transaccion.replace(R.id.contenedorFragment, fragment)
        if (agregarAlHistorial) {
            transaccion.addToBackStack(null)
        }
        transaccion.commit()
    }
}
```

Explicación. En onCreate, el if (savedInstanceState == null) garantiza que el fragment inicial solo se agregue la primera vez, evitando duplicados al girar la pantalla (el error 4 de la sección anterior). Cada boton llama a mostrarFragment con el fragment correspondiente y true para que el cambio quede en el historial. El metodo mostrarFragment concentra toda la logica de transaccion: beginTransaction() abre la transaccion, replace() saca lo que hubiera en el contenedor y mete el nuevo fragment, addToBackStack(null) (condicional) permite volver atras

con el boton del sistema, y commit() programa la ejecucion. Con este patron de un metodo reutilizable, agregar una tercera sección mañana es cuestion de un boton más y una línea: así de escalable es el enfoque de fragments.

Práctica

Ejercicio 1: Navegación con extras entre dos pantallas

Crea dos Activities: CatalogoActivity con un boton "Ver oferta del día", y OfertaActivity que muestre el nombre de la oferta y su descuento. Al tocar el boton, CatalogoActivity debe abrir OfertaActivity enviando el nombre "Laptop Gamer Pro" y el descuento 25 (entero). En OfertaActivity, lee los extras usando constantes definidas en su companion object y muestra en un TextView algo como "Laptop Gamer Pro con 25% de descuento". Si algun dato no llega, muestra valores por defecto amables.

Solución: la clave está en definir las constantes en OfertaActivity y usarlas en ambos lados. En OfertaActivity defines `const val EXTRA_NOMBRE` y `const val EXTRA_DESCUENTO` en el companion object. En onCreate lees con `intent.getStringExtra(EXTRA_NOMBRE) ?: "Oferta especial"` e `intent.getIntExtra(EXTRA_DESCUENTO, 0)`. En CatalogoActivity, dentro del listener del boton, creas `Intent(this, OfertaActivity::class.java)`, agregas los extras con `putExtra` usando `OfertaActivity.EXTRA_NOMBRE` y `OfertaActivity.EXTRA_DESCUENTO`, y llamas a `startActivity`. No olvides declarar `OfertaActivity` en el `AndroidManifest.xml`. El patron es identico al Ejemplo 1: constantes compartidas, Elvis para el String, valor por defecto para el Int.

Ejercicio 2: Formulario que devuelve un resultado

Modifica el ejercicio anterior: en lugar de abrir la oferta directamente, CatalogoActivity abre una pantalla EncuestaActivity con un EditText donde el usuario escribe su nombre y un boton "Enviar". Al enviar, la pantalla se cierra y CatalogoActivity muestra un Toast que diga "Gracias por participar, [nombre]". Si el usuario sale con el boton atras sin enviar, no debe mostrarse ningún Toast.

Solución: usa la API de Activity Result como en el Ejemplo 2. En CatalogoActivity declara como `propiedad` el `lanzador` con `registerForActivityResult(ActivityResultContracts.StartActivityForResult())` y en el lambda verifica que `resultado.resultCode == Activity.RESULT_OK` antes de leer `resultado.data?.getStringExtra(...)` y mostrar el Toast. El boton abre la pantalla con `lanzador.launch(Intent(this, EncuestaActivity::class.java))`. En EncuestaActivity, el boton "Enviar" crea un Intent vacío, le pone el extra con el texto del EditText, llama a

setResult(Activity.RESULT_OK, datos) y luego a finish(). Si el usuario presiona atras, Android cierra con RESULT_CANCELED automaticamente y el lambda no muestra nada porque el if filtra ese caso. Recuerda: el registerForActivityResult va como propiedad de la clase, nunca dentro de onCreate ni dentro del listener.

Ejercicio 3: Pantalla de "compartir y contactar" con intents implícitos

Crea una Activity llamada ContactoActivity con tres botones: "Compartir app", "Visitar sitio web" y "Cómo llegar". El primero comparte el texto "Descarga MiApp aquí: <https://www.ejemplo.com/app>" usando ACTION_SEND con chooser. El segundo abre "<https://www.lucasegaacademy.com>" en el navegador. El tercero abre la app de mapas centrada en una ubicacion (usa la Uri "geo:19.4326,-99.1332" con ACTION_VIEW, que son las coordenadas del centro de la Ciudad de Mexico). Los tres lanzamientos deben estar protegidos: si no hay app disponible, muestra un Toast avisando.

Solución: sigue el patron del Ejemplo 3. Crea un metodo privado por accion y un metodo lanzarSeguro(intent) que verifique intent.resolveActivity(packageManager) != null dentro de un try-catch de ActivityNotFoundException. Para compartir: Intent(Intent.ACTION_SEND).apply { type = "text/plain"; putExtra(Intent.EXTRA_TEXT, "...") } envuelto en Intent.createChooser(intent, "Compartir con..."). Para la web: Intent(Intent.ACTION_VIEW, Uri.parse("https://www.lucasegaacademy.com")). Para mapas: Intent(Intent.ACTION_VIEW, Uri.parse("geo:19.4326,-99.1332")). Fijate que los tres usan el mismo metodo guardian: esa es la gracia del patron, escribir la proteccion una vez y reutilizarla. El Toast de aviso va en el else y en el catch.

Ejercicio 4: App de dos secciones con fragments

Crea una Activity SeccionesActivity con dos botones ("Productos" y "Ayuda") y un FragmentContainerView. Crea ProductosFragment y AyudaFragment, cada uno con su layout que muestre un titulo distinto. Al iniciar la app debe verse ProductosFragment. Al tocar cada boton, el fragment correspondiente debe reemplazar al anterior, y el boton atras del sistema debe regresar al fragment previo. Asegurate de que al girar el teléfono no se dupliquen los fragments.

Solución: replica la estructura del Ejemplo 4. El layout lleva los dos botones arriba y el FragmentContainerView con id debajo. Cada fragment infla su layout en onCreateView y configura su titulo en onViewCreated. En SeccionesActivity crea el metodo mostrarFragment(fragment, agregarAlHistorial) con beginTransaction, replace, addToBackStack condicional y commit. En onCreate, agrega el fragment inicial SOLO si

savedInstanceState == null (esto evita el duplicado al rotar). Los botones llaman a `mostrarFragment(ProductosFragment(), true)` y `mostrarFragment(AyudaFragment(), true)`. El `true` en ambos hace que `addToBackStack` guarde cada cambio, así el botón atrás recorre el historial de fragments antes de salir de la Activity. Si al probar ves dos títulos encimados, revisa que estes usando `replace` y no `add`.

Ejercicio 5: Fragment que recibe datos con arguments

Toma el `ProductosFragment` del ejercicio anterior y modifícalo para que reciba el nombre de una categoría (por ejemplo "Electrónica") a través de `arguments`, y muestre "Categoría: Electrónica" en su título. La Activity debe crear el fragment usando un método de fábrica (companion object con una función `nuevaInstancia(categoría: String)`) que arme el Bundle de `arguments`.

Solución: en `ProductosFragment` agrega el companion object con una constante `ARG_CATEGORIA` y la función de fábrica:

```
companion object {
    private const val ARG_CATEGORIA = "arg_categoria"

    fun nuevaInstancia(categoría: String): ProductosFragment {
        val fragment = ProductosFragment()
        val args = Bundle()
        args.putString(ARG_CATEGORIA, categoría)
        fragment.arguments = args
        return fragment
    }
}
```

En `onViewCreated` lees la categoría con `arguments?.getString(ARG_CATEGORIA)` ?: "General" y la muestras en el título. En la Activity, en lugar de `ProductosFragment()` usas `ProductosFragment.nuevaInstancia("Electrónica")`. La lección clave: los `arguments` se asignan ANTES de agregar el fragment a la transacción, y se leen Después de que el fragment ya existe (típicamente en `onViewCreated`). Nunca intentes pasar datos por el constructor del fragment: Android necesita un constructor vacío para recrearlo al girar la pantalla, y los `arguments` sobreviven a esa recreación automáticamente.

Ejercicio 6: Flujo completo, login simulado con resultado y fragment de bienvenida

Combina todo: una MainActivity que al abrirse muestra un LoginFragment con un EditText para el usuario y un boton "Entrar". Al tocar "Entrar", el fragment le avisa a la Activity (llamando a un metodo público de la Activity mediante `requireActivity()`), la Activity reemplaza el LoginFragment por un BienvenidaFragment pasandole el nombre de usuario con arguments, y el BienvenidaFragment muestra "Hola, [usuario]". Además, agrega la transaccion al historial para que el boton atras regrese al login.

Solución: este ejercicio une fragments, arguments y comunicacion fragment-Activity. En LoginFragment, el boton hace (`requireActivity()` as MainActivity).`usuarioIngreso(etUsuario.text.toString())`. En MainActivity defines el metodo `usuarioIngreso(nombre: String)` que crea `BienvenidaFragment.nuevaInstancia(nombre)` (con su companion de fabrica como en el ejercicio 5) y ejecuta la transaccion con `replace` y `addToBackStack`. BienvenidaFragment lee el argumento en `onViewCreated` y arma el saludo. El cast `requireActivity()` as MainActivity funciona porque el fragment solo vive dentro de esa Activity; en proyectos grandes se prefiere una interfaz para desacoplar, pero para este nivel el cast directo es aceptable y muy usado. Prueba el flujo completo: escribe un nombre, entra, verifica el saludo, presiona atras y confirma que regresa al login con el campo vacío o con el texto anterior según tu implementacion.

Resumen de la unidad

- Un Intent es un objeto que describe una operacion a realizar; es el canal oficial para pedirle a Android que abra pantallas o ejecute acciones.
- Los intents explicitos nombran la Activity destino con su clase (`Intent(this, DetalleActivity::class.java)`) y sirven para navegar DENTRO de tu app.
- Los extras transportan datos entre pantallas con pares clave-valor usando `putExtra()` para enviar y `getStringExtra()/getIntExtra()/etc.` para recibir.
- Define las claves de extras como constantes en un companion object de la Activity destino para evitar errores de escritura y facilitar el mantenimiento.
- Maneja siempre el caso de dato ausente: operador Elvis (`?:`) para Strings anulables y valor por defecto para tipos numericos.
- Toda Activity debe estar declarada en `AndroidManifest.xml`; sin esa declaracion la navegacion se cae.

- La API de Activity Result (`registerForActivityResult`) permite abrir una pantalla esperando su respuesta; la segunda pantalla devuelve datos con `setResult(RESULT_OK, datos)` y `finish()`.
- Los intents implícitos describen una ACCION (`ACTION_SEND`, `ACTION_VIEW`, `ACTION_DIAL`) más los DATOS (Uri, tipo MIME) y dejan que Android elija la app capaz de atenderlos.
- Usa `Intent.createChooser` para compartir, `ACTION_DIAL` (no `ACTION_CALL`) para marcar números sin permisos especiales, y URIs "tel:", "geo:" y "mailto:" según el caso.
- Protege SIEMPRE los intents implícitos con `resolveActivity(packageManager)` o try-catch de `ActivityNotFoundException` para que la app nunca se caiga si no hay app disponible.
- Un Fragment es un componente de interfaz reutilizable con ciclo de vida propio que vive dentro de una Activity; no puede existir solo.
- Usa fragments para reutilizar interfaz, adaptar pantallas a teléfonos y tablets, y cambiar partes de la pantalla; usa Activities nuevas para pantallas independientes.
- El `FragmentManager` administra los fragments; las transacciones (`beginTransaction`, `add/replace/remove`, `addToBackStack`, `commit`) son cómo se agregan, reemplazan y quitan.
- Agrega el fragment inicial solo cuando `savedInstanceState == null` para evitar duplicados al girar la pantalla.
- Usa `replace()` para navegacion entre secciones, `addToBackStack()` para que el boton atras recorra el historial de fragments, y configura vistas en `onViewCreated`.
- Pasa datos iniciales a un fragment con arguments (Bundle) mediante un metodo de fabrica en el companion object; nunca uses el constructor para eso.
- La comunicacion entre fragments pasa por la Activity anfitriona: el fragment obtiene su actividad con `requireActivity()` y le invoca metodos publicos.

Ejemplo 5: Maestro-detalle adaptable, el superpoder de los fragments (clase 15)

Este ejemplo no es código para copiar tal cual, sino un diseño guiado que demuestra POR QUÉ existen los fragments. Escenario: app de recetas con una lista de recetas (maestro) y el detalle de la receta elegida (detalle).

El truco está en crear dos fragments reutilizables: ListaRecetasFragment (muestra los nombres de las recetas) y DetalleRecetaFragment (muestra ingredientes y pasos, y recibe el id de la receta por arguments, como en el Ejercicio 5). La Activity anfitriona decide COMO acomodarlos según el espacio disponible.

En un teléfono, el layout de la Activity tiene un solo FragmentContainerView. Al iniciar se muestra la lista. Cuando el usuario toca una receta, la lista le avisa a la Activity (mediante un metodo público, igual que en el Ejercicio 6), y la Activity ejecuta una transaccion `replace()` poniendo el DetalleRecetaFragment con el id recibido por arguments, agregando la transaccion al historial con `addToBackStack`. El usuario ve primero la lista, toca, ve el detalle, y con atras regresa a la lista. Navegacion clasica de teléfono.

En una tablet, el layout de la Activity (en la carpeta `res/layout-sw600dp`, que Android usa automaticamente en pantallas de al menos 600dp de ancho) tiene DOS contenedores lado a lado: uno angosto para la lista y uno ancho para el detalle. Al iniciar, la Activity pone la lista en el contenedor izquierdo y el detalle de la primera receta en el derecho. Cuando el usuario toca otra receta, la Activity solo reemplaza el fragment del contenedor derecho, SIN tocar la lista y SIN usar el historial (no hay "pantalla anterior" a la cual volver: ambas están visibles). El codigo de ListaRecetasFragment y DetalleRecetaFragment NO CAMBIA NI UNA Línea entre teléfono y tablet: solo cambia el layout de la Activity y un par de decisiones en la logica de la transaccion.

Para detectar en codigo si estas en modo "dos paneles", la Activity puede verificar si el segundo contenedor existe: `findViewById<View>(R.id.contenedorDetalle) != null`. Si existe, estas en tablet y reemplazas solo el panel derecho; si no existe, estas en teléfono y reemplazas el único contenedor agregando al historial. Este patron "maestro-detalle" es exactamente cómo funcionan las apps de correo y de configuracion en tablets, y es la demostracion perfecta de la promesa de los fragments: escribe la pieza una vez, acomodala donde quieras. Si hubieras construido la lista y el detalle como Activities separadas, el modo tablet sería casi imposible sin duplicar codigo.

Mirando al futuro: el Navigation Component

Todo lo que aprendiste en esta unidad (intents explicitos, FragmentManager, transacciones, pila de retroceso) es la base sobre la que esta construido el Navigation Component, la libreria moderna de AndroidX para navegacion con fragments. En lugar de escribir transacciones a mano, defines un "grafo de navegacion" en un archivo XML visual: cada pantalla (fragment) es un nodo, cada flecha entre nodos es una accion de navegacion, y navegar es tan simple como `findNavController().navigate(R.id.accion_detalle)`. El componente maneja por ti el historial, las animaciones de transicion y el paso de argumentos con seguridad de tipos (con el plugin Safe

Args, los argumentos viajan como propiedades tipadas en lugar de Strings sueltos en un Bundle).

Por que te cuento esto si no lo veremos a fondo en esta unidad. Primero, para que sepas que existe y no te sorprenda cuando lo veas en documentacion o en ofertas de trabajo: hoy es el estandar recomendado por Google para apps con fragments. Segundo, para que entiendas que NO es magia distinta: por dentro hace exactamente los beginTransaction, replace y addToBackStack que acabas de aprender. Quien domina las transacciones manuales entiende el Navigation Component en una tarde; quien solo sabe el componente se pierde cuando algo falla por dentro. Tu ahora perteneces al primer grupo. Cuando el curso llegue a la unidad de arquitectura moderna, retomaremos el Navigation Component y te parecera familiar desde el primer minuto.

Lista de verificación: antes de dar por terminado un flujo de navegación

Usa esta lista cada vez que construyas navegacion nueva, como un piloto antes de despegar:

1. Todas las Activities nuevas están declaradas en el AndroidManifest.xml.
2. Las claves de extras son constantes compartidas, sin Strings magicos duplicados.
3. Cada lectura de extras maneja el caso de dato ausente (Elvis o valor por defecto).
4. Todo intent implicito pasa por la verificacion de resolveActivity o try-catch.
5. El boton atras del sistema se proba en cada flujo: lleva al lugar que el usuario esperaria.
6. El fragment inicial se agrega solo con savedInstanceState == null.
7. Las transacciones que el usuario esperaria "deshacer" con atras usan addToBackStack.
8. Se usó replace() donde correspondia y no hay fragments encimados al rotar la pantalla.
9. Los arguments de fragments se asignan antes de la transaccion y se leen en onCreateViewCreated o después.
10. Ninguna referencia a vistas de un fragment sobrevive a onDestroyView.
11. Los datos que viajan en extras o arguments son pequeños (ids, textos cortos, números); lo pesado va por ViewModel o repositorio.
12. Los textos que ve el usuario están en strings.xml, no quemados en el codigo (esto facilita traducciones mañana).

Preguntas frecuentes de la unidad

P: Si los intents explícitos abren pantallas de mi app, puedo usar un intent explícito para abrir una pantalla de OTRA app. R: Técnicamente sí, si conoces el nombre exacto de la clase (con `ComponentName`), pero es una mala idea en la práctica: las apps cambian sus clases internas sin avisar y tu código se rompería. Para interactuar con otras apps usa intents implícitos, que son el mecanismo diseñado para eso. Los explícitos son para tu casa; los implícitos para el vecindario.

P:Cuál es la diferencia entre `finish()` y presionar el botón atrás. R: En la práctica, casi ninguna para el usuario: ambas cierran la Activity actual y muestran la anterior de la pila. `finish()` es la versión programática: la llamas cuando TU decides que la pantalla debe cerrarse (por ejemplo, después de guardar un formulario). El botón atrás es la versión del usuario. Un detalle: si llamas a `finish()` justo después de `startActivity()`, la pantalla actual se cierra y NO queda en la pila, así que el usuario no podrá volver a ella con atrás. Ese truco se usa en pantallas de "splash" o de login que no quieres que reaparezcan.

P: Puedo pasar un objeto completo de mi app en un extra, no solo Strings y números. R: Sí, si el objeto implementa `Parcelable` (la forma moderna y rápida en Android) o `Serializable` (más lenta, heredada de Java). Con `Parcelable` usas `putExtra("clave", miObjeto)` y `getParcelableExtra("clave", MiClase::class.java)`. Es muy útil para pasar, por ejemplo, un Producto con nombre, precio y descripción de la lista al detalle. Pero ojo con la buena práctica número 10 de la lista de verificación: pasa objetos razonablemente pequeños. Si el objeto pesa mucho, mejor pasa solo su id y que la pantalla destino lo cargue del repositorio o la base de datos.

P: Por qué mi fragment pierde los datos al girar la pantalla. R: Porque al girar, Android destruye y recrea la Activity y sus fragments. Lo que guardaste en variables normales del fragment se pierde, pero lo que pusiste en arguments SOBREVIVE, porque el sistema guarda y restaura el Bundle automáticamente. Por eso insistimos tanto en el patrón de arguments con método de fábrica: es la forma oficial de darle "memoria contra rotaciones" a un fragment. Para datos más complejos o que cambian con el tiempo existe `onSaveInstanceState` y, más adelante en el curso, los ViewModels.

P: Replace o add, cuál uso. R: `replace()` en el 95% de los casos de navegación entre secciones: quita lo anterior y pone lo nuevo, limpio y predecible. `add()` solo cuando quieras superponer fragments deliberadamente (por ejemplo, un fragment semitransparente encima del contenido). Si alguna vez ves contenido "fantasma" detrás de tu pantalla actual, busca un `add()` que debió ser `replace()`.

P: Los fragments reemplazan a las Activities. Debo hacer toda mi app con una sola Activity y muchos fragments. R: Esa arquitectura de "una sola Activity" existe y es popular en apps grandes modernas (casi siempre con el Navigation Component). Pero no es obligatoria ni siempre la mejor para empezar. Muchas apps profesionales excelentes usan varias Activities, cada una con sus fragments. La regla de esta unidad sigue valiendo: Activity para pantallas independientes, fragments para contenido modular y reutilizable. Cuando domines ambos, la arquitectura de una sola Activity te resultara natural si algun proyecto la pide.

P: Que hago si startActivity me dice que no encuentra la actividad pero SI está en el manifest.

R: Revisa tres cosas en orden: primero, que el nombre en el manifest coincida exactamente con la clase (incluyendo el paquete si usas nombre completo); segundo, que estes pasando la clase correcta en el Intent (un error de dedo en DetalleActivity::class.java no lo detecta el autocompletado si hay dos clases parecidas); tercero, limpia y reconstruye el proyecto (Build, Clean Project y luego Rebuild), porque a veces Android Studio cachea un manifest viejo. Nueve de cada diez veces el problema está en el primer punto.

Glosario de la unidad

- Intent: objeto que describe una operacion a realizar; el mensajero oficial de Android para navegacion y acciones.
- Intent explicito: intent que nombra directamente la Activity destino; para navegar dentro de tu app.
- Intent implicito: intent que describe una accion y unos datos sin nombrar destino; Android elige la app capaz de atenderlo.
- Extra: par clave-valor dentro de un Intent para transportar datos entre pantallas.
- startActivity(): metodo que lanza el Intent y abre la pantalla o pide la accion.
- Pila de actividades (back stack): pila que Android mantiene con las pantallas abiertas; el boton atras la recorre hacia abajo.
- Activity Result API: mecanismo moderno (registerForActivityResult) para abrir una pantalla esperando su respuesta.
- setResult() / finish(): metodos con los que una pantalla devuelve datos y se cierra.
- ACTION_SEND, ACTION_VIEW, ACTION_DIAL: acciones estandar de intents implicitos para compartir, abrir contenido y marcar números.
- Chooser: selector que muestra al usuario las apps capaces de atender un intent implicito.

- `resolveActivity()`: verificación que indica si existe alguna app capaz de atender un intent implícito.
- **Fragment**: componente de interfaz reutilizable con ciclo de vida propio que vive dentro de una Activity.
- **FragmentManager**: administrador de los fragments de una Activity (`supportFragmentManager`).
- **Transacción**: operación (`add`, `replace`, `remove`) que modifica los fragments, construida con `beginTransaction` y ejecutada con `commit`.
- `addToBackStack()`: guarda una transacción en el historial para que el botón atrás la revierta.
- **FragmentManager**: contenedor moderno en el layout donde se colocan los fragments con transacciones.
- **Arguments**: Bundle con los datos iniciales de un fragment; sobrevive a la recreación por rotación.
- `onCreateView` / `onViewCreated`: métodos del ciclo de vida del fragment donde se infla el layout y se configura la vista, respectivamente.

Proyecto integrador de la unidad: mini app "Mis Lugares"

Para cerrar la unidad, aquí tienes un proyecto que une todo lo aprendido en una sola app pequeña pero completa. Construirla te tomará una o dos tardes y al terminar tendrás una pieza real para tu portafolio.

La app "Mis Lugares" permite guardar lugares favoritos. Pantalla 1 (MainActivity): muestra un fragment con la lista de lugares guardados (usa dos fragments intercambiables: `ListaLugaresFragment` y `MapaResumenFragment`, cambiados con botones y transacciones con historial, como en el Ejemplo 4). Un botón flotante "Agregar" abre `AgregarLugarActivity` con un intent explícito ESPERANDO resultado: el usuario escribe el nombre del lugar y elige una categoría. Al guardar, `AgregarLugarActivity` devuelve los datos con `setResult` y `finish`, y MainActivity los recibe con la Activity Result API y actualiza la lista.

Cada lugar de la lista, al tocarlo, abre `DetalleLugarActivity` con un intent explícito llevando el nombre y las coordenadas en extras (constantes en companion object, valores por defecto si faltan). `DetalleLugarActivity` tiene tres botones con intents implícitos protegidos: "Compartir" (`ACTION_SEND` con chooser para presumir el lugar), "Ver en mapa" (`ACTION_VIEW` con Uri geo:) y "Llamar al lugar" (`ACTION_DIAL` con Uri tel.; número ficticio). Todo lanzamiento pasa por tu método `lanzarSeguro` con `resolveActivity`.

El `DetalleLugarActivity`, en tablets, muestra el detalle como dos fragments lado a lado (info + foto), reutilizando los mismos fragments del teléfono gracias al patron maestro-detalle del Ejemplo 5.

Si logras construir "Mis Lugares" sin mirar los ejemplos más que para recordar un detalle, felicidades: dominas la Unidad 5. Los intents explicitos e implicitos y los fragments ya son herramientas tuyas, no temas del libro.

UNIDAD 6: RecyclerView y listas dinámicas

Introducción

Bienvenido a la Unidad 6, una de las más prácticas de todo el curso. Hasta ahora tus pantallas muestran contenido fijo: textos que escribiste en el XML, botones que colocaste a mano. Pero las apps reales casi nunca funcionan así. Piensa en tu lista de contactos, en el feed de una red social, en el catalogo de una tienda, en la bandeja de entrada de tu correo: son LISTAS de elementos que cambian, crecen, se filtran y se actualizan. Nadie escribe mil contactos a mano en un XML. Necesitas un mecanismo que tome una coleccion de datos (una lista de objetos en Kotlin) y la convierta automaticamente en una lista visual desplazable, donde cada elemento se dibuje con el mismo diseno pero con datos distintos. Ese mecanismo se llama RecyclerView, y esta unidad lo desmenuza por completo.

Esta unidad cubre las clases 16, 17 y 18. La clase 16 presenta el RecyclerView y el patron ViewHolder: entenderas por que las listas ingenuas (como un ScrollView lleno de vistas creadas a mano) se vuelven lentas e inutilizables con cientos de elementos, y como el RecyclerView resuelve el problema reciclando las vistas que salen de la pantalla en lugar de crear miles de ellas. La analogía central es la de una cinta transportadora con charolas reutilizables, y una vez que la entiendes, el resto del componente encaja solo. La clase 17 es el corazón práctico: construir un Adapter paso a paso. El Adapter es el puente entre tus datos y la pantalla: aprenderas sus tres metodos obligatorios (onCreateViewHolder, onBindViewHolder y getItemCount), como escribir un ViewHolder que guarda las referencias a las vistas de cada fila, y como conectar todo para que la lista cobre vida. La clase 18 completa la experiencia visual e interactiva: los LayoutManagers (lista vertical, lista horizontal, cuadrícula), las CardViews para disenar tarjetas bonitas con sombras y bordes redondeados, y la deteccion de clics en cada elemento para abrir, por ejemplo, una pantalla de detalle con un intent explicito (mira como las unidades se conectan entre si).

Por que importa tanto esta unidad. El RecyclerView es, sin exagerar, el componente visual más usado en el desarrollo Android profesional. Prácticamente toda app con contenido dinámico lo usa: chats, tiendas, redes sociales, reproductores de música, apps de noticias, administradores de archivos. En entrevistas de trabajo para puestos Android junior, saber explicar el patron ViewHolder y escribir un Adapter desde cero es casi una pregunta obligada. No es un tema decorativo: es una habilidad troncal que usarás en todos tus proyectos de ahora en adelante. Además, el RecyclerView te enseña un patron de diseno (el adaptador) que aparece en muchos otros lugares de la programacion: la idea de un objeto que traduce datos a una representacion visual es universal.

Al terminar esta unidad sabras explicar con tus palabras qué es un RecyclerView, por qué recicla vistas y qué problema resuelve. Sabras crear el layout XML de un item (la plantilla visual que se repite), escribir un ViewHolder que conserva las referencias a las vistas de ese item, y construir un Adapter completo que conecte una lista de objetos Kotlin con la pantalla. Sabras configurar un RecyclerView en una Activity: asignarle un LayoutManager (lineal vertical, horizontal o cuadrícula), conectarle el Adapter y actualizar los datos cuando cambien. Sabras envolver tus items en CardViews para lograr tarjetas con elevacion, esquinas redondeadas y un aspecto profesional. Sabras detectar el clic en cada elemento de la lista y reaccionar a el, por ejemplo abriendo una pantalla de detalle con los datos del elemento tocado. Y conoceras los errores tipicos (olvidar el LayoutManager y ver una pantalla vacia, no notificar cambios al Adapter, inflar layouts con el contexto equivocado) y como evitarlos.

La unidad esta organizada como la anterior: explicación detallada con analogias de la vida real, ejemplos paso a paso con codigo Kotlin y XML explicado línea por línea (incluyendo el ejemplo completo obligatorio: data class, item XML, Adapter con ViewHolder y la Activity que lo usa), práctica con ejercicios numerados y soluciones explicadas, y resumen final. Un consejo especifico para esta unidad: el RecyclerView tiene varias piezas que deben encajar (layout del item, ViewHolder, Adapter, LayoutManager, Activity). Si algo no se ve en pantalla, casi siempre es que una de esas piezas falta o está mal conectada. La sección de errores comunes te dara un protocolo de diagnostico en orden para esos casos. Escribe el codigo, ejecútalo, rompe cosas a proposito y arreglalas: así se aprende de verdad este componente.

Explicación detallada

Clase 16: RecyclerView y el patrón ViewHolder

Imagina que trabajas en un restaurante de sushi con cinta transportadora. Los platos pasan frente a los clientes, cada uno toma el que quiere, y los platos vacios regresan a la cocina,

donde los lavan y los vuelven a llenar con sushi nuevo. El restaurante NO fabrica un plato nuevo para cada cliente: reutiliza los mismos platos una y otra vez. El RecyclerView funciona exactamente igual. La "cinta" es la lista desplazable que ve el usuario. Los "platos" son las vistas de cada fila (los items). Cuando una fila sale de la pantalla por arriba al hacer scroll, el RecyclerView NO la destruye: la guarda en una pila de vistas recicladas, la limpia y la reutiliza para mostrar el siguiente dato que entra por abajo. Con este truco, una lista de diez mil elementos solo necesita crear en memoria las vistas que caben en pantalla más un par extra: quizá doce o quince vistas en total, en lugar de diez mil. Esa es la diferencia entre una app fluida y una app que se traba y se queda sin memoria.

Para apreciar el truco, piensa en la alternativa ingenua: un ScrollView con un LinearLayout al que le agregas mil vistas creadas en código. Funciona con veinte elementos. Con doscientos empieza a tardar en abrir. Con dos mil, la app se congela o truena por falta de memoria, porque cada vista es un objeto pesado con su jerarquía de dibujo. El RecyclerView nació precisamente para reemplazar a ese enfoque y al viejo ListView, que reciclaba vistas pero de forma torpe y sin una arquitectura limpia. La palabra clave del nombre es "Recycler" (reciclador): todo el componente está diseñado alrededor de la reutilización.

Ahora, el patrón ViewHolder. Cada vez que el RecyclerView necesita mostrar un dato en una fila, tiene que buscar dentro del layout de la fila los TextView e ImageView para ponerles el contenido. Buscar vistas con findViewById es una operación costosa: recorre la jerarquía de vistas comparando identificadores. Si lo hicieras en cada scroll para cada fila, la lista se movería a tirones. La solución es buscar UNA SOLA VEZ y guardar los resultados. Un ViewHolder es una pequeña clase contenedora que guarda las referencias a las vistas de UNA fila (por ejemplo, un TextView para el nombre y otro para el precio). Cuando la fila se crea, se buscan sus vistas una vez y se guardan en el ViewHolder; a partir de ahí, cada vez que esa fila se recicla para mostrar otro dato, el código usa las referencias guardadas sin volver a buscar nada. Volviendo al restaurante: el ViewHolder es como pegarle una etiqueta al plato que dice "aquí va el sushi de salmon, aquí el de atun", para que la cocina no tenga que adivinar la forma del plato cada vez.

El RecyclerView no trabaja solo: es un equipo de tres piezas con responsabilidades muy claras. El LayoutManager decide COMO se acomodan las filas: en lista vertical (LinearLayoutManager), en lista horizontal, o en cuadrícula (GridLayoutManager). El Adapter es el puente entre tus datos y las filas: él sabe cuántos datos hay, crea los ViewHolders cuando se necesitan y "ata" (bind) cada dato a las vistas de su fila. El ViewHolder, como vimos, guarda las referencias a las vistas de una fila. Y el RecyclerView mismo es el coordinador: detecta el scroll, pide ViewHolders al Adapter, los recicla y le dice al LayoutManager donde colocarlos. Esta separación de responsabilidades es elegante: si

mañana quieres cambiar de lista vertical a cuadrícula, solo cambias el `LayoutManager` en UNA línea; el `Adapter` y los datos ni se enteran.

El flujo de trabajo del `RecyclerView`, paso a paso, es así. Primero, tú le das al `RecyclerView` su `LayoutManager` y su `Adapter` (normalmente en el `onCreate` de la `Activity`). El `RecyclerView` le pregunta al `Adapter` "cuantos elementos hay" (`getItemCount`). Si hay, le pide "creame un `ViewHolder` para una fila" (`onCreateViewHolder`): ahí se infla el layout XML del item y se guardan las referencias. Luego le dice "ata el dato número 5 a este `ViewHolder`" (`onBindViewHolder`): ahí tu código toma el objeto de la posición 5 y pone sus valores en los `TextView` e `ImageView` del holder. El `LayoutManager` coloca la fila en pantalla. Cuando el usuario hace scroll y la fila 0 sale por arriba, el `RecyclerView` la mete a la pila de reciclaje; cuando necesita mostrar la fila 12 que entra por abajo, saca una vista reciclada de la pila y le pide al `Adapter` que la "re-ate" con el dato 12 (`onBindViewHolder` de nuevo, SIN pasar por `onCreateViewHolder` porque la vista ya existe). Crear es caro y pasa pocas veces; atar es barato y pasa muchas veces. Ese desbalance es toda la magia del rendimiento.

Un detalle importante: el `Adapter` NO sabe nada de `Activities` ni de navegación. Su único trabajo es convertir datos en filas. Si quieres que al tocar una fila se abra otra pantalla, el `Adapter` avisa "tocaron la posición 7" mediante un listener (una interfaz de callback o un lambda), y la `Activity` decide que hacer (por ejemplo, lanzar un intent explícito al detalle). Esta separación mantiene el `Adapter` reutilizable: el mismo `Adapter` de productos puede usarse en el catálogo (donde el clic abre el detalle) y en el carrito (donde el clic elimina el producto), cambiando solo el listener. Lo veremos en código en la clase 18.

Por último, una comparación que te harán en entrevistas: `RecyclerView` vs `ListView`. `ListView` fue el componente de listas del Android antiguo: reciclaba vistas, pero el patrón `ViewHolder` era opcional y manual (muchos tutoriales viejos ni lo usaban), solo soportaba lista vertical, y no tenía animaciones de cambios integradas. `RecyclerView` lo reemplazo por completo: `ViewHolder` obligatorio por diseño, `LayoutManagers` intercambiables, animaciones automáticas al insertar o eliminar elementos, y una arquitectura extensible. Hoy, `ListView` solo existe por compatibilidad con código viejo. Si ves un tutorial que usa `ListView`, cierralo: está desactualizado. Todo lo que aprendas aquí sobre `RecyclerView` es lo vigente y lo que te pedirán en el mundo real.

Clase 17: El Adapter paso a paso, el puente entre datos y pantalla

Si el `RecyclerView` es la cinta transportadora del restaurante, el `Adapter` es el cocinero: recibe los ingredientes crudos (tu lista de objetos Kotlin), sabe cuantos platos debe preparar, arma cada plato en su charola (`ViewHolder`) y lo decora con los ingredientes correctos antes de mandarlo a la cinta. Técnicamente, un `Adapter` es una clase que hereda de

RecyclerView.Adapter<MiViewHolder> y que implementa tres metodos obligatorios. Vamos a diseccionarlos uno por uno, porque entender que hace cada uno (y cuando lo llama el RecyclerView) es el nucleo de esta clase.

Antes de los metodos, la declaracion de la clase. Un Adapter típico se ve así en su encabezado:

```
class ProductoAdapter(private val productos: List<Producto>) :  
    RecyclerView.Adapter<ProductoAdapter.ProductoViewHolder>() {
```

Lee esto como una oracion: "ProductoAdapter recibe una lista de productos y es un Adapter que trabaja con ProductoViewHolder". El parametro productos es la fuente de datos: la lista de objetos que se mostraran, un elemento por fila. El tipo generico <ProductoAdapter.ProductoViewHolder> le dice al RecyclerView que clase de ViewHolder usara este Adapter; el ViewHolder suele declararse como clase interna (inner class) del Adapter porque solo a el le sirve. Fijate que la lista se recibe en el constructor: el Adapter no va a buscar datos a una base de datos ni a internet; alguien (la Activity) se los entrega ya listos. Esa es otra muestra de la separacion de responsabilidades: el Adapter pinta, no investiga.

Metodo 1: onCreateViewHolder. Su firma es:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
    ProductoViewHolder
```

Su trabajo: crear UNA fila nueva desde cero. Recibe el parent (el RecyclerView, que sirve para obtener el contexto y los parametros de layout) y el viewType (un entero que distingue tipos de fila cuando hay varios disenos distintos; si todas tus filas son iguales, lo ignoras). Adentro, inflas el layout XML del item con LayoutInflater.from(parent.context).inflate(R.layout.item_producto, parent, false) y devuelves un ProductoViewHolder construido con esa vista. El false final es crucial: significa "crea la vista pero no la agregues todavia al parent"; el RecyclerView se encarga de agregarla cuando corresponda. Este metodo se llama POCAS veces: solo cuando el RecyclerView necesita filas nuevas porque su pila de reciclaje esta vacia (típicamente al inicio, para llenar la pantalla). Si tu lista muestra 8 filas visibles, onCreateViewHolder se llamara unas 10 o 12 veces en total, aunque tengas mil datos.

Metodo 2: onBindViewHolder. Su firma es:

```
override fun onBindViewHolder(holder: ProductoViewHolder, position: Int)
```

Su trabajo: tomar el dato de la posición indicada y "atarlo" a las vistas del ViewHolder recibido. Adentro haces algo como `val producto = productos[position]` y luego `holder.tvNombre.text = producto.nombre`, `holder.tvPrecio.text = producto.precio`, etc. Este método se llama MUCHAS veces: cada vez que una fila (nueva o reciclada) necesita mostrar un dato, incluyendo cada scroll. Por eso debe ser RÁPIDO: nada de consultas a base de datos, nada de cargar imágenes pesadas de forma sincrónica, nada de operaciones costosas. Solo toma el objeto y asigna valores a vistas. Si alguna vez tu lista se mueve a tirones, el primer sospechoso es un `onBindViewHolder` que hace demasiado trabajo.

Método 3: `getItemCount`. Su firma es:

```
override fun getItemCount(): Int = productos.size
```

Su trabajo: decirle al `RecyclerView` cuantos elementos hay en total. Casi siempre es una sola línea que devuelve el tamaño de la lista. Parece trivial, pero es la base de todo: el `RecyclerView` lo usa para saber cuántas filas existen, cuando dejar de pedir más y como calcular el tamaño del scroll. Si devuelves 0 por error (por ejemplo, porque la lista aun no se había cargado cuando creaste el `Adapter`), verás una pantalla vacía aunque tus datos existan: error clásico que veremos en la sección de diagnóstico.

Ahora el ViewHolder, la clase interna:

```
inner class ProductoViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)
{
    val tvNombre: TextView = itemView.findViewById(R.id.tvNombre)
    val tvPrecio: TextView = itemView.findViewById(R.id.tvPrecio)
}
```

Hereda de `RecyclerView.ViewHolder` y recibe la vista raíz de la fila (`itemView`). En su cuerpo busca UNA VEZ los `TextView` con `itemView.findViewById` y los guarda en propiedades. Nota el detalle: se busca en `itemView`, no en la `Activity`. Cada fila inflada tiene su propio ViewHolder con sus propias referencias; cuando la fila se recicla, el mismo ViewHolder se reusa con otro dato. En código moderno con View Binding estas líneas se ven un poco distintas, pero la idea es idéntica: buscar una vez, guardar, reutilizar.

Cómo se actualizan los datos. Si la lista cambia (agregas un producto, filtras por una búsqueda), NO basta con modificar la lista: debes avisarle al `Adapter` para que repinte. El método más simple es `notifyDataSetChanged()`, que dice "todo cambio, vuelve a atar todo lo visible". Funciona, pero es torpe: repinta de más y sin animación. Los métodos finos son `notifyItemInserted(pos)`, `notifyItemRemoved(pos)` y `notifyItemChanged(pos)`, que avisan de UN

cambio puntual y producen una bonita animacion de entrada o salida. La forma moderna y recomendada para listas que cambian mucho es usar ListAdapter con DiffUtil, que calcula automaticamente las diferencias entre la lista vieja y la nueva y anima solo lo necesario; es un poco más de código inicial a cambio de listas perfectamente eficientes. Para este nivel, domina primero notifyDataSetChanged y los notifyItem*, y deja ListAdapter/DiffUtil como tu siguiente paso cuando te sientas comodo (los mencionaremos en los ejercicios).

Un patron útil: si tu Adapter necesita modificar su lista internamente (por ejemplo, un metodo agregarProducto()), declara la fuente como MutableList en el constructor o guarda una referencia mutable privada. Pero cuidado con una trampa: si la Activity también conserva la lista original y la modifica por su cuenta sin avisar al Adapter, la pantalla y los datos se desincronizan. Regla simple: los cambios a los datos SIEMPRE van acompañados de su notificación al Adapter, en el mismo momento y en el mismo lugar del código. Datos y notificación son una pareja inseparable.

Para cerrar la clase, el ciclo de vida completo de una fila en una frase memorable: el RecyclerView CREA pocas veces (onCreateViewHolder), ATA muchas veces (onBindViewHolder) y CUENTA una vez por consulta (getItemCount). Si interiorizas ese ritmo, leer y escribir Adapters se vuelve mecánico, en el buen sentido: un patron que aplicas igual para productos, contactos, mensajes o canciones.

Clase 18: LayoutManager, CardView y clics en la lista

Con el Adapter dominado, esta clase viste la lista de gala y la hace interactiva. Tres temas: cómo se acomodan las filas (LayoutManager), cómo se ven bonitas (CardView) y como responden al tacto (clics).

El LayoutManager es el "decorador de interiores" del RecyclerView: decide la posición y el tamaño de cada fila en pantalla. Lo asignas con una sola línea: `recyclerView.layoutManager = LinearLayoutManager(this)`. Los tres managers que debes conocer son: `LinearLayoutManager`, que acomoda las filas en una lista vertical (el valor por defecto que usarás el 80% de las veces); el mismo `LinearLayoutManager` pero en horizontal, pasándole la orientación: `LinearLayoutManager(this, LinearLayoutManager.HORIZONTAL, false)`, perfecto para carruseles de imágenes o listas de categorías que se deslizan de lado; y `GridLayoutManager(this, 2)`, que organiza las filas en una cuadrícula de N columnas (aquí, 2), ideal para galerías de fotos o catálogos de productos. Cambiar entre estos tres es cuestión de UNA línea, sin tocar el Adapter ni los datos: esa es la belleza de la separación de responsabilidades que vimos en la clase 16. También existe `StaggeredGridLayoutManager` para cuadrículas con filas de alturas distintas (estilo Pinterest), pero con los tres básicos cubres casi todo.

Un detalle que muerde a los principiantes: si OMITES el `LayoutManager`, el `RecyclerView` se muestra vacío aunque el `Adapter` tenga datos. No hay error, no hay crash, solo una pantalla en blanco que te vuelve loco. Por eso la primera pregunta del protocolo de diagnostico (que verás más abajo) es siempre "le pusiste `LayoutManager`". Grabatelo: `RecyclerView` sin `LayoutManager` es como un marco sin colgar: existe, pero no muestra nada.

Ahora la `CardView`. Una `CardView` es un contenedor (hereda de `FrameLayout`) que dibuja a sus hijos dentro de una "tarjeta": un rectángulo con esquinas redondeadas, fondo de color configurable y una sombra de elevación que le da profundidad. Es el ingrediente visual que hace que una lista se vea moderna y profesional en lugar de una simple columna de texto. Para usarla necesitas la dependencia de Material (`com.google.android.material:material`) y el widget `androidx.cardview.widget.CardView` o, mejor aun, `com.google.android.material.card.MaterialCardView`, que es la versión moderna con más opciones de estilo. En el layout del ítem envuelves tu contenido en la `CardView` y ajustas tres atributos clave: `cardCornerRadius` (que tan redondeadas son las esquinas, por ejemplo 12dp), `cardElevation` (que tan marcada es la sombra, por ejemplo 4dp) y `cardUseCompatPadding="true"` (agrega un pequeño relleno para que la sombra no se corte en los bordes). Dentro de la tarjeta pones tu diseño normal: un `LinearLayout` con el nombre, el precio, una descripción. El resultado visual es inmediato: cada fila parece una tarjeta flotando sobre el fondo, y la lista entera sube de nivel estético con muy poco esfuerzo.

Consejo de diseño: no abuses. Elevaciones gigantes y radios enormes en listas largas cansan la vista y pueden afectar el rendimiento de dibujo en teléfonos viejos. Una tarjeta limpia (radio 8-12dp, elevación 2-4dp, buen margen entre tarjetas de 8dp) se ve profesional sin excesos. Y recuerda poner un fondo de color suave al `RecyclerView` o a la pantalla para que las tarjetas blancas destaquen por contraste.

Finalmente, los clics. El RecyclerView NO trae un "onItemClickListener" incorporado como lo tenía el viejo ListView; tú lo implementas, y hay dos formas principales. La forma recomendada para este nivel: el Adapter recibe un lambda (o una interfaz) en su constructor, por ejemplo `private val onClic: (Producto) -> Unit`, y en `onBindViewHolder` asigna `holder.itemView.setOnClickListener { onClic(producto) }`. La Activity, al crear el Adapter, pasa el lambda con la acción deseada: `ProductoAdapter(lista) { producto -> abrirDetalle(producto) }`. Fíjate en la elegancia: el Adapter no sabe que es un "detalle" ni conoce intents; solo avisa "tocaron este producto" y la Activity decide. La segunda forma es definir una interfaz dentro del Adapter (`interface OnProductoClickListener { fun onClick(producto: Producto) }`) que la Activity implementa; es más verbosa pero útil cuando hay VARIAS acciones por fila (clic normal, clic largo, botón de borrar dentro de la fila). Para botones DENTRO de la fila (un botón "agregar al carrito" en cada tarjeta), el listener se asigna a ese botón específico dentro del ViewHolder o en `onBindViewHolder`, no a toda la fila.

Un error sutil con los clics: usar la posición capturada en el lambda de forma ingenua. Si escribes `holder.itemView.setOnClickListener { onClic(productos[position]) }` usando el parámetro `position` de `onBindViewHolder`, en la mayoría de los casos funciona, pero si la lista cambia (se elimina un elemento) entre el atado y el clic, la posición puede quedar desactualizada. La forma robusta es capturar el OBJETO (producto) en lugar de la posición, como en el ejemplo anterior, o usar `holder.bindingAdapterPosition` dentro del listener para obtener la posición actualizada al momento del clic. Para este nivel, capturar el objeto es simple y seguro: adoptalo como hábito.

Con `LayoutManager`, `CardView` y clics, tu lista ya es una lista profesional: se acomoda como quieras, se ve bonita y responde al usuario. En la sección de ejemplos lo verás todo junto en código real, y en la práctica lo extenderás con búsqueda, eliminación y cuadrículas.

Errores comunes de principiantes y cómo evitarlos

Error 1: la lista se ve vacía aunque hay datos. Diagnóstico en orden: primero verifica que asignaste el `LayoutManager` (`recyclerView.layoutManager = LinearLayoutManager(this)`); sin él, nada se muestra y no hay error que te avise. Segundo, verifica que `getItemCount()` devuelva el tamaño real (si creaste el Adapter con una lista vacía y luego llenaste OTRA lista, el Adapter sigue viendo la vacía). Tercero, verifica que el RecyclerView tenga tamaño en el layout (si su `height` es `0dp` sin constraints o `weight` que lo estire, es invisible). Cuarto, revisa el Logcat: un crash silencioso en `onCreateViewHolder` (por ejemplo, un layout mal inflado) también deja la lista vacía. Este protocolo ordenado te ahorra horas: el 90% de las "listas vacías" caen en los dos primeros puntos.

Error 2: modificar la lista sin notificar al Adapter. Sintoma: agregas un elemento y no aparece, o eliminas uno y la app se cae con un `IndexOutOfBoundsException` al hacer scroll. Causa: cambiaste los datos pero no llamaste a `notifyDataSetChanged()` ni a `notifyItem*`. Como evitarlo: convierte en regla inquebrantable que todo cambio de datos vaya acompañado de su notificación, en la misma función. Si tienes un método `agregarProducto()` en la Activity, que termine con `adapter.notifyItemInserted(posicion)`. Si el Adapter expone métodos que mutan su lista interna, que esos métodos notifiquen por dentro.

Error 3: `findViewById` en `onBindViewHolder`. Sintoma: la lista funciona pero se mueve a tirones, con scroll poco fluido. Causa: estas buscando vistas en cada `atado` en lugar de hacerlo una vez en el `ViewHolder`. Como evitarlo: TODA búsqueda de vistas vive en el `ViewHolder`; `onBindViewHolder` solo asigna valores a las referencias ya guardadas. Si ves un `findViewById` dentro de `onBindViewHolder`, muevelo al `ViewHolder` de inmediato.

Error 4: inflar el item con el contexto o los parametros equivocados. Sintoma: las filas se ven con tamaño raro (gigantes, o de ancho incorrecto). Causa típica: usar `inflate(R.layout.item_producto, null)` en lugar de `inflate(R.layout.item_producto, parent, false)`. Sin el `parent`, el layout pierde sus `LayoutParams` y el `RecyclerView` no sabe cómo medir la fila. Como evitarlo: usa siempre la forma completa con `parent` y `false`. Es una de esas líneas que se copian igual en todos los Adapters: memoriza el patrón.

Error 5: el clic abre el detalle del producto equivocado. Sintoma: tocas la fila 5 y se abre el detalle de la fila 3. Causa: capturaste la posición en el listener en lugar del objeto, y la lista cambio entre el `atado` y el clic (o el `ViewHolder` se reciclo). Como evitarlo: captura el objeto (producto) en el lambda, no la posición. Si necesitas la posición por alguna razón, usa `holder.bindingAdapterPosition` DENTRO del listener, que devuelve la posición actualizada al momento del clic.

Error 6: `NullPointerException` con listas que llegan tarde. Sintoma: crash al abrir la pantalla porque el Adapter se creó con una lista null o porque los datos llegan de internet después. Causa: pasar null como fuente de datos o no contemplar la carga asíncrona. Como evitarlo: el Adapter siempre recibe una lista no nula (aunque sea vacía al inicio: `mutableListOf()`), y cuando los datos llegan, actualizas la lista y notificas. Nunca disenes un Adapter que acepte null como fuente.

Error 7: poner TODA la lógica de negocio dentro del Adapter. Sintoma: un Adapter de 300 líneas que hace peticiones de red, abre pantallas y formatea fechas. Causa: confundir "puente visual" con "cerebro de la app". Como evitarlo: el Adapter solo pinta filas y reporta clics. La carga de datos, la navegación y las reglas de negocio viven en la Activity, el Fragment o el `ViewModel`. Un Adapter sano rara vez pasa de 100 líneas.

Buenas prácticas

1. Nombra con claridad: `ProductoAdapter`, `ProductoViewHolder`, `item_producto.xml`. Quien lea tu código debe ubicar cada pieza sin esfuerzo.
2. Usa `data class` para tus modelos (`data class Producto(val nombre: String, val precio: Double)`): te da `equals`, `toString` y `copy` gratis, y expresa que el objeto es un contenedor de datos.
3. Mantén `onBindViewHolder` ligero: solo asignaciones. El trabajo pesado (cargar imágenes de internet) va con librerías como `Coil` o `Glide`, que lo hacen en segundo plano.
4. Prefiere `notifyItemInserted/Removed/Changed` sobre `notifyDataSetChanged` cuando el cambio es puntual: mejor rendimiento y animaciones gratis.
5. Cuando una fila tenga varios elementos interactivos (botón de favorito, botón de borrar), dale a cada uno su listener en el `ViewHolder` o en `onBindViewHolder`, y reporta cada acción por separado al listener.
6. Extrae los textos visibles al archivo `strings.xml` y los colores a `colors.xml` desde el primer día; las listas son lo primero que se traduce o se retematiza en una app real.
7. Si la lista puede quedar vacía, diseña un "estado vacío" amable (un `TextView` que diga "No hay productos todavía") y alterna su visibilidad según `getItemCount() == 0`. Una pantalla en blanco parece un bug; un mensaje claro parece una decisión de diseño.
8. Prueba tu lista con MUCHOS elementos (genera 500 de prueba en código) y con UNO solo, y con CERO. Los tres casos revelan bugs distintos: rendimiento, diseño de fila única y estado vacío.
9. Reutiliza `Adapters` cuando el diseño de fila coincide: el mismo `Adapter` de contactos sirve para "todos los contactos" y para "resultados de búsqueda" si la única diferencia es la lista que recibe.
10. Documenta con un comentario breve que representa cada `ViewHolder` cuando un `Adapter` maneja varios tipos de fila (`viewTypes`): el código de `viewTypes` se vuelve confuso rápido sin esas notas.

Ejemplos paso a paso

Ejemplo 1: El modelo y los layouts (data class, item XML con CardView, layout de la Activity)

Escenario: app "Cursos" que muestra una lista de cursos de la academia. Cada fila es una tarjeta con el titulo del curso, su categoría y su duracion en horas. Al tocar una tarjeta se abra el detalle (eso lo conectamos en el Ejemplo 3).

Paso 1: el modelo de datos. Una data class simple:

```
package com.lucasega.cursos

data class Curso(
    val titulo: String,
    val categoria: String,
    val horas: Int
)
```

Explicación. data class genera automaticamente equals(), hashCode(), toString() y copy(): justo lo que necesitas para comparar y duplicar objetos de datos sin escribir codigo repetitivo. Las tres propiedades son val (inmutables): un curso no cambia su titulo a mitad de la lista, y la inmutabilidad evita una familia entera de bugs. Este objeto es el "ingrediente crudo" que el Adapter convertira en filas.

Paso 2: el layout del item, item_curso.xml. Es la plantilla visual que se repetira por cada curso, envuelta en una CardView:

```

<?xml version="1.0" encoding="utf-8"?>
<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardCornerRadius="12dp"
    app:cardElevation="4dp"
    app:cardUseCompatPadding="true">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="16dp">

        <TextView
            android:id="@+id/tvTitulo"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="18sp"
            android:textStyle="bold"
            android:text="Titulo del curso" />

        <TextView
            android:id="@+id/tvCategoria"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="4dp"
            android:textSize="14sp"
            android:text="Categoria" />

        <TextView
            android:id="@+id/tvHoras"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="4dp"
            android:textSize="14sp"
            android:text="0 horas" />

    </LinearLayout>

</com.google.android.material.card.MaterialCardView>

```

Explicación línea por línea. La raíz es `MaterialCardView` (necesitas la dependencia `Material` en tu `build.gradle`). Su ancho es `match_parent` para que la tarjeta ocupe todo el ancho de la lista, y su alto `wrap_content` para ajustarse al contenido. `android:layout_margin="8dp"` separa cada tarjeta de sus vecinas y de los bordes. Los tres atributos `app:` son la magia visual: `cardCornerRadius="12dp"` redondea las esquinas, `cardElevation="4dp"` dibuja la sombra que hace "flotar" la tarjeta, y `cardUseCompatPadding="true"` evita que la sombra se recorte en los bordes en versiones viejas de Android. Dentro, un `LinearLayout` vertical con padding de 16dp contiene los tres `TextView`: título en negrita y más grande, categoría y horas más pequeños. Los ids (`tvTitulo`, `tvCategoria`, `tvHoras`) son los que el `ViewHolder` buscara una sola vez. Los textos de ejemplo ("Titulo del curso", etc.) son solo marcadores: el `Adapter` los reemplazara con datos reales.

Paso 3: el layout de la Activity, `activity_cursos.xml`, con el `RecyclerView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:background="#F2F2F2">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp"
        android:text="Cursos disponibles"
        android:textSize="22sp"
        android:textStyle="bold" />

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/rvCursos"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />

</LinearLayout>
```

Explicación. Un encabezado fijo con el título de la sección y debajo el `RecyclerView` ocupando todo el espacio restante (`height 0dp` más `weight 1`). El fondo gris claro (`#F2F2F2`) hace que las tarjetas blancas destaquen por contraste, tal como recomienda la buena práctica de diseño. Nota que aquí NO se define cómo se acomodan las filas: eso lo decidirá el `LayoutManager` en código, y podrás cambiarlo sin tocar este XML.

Ejemplo 2: El Adapter completo con ViewHolder (clase 17)

Este es el corazón de la unidad. El mismo archivo contiene el Adapter y su ViewHolder como clase interna:

```
package com.lucasega.cursos

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

class CursoAdapter(
    private val cursos: List<Curso>,
    private val onClick: (Curso) -> Unit
) : RecyclerView.Adapter<CursoAdapter.CursoViewHolder>() {

    inner class CursoViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView)
    {
        val tvTitulo: TextView = itemView.findViewById(R.id.tvTitulo)
        val tvCategoria: TextView = itemView.findViewById(R.id.tvCategoria)
        val tvHoras: TextView = itemView.findViewById(R.id.tvHoras)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
    CursoViewHolder {
        val vista = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_curso, parent, false)
        return CursoViewHolder(vista)
    }

    override fun onBindViewHolder(holder: CursoViewHolder, position: Int) {
        val curso = cursos[position]
        holder.tvTitulo.text = curso.titulo
        holder.tvCategoria.text = curso.categoria
        holder.tvHoras.text = "${curso.horas} horas"
        holder.itemView.setOnClickListener { onClick(curso) }
    }

    override fun getItemCount(): Int = cursos.size
}
```

Explicación línea por línea. El encabezado declara que `CursoAdapter` recibe dos cosas: la lista de cursos a mostrar y un lambda `onClic` que se ejecutara con el curso tocado. Hereda de `RecyclerView.Adapter` parametrizado con `CursoViewHolder`, su propio `ViewHolder` interno. Fijate que el `Adapter` NO recibe ningún contexto ni conoce `Activities`: solo datos y un callback. Esa es la separacion de responsabilidades en accion.

La inner class `CursoViewHolder` hereda de `RecyclerView.ViewHolder` y recibe `itemView`, la vista raíz de la fila. En su cuerpo busca los tres `TextView` UNA SOLA VEZ con `itemView.findViewById` y los guarda en propiedades `val`. Cada fila que se crea obtiene su propio `ViewHolder` con sus propias referencias; cuando la fila se recicla, el `ViewHolder` viaja con ella y las referencias siguen valiendo. Aquí se paga el costo de buscar vistas una unica vez por fila.

`onCreateViewHolder`: toma el parent (el `RecyclerView`) para obtener su contexto con `parent.context`, infla `item_curso.xml` con el patron correcto (`parent, false`) y devuelve un `CursoViewHolder` envolviendo la vista. Recuerda: este metodo se llama pocas veces, solo para crear las filas que necesita la pantalla más un pequeño colchon.

`onBindViewHolder`: el metodo que más se ejecuta. Toma el curso de la posicion, asigna sus valores a los `TextView` del holder (fijate en el string template `"${curso.horas} horas"` que arma el texto "30 horas"), y asigna el listener de clic a la fila completa con `holder.itemView.setOnClickListener`. El lambda captura el OBJETO curso, no la posicion: así, aunque la lista cambie después, el clic siempre reporta el curso correcto (evitamos el Error 5). Al tocar la fila, se invoca `onClic(curso)`, que es el lambda que la `Activity` proporcione al construir el `Adapter`.

`getItemCount`: devuelve `cursos.size` en una sola línea. Si la lista esta vacia, devuelve 0 y el `RecyclerView` no pide ninguna fila.

Nota de diseno: este `Adapter` es reutilizable tal cual en cualquier pantalla que muestre cursos con este diseno de tarjeta. Si otra pantalla quiere que el clic haga algo distinto (abrir el detalle vs agregar a favoritos), solo cambia el lambda al construir el `Adapter`. El `Adapter` pinta; la `Activity` decide.

Una variante que verás en codigo profesional: en lugar del lambda, el `Adapter` define una interfaz `OnCursoClickListener`. El lambda es más corto y es el estilo moderno en Kotlin; la interfaz es más explicita cuando hay multiples acciones por fila. Para este curso, quedate con el lambda hasta que necesites más de una accion.

Ejemplo 3: La Activity que usa el RecyclerView (clases 16-18 juntas)

Ahora conectamos las piezas en CursosActivity: obtiene el RecyclerView del layout, le asigna un layoutManager, crea el Adapter con datos de ejemplo y define qué pasa al tocar una tarjeta (abrir el detalle con un intent explicito, uniendo la Unidad 5 con la 6):

```

package com.lucasega.cursos

import android.content.Intent
import android.os.Bundle
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView

class CursosActivity : AppCompatActivity() {

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

        val rvCursos = findViewById<RecyclerView>(R.id.rvCursos)

        rvCursos.layoutManager = LinearLayoutManager(this)

        val cursos = listOf(
            Curso("Desarrollo Android desde cero", "Programacion", 30),
            Curso("Kotlin para principiantes", "Programacion", 20),
            Curso("Diseno de interfaces con XML", "Diseno", 15),
            Curso("Bases de datos con Room", "Programacion", 18),
            Curso("Publicar apps en Play Store", "Negocios", 8)
        )

        val adapter = CursoAdapter(cursos) { curso ->
            abrirDetalle(curso)
        }
        rvCursos.adapter = adapter
    }

    private fun abrirDetalle(curso: Curso) {
        val intent = Intent(this, DetalleCursoActivity::class.java)
        intent.putExtra(DetalleCursoActivity.EXTRA_TITULO, curso.titulo)
        intent.putExtra(DetalleCursoActivity.EXTRA_CATEGORIA, curso.categoria)
        intent.putExtra(DetalleCursoActivity.EXTRA_HORAS, curso.horas)
        startActivity(intent)
    }
}

```

Explicación línea por línea. En onCreate, después de setContentView, buscamos el RecyclerView por su id. La primera configuracion obligatoria es el layoutManager:

`LinearLayoutManager(this)` acomoda las tarjetas en lista vertical. Sin esta línea, la lista saldría vacía (Error 1). Luego creamos la lista de cursos de ejemplo con `listOf`; en una app real estos datos vendrían de una base de datos o de internet, pero para el ejemplo una lista fija es perfecta.

Creamos el Adapter pasándole la lista y el lambda de clic: `{ curso -> abrirDetalle(curso) }`. Ese lambda es el `onClic` que el Adapter invocará en `onBindViewHolder` cuando toquen una tarjeta. Finalmente, `rvCursos.adapter = adapter` conecta el puente: a partir de aquí el `RecyclerView` empieza a pedirle `ViewHolders` y conteos al Adapter, y las tarjetas aparecen.

El método `abrirDetalle` muestra la conexión entre unidades: es un intent explícito clásico de la Unidad 5 que lleva los datos del curso tocado a `DetalleCursoActivity` usando extras con constantes (el patrón que ya dominas). Así se cierra el círculo: la lista muestra, el clic avisa, la Activity navega.

Tres variaciones rápidas para experimentar (cambia UNA línea cada vez y ejecuta):

1. Lista horizontal (carrusel): reemplaza el `LayoutManager` por `LinearLayoutManager(this, LinearLayoutManager.HORIZONTAL, false)`. Las tarjetas se deslizarán de lado. Nota: con orientación horizontal conviene que el item tenga un ancho fijo (por ejemplo 280dp) en lugar de `match_parent`, o cada tarjeta ocupará toda la pantalla.
2. Cuadrícula de 2 columnas: usa `GridLayoutManager(this, 2)`. Perfecta para catálogos. Las tarjetas se acomodan solas en dos columnas sin tocar el Adapter.
3. Agregar un curso en caliente: guarda el adapter en una propiedad de la clase, convierte la lista a `mutableListOf`, agrega un elemento y llama a `adapter.notifyItemInserted(0)` para ver la animación de entrada. Así practicas el patrón "cambio de datos + notificación".

Con estos tres ejemplos tienes el circuito completo y funcional: modelo, item con `CardView`, Adapter con `ViewHolder` y Activity que configura, conecta y reacciona. Todo lo que sigue (ejercicios y resumen) es practicar variaciones sobre esta base.

El ciclo de vida de una fila, visto con lupa

Para que el reciclaje deje de ser teoría, sigamos con números lo que pasa cuando abres la pantalla de cursos con 100 elementos y solo caben 8 en pantalla.

Al abrir la pantalla, el `RecyclerView` pregunta `getItemCount` y recibe 100. Calcula que necesita unas 10 filas para llenar la pantalla más un pequeño colchón. Llama a `onCreateViewHolder` 10 veces: se inflan 10 tarjetas y nacen 10 `ViewHolders`. Inmediatamente llama a `onBindViewHolder` 10 veces con las posiciones 0 a 9: cada tarjeta muestra su curso. Hasta aquí, 10 creaciones y 10 atados.

Haces scroll hacia abajo. La fila de la posición 0 sale por arriba: el RecyclerView la despega y la guarda en su pila de reciclaje (no la destruye, no libera sus vistas, solo la aparta). Casi al mismo tiempo necesita mostrar la posición 10 que entra por abajo: en lugar de crear una fila nueva, saca la fila reciclada de la pila y llama a `onBindViewHolder` con la posición 10. El mismo ViewHolder que mostraba el curso 0 ahora muestra el curso 10: sus TextView simplemente reciben texto nuevo. Sigues bajando hasta la posición 50: en total se habrán hecho unas 10 o 12 creaciones (algunas filas extra para el colchon) y unos 60 atados. La memoria usada es la de 12 tarjetas, no la de 100.

Ahora subes de nuevo. La fila de la posición 50 sale por abajo, entra a la pila, y la posición 9 que reaparece por arriba reutiliza una fila reciclada con un nuevo atado. El sistema nunca "recuerda" que esa vista antes mostraba el curso 50: cada atado escribe TODOS los valores visibles desde cero. Y aquí hay una lección práctica escondida: en `onBindViewHolder` debes asignar SIEMPRE todos los campos visibles. Si una fila a veces muestra un icono de "favorito" y otras no, no basta con ponerlo visible cuando toca: debes ponerlo invisible (o gone) explícitamente cuando NO toca. Porque la fila reciclada puede traer el icono visible de su vida anterior, y si tu código no lo oculta, verás iconos fantasma en filas que no deberían tenerlos. Regla de oro del reciclaje: nunca dejes un estado visual "heredado"; cada atado pinta la fila completa como si fuera nueva.

Este modelo también explica por qué `onCreateViewHolder` recibe el `viewType`: cuando tu lista tiene DOS diseños de fila distintos (por ejemplo, encabezados y elementos, que veremos en la siguiente sección), el RecyclerView mantiene una pila de reciclaje SEPARADA por cada tipo, para no mezclar una tarjeta de curso con un encabezado de sección. Cada tipo se crea con su propio layout y se ata con su propio ViewHolder.

ViewHolder moderno con View Binding

En proyectos actuales es común usar View Binding en lugar de `findViewById`: el plugin genera una clase por cada layout XML con propiedades tipadas para cada vista con id, eliminando las búsquedas manuales y los casts. El patrón ViewHolder no cambia en esencia, solo se escribe más limpio. Así se vería nuestro Adapter con View Binding (suponiendo que el binding de `item_curso.xml` se llama `ItemCursoBinding`):

```

class CursoAdapter(
    private val cursos: List<Curso>,
    private val onClick: (Curso) -> Unit
) : RecyclerView.Adapter<CursoAdapter.CursoViewHolder>() {

    inner class CursoViewHolder(
        private val binding: ItemCursoBinding
    ) : RecyclerView.ViewHolder(binding.root) {

        fun atar(curso: Curso) {
            binding.tvTitulo.text = curso.titulo
            binding.tvCategoria.text = curso.categoria
            binding.tvHoras.text = "${curso.horas} horas"
            binding.root.setOnClickListener { onClick(curso) }
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
CursoViewHolder {
        val binding = ItemCursoBinding.inflate(
            LayoutInflater.from(parent.context), parent, false
        )
        return CursoViewHolder(binding)
    }

    override fun onBindViewHolder(holder: CursoViewHolder, position: Int) {
        holder.atar(cursos[position])
    }

    override fun getItemCount(): Int = cursos.size
}

```

Explicación de las diferencias. El ViewHolder ahora recibe el binding en lugar de la vista cruda, y hereda pasando `binding.root` (la vista raíz del layout) al constructor padre. Ya no hay `findViewById`: `binding.tvTitulo` es una propiedad generada, con el tipo correcto (`TextView`) y sin posibilidad de un id mal escrito que trueque en ejecución (si el id no existe, el código ni compila). Además introducimos un método `atar(curso)` dentro del ViewHolder: mueve la lógica de "pintar una fila" al propio holder, dejando `onBindViewHolder` en una sola línea que delega. Es una organización más limpia que muchos equipos prefieren: el Adapter decide CUANDO atar, el ViewHolder sabe COMO atar. Para activar View Binding en tu proyecto agregas `viewBinding { enabled = true }` en el bloque android del `build.gradle` del módulo. No es obligatorio para este curso, pero es el estándar en la industria y conviene que lo reconozcas cuando lo veas.

Listas con dos tipos de fila: encabezados y elementos

Un caso real muy común: una lista de cursos agrupada por categoría, donde cada grupo lleva un encabezado ("Programación", "Diseño") seguido de sus cursos. Eso son DOS diseños de fila distintos en el mismo RecyclerView, y se resuelve con `viewTypes`. La idea: el Adapter decide, según la posición, que tipo de fila toca; crea el ViewHolder adecuado para cada tipo; y ata cada uno con su lógica.

Primero modelamos la lista como una lista de "elementos de lista" sellada:

```
sealed class ElementoLista {
    data class Encabezado(val titulo: String) : ElementoLista()
    data class CursoItem(val curso: Curso) : ElementoLista()
}
```

Una sealed class es perfecta aquí: la lista solo puede contener estos dos subtipos, y el compilador lo sabe. Luego el Adapter:

```
class AgendaAdapter(
    private val elementos: List<ElementoLista>,
    private val onClickCurso: (Curso) -> Unit
) : RecyclerView.Adapter<RecyclerView.ViewHolder>() {

    companion object {
        private const val TIPO_ENCABEZADO = 0
        private const val TIPO_CURSO = 1
    }

    inner class EncabezadoViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView) {
```

```

        val tvEncabezado: TextView = itemView.findViewById(R.id.tvEncabezado)
    }

    inner class CursoViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val tvTitulo: TextView = itemView.findViewById(R.id.tvTitulo)
        val tvCategoria: TextView = itemView.findViewById(R.id.tvCategoria)
    }

    override fun getItemViewType(position: Int): Int {
        return when (elementos[position]) {
            is ElementoLista.Encabezado -> TIPO_ENCABEZADO
            is ElementoLista.CursoItem -> TIPO_CURSO
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
RecyclerView.ViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        return when (viewType) {
            TIPO_ENCABEZADO -> {
                val vista = inflater.inflate(R.layout.item_encabezado, parent, false)
                EncabezadoViewHolder(vista)
            }
            else -> {
                val vista = inflater.inflate(R.layout.item_curso_simple, parent,
false)
                CursoViewHolder(vista)
            }
        }
    }

    override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
        when (val elemento = elementos[position]) {
            is ElementoLista.Encabezado ->
                (holder as EncabezadoViewHolder).tvEncabezado.text = elemento.titulo
            is ElementoLista.CursoItem -> {
                val vh = holder as CursoViewHolder
                vh.tvTitulo.text = elemento.curso.titulo
                vh.tvCategoria.text = elemento.curso.categoria
                vh.itemView.setOnClickListener { onClicCurso(elemento.curso) }
            }
        }
    }

    override fun getItemCount(): Int = elementos.size

```

```
}
```

Explicación. El Adapter ahora trabaja con RecyclerView.ViewHolder generico (el padre común) porque maneja dos holders distintos. getItemViewType() es el metodo nuevo: recibe la posicion y devuelve un entero que identifica el tipo de fila; el RecyclerView lo usa para mantener pilas de reciclaje separadas por tipo, como explicamos antes. En onCreateViewHolder, un when sobre el viewType infla el layout correspondiente (item_encabezado.xml para los titulos de sección, item_curso_simple.xml para los cursos) y devuelve el holder adecuado. En onBindViewHolder, otro when distingue el elemento y hace el cast al holder correcto para atar sus vistas. Para construir la lista de elementos, la Activity intercala encabezados y cursos: Encabezado("Programacion"), CursorItem(curso1), CursorItem(curso2), Encabezado("Diseno"), CursorItem(curso3)... Este patron aparece en apps de verdad constantemente (chats con fechas separadoras, agendas con encabezados de día, ajustes con titulos de sección). Dominarlo te pone por encima del nivel "solo se hacer listas simples".

Preguntas frecuentes y glosario de la unidad

P: Mi lista se ve bien pero al hacer scroll rápido las imágenes parpadean o se mezclan. R: Es el reciclaje en accion con trabajo asincrono: una fila reciclada muestra brevemente la imagen VIEJA mientras carga la nueva. La solución tiene dos partes: en onBindViewHolder, pon siempre una imagen de placeholder ANTES de lanzar la carga; y usa una libreria como Coil o Glide, que cancelan automaticamente la carga anterior cuando la vista se recicla. Nunca cargues imágenes de red con codigo manual en onBindViewHolder.

P: Puedo poner un RecyclerView dentro de un ScrollView. R: Tecnicamente si, con setNestedScrollingEnabled(false), pero casi siempre es mala idea: el RecyclerView pierde su superpoder porque el ScrollView padre le pide medir TODAS las filas de golpe, y volvemos al problema de las mil vistas en memoria. Si necesitas encabezados o secciones encima de la lista, la solución correcta es la de viewTypes (encabezados como filas) o un ConcatAdapter, no anidar scrolls.

P: Cuantos elementos aguanta un RecyclerView. R: En la práctica, decenas de miles sin despeinarse, porque solo mantiene en memoria las filas visibles. El limite real lo ponen tus datos (una lista gigante en memoria) no el componente. Para cantidades enormes con paginacion existe la libreria Paging, que carga los datos por bloques: tema avanzado para cuando lo necesites.

P: Adapter o ListAdapter, cuál uso. R: Si estas aprendiendo o la lista es pequeña y cambia poco, RecyclerView.Adapter con notifyItem* es perfecto y es lo que evalúan en entrevistas

junior. Si la lista cambia mucho (un chat, un feed en vivo), ListAdapter con DiffUtil te da eficiencia y animaciones automaticas a cambio de un poco más de estructura. Aprende primero el Adapter clásico (ya lo hiciste) y migra a ListAdapter cuando el proyecto lo pida.

P: Por que mi CardView no muestra la sombra en algunos teléfonos. R: Revisa `cardUseCompatPadding="true"` y que la tarjeta tenga margen suficiente para que la sombra quepa sin recortarse. En fondos del mismo color que la tarjeta la sombra se nota menos: el contraste del fondo ayuda. También verifica que no estes usando hardware antiguo con elevaciones extremas.

Glosario de la unidad:

- RecyclerView: componente que muestra listas desplazables reciclando las vistas de las filas.
- ViewHolder: contenedor que guarda las referencias a las vistas de una fila para no buscarlas en cada atado.
- Adapter: puente entre la lista de datos y las filas; crea ViewHolders y ata datos a ellos.
- onCreateViewHolder: crea una fila nueva inflando su layout; se llama pocas veces.
- onBindViewHolder: ata el dato de una posicion a un ViewHolder; se llama muchas veces, debe ser ligero.
- getItemCount: devuelve el total de elementos; base para que el RecyclerView sepa cuantas filas existen.
- LayoutManager: decide el acomodo de las filas (LinearLayoutManager vertical/horizontal, GridLayoutManager cuadrícula).
- CardView / MaterialCardView: contenedor que dibuja su contenido como tarjeta con esquinas redondeadas y sombra.
- notifyDataSetChanged / notifyItemInserted / notifyItemRemoved / notifyItemChanged: avisos al Adapter de que los datos cambiaron.
- DiffUtil / ListAdapter: herramientas modernas para calcular diferencias entre listas y animar cambios eficientemente.
- viewType: entero que distingue disenos de fila distintos dentro del mismo RecyclerView.
- sealed class: jerarquia cerrada de clases ideal para modelar los tipos de elemento de una lista heterogenea.
- View Binding: generacion de clases de acceso tipado a las vistas de un layout, alternativa moderna a findViewById.

- Estado vacío: diseño alternativo que se muestra cuando la lista no tiene elementos, en lugar de una pantalla en blanco.

Práctica

Ejercicio 1: Tu primer RecyclerView desde cero

Crea una app que muestre una lista de tus 8 películas favoritas. Cada elemento es una data class `Pelicula`(titulo: String, año: Int). El item es un diseño simple (sin `CardView` por ahora): un `LinearLayout` vertical con dos `TextView`. Crea el `Adapter` con su `ViewHolder`, configura el `RecyclerView` con `LinearLayoutManager` vertical en la `Activity` y muestra la lista. Verifica que el scroll sea fluido.

Solución: sigue el patrón de los Ejemplos 1-3 simplificado. La data class `Pelicula` con sus dos propiedades. El `item_pelicula.xml` con `LinearLayout` vertical, dos `TextView` (`tvTitulo`, `tvAño`) y padding. El `PeliculaAdapter` con inner class `PeliculaViewHolder` que guarda las dos referencias, `onCreateViewHolder` inflando con (`parent`, `false`), `onBindViewHolder` asignando título y año, y `getItemCount` devolviendo la lista. En la `Activity`: `findViewById` del `RecyclerView`, `layoutManager = LinearLayoutManager(this)`, lista de 8 películas con `listOf`, `adapter = PeliculaAdapter(lista)`, `rvPelículas.adapter = adapter`. Si la lista sale vacía, aplica el protocolo del Error 1: `LayoutManager` primero, `getItemCount` después. Este ejercicio parece simple, pero hacerlo sin mirar los ejemplos es la prueba de que el patrón se te quedó.

Ejercicio 2: Tarjetas bonitas con CardView y cuadrícula

Toma el ejercicio anterior y envuelve el item en una `MaterialCardView` con esquinas redondeadas de 12dp y elevación de 4dp (no olvides la dependencia `Material` y `cardUseCompatPadding`). Además, cambia el `LayoutManager` a `GridLayoutManager(this, 2)` para mostrar las películas en cuadrícula de 2 columnas. Agrega un tercer `TextView` con el género de la película (actualiza la data class).

Solución: en `item_pelicula.xml`, envuelve el `LinearLayout` existente en `MaterialCardView` con los atributos `app:cardCornerRadius="12dp"`, `app:cardElevation="4dp"`, `android:layout_margin="8dp"` y `app:cardUseCompatPadding="true"`. Agrega `val genero: String` a la data class y un `tvGenero` al layout y al `ViewHolder` (recuerda: la búsqueda del nuevo `TextView` va en el `ViewHolder`, no en `onBindViewHolder`). En la `Activity` cambia la línea del `LayoutManager` por `GridLayoutManager(this, 2)`. El `Adapter` no se toca: ni una línea. Este es el momento "ajá" de la separación de responsabilidades: cambiaste el acomodo y el diseño sin modificar la lógica del puente de datos.

Ejercicio 3: Clic que abre el detalle

Agrega interactividad: al tocar una película debe abrirse `DetallePeliculaActivity` mostrando título, año y género. Modifica el `Adapter` para recibir un lambda `onClic: (Pelicula) -> Unit`, asignalo en `onBindViewHolder` capturando el objeto película, y en la `Activity` pasa el lambda que construye un `intent` explícito con los tres datos como extras (usa constantes en el `companion object` de `DetallePeliculaActivity`, como aprendiste en la Unidad 5).

Solución: en el constructor del `Adapter` agrega el parámetro `private val onClic: (Pelicula) -> Unit`. En `onBindViewHolder`, después de asignar los textos, agrega `holder.itemView.setOnClickListener { onClic(pelicula) }`. En la `Activity`: `val adapter = PeliculaAdapter(peliculas) { pelicula -> val intent = Intent(this, DetallePeliculaActivity::class.java); intent.putExtra(DetallePeliculaActivity.EXTRA_TITULO, pelicula.titulo); ...; startActivity(intent) }`. En `DetallePeliculaActivity` lee los extras con `Elvis` y valores por defecto. No olvides declarar la `Activity` en el `manifest`. Si al tocar se abre el detalle equivocado, revisa que estes capturando el objeto y no la posición (Error 5).

Ejercicio 4: Agregar y eliminar elementos con animacion

Agrega a la `Activity` un botón "Agregar película" que inserte una película nueva al INICIO de la lista con animación, y haz que un clic LARGO sobre una tarjeta la elimine, también con animación. La lista del `Adapter` debe ser mutable.

Solución: cambia la fuente de datos a `mutableListOf<Pelicula>()` (o guarda una `MutableList`). Guarda el `adapter` en una propiedad de la `Activity` para usarlo fuera de `onCreate`. Para agregar: en el listener del botón, `peliculas.add(0, Pelicula("Nueva película", 2026, "Drama"))` y luego `adapter.notifyItemInserted(0)`; el `RecyclerView` animará la entrada en la posición 0. Para eliminar con clic largo: en el `Adapter` agrega otro lambda `onClicLargo: (Pelicula) -> Unit` (o mejor, `(Int) -> Unit` con la posición actualizada), asignalo con `holder.itemView.setOnLongClickListener { onClicLargo(pelicula); true }` (debe devolver `true` para indicar que el evento se consumió). En la `Activity`, el lambda busca el índice (`val pos = peliculas.indexOf(pelicula)`), lo remueve (`peliculas.removeAt(pos)`) y notifica (`adapter.notifyItemRemoved(pos)`). La lección: dato y notificación siempre juntos; y los `notifyItem*` puntuales dan animación gratis frente al `notifyDataSetChanged` genérico.

Ejercicio 5: Buscador que filtra la lista

Agrega un `EditText` de búsqueda encima del `RecyclerView`: al escribir, la lista debe mostrar solo las películas cuyo título contenga el texto buscado (sin importar mayúsculas). Implementa en el `Adapter` un método `filtrar(texto: String)` que actualice lo que se muestra.

Solución: el Adapter necesita DOS listas: la lista completa original (inmutable, de respaldo) y la lista visible (mutable, la que usa getItemCount y onBindViewHolder). El metodo filtrar hace: listaVisible.clear(); si el texto está vacío, agrega todo el respaldo; si no, agrega solo los que cumplan pelicula.titulo.contains(texto, ignoreCase = true); y termina con notifyDataSetChanged(). En la Activity, agrega un TextWatcher al EditText (o usa el listener doOnTextChanged de AndroidX) que llame a adapter.filtrar(texto) en cada cambio. Este ejercicio te enseña un patron muy común en apps reales: el Adapter como dueño de "que se ve", separado de "que existe". Como reto extra, muestra el estado vacío amable ("No se encontraron peliculas") cuando la lista visible quede en cero (buena práctica 7).

Ejercicio 6: Lista horizontal de categorías + investigacion de DiffUtil

Primera parte: crea un segundo RecyclerView HORIZONTAL encima del principal que muestre categorías ("Todas", "Accion", "Comedia", "Drama"...). Al tocar una categoría, la lista principal se filtra por ese genero (reutiliza la logica del Ejercicio 5). Usa LinearLayoutManager horizontal y un item sencillo de ancho fijo. Segunda parte (investigacion): lee la documentacion oficial de ListAdapter y DiffUtil y escribe en tus notas con tus palabras: que problema resuelven frente a notifyDataSetChanged, y en que casos conviene usarlos.

Solución: primera parte: es el mismo patron aplicado dos veces en una pantalla: dos RecyclerViews, dos Adapters, dos LayoutManagers (uno vertical, uno horizontal). El Adapter de categorías puede ser minimalista (una lista de Strings, un TextView por item). Su lambda onClick llama a adapterPeliculas.filtrarPorGenero(genero), un metodo analogo a filtrar pero comparando el genero exacto (o mostrando todo si es "Todas"). Verás que una pantalla puede hospedar varios RecyclerViews independientes sin que se estorben: cada uno es su propio equipo de cinta, cocinero y decorador. Segunda parte: DiffUtil calcula la diferencia minima entre la lista vieja y la nueva (que se inserto, que se movio, que cambio) y ListAdapter lo usa para animar solo esos cambios con el máximo rendimiento; conviene cuando la lista cambia frecuentemente o es muy larga (chats, feeds), mientras que notifyDataSetChanged basta para listas pequeñas y estaticas. Si lograste explicar esto con tus palabras, estas listo para el siguiente nivel.

Decoraciones y animaciones: el toque final profesional

Dos detalles separan una lista funcional de una lista que se siente premium: la separacion visual entre elementos y las animaciones al cambiar datos. Ambos se configuran en una línea o dos.

Primero, el espaciado. En los ejemplos usamos `android:layout_margin="8dp"` en la `CardView` del item para separar las tarjetas. Funciona, pero tiene un efecto colateral: la primera tarjeta queda con margen arriba y la última con margen abajo, lo cual a veces se ve bien y a veces no. La herramienta profesional para esto es `ItemDecoration`: un objeto que el `RecyclerView` consulta para agregar dibujo o espacio alrededor de cada fila, SIN modificar el layout del item. La decoracion más útil es un espaciador simple:

```
class EspacioItem(private val espacio: Int) : RecyclerView.ItemDecoration() {
    override fun getItemOffsets(
        rect: Rect,
        vista: View,
        parent: RecyclerView,
        estado: RecyclerView.State
    ) {
        rect.set(espacio, espacio, espacio, espacio)
    }
}
```

Y se aplica con `rvCursos.addItemDecoration(EspacioItem(16))`. Cada fila recibe 16 pixeles de espacio en sus cuatro lados. La ventaja sobre el margen en el XML: la decoracion vive en la pantalla, no en el item, así que el MISMO item puede mostrarse con espaciado distinto en dos pantallas diferentes sin duplicar layouts. También existen decoraciones que dibujan divisores (líneas) entre filas, muy usadas en listas de ajustes o contactos: `DividerItemDecoration` es la version lista para usar (`rvCursos.addItemDecoration(DividerItemDecoration(this, DividerItemDecoration.VERTICAL))`). Para cuadrículas, el espaciado uniforme con `ItemDecoration` evita el problema del margen duplicado entre columnas vecinas.

Segundo, las animaciones. El `RecyclerView` trae por defecto un `DefaultItemAnimator` que anima automaticamente las inserciones, eliminaciones y cambios CUANDO usas los metodos puntuales `notifyItemInserted`, `notifyItemRemoved` y `notifyItemChanged`. Por eso insistimos en preferirlos sobre `notifyDataSetChanged`: no solo rinden mejor, se ven mejor. La tarjeta nueva se desliza entrando, la eliminada se desvanece saliendo, la modificada hace un pequeño parpadeo. Todo gratis, sin una línea extra de codigo de animacion. Si quieres personalizar (duraciones, efectos), puedes asignar tu propio `RecyclerView.ItemAnimator`, pero el valor por defecto ya se ve profesional.

Un truco final de pulido: cuando la lista carga datos de forma asincrona (de internet o base de datos), evita el "salto" visual mostrando primero unas tarjetas de placeholder con efecto shimmer (brillo deslizante) y reemplazandolas por los datos reales al llegar. Hay librerias para el efecto shimmer, pero incluso unos placeholders grises simples con la misma forma de

tarjeta mejoran enormemente la percepcion de velocidad. El usuario perdona una espera corta si la pantalla le comunica que algo viene en camino; lo que no perdona es una pantalla en blanco que parece trabada.

Proyecto integrador de la unidad: catálogo completo de la academia

Une todo en una app de verdad: el catálogo de cursos de Lucas EGA Academy. Pantalla principal con un buscador arriba, un carrusel horizontal de categorías debajo, y la lista principal de cursos en tarjetas con CardView. Los datos: una lista de al menos 20 cursos (inventa titulos reales del mundo Android: "Kotlin desde cero", "Intents y navegacion", "RecyclerView a fondo", "Room y bases de datos", "Corrutinas", "Jetpack Compose", etc.), cada uno con titulo, categoría, horas y nivel ("Principiante", "Intermedio", "Avanzado").

Requisitos: el carrusel de categorías filtra la lista principal (reutiliza el patron de los Ejercicios 5 y 6); el buscador filtra por titulo ignorando mayusculas; el clic en una tarjeta abre DetalleCursoActivity con un intent explicito llevando todos los datos en extras (constantes en companion object, patron de la Unidad 5); el detalle muestra la informacion completa y un boton "Compartir curso" con un intent implicito ACTION_SEND protegido con resolveActivity (Unidad 5 de nuevo); agregar y quitar cursos de favoritos con un boton en cada tarjeta que cambie de estado visual, notificando con notifyItemChanged para ver la actualizacion puntual; y un estado vacío amable cuando la busqueda no arroje resultados.

Si quieres el nivel experto, agrupa la lista por nivel usando viewTypees con encabezados ("Principiante", "Intermedio", "Avanzado") como en la sección de dos tipos de fila, y prueba todo con 200 cursos generados en codigo para sentir el scroll fluido del reciclaje.

Si construyes este catalogo sin copiar los ejemplos más que para recordar detalles, dominas la Unidad 6: RecyclerView, ViewHolder, Adapter, LayoutManagers, CardView y clics ya son herramientas tuyas. Y fijate en algo hermoso: este proyecto usa las DOS unidades de tu parte del libro juntas, intents y listas, igual que las apps profesionales.

Resumen de la unidad

- El RecyclerView muestra listas desplazables de datos reciclando las vistas que salen de la pantalla, en lugar de crear una vista por elemento: así maneja miles de elementos con solo una docena de vistas en memoria.
- El patron ViewHolder guarda las referencias a las vistas de una fila (buscadas una sola vez con findViewById) para que atar datos sea rápido y el scroll sea fluido.

- El RecyclerView trabaja en equipo: el LayoutManager decide el acomodo (lista vertical, horizontal, cuadrícula), el Adapter conecta datos con filas, y el ViewHolder conserva las vistas de cada fila.
- Un Adapter hereda de RecyclerView.Adapter<MiViewHolder>, recibe la lista de datos (y opcionalmente un listener) en su constructor, e implementa tres metodos: onCreateViewHolder (crea filas nuevas, pocas veces), onBindViewHolder (ata un dato a una fila, muchas veces) y getItemCount (dice cuantos elementos hay).
- En onCreateViewHolder se infla el layout del item con inflate(R.layout.item_x, parent, false); el false es obligatorio para no romper los parametros de layout.
- onBindViewHolder debe ser ligero: solo asignar valores a las vistas del holder; nada de trabajo pesado ni findViewById ahi dentro.
- Todo cambio en los datos exige notificar al Adapter: notifyDataSetChanged() para cambios generales, notifyItemInserted/Removed/Changed() para cambios puntuales con animacion.
- Sin LayoutManager el RecyclerView muestra una pantalla vacia sin errores: asignarlo es el primer paso del diagnostico ante una lista que no aparece.
- LinearLayoutManager sirve para listas verticales y horizontales; GridLayoutManager para cuadrículas; cambiar entre ellos es una sola línea sin tocar el Adapter.
- La CardView (MaterialCardView) envuelve el contenido del item en una tarjeta con esquinas redondeadas (cardCornerRadius), sombra (cardElevation) y relleno compatible (cardUseCompatPadding).
- El RecyclerView no trae clic por defecto: el Adapter recibe un lambda o interfaz de callback, lo invoca en onBindViewHolder capturando el OBJETO (no la posicion), y la Activity decide la accion (por ejemplo, abrir el detalle con un intent).
- Usa data class para los modelos, manten el Adapter enfocado solo en pintar filas, y prueba siempre con cero, uno y cientos de elementos.
- Para listas que cambian mucho, el siguiente nivel es ListAdapter con DiffUtil, que calcula diferencias y anima solo lo necesario.

UNIDAD 7: Persistencia de datos

Introducción

Bienvenido a la Unidad 7 del Curso Completo de Desarrollo Android. Hasta este punto del curso hemos construido aplicaciones que muestran información, responden a toques y navegan entre pantallas. Sin embargo, todas esas aplicaciones tenían algo en común: cuando el usuario cerraba la app y la volvía a abrir, todo se había perdido. Las listas que cargaste, los datos que el usuario escribió, la configuración que eligió, todo desaparecía como si nunca hubiera existido. En esta unidad vamos a cambiar eso de forma definitiva: vas a aprender a guardar datos en el dispositivo para que sobrevivan al cierre de la aplicación, al reinicio del teléfono e incluso a las actualizaciones de la app.

La persistencia de datos es una de las habilidades más importantes de todo desarrollador Android. Piensa en las aplicaciones que usas todos los días: WhatsApp conserva tus conversaciones, Instagram recuerda tu sesión, Spotify guarda tus canciones descargadas, tu aplicación de notas no borra lo que escribiste ayer. Ninguna de esas apps sería útil si olvidara todo cada vez que la cierras. Dominar la persistencia significa convertir tu app de un juguete desechable en una herramienta de verdad, una que los usuarios pueden usar día tras día construyendo sobre lo que ya hicieron.

En esta unidad trabajaremos con tres tecnologías fundamentales, organizadas en tres clases:

En la clase 19 aprenderás `SharedPreferences`, la forma más sencilla de guardar datos en Android. Es perfecta para ajustes y configuraciones pequeñas: el nombre del usuario, si ya vio el tutorial de bienvenida, el tema oscuro o claro, el último nivel alcanzado en un juego. Son datos sencillos, del tipo clave-valor, que leerás y escribirás constantemente. También verás cómo gestionar sesiones de usuario, es decir, cómo recordar que alguien ya inició sesión para no pedirle su contraseña cada vez que abre la app.

En la clase 20 nos meteremos de lleno con `SQLite`, la base de datos relacional que vive dentro de cada teléfono Android. Aprenderás a crear tablas con `SQLiteOpenHelper`, a insertar registros, a consultarlos con filtros y ordenamientos, a actualizarlos y a borrarlos. Esta es la operación que en el mundo de las bases de datos se conoce como CRUD: Create, Read, Update, Delete. El CRUD es el corazón de casi cualquier aplicación que maneje información: una agenda de contactos, una lista de tareas, un inventario, un diario personal. Todo eso se construye sobre estas cuatro operaciones.

En la clase 21 conocerás Room, la biblioteca moderna que Google recomienda hoy en día para trabajar con bases de datos en Android. Room no reemplaza a SQLite, la envuelve: por debajo sigue siendo SQLite, pero Room elimina casi todo el código repetitivo y peligroso que SQLite exige. Verás por qué los proyectos profesionales actuales usan Room en lugar de SQLite crudo, y cómo migrar tu mentalidad del SQL manual al enfoque moderno con entidades, DAOs y bases de datos anotadas.

¿Por qué importa todo esto? Porque una aplicación sin persistencia es una aplicación sin memoria, y una aplicación sin memoria no genera lealtad. Los usuarios vuelven a las apps donde ya tienen algo construido: su perfil, su historial, sus archivos, su progreso. Cada vez que tu app recuerda algo del usuario, estás creando una razón para que regrese. Además, desde el punto de vista profesional, las empresas esperan que todo desarrollador Android domine estas tres tecnologías. En entrevistas de trabajo para puestos Android es extremadamente común que te pidan explicar la diferencia entre SharedPreferences y una base de datos, o que implementes un CRUD básico en vivo. Esta unidad te prepara para responder eso con seguridad.

Al terminar esta unidad sabrás: guardar y leer preferencias del usuario con SharedPreferences, incluyendo el manejo de sesiones de inicio de sesión; crear una base de datos SQLite desde cero con SQLiteOpenHelper, definir su esquema con tablas y columnas, e implementar las cuatro operaciones CRUD completas y seguras; entender las ventajas de Room sobre SQLite crudo y escribir tus primeras entidades, DAOs y bases de datos con la sintaxis moderna; y lo más importante, sabrás decidir cuándo usar cada tecnología: preferencias para ajustes pequeños, SQLite o Room para colecciones de datos estructurados.

Un consejo antes de empezar: la persistencia parece aburrida al principio porque no se ve. No hay animaciones ni colores nuevos. Pero es el cimiento invisible sobre el que se construye todo lo demás. Tómate tu tiempo con los ejemplos, escríbelos a mano en Android Studio en lugar de solo leerlos, y ejecuta cada fragmento de código para ver los datos aparecer y persistir. La mejor forma de aprender persistencia es cerrar la app, volverla a abrir y comprobar con tus propios ojos que tus datos siguen ahí. Esa pequeña magia nunca se vuelve vieja.

Vamos a darle. Tu app está a punto de conseguir memoria.

Explicación detallada

Qué significa persistir datos

Persistir un dato significa guardarlo de forma que sobreviva al ciclo de vida de la aplicación. En Android, cuando una Activity se destruye, todas sus variables desaparecen. Cuando el proceso de la app muere (porque el sistema necesitó memoria o porque el usuario la cerró), todo lo que estaba en RAM se pierde. Persistir es escribir la información en un almacenamiento duradero: el almacenamiento interno del dispositivo, de modo que la próxima vez que la app arranque pueda leerla de nuevo.

Android ofrece varios niveles de persistencia, y elegir el correcto es una decisión de diseño que tomarás en cada proyecto:

1. **SharedPreferences:** para datos pequeños tipo clave-valor. Ajustes, banderas, sesiones, preferencias del usuario. Piensa en ello como una libretita de notas rápidas.
2. **Archivos:** para documentos, imágenes, textos largos. Guardas archivos en el almacenamiento interno o externo. No lo veremos a fondo en esta unidad, pero debes saber que existe.
3. **Bases de datos (SQLite / Room):** para colecciones estructuradas de datos que necesitas buscar, filtrar, ordenar y relacionar. Contactos, tareas, productos, mensajes. Piensa en ello como un archivero con cajones etiquetados.
4. **Almacenamiento en la nube:** para datos que deben sobrevivir incluso si el usuario cambia de teléfono. Firebase, tu propio servidor, etc. Eso lo tocaremos en la Unidad 8 cuando hablemos de red.

La analogía de la vida real que mejor funciona es la de una oficina. SharedPreferences es el corcho de notas junto a tu escritorio: anotas rápido "llamar a Juan", "tema oscuro activado", cosas cortas que consultas de un vistazo. La base de datos es el archivero metálico con carpetas: cada carpeta es una tabla, cada hoja dentro es un registro, y puedes buscar hojas por fecha, por nombre, por categoría. Los archivos sueltos son la pila de papeles sobre el escritorio. Y la nube es la bodega externa donde guardas copias de seguridad. Un buen oficinista no guarda todo en el corcho ni todo en el archivero: usa cada cosa para lo que sirve.

Clase 19: SharedPreferences a fondo

SharedPreferences es un mecanismo que guarda pares de clave y valor en un archivo XML dentro del almacenamiento interno privado de tu aplicación. Cada valor tiene un tipo: String, Int, Long, Float, Boolean o Set de Strings. Eso es todo lo que puede guardar, y esa limitación es también su virtud: es simple, rápido y suficiente para el 90 por ciento de los ajustes de una app.

Hay dos formas de obtener un `SharedPreferences`. La primera es con `getSharedPreferences(nombre, modo)`, que te da un archivo de preferencias con el nombre que tú elijas; puedes tener varios archivos para organizar distintos grupos de ajustes. La segunda es con `getPreferences(modo)` dentro de una `Activity`, que usa un archivo privado asociado a esa `Activity`. En la práctica moderna, la mayoría de los proyectos usan un único archivo con nombre descriptivo, algo como "ajustes_app" o "sesion_usuario".

El modo casi siempre es `MODE_PRIVATE`, que significa que solo tu aplicación puede leer ese archivo. Los otros modos existieron en versiones antiguas de Android y hoy están obsoletos o prohibidos por seguridad. Regla de oro: siempre `MODE_PRIVATE`.

Para escribir datos usas un `Editor`. El flujo es: obtienes el editor con `edit()`, pones valores con `putString()`, `putInt()`, `putBoolean()`, etc., y luego confirmas con `apply()` o `commit()`. La diferencia entre `apply()` y `commit()` es importante y es pregunta clásica de entrevista: `commit()` escribe de forma síncrona y devuelve un booleano indicando si tuvo éxito, bloqueando el hilo hasta terminar; `apply()` escribe de forma asíncrona en segundo plano y no devuelve nada. En el 99 por ciento de los casos debes usar `apply()`, porque no quieres congelar la interfaz esperando a que se escriba un archivo. Usa `commit()` solo cuando necesites saber con certeza inmediata que el dato quedó guardado, por ejemplo justo antes de un proceso crítico.

Para leer datos usas los métodos `get`: `getString(clave, valorPorDefecto)`, `getInt(clave, valorPorDefecto)`, `getBoolean(clave, valorPorDefecto)`, etc. El valor por defecto es lo que se devuelve cuando la clave aún no existe, es decir, la primera vez que se abre la app. Elegir buenos valores por defecto es parte del diseño: si preguntas `getBoolean("modo_oscuro", false)`, estás diciendo "si nunca se configuró, asumimos modo claro".

Un error muy común de principiantes es usar `SharedPreferences` para guardar listas grandes de objetos o datos estructurados complejos. Técnicamente puedes serializar una lista a JSON y guardarla como `String`, y para casos muy pequeños eso funciona, pero en cuanto la lista crece o necesitas buscar dentro de ella, se vuelve un desastre de rendimiento y mantenimiento. Regla práctica: si necesitas guardar más de unas decenas de valores simples, o si necesitas buscar, filtrar u ordenar, eso ya no es trabajo de `SharedPreferences`, es trabajo de una base de datos.

Otro error típico es guardar contraseñas en texto plano en `SharedPreferences`. El archivo es privado para tu app, sí, pero en un dispositivo con root cualquiera puede leerlo. Nunca guardes contraseñas ni tokens sensibles sin cifrado. Para sesiones, lo correcto es guardar un token de sesión (que el servidor puede invalidar) y no la contraseña misma. En aplicaciones reales se usa la biblioteca `EncryptedSharedPreferences` de `AndroidX` para datos sensibles, que cifra automáticamente los valores.

Las sesiones de usuario merecen atención especial porque son el caso de uso más pedido. El patrón estándar es: cuando el usuario inicia sesión correctamente, guardas un booleano "sesion_activa" en true y el nombre o identificador del usuario. En la pantalla de arranque (splash) de la app, lees ese booleano: si es true, vas directo a la pantalla principal; si es false, muestras el login. Cuando el usuario cierra sesión, pones el booleano en false y borras los datos de la sesión. Simple, efectivo, y es exactamente lo que hacen miles de apps en producción.

Buenas prácticas con SharedPreferences: define las claves como constantes para evitar errores de escritura (un typo en una clave significa leer un valor que nunca se escribió); agrupa la lógica de lectura y escritura en una clase de ayuda, por ejemplo PreferenciasManager o SesionManager, en lugar de esparcir getSharedPreferences por todas las Activities; y no abuses de apply() en bucles masivos, mejor acumula todos los put en un solo editor y aplica una vez.

Clase 20: SQLite y SQLiteOpenHelper

SQLite es un motor de base de datos relacional completo que vive dentro de cada dispositivo Android. No necesita servidor, no necesita instalación, no necesita configuración de red: es simplemente una biblioteca que lee y escribe un archivo en el disco. A pesar de su tamaño diminuto, SQLite soporta SQL estándar casi completo: crear tablas, índices, consultas con JOIN, transacciones, etc. Es la base de datos más usada del mundo precisamente porque está en cada teléfono, cada navegador y cada sistema embebido.

En Android trabajas con SQLite a través de la clase SQLiteOpenHelper. Esta clase hace dos trabajos: primero, gestiona la creación de la base de datos y sus versiones (el famoso onCreate y onUpgrade que veremos en el ejemplo); segundo, te da objetos SQLiteDatabase para leer y escribir, con getWritableDatabase() para escritura y getReadableDatabase() para lectura.

El concepto central es la tabla. Una tabla tiene un nombre y un conjunto de columnas, cada columna con un tipo de dato. SQLite usa tipos dinámicos, pero en la práctica declaras INTEGER, TEXT, REAL y BLOB. Cada tabla debería tener una columna de identificador único, por convención llamada _id o id, de tipo INTEGER PRIMARY KEY AUTOINCREMENT. Esa columna es tu llave maestra: cada registro tiene un número único que nunca se repite y que usarás para actualizar o borrar ese registro específico.

El CRUD son las cuatro operaciones:

- Create: insertar un registro nuevo con el método `insert()`. Le pasas el nombre de la tabla y un `ContentValues`, que es un contenedor de pares columna-valor. Te devuelve el id del registro creado, o -1 si falló.
- Read: consultar con el método `query()`. Le pasas la tabla, las columnas que quieres, la condición WHERE, los argumentos de la condición, y el ordenamiento. Te devuelve un `Cursor`, que es como un apuntador que recorre las filas resultado una por una.
- Update: actualizar con el método `update()`. Le pasas la tabla, un `ContentValues` con los nuevos valores, la condición WHERE que identifica qué filas cambiar, y sus argumentos. Te devuelve cuántas filas se actualizaron.
- Delete: borrar con el método `delete()`. Le pasas la tabla, la condición WHERE y sus argumentos. Te devuelve cuántas filas se borraron.

El `Cursor` merece explicación porque es donde más se atorán los principiantes. Un `Cursor` no es una lista: es un iterador posicionado antes de la primera fila. Debes llamar `moveToFirst()` para ir a la primera fila (devuelve false si no hay resultados), luego leer columnas con `getString()`, `getInt()`, etc., usando el índice de la columna (que obtienes con `getColumnIndexOrThrow(nombre)`), y avanzar con `moveToNext()` hasta que devuelva false. Y lo más importante: siempre cierra el `Cursor` con `close()` cuando termines, porque un `Cursor` abierto retiene recursos de la base de datos. El olvido del `close()` es una de las fugas de recursos más comunes en código Android antiguo.

La seguridad en las consultas es otro punto crítico. Nunca construyas una condición WHERE concatenando texto del usuario, como `"nombre = " + nombre + ""`. Eso abre la puerta a inyección SQL: un usuario malicioso podría escribir en el campo de búsqueda algo que altere tu consulta y lea o borre datos ajenos. La forma correcta es usar argumentos con el marcador `?`: `"nombre = ?"` y pasar los valores en el arreglo de `selectionArgs`. SQLite se encarga de escaparlos de forma segura. Esta regla no es opcional: es la diferencia entre una app segura y una vulnerable.

El versionado de la base de datos es el tema que más dolores de cabeza causa cuando se ignora. `SQLiteOpenHelper` recibe un número de versión. Cuando instalas la app por primera vez, se ejecuta `onCreate` y se crean las tablas. Si después publicas una actualización que necesita una columna nueva, aumentas el número de versión y escribes en `onUpgrade` el código que migra la tabla vieja a la nueva (por ejemplo, un `ALTER TABLE`). Si no implementas `onUpgrade` correctamente, los usuarios que actualicen perderán datos o la app tronará. Regla: cada cambio de esquema implica subir la versión y escribir la migración.

Errores comunes de principiantes con SQLite: olvidar cerrar la base de datos o el `Cursor`; hacer operaciones de base de datos en el hilo principal con tablas grandes (la UI se congela);

no usar transacciones cuando insertan muchos registros en un ciclo (cada insert sin transacción es lento; envolver en transacción lo acelera decenas de veces); y el clásico de guardar la base de datos con un nombre distinto en dos partes del código, creando dos archivos y preguntándose por qué los datos "desaparecen".

Buenas prácticas: crea una clase de contrato que defina los nombres de tablas y columnas como constantes (así evitas typos en strings SQL); centraliza todo el acceso a datos en una clase de ayuda o repositorio en lugar de esparcir SQL por las Activities; usa transacciones para operaciones múltiples; y valida siempre que el Cursor tenga datos con moveToFirst() antes de leer.

Clase 21: Room, la forma moderna

Room es una biblioteca de persistencia que forma parte de Android Jetpack. No es una base de datos distinta: por debajo usa SQLite, el mismo motor de siempre. Lo que Room hace es generar por ti todo el código repetitivo y propenso a errores que acabas de ver: ya no escribes strings SQL a mano para las operaciones comunes, ya no manipulas Cursores, ya no conviertes filas en objetos manualmente. Tú defines el modelo con anotaciones y Room genera la implementación.

Room se organiza en tres componentes:

1. Entity (entidad): una data class de Kotlin anotada con @Entity que representa una tabla. Cada propiedad es una columna. Con @PrimaryKey marcas la llave primaria.
2. DAO (Data Access Object): una interfaz anotada con @Dao donde declaras métodos anotados con @Insert, @Update, @Delete y @Query. En los @Query escribes SQL, pero Room lo verifica en tiempo de compilación: si el nombre de una tabla o columna está mal escrito, el proyecto no compila en lugar de tronar en ejecución. Esa verificación es oro.
3. Database: una clase abstracta anotada con @Database que lista las entidades y los DAOs, con un número de versión. Es el punto de entrada.

¿Por qué Google recomienda Room sobre SQLite crudo? Tres razones de peso. Primera: seguridad en tiempo de compilación, como ya dijimos. Segunda: integración con el resto de Jetpack, especialmente con LiveData, Flow y corrutinas, lo que permite observar cambios en la base de datos de forma reactiva (la UI se actualiza sola cuando los datos cambian). Tercera: migraciones estructuradas y soporte para pruebas. En resumen: menos código, menos errores, más potencia.

La analogía: si SQLite crudo es cocinar desde cero picando cada verdura, Room es tener un sous-chef que pica, mide y prepara todo siguiendo tu receta. Tú sigues siendo el chef (defines el menú, es decir, el esquema), pero el trabajo mecánico y repetitivo desaparece.

Un detalle importante para tu carrera: aunque Room sea lo recomendado, aprender SQLite primero no es tiempo perdido. Primero, porque Room genera SQLite por debajo y entender qué pasa te hace mejor depurando. Segundo, porque muchísimo código existente y muchísimas entrevistas siguen hablando de SQLiteOpenHelper. Tercero, porque los conceptos (tablas, llaves primarias, CRUD, WHERE, migraciones) son idénticos; solo cambia la sintaxis. Piensa en SQLite como aprender a manejar estándar y Room como el automático: el que sabe estándar maneja ambos.

Errores comunes al empezar con Room: olvidar agregar las dependencias correctas en el build.gradle (Room necesita la biblioteca runtime, el compilador de anotaciones ksp o kapt, y opcionalmente las extensiones de corrutinas); declarar la base de datos como clase instanciable en lugar de abstracta; intentar hacer operaciones de base de datos en el hilo principal (Room directamente lanza una excepción si lo intentas, lo cual es una ayuda disfrazada); y no manejar las migraciones al cambiar la versión, igual que con SQLite.

Buenas prácticas con Room: usa el patrón singleton para la instancia de la base de datos (una sola instancia en toda la app); nombra las entidades y columnas con claridad; usa suspend functions en el DAO combinadas con corrutinas para operaciones asíncronas; y para desarrollo rápido puedes usar fallbackToDestructiveMigration(), pero jamás en producción, porque borra los datos del usuario al migrar.

Al final de esta unidad tendrás el mapa completo: preferencias para lo pequeño, SQLite para entender el fundamento, Room para construir en serio. Esa es exactamente la progresión que sigue un desarrollador profesional.

Profundizando: transacciones, relaciones e índices

Hay tres conceptos de bases de datos que separan al principiante del desarrollador competente: las transacciones, las relaciones entre tablas y los índices. Vamos a verlos con calma porque aparecen en entrevistas y en código real todos los días.

Una transacción es un conjunto de operaciones que se ejecutan como una sola unidad: o todas tienen éxito, o ninguna se aplica. Imagina una transferencia bancaria: quitar 500 pesos de la cuenta A y poner 500 en la cuenta B. Si el teléfono se apaga justo después de quitar el dinero pero antes de ponerlo, esos 500 pesos se esfumarían. Con una transacción, si algo falla a la mitad, la base de datos revierte todo al estado anterior, como si nada hubiera pasado. En SQLite usas beginTransaction(), luego tus operaciones, luego setTransactionSuccessful()

para confirmar, y `endTransaction()` en un bloque `finally` para cerrar. En el ejemplo paso a paso verás este patrón completo. Regla práctica: siempre que hagas dos o más escrituras que dependen entre sí, úsalas dentro de una transacción. Además, las transacciones aceleran enormemente las inserciones masivas: insertar 1000 registros uno por uno puede tardar varios segundos, mientras que dentro de una transacción tarda una fracción de segundo, porque SQLite escribe al disco una sola vez al confirmar.

Las relaciones entre tablas son cómo conectas información. Ejemplo clásico: una app de biblioteca tiene una tabla de autores y una tabla de libros. Cada libro pertenece a un autor. En lugar de repetir el nombre del autor en cada fila de libro (desperdicio y fuente de inconsistencias), guardas en la tabla de libros una columna `autor_id` que contiene el id del autor en la otra tabla. Esa columna es una llave foránea (foreign key). Con un `JOIN` puedes luego consultar "dame todos los libros con el nombre de su autor" en una sola consulta. Los principiantes suelen meter todo en una sola tabla gigante con columnas repetidas; funciona con 10 registros y se vuelve un pantano con 10,000. Aprender a normalizar (separar en tablas relacionadas) es una de las habilidades que más se notan en la calidad de tu código.

Los índices aceleran las búsquedas. Sin índice, buscar un contacto por nombre obliga a SQLite a revisar fila por fila (búsqueda lineal). Con un índice sobre la columna nombre, SQLite salta directo a las filas correctas. La analogía es el índice alfabético al final de un libro: sin él, buscar un tema es hojear página por página; con él, vas directo. Se crean con `CREATE INDEX`. El costo es que cada escritura se vuelve un poco más lenta (hay que actualizar el índice) y ocupa espacio, así que se ponen índices en las columnas por las que buscas u ordenas frecuentemente, no en todas.

Tipos de datos en SQLite y sus trampas

SQLite tiene una particularidad famosa: su sistema de tipos es dinámico y flexible, más que el de otras bases de datos. Las afinidades de tipo que usarás son:

- **INTEGER**: números enteros. Para ids, cantidades, edades, timestamps.
- **TEXT**: cadenas de texto. Nombres, descripciones, correos.
- **REAL**: números con decimales. Precios, promedios, coordenadas.
- **BLOB**: datos binarios. Imágenes pequeñas, archivos. Úsalo con moderación: guardar fotos grandes en la base de datos la hace lenta y pesada; lo profesional es guardar la imagen como archivo y en la base solo la ruta.
- **NUMERIC**: para valores numéricos flexibles, poco usado en apps comunes.

La trampa clásica: SQLite te deja guardar un texto en una columna declarada INTEGER sin quejarse. Eso no significa que debas hacerlo. Sé disciplinado con los tipos aunque el motor sea permisivo, porque Room y tu propio código asumirán los tipos declarados.

Otro detalle: SQLite no tiene un tipo BOOLEAN nativo. La convención es usar INTEGER con 0 para falso y 1 para verdadero. Tampoco tiene tipo DATE: las fechas se guardan como TEXT en formato ISO ("2026-09-19") o como INTEGER con el timestamp en milisegundos. El formato ISO como texto tiene la ventaja de que ordena cronológicamente con un simple ORDER BY, así que es el favorito para fechas legibles.

El ciclo de vida de la base de datos en la app real

Vamos a conectar los puntos con el ciclo de vida real de una aplicación publicada:

1. El usuario instala la versión 1.0. Al primer acceso a datos, SQLiteOpenHelper ejecuta onCreate: se crean las tablas vacías. Versión = 1.
2. Meses después publicas la versión 2.0 que necesita una columna nueva "teléfono" en la tabla contactos. Subes la versión a 2 y escribes en onUpgrade: si la versión vieja es 1, ejecuta ALTER TABLE contactos ADD COLUMN teléfono TEXT.
3. Un usuario actualiza de 1.0 a 2.0: Android detecta que la versión guardada (1) es menor que la pedida (2) y ejecuta onUpgrade. Sus contactos se conservan y la tabla gana la columna nueva.
4. Un usuario nuevo instala directamente la 2.0: se ejecuta onCreate, que ya debe crear el esquema completo actualizado (con la columna teléfono incluida).

El error mortal aquí es escribir onUpgrade pero olvidar actualizar onCreate con el esquema nuevo. Entonces los usuarios nuevos obtienen tablas viejas y todo lo que asuma la columna nueva truena. Regla: onCreate siempre refleja el esquema más reciente; onUpgrade contiene los pasos incrementales para quienes vienen de versiones anteriores.

En Room el concepto es idéntico pero con clases Migration en lugar de código suelto en onUpgrade. Cada migración declara de qué versión a qué versión va y ejecuta el SQL necesario. Room además puede validar el esquema para asegurarse de que la migración dejó la base cómo se esperaba.

Cuándo usar cada tecnología: la guía de decisión

Memoriza esta guía porque la usarás en cada proyecto:

- Usa SharedPreferences cuando: el dato es un valor simple (texto, número, booleano); son pocos valores (decenas, no miles); no necesitas buscar, filtrar ni ordenar; ejemplos: nombre de usuario, tema oscuro, última pantalla visitada, bandera de "ya vio el tutorial", token de sesión.
- Usa SQLite/Room cuando: tienes colecciones de elementos del mismo tipo; necesitas buscar por criterios, ordenar o filtrar; los datos se relacionan entre sí; la cantidad puede crecer sin límite claro; ejemplos: contactos, tareas, notas, productos, mensajes, registros de ejercicio.
- Usa archivos cuando: el dato es un documento completo, una imagen, un audio, un PDF; ejemplos: foto de perfil, nota de voz, archivo descargado.
- Usa la nube cuando: el dato debe estar disponible en varios dispositivos o sobrevivir a la desinstalación; ejemplos: respaldo de conversaciones, perfil público, compras.

Un antipatrón muy común es el "SharedPreferences gigante": guardar listas serializadas en JSON dentro de preferencias porque "era más rápido que hacer una base de datos". Al mes, ese código es imposible de mantener: no puedes buscar un elemento sin deserializar todo, cualquier cambio de formato rompe datos viejos, y el archivo crece sin control. Si te descubres serializando colecciones en preferencias, es señal de que necesitas una base de datos. Y al revés: crear una tabla SQL para guardar un simple booleano de "modo oscuro" es usar un cañón para matar una mosca.

Seguridad básica de datos locales

Cierra esta sección con seguridad, porque los datos persistentes son datos que un atacante puede intentar leer:

1. Todo lo que guardes en el almacenamiento interno privado solo lo puede leer tu app... en un dispositivo normal. En un dispositivo con root, cualquiera con acceso puede leerlo. Asume que un atacante motivado puede leer tus archivos locales.
2. Nunca guardes contraseñas en texto plano, ni en SharedPreferences ni en SQLite. Guarda tokens de sesión que el servidor pueda revocar, o usa EncryptedSharedPreferences para valores sensibles.
3. No guardes en la base de datos local información que no necesites. Si un dato sensible no es imprescindible offline, no lo persistas.
4. Cuidado con los respaldos: por defecto, Android incluye los datos de tu app en el respaldo automático a la nube. Si guardas datos sensibles, considera excluirlos del respaldo con las reglas correspondientes.

5. La inyección SQL que mencionamos antes: usa siempre `selectionArgs`, nunca concatenes entrada del usuario en el SQL.

Con esto tienes la teoría completa. Ahora vamos a ensuciarnos las manos con código real, paso a paso, como debe ser.

Ejemplos paso a paso

Ejemplo 1: `SharedPreferences` para ajustes de la app

Empezamos con lo más sencillo: una pantalla de ajustes donde el usuario elige su nombre, activa o desactiva el modo oscuro y elige cuántos elementos mostrar por página. Todo se guarda en `SharedPreferences` y sobrevive al reinicio de la app.

Primero, creamos una clase que centraliza todo el acceso a las preferencias. Esta es la buena práctica que mencionamos: una sola clase, claves como constantes, nadie más toca `SharedPreferences` directamente.

```

class AjustesManager(context: Context) {

    companion object {
        private const val ARCHIVO = "ajustes_app"
        private const val CLAVE_NOMBRE = "nombre_usuario"
        private const val CLAVE_MODO_OSCURO = "modo_oscuro"
        private const val CLAVE_ELEMENTOS_POR_PAGINA = "elementos_por_pagina"
    }

    private val prefs: SharedPreferences =
        context.getSharedPreferences(ARCHIVO, Context.MODE_PRIVATE)

    fun guardarNombre(nombre: String) {
        prefs.edit().putString(CLAVE_NOMBRE, nombre).apply()
    }

    fun obtenerNombre(): String {
        return prefs.getString(CLAVE_NOMBRE, "Invitado") ?: "Invitado"
    }

    fun guardarModoOscuro(activo: Boolean) {
        prefs.edit().putBoolean(CLAVE_MODO_OSCURO, activo).apply()
    }

    fun esModoOscuro(): Boolean {
        return prefs.getBoolean(CLAVE_MODO_OSCURO, false)
    }

    fun guardarElementosPorPagina(cantidad: Int) {
        prefs.edit().putInt(CLAVE_ELEMENTOS_POR_PAGINA, cantidad).apply()
    }

    fun obtenerElementosPorPagina(): Int {
        return prefs.getInt(CLAVE_ELEMENTOS_POR_PAGINA, 20)
    }
}

```

Explicación línea por línea:

- ``class AjustesManager(context: Context)``: la clase recibe un Context porque `getSharedPreferences` lo necesita. Recibirlo en el constructor evita pasarlo en cada método.

- El bloque ``companion object`` guarda las constantes: el nombre del archivo y cada clave. Si mañana cambias una clave, la cambias en un solo lugar.
- ``context.getSharedPreferences(ARCHIVO, Context.MODE_PRIVATE)``: abre (o crea) el archivo "ajustes_app" en modo privado. Se guarda una sola vez como propiedad para reutilizarlo.
- ``prefs.edit().putString(CLAVE_NOMBRE, nombre).apply()``: el patrón de escritura en una línea. `edit()` abre el editor, `putString()` pone el valor, `apply()` lo guarda en segundo plano sin bloquear.
- ``prefs.getString(CLAVE_NOMBRE, "Invitado") ?: "Invitado"``: lee el nombre; si la clave no existe devuelve "Invitado". El operador Elvis ``?:`` protege contra el null que `getString` puede devolver.
- Los métodos de modo oscuro y elementos por página siguen el mismo patrón con `putBoolean/getBoolean` y `putInt/getInt`. Observa los valores por defecto: `false` para el modo oscuro (empieza en claro) y 20 como paginación razonable.

Ahora usamos esta clase en una Activity de ajustes:

```

class AjustesActivity : AppCompatActivity() {

    private lateinit var ajustes: AjustesManager

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

        ajustes = AjustesManager(this)

        val campoNombre = findViewById<EditText>(R.id.campoNombre)
        val switchOscuro = findViewById<Switch>(R.id.switchOscuro)
        val botonGuardar = findViewById<Button>(R.id.botonGuardar)

        // Cargar los valores guardados al abrir la pantalla
        campoNombre.setText(ajustes.obtenerNombre())
        switchOscuro.isChecked = ajustes.esModoOscuro()

        botonGuardar.setOnClickListener {
            ajustes.guardarNombre(campoNombre.text.toString())
            ajustes.guardarModoOscuro(switchOscuro.isChecked)
            Toast.makeText(this, "Ajustes guardados", Toast.LENGTH_SHORT).show()
        }
    }
}

```

Al abrir la pantalla, leemos los valores guardados y los mostramos; al pulsar guardar, los escribimos. Cierra la app, vuelve a abrirla: los valores siguen ahí. Eso es persistencia.

Ejemplo 2: Gestor de sesión de usuario con SharedPreferences

Este es el patrón de login que usan miles de apps. La idea: al iniciar sesión correctamente, guardamos una bandera y los datos básicos del usuario. Una pantalla de arranque decide a dónde llevar al usuario según esa bandera.

```

class SesiónManager(context: Context) {

    companion object {
        private const val ARCHIVO = "sesión_usuario"
        private const val CLAVE_ACTIVA = "sesión_activa"
        private const val CLAVE_NOMBRE = "usuario_nombre"
        private const val CLAVE_CORREO = "usuario_correo"
        private const val CLAVE_TOKEN = "usuario_token"
    }

    private val prefs =
        context.getSharedPreferences(ARCHIVO, Context.MODE_PRIVATE)

    fun iniciarSesión(nombre: String, correo: String, token: String) {
        prefs.edit()
            .putBoolean(CLAVE_ACTIVA, true)
            .putString(CLAVE_NOMBRE, nombre)
            .putString(CLAVE_CORREO, correo)
            .putString(CLAVE_TOKEN, token)
            .apply()
    }

    fun cerrarSesión() {
        prefs.edit().clear().apply()
    }

    fun haySesiónActiva(): Boolean {
        return prefs.getBoolean(CLAVE_ACTIVA, false)
    }

    fun obtenerNombre(): String? {
        return prefs.getString(CLAVE_NOMBRE, null)
    }

    fun obtenerToken(): String? {
        return prefs.getString(CLAVE_TOKEN, null)
    }
}

```

Explicación de lo nuevo:

- `iniciarSesión` encadena varios put en el mismo editor y hace un solo apply() al final. Esto es más eficiente que cuatro editores separados y garantiza que todo se guarde junto.

- Guardamos un token, no la contraseña. En una app real ese token lo genera tu servidor al validar el login; aquí lo simulamos.
- `cerrarSesion()` usa clear(), que borra todas las claves del archivo de una vez. Es la forma limpia de "olvidar" al usuario.
- `haySesionActiva()` devuelve false por defecto: si nunca hubo login, no hay sesión. Simple y seguro.

Y la pantalla de arranque que decide el destino:

```
class SplashActivity : AppCompatActivity() {

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

        val sesion = SesionManager(this)

        val destino = if (sesion.haySesionActiva()) {
            Intent(this, PrincipalActivity::class.java)
        } else {
            Intent(this, LoginActivity::class.java)
        }
        startActivity(destino)
        finish()
    }
}
```

Sin layouts complicados: la SplashActivity solo decide y redirige. El usuario que ya inició sesión jamás vuelve a ver el login hasta que cierre sesión explícitamente. En el LoginActivity, tras validar las credenciales, llamarías a sesion.iniciarSesion(nombre, correo, token) y navegarías a la pantalla principal.

Nota de seguridad: este ejemplo guarda el token en texto plano para mantenerlo didáctico. En una app que publiques, usa EncryptedSharedPreferences de AndroidX para el archivo de sesión. El patrón es el mismo, solo cambia cómo se crea el objeto prefs.

Ejemplo 3: SQLiteOpenHelper completo con CRUD (parte 1: contrato y helper)

Vamos a construir una agenda de contactos con SQLite desde cero. Este es el ejemplo central de la unidad: un SQLiteOpenHelper completo con las cuatro operaciones CRUD, escrito con buenas prácticas. Lo dividimos en dos partes para digerirlo bien.

Primero el contrato: una clase que define los nombres de la tabla y las columnas como constantes. Esto elimina los errores de escritura en los strings SQL.

```
object ContactoContrato {  
  
    const val NOMBRE_TABLA = "contactos"  
  
    const val COL_ID = "_id"  
    const val COL_NOMBRE = "nombre"  
    const val COL_TELEFONO = "telefono"  
    const val COL_CORREO = "correo"  
    const val COL_FAVORITO = "favorito"  
  
}
```

Simple pero poderoso: cada vez que necesites el nombre de una columna en un SQL, un query o un ContentValues, usas la constante. Si un día renombras una columna, la cambias aquí y todo el código sigue funcionando.

Ahora la data class que representa un contacto en Kotlin. Es el puente entre el mundo de objetos y el mundo de filas:

```
data class Contacto(  
    val id: Long = 0,  
    val nombre: String,  
    val telefono: String,  
    val correo: String,  
    val favorito: Boolean = false  
)
```

Y ahora el SQLiteOpenHelper. Lee con calma cada método:

```

class AgendaDbHelper(context: Context) :
    SQLiteOpenHelper(context, NOMBRE_BD, null, VERSION_BD) {

    companion object {
        private const val NOMBRE_BD = "agenda.db"
        private const val VERSION_BD = 1

        private const val SQL_CREAR_TABLA =
            "CREATE TABLE ${ContactoContrato.NOMBRE_TABLA} (" +
            "${ContactoContrato.COL_ID} INTEGER PRIMARY KEY AUTOINCREMENT, " +
            "${ContactoContrato.COL_NOMBRE} TEXT NOT NULL, " +
            "${ContactoContrato.COL_TELEFONO} TEXT NOT NULL, " +
            "${ContactoContrato.COL_CORREO} TEXT, " +
            "${ContactoContrato.COL_FAVORITO} INTEGER NOT NULL DEFAULT 0)"
    }

    override fun onCreate(db: SQLiteDatabase) {
        db.execSQL(SQL_CREAR_TABLA)
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        // Versión 1: no hay migraciones todavía.
        // Cuando agregues la versión 2, escribe aquí el ALTER TABLE
        correspondiente.
    }
}

```

Explicación línea por línea:

- ``class AgendaDbHelper(context: Context) : SQLiteOpenHelper(context, NOMBRE_BD, null, VERSION_BD)``: heredamos de `SQLiteOpenHelper` pasándole el nombre del archivo ("agenda.db"), un factory de cursores null (casi nunca se usa) y la versión 1.
- ``NOMBRE_BD`` y ``VERSION_BD`` como constantes privadas: el nombre del archivo y la versión actual del esquema.
- ``SQL_CREAR_TABLA``: la sentencia `CREATE TABLE`. Observa cada columna: `_id` es `INTEGER PRIMARY KEY AUTOINCREMENT` (identificador único automático); nombre y teléfono son `TEXT NOT NULL` (obligatorios); correo es `TEXT` sin `NOT NULL` (opcional, puede quedar vacío); favorito es `INTEGER NOT NULL DEFAULT 0` (guardamos el booleano como 0/1, con 0 por defecto). Usamos las constantes del contrato interpoladas con `${}` para no escribir strings a mano.

- ``onCreate(db)``: se ejecuta una sola vez, cuando la base de datos se crea. Aquí va el esquema inicial completo.
- ``onUpgrade(db, oldVersion, newVersion)``: se ejecuta cuando la versión pedida es mayor que la guardada. Por ahora está vacío porque estamos en la versión 1, pero dejamos el comentario como recordatorio de dónde irán las migraciones futuras.

Antes de seguir, un ejemplo de migración para que veas cómo se vería la versión 2. Imagina que después quieres agregar una columna "notas":

```
override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
    if (oldVersion < 2) {
        db.execSQL("ALTER TABLE ${ContactoContrato.NOMBRE_TABLA} " +
            "ADD COLUMN notas TEXT")
    }
}
```

Y subirías `VERSION_BD` a 2. Los usuarios de la versión 1 conservan sus contactos y ganan la columna nueva; los usuarios nuevos obtienen el esquema completo desde `onCreate` (que también deberías actualizar para incluir "notas"). Este es el ciclo de versionado del que hablamos en la teoría.

Ejemplo 3 (continuación): las operaciones CRUD

Ahora agregamos a `AgendaDbHelper` los cuatro métodos CRUD. Cada uno sigue el patrón oficial de Android.

```
// CREATE: inserta un contacto nuevo, devuelve su id o -1 si falló
fun insertar(contacto: Contacto): Long {
    val db = writableDatabase
    val valores = ContentValues().apply {
        put(ContactoContrato.COL_NOMBRE, contacto.nombre)
        put(ContactoContrato.COL_TELEFONO, contacto.telefono)
        put(ContactoContrato.COL_CORREO, contacto.correo)
        put(ContactoContrato.COL_FAVORITO, if (contacto.favorito) 1 else 0)
    }
    val id = db.insert(ContactoContrato.NOMBRE_TABLA, null, valores)
    db.close()
    return id
}
```

- ``writableDatabase``: abre la base en modo escritura (la crea si no existe).

- ``ContentValues().apply { ... }``: el contenedor de pares columna-valor. `put()` con la constante de columna y el valor del objeto. El booleano se convierte a 1 o 0 porque SQLite no tiene booleanos.
- ``db.insert(tabla, null, valores)``: inserta la fila. El segundo parámetro (`nullColumnHack`) casi siempre va en `null`. Devuelve el id generado o -1 si algo falló.
- ``db.close()``: liberamos la conexión. En apps pequeñas este patrón es aceptable; en apps grandes se suele mantener una sola instancia abierta.

```
// READ: obtiene todos los contactos ordenados por nombre
fun obtenerTodos(): List<Contacto> {
    val lista = mutableListOf<Contacto>()
    val db = readableDatabase

    val cursor = db.query(
        ContactoContrato.NOMBRE_TABLA,
        null,          // columnas: null = todas
        null,          // selection (WHERE): null = sin filtro
        null,          // selectionArgs
        null,          // groupBy
        null,          // having
        "${ContactoContrato.COL_NOMBRE} ASC" // orden: nombre ascendente
    )

    cursor.use {
        while (it.moveToNext()) {
            val contacto = Contacto(
                id =
it.getLong(it.getColumnIndexOrThrow(ContactoContrato.COL_ID)),
                nombre =
it.getString(it.getColumnIndexOrThrow(ContactoContrato.COL_NOMBRE)),
                telefono =
it.getString(it.getColumnIndexOrThrow(ContactoContrato.COL_TELEFONO)),
                correo =
it.getString(it.getColumnIndexOrThrow(ContactoContrato.COL_CORREO)),
                favorito =
it.getInt(it.getColumnIndexOrThrow(ContactoContrato.COL_FAVORITO)) == 1
            )
            lista.add(contacto)
        }
    }
    db.close()
    return lista
}
```

- ``db.query(...)``: la consulta estructurada. Con null en columnas traemos todas; con null en selection traemos todas las filas; el último parámetro ordena por nombre ascendente.
- ``cursor.use { ... }``: la forma moderna y segura de trabajar con Cursor. `use()` lo cierra automáticamente al terminar el bloque, incluso si hay una excepción. Ya no hay riesgo de olvidar el `close()`.
- ``moveToNext()``: avanza fila por fila; devuelve false al final. No necesitamos `moveToFirst()` porque el while maneja el recorrido completo.
- ``getColumnIndexOrThrow(COL)``: obtiene el índice de la columna por nombre. La versión `OrThrow` lanza excepción si la columna no existe, lo cual es preferible a la versión silenciosa que devuelve -1 y luego truena en un lugar confuso.
- Convertimos cada fila en un objeto Contacto, reconvirtiendo el 0/1 a booleano con ``== 1``.

```

// READ con filtro: busca contactos por nombre (búsqueda segura)
fun buscarPorNombre(texto: String): List<Contacto> {
    val lista = mutableListOf<Contacto>()
    val db = readableDatabase

    val cursor = db.query(
        ContactoContrato.NOMBRE_TABLA,
        null,
        "${ContactoContrato.COL_NOMBRE} LIKE ?",
        arrayOf("%$texto%"),
        null, null, null
    )

    cursor.use {
        while (it.moveToNext()) {
            lista.add(
                Contacto(
                    id =
it.getLong(it.getColumnIndexOrThrow(ContactoContrato.COL_ID)),
                    nombre =
it.getString(it.getColumnIndexOrThrow(ContactoContrato.COL_NOMBRE)),
                    telefono =
it.getString(it.getColumnIndexOrThrow(ContactoContrato.COL_TELEFONO)),
                    correo =
it.getString(it.getColumnIndexOrThrow(ContactoContrato.COL_CORREO)),
                    favorito =
it.getInt(it.getColumnIndexOrThrow(ContactoContrato.COL_FAVORITO)) == 1
                )
            )
        }
    }
    db.close()
    return lista
}

```

- El WHERE es ``nombre LIKE ?`` y el valor va en selectionArgs como ``"%\$texto%"``. Los porcentajes son comodines: encuentra el texto en cualquier posición del nombre. Lo crucial: el texto del usuario viaja como argumento, jamás concatenado al SQL. Así evitas la inyección SQL.

```

// UPDATE: actualiza un contacto existente, devuelve filas afectadas
fun actualizar(contacto: Contacto): Int {
    val db = writableDatabase
    val valores = ContentValues().apply {
        put(ContactoContrato.COL_NOMBRE, contacto.nombre)
        put(ContactoContrato.COL_TELEFONO, contacto.telefono)
        put(ContactoContrato.COL_CORREO, contacto.correo)
        put(ContactoContrato.COL_FAVORITO, if (contacto.favorito) 1 else 0)
    }
    val filas = db.update(
        ContactoContrato.NOMBRE_TABLA,
        valores,
        "${ContactoContrato.COL_ID} = ?",
        arrayOf(contacto.id.toString())
    )
    db.close()
    return filas
}

// DELETE: borra un contacto por su id, devuelve filas afectadas
fun eliminar(id: Long): Int {
    val db = writableDatabase
    val filas = db.delete(
        ContactoContrato.NOMBRE_TABLA,
        "${ContactoContrato.COL_ID} = ?",
        arrayOf(id.toString())
    )
    db.close()
    return filas
}

```

- En update y delete, el WHERE identifica la fila por su `\_id` con el marcador `?` y el id como argumento. Esto garantiza que solo se toca el registro exacto.
- Ambos devuelven el número de filas afectadas: si actualizar devuelve 0, ese id no existía. Úsalo para validar.

Y un método extra muy útil: inserción masiva con transacción.

```

// Inserta varios contactos en una sola transacción (mucho más rápido)
fun insertarVarios(contactos: List<Contacto>) {
    val db = writableDatabase
    db.beginTransaction()
    try {
        for (c in contactos) {
            val valores = ContentValues().apply {
                put(ContactoContrato.COL_NOMBRE, c.nombre)
                put(ContactoContrato.COL_TELEFONO, c.telefono)
                put(ContactoContrato.COL_CORREO, c.correo)
                put(ContactoContrato.COL_FAVORITO, if (c.favorito) 1 else 0)
            }
            db.insert(ContactoContrato.NOMBRE_TABLA, null, valores)
        }
        db.setTransactionSuccessful()
    } finally {
        db.endTransaction()
    }
    db.close()
}

```

- `beginTransaction()` abre la transacción; si todo el ciclo termina sin excepciones, `setTransactionSuccessful()` marca que se puede confirmar; `endTransaction()` en el finally confirma o revierte. Si algo truena a la mitad, ninguna fila parcial queda guardada. Además, como dijimos en la teoría, esto es decenas de veces más rápido que insertar sin transacción.

Ejemplo 3 (uso): el CRUD en acción desde una Activity

Con el helper completo, usar la base de datos desde la interfaz es directo. Este ejemplo muestra el ciclo completo: insertar, leer, actualizar y borrar.

```

class AgendaActivity : AppCompatActivity() {

    private lateinit var dbHelper: AgendaDbHelper

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

        dbHelper = AgendaDbHelper(this)

        // CREATE: agregar un contacto nuevo
        val nuevoId = dbHelper.insertar(

```



```

        Contacto(
            nombre = "María González",
            telefono = "5551234567",
            correo = "maria@ejemplo.com",
            favorito = true
        )
    )
    Log.d("Agenda", "Contacto insertado con id: $nuevoId")

    // READ: leer todos y mostrarlos en el log
    val todos = dbHelper.obtenerTodos()
    for (c in todos) {
        Log.d("Agenda", "${c.id}: ${c.nombre} - ${c.telefono}")
    }

    // UPDATE: cambiar el teléfono del contacto recién creado
    val filasActualizadas = dbHelper.actualizar(
        Contacto(
            id = nuevoId,
            nombre = "María González",
            telefono = "5559998888",
            correo = "maria@ejemplo.com",
            favorito = true
        )
    )
    Log.d("Agenda", "Filas actualizadas: $filasActualizadas")

    // READ con filtro: buscar por nombre
    val resultados = dbHelper.buscarPorNombre("María")
    Log.d("Agenda", "Coincidencias: ${resultados.size}")

    // DELETE: borrar el contacto (descomenta para probar)
    // val filasBorradas = dbHelper.eliminar(nuevoId)
    // Log.d("Agenda", "Filas borradas: $filasBorradas")
}

override fun onDestroy() {
    dbHelper.close()
    super.onDestroy()
}
}

```

Puntos a observar: cada operación devuelve información útil (el id nuevo, la lista, el conteo de filas) que puedes usar para actualizar la UI o validar. En una app real, estas llamadas irían en un hilo secundario o corrutina si la tabla puede crecer, y los resultados alimentarían un RecyclerView. El onDestroy cierra el helper como limpieza final.

Si ejecutas esto, abres el Logcat y ves los mensajes, luego cierras la app y la vuelves a abrir, notarás algo importante: el contacto se duplica, porque el onCreate inserta de nuevo cada vez. Eso es esperado en este ejemplo didáctico; en una app real la inserción ocurriría solo cuando el usuario pulse un botón de "agregar".

Ejemplo 4: los mismos contactos, ahora con Room

Vamos a reescribir la agenda con Room para que compares. Primero las dependencias en el build.gradle del módulo app (sintaxis moderna con Kotlin DSL y ksp):

```
dependencies {
    val roomVersion = "2.6.1"
    implementation("androidx.room:room-runtime:$roomVersion")
    implementation("androidx.room:room-ktx:$roomVersion")
    ksp("androidx.room:room-compiler:$roomVersion")
}
```

Necesitas también el plugin ksp en el build.gradle del proyecto. room-runtime es la biblioteca base, room-ktx agrega soporte para corrutinas (las funciones suspend), y room-compiler es el generador de código que se ejecuta al compilar.

Paso 1: la entidad. Es la misma data class Contacto, pero anotada:

```
@Entity(tableName = "contactos")
data class ContactoRoom(
    @PrimaryKey(autoGenerate = true) val id: Long = 0,
    val nombre: String,
    val telefono: String,
    val correo: String,
    val favorito: Boolean = false
)
```

- `@Entity(tableName = "contactos")`: esta clase es una tabla llamada "contactos".
- `@PrimaryKey(autoGenerate = true)`: el id es llave primaria autogenerada, equivalente al AUTOINCREMENT de antes.

- Observa que favorito ahora es Boolean directamente: Room hace la conversión a 0/1 por ti. Ya no hay conversiones manuales.

Paso 2: el DAO, donde declaras qué operaciones necesitas:

```
@Dao
interface ContactoDao {

    @Insert
    suspend fun insertar(contacto: ContactoRoom): Long

    @Update
    suspend fun actualizar(contacto: ContactoRoom): Int

    @Delete
    suspend fun eliminar(contacto: ContactoRoom): Int

    @Query("SELECT * FROM contactos ORDER BY nombre ASC")
    suspend fun obtenerTodos(): List<ContactoRoom>

    @Query("SELECT * FROM contactos WHERE nombre LIKE '%' || :texto || '%'")
    suspend fun buscarPorNombre(texto: String): List<ContactoRoom>

    @Query("SELECT * FROM contactos WHERE favorito = 1")
    suspend fun obtenerFavoritos(): List<ContactoRoom>
}
```

Compara con el SQLiteOpenHelper: las operaciones básicas (@Insert, @Update, @Delete) no llevan ni una línea de SQL, Room las genera. Solo escribes SQL en los @Query, y Room verifica en compilación que la tabla "contactos" y las columnas existan. Si escribes "contacto" en singular por error, el proyecto no compila y te dice exactamente dónde está el problema. Con SQLite crudo, ese error tronaba en ejecución frente al usuario.

Todas las funciones son suspend: se ejecutan en corrutinas, fuera del hilo principal, sin bloquear la UI. Room además te impide por la fuerza hacer consultas en el hilo principal: si lo intentas, lanza una excepción. Es una de esas restricciones que parecen molestas y en realidad te salvan.

El truco del LIKE con parámetros: ``%' || :texto || '%`` concatena los comodines dentro del SQL usando el operador || de SQLite, manteniendo el texto como parámetro seguro. Es la versión Room de lo que hicimos con selectionArgs.

Paso 3: la base de datos que une todo:

```

@Database(entities = [ContactoRoom::class], version = 1)
abstract class AgendaRoomDb : RoomDatabase() {

    abstract fun contactoDao(): ContactoDao

    companion object {
        @Volatile
        private var INSTANCIA: AgendaRoomDb? = null

        fun obtener(context: Context): AgendaRoomDb {
            return INSTANCIA ?: synchronized(this) {
                val nueva = Room.databaseBuilder(
                    context.applicationContext,
                    AgendaRoomDb::class.java,
                    "agenda_room.db"
                ).build()
                INSTANCIA = nueva
                nueva
            }
        }
    }
}

```

- ``@Database(entities = [...], version = 1)``: declara qué entidades contiene y la versión del esquema, igual que el número de versión del SQLiteOpenHelper.
- La clase es abstracta y expone el DAO con una función abstracta; Room genera la implementación.
- El companion object implementa el patrón singleton con doble verificación: una sola instancia de la base de datos para toda la app. ``@Volatile`` garantiza que todos los hilos vean la misma instancia. Crear varias instancias de la base de datos es un error común que causa problemas de concurrencia y desperdicia memoria.

Paso 4: usarlo desde una coroutine:

```

class AgendaRoomActivity : AppCompatActivity() {

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

        val db = AgendaRoomDb.obtener(this)
        val dao = db.contactoDao()

        lifecycleScope.launch {
            val id = dao.insertar(
                ContactoRoom(
                    nombre = "Carlos Ruiz",
                    telefono = "5554443322",
                    correo = "carlos@ejemplo.com",
                    favorito = false
                )
            )
            Log.d("AgendaRoom", "Insertado con id: $id")

            val todos = dao.obtenerTodos()
            for (c in todos) {
                Log.d("AgendaRoom", "${c.id}: ${c.nombre}")
            }
        }
    }
}

```

- ``lifecycleScope.launch { ... }``: lanza una corrutina ligada al ciclo de vida de la Activity. Dentro de ella puedes llamar a las funciones suspend del DAO sin bloquear la interfaz.
- Cuenta las líneas: la misma funcionalidad que el SQLiteOpenHelper en una fracción del código, sin Cursores, sin ContentValues, sin conversiones manuales de booleanos. Esa es la razón por la que la industria migró a Room.

Para cerrar el ejemplo, una migración en Room por si algún día agregas una columna:

```

val MIGRACION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("ALTER TABLE contactos ADD COLUMN notas TEXT")
    }
}

// Al construir la base:
Room.databaseBuilder(...)
    .addMigrations(MIGRACION_1_2)
    .build()

```

Y subirías la versión a 2 en la anotación `@Database`. El concepto es el mismo que en `onUpgrade`, pero organizado en objetos `Migration` reutilizables y verificables.

Práctica

Ejercicio 1: pantalla de bienvenida que solo aparece una vez

Muchas apps muestran un tutorial la primera vez que se abren y nunca más. Implementa ese comportamiento: usa `SharedPreferences` con una clave `"tutorial_visto"`. En la `Activity` principal, si la clave es `false` (valor por defecto), muestra el tutorial y luego guarda `true`. Si ya es `true`, ve directo al contenido.

Solución: la clave está en el valor por defecto. Como la primera vez la clave no existe, `getBoolean("tutorial_visto", false)` devuelve `false` y mostramos el tutorial. Al terminar el tutorial, guardamos `putBoolean("tutorial_visto", true)`. La segunda vez que se abre la app, la lectura devuelve `true` y saltamos el tutorial. El patrón completo:

```

val prefs = getSharedPreferences("app", Context.MODE_PRIVATE)
if (!prefs.getBoolean("tutorial_visto", false)) {
    // Mostrar tutorial
    mostrarTutorial {
        prefs.edit().putBoolean("tutorial_visto", true).apply()
        irAlContenido()
    }
} else {
    irAlContenido()
}

```

El error común aquí es guardar el true antes de que el usuario termine el tutorial (por ejemplo en el onCreate). Si la app se cierra a la mitad del tutorial, el usuario jamás lo verá completo. Guarda la bandera solo cuando el tutorial se completa de verdad.

Ejercicio 2: contador de visitas persistente

Crea una app que muestre "Has abierto esta app N veces". Cada vez que se abre la Activity principal, lee el contador de SharedPreferences, súmale 1, guárdalo y muéstralo en un TextView.

Solución:

```
val prefs = getSharedPreferences("estadisticas", Context.MODE_PRIVATE)
val visitas = prefs.getInt("veces_abierta", 0) + 1
prefs.edit().putInt("veces_abierta", visitas).apply()
textoVisitas.text = "Has abierto esta app $visitas veces"
```

Tres líneas para leer, incrementar y guardar. El valor por defecto 0 maneja la primera apertura. Este ejercicio parece trivial pero enseña el ciclo fundamental de la persistencia: leer, transformar, escribir. Casi todo lo que harás con datos locales es una variación de ese ciclo.

Ejercicio 3: extender la agenda con borrado de no favoritos

Agrega al AgendaDbHelper un método eliminarNoFavoritos() que borre todos los contactos donde favorito = 0, y devuelva cuántos se borraron. Luego escribe el código que lo llama y muestra el resultado en un Toast.

Solución: el método en el helper:

```
fun eliminarNoFavoritos(): Int {
    val db = writableDatabase
    val filas = db.delete(
        ContactoContrato.NOMBRE_TABLA,
        "${ContactoContrato.COL_FAVORITO} = ?",
        arrayOf("0")
    )
    db.close()
    return filas
}
```

Y su uso:

```
val borrados = dbHelper.eliminarNoFavoritos()
Toast.makeText(this, "Se borraron $borrados contactos", Toast.LENGTH_SHORT).show()
```

La lección: el WHERE no tiene que ser por id. Puedes borrar o actualizar conjuntos completos de filas con una condición. Pero cuidado: un delete con selection en null borra TODA la tabla. Por eso siempre escribe la condición primero y revísala dos veces antes de ejecutar. En desarrollo, prueba estos métodos con datos de prueba, nunca con datos reales del usuario.

Ejercicio 4: marcar favorito con un solo UPDATE

Agrega un método `alternarFavorito(id: Long)` al helper que invierta el valor de favorito para un contacto (si era 1 pasa a 0 y viceversa), sin necesidad de leer el contacto primero. Pista: puedes hacerlo con `execSQL` y una expresión SQL.

Solución: con SQL directo es una sola sentencia elegante:

```
fun alternarFavorito(id: Long) {
    val db = writableDatabase
    db.execSQL(
        "UPDATE ${ContactoContrato.NOMBRE_TABLA} " +
            "SET      ${ContactoContrato.COL_FAVORITO}    =    1    - " +
        "${ContactoContrato.COL_FAVORITO} " +
            "WHERE ${ContactoContrato.COL_ID} = ?",
        arrayOf(id)
    )
    db.close()
}
```

El truco está en ``1 - favorito``: si vale 1, queda 0; si vale 0, queda 1. Es una operación atómica: no hay riesgo de que otro hilo lea el valor entre la lectura y la escritura, cosa que sí podría pasar si primero lees y luego escribes en dos pasos. Este patrón de "modificar en el SQL mismo" es muy usado en producción para contadores e interruptores.

Ejercicio 5: tabla de notas relacionada con contactos

Diseña el SQL para una segunda tabla "notas" donde cada nota pertenece a un contacto (columnas: `_id`, `contacto_id`, `texto`, `fecha`). Escribe el `CREATE TABLE` con llave foránea y una consulta que traiga todas las notas de un contacto ordenadas por fecha descendente.

Solución: el `CREATE TABLE` con la llave foránea:


```
CREATE TABLE notas (  
    _id INTEGER PRIMARY KEY AUTOINCREMENT,  
    contacto_id INTEGER NOT NULL,  
    texto TEXT NOT NULL,  
    fecha TEXT NOT NULL,  
    FOREIGN KEY(contacto_id) REFERENCES contactos(_id)  
)
```

La cláusula FOREIGN KEY declara la relación: contacto_id debe corresponder a un _id existente en contactos. Nota: en SQLite las llaves foráneas no se verifican a menos que actives PRAGMA foreign_keys = ON; Room las maneja con @ForeignKey y validaciones propias.

La consulta de las notas de un contacto:

```
SELECT * FROM notas WHERE contacto_id = ? ORDER BY fecha DESC
```

Con fecha en formato ISO ("2026-09-19 10:30:00"), el ORDER BY DESC las ordena de la más reciente a la más antigua correctamente, porque el orden alfabético coincide con el cronológico. Este es el motivo por el que recomendamos el formato ISO para fechas como texto.

Ejercicio 6: migrar la agenda de SQLite crudo a Room

Toma el AgendaDbHelper de los ejemplos y describe los pasos concretos para migrar la app a Room sin perder los datos de los usuarios que ya tienen la versión con SQLite crudo instalada.

Solución: los pasos son:

1. Agregar las dependencias de Room (runtime, ktx, compiler con ksp).
2. Crear la entidad ContactoRoom con @Entity(tableName = "contactos") usando exactamente los mismos nombres de tabla y columnas que el esquema SQLite existente. Esto es lo crucial: si los nombres coinciden, Room puede abrir el mismo archivo "agenda.db" y leer las tablas ya creadas.
3. Crear el ContactoDao con las operaciones equivalentes (@Insert, @Update, @Delete, @Query).
4. Crear la clase AgendaRoomDb con @Database(entities = [ContactoRoom::class], version = 1) apuntando al mismo nombre de archivo "agenda.db".

5. Reemplazar en las Activities las llamadas al helper por llamadas al DAO dentro de corrutinas.
6. Probar la migración instalando primero la versión vieja, agregando datos, y luego instalando la versión Room encima: los datos deben seguir ahí.

El punto clave que muchos olvidan: Room no necesita que "conviertas" el archivo; SQLite es SQLite. Mientras el esquema que Room espera coincida con el que existe en el archivo, todo funciona. Por eso mantener nombres de tablas y columnas consistentes desde el inicio paga dividendos cuando migras.

Resumen de la unidad

- Persistir datos significa guardarlos en almacenamiento duradero para que sobrevivan al cierre de la app y al reinicio del dispositivo.
- SharedPreferences guarda pares clave-valor simples (String, Int, Long, Float, Boolean, Set de Strings) en un archivo XML privado; es ideal para ajustes, banderas y sesiones.
- Siempre usa MODE_PRIVATE; escribe con edit() + put + apply() (asíncrono) y lee con get + un valor por defecto sensato.
- Para sesiones: guarda una bandera de sesión activa más un token (nunca la contraseña en texto plano); la pantalla de arranque decide el destino según esa bandera.
- SQLite es una base de datos relacional completa embebida en cada dispositivo Android, sin servidor ni configuración.
- SQLiteOpenHelper gestiona la creación y el versionado: onCreate crea el esquema inicial, onUpgrade migra entre versiones cuando subes el número de versión.
- El CRUD son las cuatro operaciones: insert() para crear, query() para leer (devuelve un Cursor), update() para actualizar y delete() para borrar.
- Usa siempre selectionArgs con el marcador ? en lugar de concatenar texto del usuario: así evitas la inyección SQL.
- Cierra los Cursor con use() o close(), usa transacciones para escrituras múltiples relacionadas o masivas, y pon índices en las columnas por las que buscas u ordenas.
- Las fechas se guardan como TEXT en formato ISO o INTEGER como timestamp; los booleanos como INTEGER 0/1 porque SQLite no tiene tipo booleano.

- Room es la forma moderna recomendada: envuelve SQLite con entidades (@Entity), DAOs (@Dao) y base de datos (@Database), verifica tu SQL en tiempo de compilación y elimina el código repetitivo.
- En Room las operaciones del DAO son funciones suspend que se llaman desde corrutinas; la base de datos se expone como singleton.
- Guía de decisión: SharedPreferences para valores simples y pocos; SQLite/Room para colecciones estructuradas que se buscan, filtran u ordenan; archivos para documentos e imágenes; nube para datos entre dispositivos.
- Nunca guardes contraseñas en texto plano en almacenamiento local; para datos sensibles usa EncryptedSharedPreferences.

Extra: DataStore, el sucesor moderno de SharedPreferences

Para cerrar la unidad con visión de futuro, debes conocer DataStore, la biblioteca de Jetpack que Google posiciona como el reemplazo moderno de SharedPreferences. No es obligatorio dominarla hoy, pero en ofertas de trabajo recientes ya aparece como "deseable", así que conviene entender qué resuelve.

SharedPreferences tiene defectos conocidos que DataStore corrige: no es seguro ante escrituras concurrentes (dos hilos escribiendo a la vez pueden corromper datos), el apply() asíncrono puede perder escrituras si el proceso muere antes de volcar a disco, no avisa cuando un valor cambia (tienes que releer manualmente), y los errores de lectura/escritura se tragan en silencio. DataStore se construyó sobre corrutinas y Flow: lees valores como un flujo reactivo que emite automáticamente cada vez que el dato cambia, las escrituras son transaccionales y seguras, y los errores se propagan de forma controlada.

Hay dos variantes: Preferences DataStore (pares clave-valor, el reemplazo directo de SharedPreferences) y Proto DataStore (objetos tipados con Protocol Buffers, para esquemas más estructurados). Para el 90 por ciento de los casos, Preferences DataStore es suficiente.

Veamos un ejemplo mínimo para que veas el sabor:

```
// Definición del DataStore como propiedad delegada
val Context.ajustesDataStore: DataStore<Preferences> by preferencesDataStore(name =
"ajustes")

// Claves tipadas
val MODOS_OSCURO = booleanPreferencesKey("modo_oscuro")
val NOMBRE_USUARIO = stringPreferencesKey("nombre_usuario")

class AjustesRepository(private val context: Context) {

    // Lectura reactiva: un Flow que emite cada vez que cambia el valor
    val modoOscuro: Flow<Boolean> = context.ajustesDataStore.data
        .map { prefs -> prefs[MODOS_OSCURO] ?: false }

    // Escritura transaccional con corrutinas
    suspend fun guardarModoOscuro(activo: Boolean) {
        context.ajustesDataStore.edit { prefs ->
            prefs[MODOS_OSCURO] = activo
        }
    }
}
```

Y en la Activity lo observas así:

```
lifecycleScope.launch {
    repositorio.modoOscuro.collect { activo ->
        switchOscuro.isChecked = activo
        aplicarTema(activo)
    }
}
```

La diferencia filosófica es grande: con `SharedPreferences` tú preguntas el valor cuando lo necesitas; con `DataStore` te suscribes y el sistema te avisa cuando cambia. Si el usuario cambia el tema en la pantalla de ajustes, cualquier pantalla suscrita se entera sola. Ese modelo reactivo es el mismo que usa `Room` con `Flow`, así que aprenderlo aquí te prepara para la Unidad 8 y para el desarrollo Android moderno en general.

¿Cuándo migrar? Si tu app ya funciona con `SharedPreferences`, no hay prisa: Google mantiene `SharedPreferences` y no va a desaparecer. Pero en proyectos nuevos, considera `DataStore` desde el inicio, sobre todo si ya usas corrutinas (que a estas alturas del curso ya manejas). Existe incluso una guía oficial de migración que lee tu `SharedPreferences` viejo y lo vuelca a `DataStore` automáticamente.

Glosario rápido de la unidad

- Persistencia: guardar datos en almacenamiento duradero para que sobrevivan al ciclo de vida de la app.
- Clave-valor: modelo donde cada dato tiene un nombre único (clave) y un valor asociado.
- CRUD: Create, Read, Update, Delete; las cuatro operaciones básicas sobre datos.
- Tabla: estructura de filas y columnas dentro de una base de datos relacional.
- Registro (fila): una entrada individual en una tabla, por ejemplo un contacto.
- Columna (campo): un atributo de la tabla, por ejemplo "teléfono".
- Llave primaria: columna que identifica de forma única cada fila, normalmente un id autoincremental.
- Llave foránea: columna que referencia el id de otra tabla, creando una relación.
- Cursor: iterador sobre los resultados de una consulta SQLite, fila por fila.
- ContentValues: contenedor de pares columna-valor para insertar o actualizar en SQLite.
- Transacción: conjunto de operaciones que se confirman todas juntas o se revierten todas.
- Migración: código que transforma el esquema viejo de la base de datos al nuevo al actualizar la app.
- Inyección SQL: ataque que manipula consultas concatenando texto malicioso; se evita con selectionArgs.
- Entidad (Room): clase anotada con @Entity que representa una tabla.
- DAO (Room): interfaz anotada con @Dao que declara las operaciones de acceso a datos.
- Flow: flujo reactivo de corrutinas que emite valores a lo largo del tiempo, usado por DataStore y Room para observar cambios.

Con esto, la Unidad 7 queda completa: dominas las preferencias, el SQLite clásico con su CRUD, la visión moderna con Room y DataStore, y los criterios para elegir entre ellos. En la siguiente unidad conectaremos tu app con el mundo: APIs, red y datos que viven en servidores.

Caso de estudio: depurando la persistencia como un profesional

La teoría y los ejemplos te enseñan a escribir código que funciona; esta sección te enseña a descubrir por qué no funciona cuando algo sale mal. La depuración de persistencia tiene sus propias herramientas y sus propios errores clásicos. Vamos a verlos.

Inspeccionar la base de datos con Database Inspector

Android Studio incluye el Database Inspector (en versiones recientes, dentro del App Inspection). Con la app corriendo en un emulador o dispositivo conectado, puedes abrir la base de datos en vivo: ver las tablas, su contenido fila por fila, e incluso ejecutar consultas SQL directamente y modificar valores para probar. Es la herramienta más valiosa de esta unidad.

El flujo típico de depuración es: "mi lista sale vacía". Abres el inspector, miras la tabla contactos y descubres una de tres cosas: la tabla está vacía (el insert nunca ocurrió o falló), la tabla tiene datos pero tu consulta los filtra (el WHERE está mal), o la tabla ni siquiera existe (el onCreate no se ejecutó o el nombre está mal). Cada escenario apunta a una causa distinta y te ahorra horas de adivinar. Acostúmbrate a verificar los datos en el inspector antes de culpar a la UI.

Para SharedPreferences, el archivo XML vive en `/data/data/tu.paquete/shared_prefs/`. En un emulador con root puedes abrirlo con el Device File Explorer de Android Studio y leer los valores tal cual. Cuando "el ajuste no se guarda", casi siempre es una de estas: escribiste con una clave y lees con otra (typo), olvidaste el `apply()`, o estás leyendo de un archivo distinto al que escribiste (dos nombres de archivo diferentes).

Los cinco errores que verás en Logcat (y qué significan)

1. `android.database.sqlite.SQLiteException: no such table: contactos`. La tabla no existe. Causas: `onCreate` no se ejecutó (la base ya existía de una instalación anterior sin esa tabla), o el nombre en el `CREATE TABLE` no coincide con el del query. Solución: desinstala la app para forzar `onCreate` durante el desarrollo, o escribe la migración en `onUpgrade` si ya hay usuarios.
2. `android.database.sqlite.SQLiteException: no such column: teléfono`. La columna no existe. Casi siempre es un typo entre el `CREATE TABLE` y el query, o una migración olvidada. El contrato de constantes previene la mitad de estos casos.
3. `android.database.CursorIndexOutOfBoundsException`. Intentaste leer una columna de un Cursor posicionado fuera de las filas, típicamente porque olvidaste `moveToFirst()` o `moveToNext()`, o porque la consulta devolvió cero filas y leíste de todos modos. Solución: verifica siempre con `moveToFirst()` o itera con `while (cursor.moveToNext())`.
4. `IllegalStateException: Cannot access database on the main thread`. Es Room protegiéndote: intentaste una operación de base de datos en el hilo principal. Solución: usa corrutinas (`lifecycleScope.launch`) o mueve la llamada a un hilo secundario. Con SQLite crudo no hay excepción, solo la UI congelada, que es peor porque no te avisa.

5. Los datos "desaparecen" al reinstalar o entre ejecuciones. Casi nunca es un bug de SQLite: es que estás insertando datos de prueba en cada onCreate (como en nuestro ejemplo didáctico), o que dos partes del código usan nombres de archivo de base de datos distintos, o que el usuario desinstaló (al desinstalar se borra todo, es normal). Verifica el nombre del archivo y dónde ocurre cada escritura.

Estrategia general de depuración

Cuando algo falla con persistencia, sigue este orden, del más probable al menos probable:

1. Verifica el dato en el inspector o en el archivo: ¿se escribió? ¿con qué valores?
2. Revisa las claves y nombres: typos en claves de preferencias, nombres de tablas y columnas.
3. Revisa el ciclo de vida: ¿el onCreate/onUpgrade se ejecutó? ¿aumentaste la versión al cambiar el esquema?
4. Revisa los hilos: ¿bloqueas la UI? ¿Room te está deteniendo por acceso en hilo principal?
5. Revisa las condiciones: imprime el WHERE y los argumentos en el log antes de ejecutar.

Y una regla de oro del desarrollo: durante la fase de pruebas, no tengas miedo de desinstalar la app para empezar con una base limpia. Muchos "bugs misteriosos" son simplemente esquemas viejos de ayer conviviendo con código nuevo de hoy. En producción los resolverás con migraciones; en desarrollo, con una desinstalación.

Con esto cerramos la Unidad 7. Tienes la teoría, el código completo, la práctica, el glosario y las herramientas de depuración. La persistencia local ya no tiene secretos para ti.

UNIDAD 8: Consumo de APIs y trabajo en red

Introducción

Bienvenido a la Unidad 8. Hasta ahora, todas las aplicaciones que hemos construido viven encerradas en el teléfono: sus datos nacen y mueren en el dispositivo. Pero las aplicaciones que usas todos los días no funcionan así. Cuando abres YouTube, los videos vienen de servidores de Google. Cuando pides un Uber, tu ubicación viaja a sus servidores y el conductor asignado regresa por la red. Cuando consultas el clima, tu app le pregunta a un servicio meteorológico. La magia de las apps modernas no está solo en lo que muestran, sino en que están conectadas al mundo. En esta unidad tu app va a salir del teléfono y a conversar con internet.

El puente entre tu app y esos servidores se llama API, y el idioma en que conversan se llama JSON, viajando sobre el protocolo HTTP. Dominar este tema te convierte en un desarrollador completo: ya sabes guardar datos localmente (Unidad 7), y ahora aprenderás a traer datos del mundo y a enviar los tuyos. La combinación de ambas habilidades es lo que permite construir casi cualquier aplicación real: redes sociales, tiendas en línea, apps de noticias, clientes de música, sistemas de reservas. Todas siguen el mismo patrón: piden datos a un servidor, los muestran, y a veces guardan una copia local para funcionar sin conexión.

En esta unidad trabajaremos con la pila estándar de la industria Android, organizada en tres clases:

En la clase 22 entenderás qué es una API REST y qué es JSON. Verás cómo funciona el protocolo HTTP por debajo: los métodos GET, POST, PUT y DELETE, las URLs con sus rutas y parámetros, los códigos de estado (ese famoso 404 que ya conoces, el 200 que significa éxito, el 500 que significa que el servidor se rompió). Aprenderás a leer documentación de APIs y a probar endpoints con herramientas antes de escribir una sola línea de código. Usaremos APIs públicas reales para practicar, así que todo lo que hagas en esta unidad funcionará de verdad, no con datos inventados.

En la clase 23 conocerás Retrofit, la biblioteca que el 90 por ciento de las apps Android profesionales usa para hablar con APIs. Aprenderás a declarar sus dependencias, a definir una interface donde cada método representa un endpoint, a hacer llamadas GET para traer datos y POST para enviarlos, y a usar Gson o Moshi para convertir automáticamente el JSON en objetos Kotlin. Al final, mostrarás esos datos en un RecyclerView, cerrando el círculo con lo que aprendiste en unidades anteriores: datos reales de internet, en una lista real, en tu app real.

En la clase 24 entraremos a las corrutinas de Kotlin, el mecanismo moderno para hacer trabajo en segundo plano sin bloquear la interfaz. Entenderás por qué Android prohíbe hacer red en el hilo principal, cómo las corrutinas resuelven ese problema con código que se lee como si fuera secuencial, y cómo manejar los errores de red con elegancia: sin internet, servidor caído, respuestas malformadas. Una app que se congela o se cierra cuando falla la red es una app que los usuarios desinstalan; una app que muestra un mensaje claro y reintenta es una app profesional.

¿Por qué importa todo esto? Porque el trabajo en red es donde viven los empleos. La inmensa mayoría de las aplicaciones comerciales son, en esencia, clientes de APIs: pantallas bonitas sobre datos que viven en servidores. En las entrevistas de trabajo para Android, "consume esta API y muestra los resultados en una lista" es probablemente el ejercicio práctico más común de todos. Si dominas Retrofit + corrutinas + RecyclerView, puedes resolver ese ejercicio dormido. Además, entender HTTP y JSON te sirve más allá de Android: es el lenguaje común de todo el desarrollo moderno, web incluido.

Al terminar esta unidad sabrás: explicar qué es una API REST y leer su documentación; interpretar JSON y mapearlo a data classes de Kotlin; configurar Retrofit con sus dependencias, definir interfaces de endpoints y ejecutar llamadas GET y POST; convertir JSON a objetos automáticamente con Gson o Moshi; usar corrutinas de Kotlin para ejecutar la red en segundo plano sin congelar la UI, con los dispatchers correctos y el ciclo de vida respetado; manejar errores de red (sin conexión, timeouts, errores del servidor, respuestas inesperadas) mostrando estados adecuados al usuario; y mostrar datos remotos en un RecyclerView con estados de carga, éxito y error.

Un consejo práctico: en esta unidad vas a depender de internet y de servidores ajenos, así que algunas cosas escapan a tu control: una API pública puede estar caída justo cuando practicas, o cambiar su formato. No te frustres: eso también es parte del aprendizaje real, y por eso dedicamos una clase entera al manejo de errores. Trabaja con las APIs que te propongo, ten siempre un plan B (otra API pública), y prueba cada endpoint en el navegador o en una herramienta antes de programarlo. Ver el JSON crudo con tus propios ojos antes de parsearlo te ahorrará muchísimos dolores de cabeza.

Vamos a conectar tu app con el mundo.

Explicación detallada

Qué es una API y qué es REST

API significa Interfaz de Programación de Aplicaciones. En términos sencillos, es un contrato: un servidor publica una serie de URLs, y te promete que si las llamas de cierta forma, te responderá con datos en cierto formato. Tú no necesitas saber cómo está programado ese servidor, en qué lenguaje, ni dónde está físicamente. Solo necesitas el contrato: la dirección, qué pedir y qué esperar de vuelta.

La analogía clásica es el mesero de un restaurante. Tú (la app) no entras a la cocina (el servidor) a cocinar: le dices al mesero (la API) qué quieres del menú (los endpoints disponibles), él lo lleva a la cocina, y regresa con tu platillo (los datos). El menú te dice qué puedes pedir y cómo; si pides algo que no está en el menú, el mesero te dice que no existe (error 404). Tú nunca ves la cocina, y la cocina nunca te ve a ti. Esa separación es poderosa: el servidor puede cambiar toda su tecnología interna y mientras el menú siga igual, tu app sigue funcionando.

REST (Transferencia de Estado Representacional) es el estilo más común para diseñar esas APIs. No es una tecnología ni una biblioteca: es un conjunto de convenciones. Las ideas centrales son:

1. Todo es un recurso identificado por una URL. Los usuarios viven en /usuarios, un usuario específico en /usuarios/42, sus pedidos en /usuarios/42/pedidos. Las URLs son sustantivos, no verbos.
2. Se usan los métodos HTTP estándar para las acciones: GET para leer, POST para crear, PUT para reemplazar/actualizar, DELETE para borrar. Es el CRUD de la Unidad 7, pero a través de la red.
3. No hay memoria entre llamadas (stateless): cada petición lleva todo lo necesario, típicamente un token de autenticación. El servidor no "recuerda" tu llamada anterior.
4. Los datos viajan normalmente en JSON, un formato de texto ligero que tanto humanos como máquinas leen fácilmente.

Cuando una API sigue estas convenciones se le llama API REST o RESTful. Casi todas las APIs públicas que usarás (JSONPlaceholder, PokeAPI, OpenWeather, la API de GitHub) son REST.

HTTP por debajo: lo que realmente viaja por el cable

Cada vez que tu app "llama a una API", ocurre un intercambio de mensajes HTTP. Entender ese intercambio te hace mejor depurando, porque cuando algo falla, el error casi siempre está en alguna parte de ese mensaje.

Una petición HTTP tiene: un método (GET, POST...), una URL (que incluye el host, la ruta y opcionalmente parámetros de consulta como `?página=2&limite=20`), unas cabeceras (headers: metadatos como el tipo de contenido o el token de autenticación), y opcionalmente un cuerpo (body: los datos que envías en un POST, normalmente en JSON).

Una respuesta HTTP tiene: un código de estado (el número famoso), unas cabeceras, y un cuerpo (los datos pedidos, normalmente en JSON).

Los códigos de estado se agrupan por centenas y vale la pena memorizar los más comunes:

- 200 OK: todo salió bien, aquí están tus datos. El pan de cada día.
- 201 Created: se creó un recurso nuevo (típico tras un POST exitoso).
- 204 No Content: éxito sin datos que devolver (típico tras un DELETE).
- 400 Bad Request: tu petición está mal formada; revisa los parámetros.
- 401 Unauthorized: falta autenticación o el token es inválido; necesitas login.
- 403 Forbidden: estás autenticado pero no tienes permiso para ese recurso.
- 404 Not Found: la URL no existe; revisa la ruta o el id.
- 429 Too Many Requests: hiciste demasiadas llamadas muy rápido; espera y reintenta.
- 500 Internal Server Error: el servidor se rompió; no es tu culpa, pero debes manejarlo.
- 502/503: el servidor o un intermediario falló o está en mantenimiento.

Regla de oro para tu código: nunca asumas que la respuesta será 200. Toda llamada de red puede fallar de una docena de formas, y tu app debe manejar cada familia de errores con un mensaje útil en lugar de cerrarse o quedarse congelada. La clase 24 está dedicada exactamente a eso.

Los parámetros de consulta (query params) merecen atención porque los usarás constantemente: son los pares `clave=valor` después del `?` en la URL. Por ejemplo, `GET /usuarios?rol=admin&activos=true` pide los usuarios filtrados. En Retrofit los declararás con la anotación `@Query` y la biblioteca construirá la URL por ti, escapando caracteres especiales automáticamente. Las rutas con variables, como `/usuarios/42`, usan `@Path`: tú pasas el 42 como argumento y Retrofit lo inserta en la URL.

JSON: el idioma de las APIs

JSON (Notación de Objetos de JavaScript) es un formato de texto para representar datos estructurados. A pesar del nombre, hoy lo usan todos los lenguajes. Su gramática es mínima: objetos entre llaves con pares clave-valor, arreglos entre corchetes, y valores que pueden ser texto, número, booleano, null, otro objeto u otro arreglo.

Un ejemplo realista, como el que devuelve JSONPlaceholder para un usuario:

```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "city": "Gwenborough",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona"
  }
}
```

Lee esa estructura: un objeto con campos simples (id, name), objetos anidados (address contiene geo) y campos que podrías ignorar. En Kotlin lo modelas con data classes anidadas, y Gson o Moshi hacen la conversión automática. La correspondencia es directa: cada clave JSON se mapea a una propiedad Kotlin del mismo nombre (o al nombre que indiques con una anotación si difieren).

Errores comunes de principiantes con JSON: asumir que un campo siempre vendrá (los servidores a veces omiten campos o los mandan null; usa tipos nulos String? cuando el campo pueda faltar); confundir un objeto con un arreglo (si el JSON empieza con [es una lista, si empieza con { es un objeto; tu data class debe corresponder); y no verificar el JSON crudo antes de programar (abre la URL en el navegador primero y mira la estructura real).

Retrofit: el estándar de la industria

Retrofit es una biblioteca de Square que convierte tu API REST en una interface de Kotlin. Tú declaras métodos anotados que describen cada endpoint, y Retrofit genera la implementación que hace la llamada HTTP real, envía los parámetros, recibe la respuesta y la convierte a tus objetos. Es declarativo: describes QUÉ quieres, no CÓMO se hace.

Sus piezas:

1. Las dependencias en build.gradle: retrofit, el convertidor (converter-gson o converter-moshi) y, si usas corrutinas, nada extra porque Retrofit las soporta de forma nativa desde la versión 2.6.
2. La interface del servicio: cada método lleva una anotación con el método HTTP y la ruta (@GET("usuarios"), @POST("usuarios")), los parámetros (@Query, @Path, @Body) y el tipo de retorno, que con corrutinas es directamente tu objeto o Response<TuObjeto>.
3. El objeto Retrofit construido con Builder: le das la base URL, agregas el convertidor con addConverterFactory, y creas el servicio con create().
4. Las data classes que modelan el JSON, con anotaciones @SerializedName (Gson) o @Json (Moshi) cuando el nombre del campo JSON difiere del nombre Kotlin.

La diferencia entre que el método devuelva List<Usuario> directamente o Response<List<Usuario>> es importante: con Response tienes acceso al código de estado, las cabeceras y puedes distinguir un 404 de un error de red; sin Response, Retrofit lanza una excepción ante cualquier respuesta no exitosa. Para código profesional, prefiere Response y maneja los códigos explícitamente, como verás en los ejemplos.

Gson vs Moshi

Ambos convierten JSON a objetos Kotlin y viceversa. Gson es el veterano: lleva años, tiene muchísima documentación y ejemplos. Moshi es el sucesor espiritual, también de Square (los mismos de Retrofit), diseñado pensando en Kotlin desde el inicio: maneja mejor los tipos nulos, los valores por defecto de las data classes y genera adaptadores en tiempo de compilación (más rápido y seguro) con la anotación @JsonClass(generateAdapter = true).

Para este curso usaremos Gson en los ejemplos principales porque su sintaxis es la más simple y la encontrarás en más tutoriales y código existente, pero te mostraré el equivalente en Moshi para que conozcas ambas. En un proyecto nuevo profesional, Moshi (o kotlinx.serialization, la opción oficial de JetBrains) sería la recomendación moderna. Lo importante es que el patrón es idéntico: data classes + anotaciones + convertidor en el Builder.

Un detalle que atrapa a principiantes: los nombres de campos. Si el JSON trae "user_name" y tu propiedad Kotlin se llama userName, necesitas `@SerializedName("user_name")` para que Gson haga el puente. Sin la anotación, el campo quedará null o con su valor por defecto y pasarás horas creyendo que la API no manda el dato. Cuando algo "no llega", lo primero que se revisa son los nombres de campos.

Corrutinas de Kotlin: red sin congelar la app

Android tiene una regla inquebrantable: el hilo principal (también llamado hilo de UI) no debe bloquearse. Ese hilo dibuja la pantalla 60 veces por segundo y procesa tus toques; si lo ocupas esperando una respuesta de red que tarda 3 segundos, la app se congela 3 segundos y el sistema muestra el temido diálogo "la aplicación no responde" (ANR). Por eso la red siempre va en segundo plano.

Las corrutinas son la forma moderna de Kotlin para programar trabajo asíncrono. La idea brillante: escribes código que se lee como secuencial (primero pido los datos, luego los muestro), pero que se suspende sin bloquear el hilo mientras espera. La palabra clave es `suspend`: una función `suspend` puede pausarse y reanudarse después, liberando el hilo mientras tanto.

Los conceptos mínimos que necesitas:

- `launch`: lanza una corrutina "dispara y olvida", para trabajo que no devuelve un valor directo (como actualizar la UI tras cargar datos).
- `async`: lanza una corrutina que devuelve un `Deferred`, para trabajo que produce un resultado que esperarás con `await()`. Útil para lanzar dos llamadas en paralelo y esperar ambas.
- `Dispatchers.IO`: el despachador optimizado para operaciones de entrada/salida (red, disco, base de datos). Las llamadas de red van aquí.
- `Dispatchers.Main`: el hilo principal, donde actualizas la UI. Retrofit con corrutinas ya regresa al hilo que llamó, así que normalmente lanzas en Main y la suspensión ocurre en IO internamente; verás el patrón exacto en los ejemplos.
- `lifecycleScope`: el alcance ligado al ciclo de vida de la Activity o Fragment. Si el usuario sale de la pantalla, las corrutinas se cancelan automáticamente. Esto previene el error clásico de actualizar una pantalla que ya no existe.
- `viewModelScope`: lo mismo pero ligado a un `ViewModel`; sobrevive a rotaciones de pantalla. Es el estándar en arquitectura profesional.

El manejo de errores con corrutinas usa try/catch normal, lo cual es otra de sus bellezas: el código asíncrono se maneja con las mismas herramientas que el código normal. IOException para problemas de red (sin conexión, timeout), HttpException para respuestas HTTP no exitosas cuando no usas Response, y una rama genérica para lo inesperado. En los ejemplos verás una función que traduce cada excepción a un mensaje amigable para el usuario.

Errores comunes con corrutinas: lanzar corrutinas en GlobalScope (viven para siempre, fugan memoria y actualizan pantallas muertas; casi nunca es lo correcto); olvidar que lifecycleScope se cancela al salir (si necesitas que la descarga termine aunque el usuario navegue, ese trabajo va en otro alcance o en WorkManager); y atrapar excepciones demasiado pronto, ocultando errores reales (registra siempre el error en el log aunque muestres un mensaje amigable).

Arquitectura mínima profesional: separando responsabilidades

Antes de los ejemplos de código, una nota de arquitectura que te servirá toda la carrera. El código de red no debe vivir en la Activity. El patrón profesional mínimo tiene tres capas:

1. La capa de red: la interface Retrofit y el Builder (a veces envueltos en una clase ApiClient).
2. El repositorio: una clase que expone funciones suspend como obtenerUsuarios() y por dentro llama a Retrofit, maneja errores y traduce a un resultado limpio (éxito con datos o fallo con mensaje). La UI nunca ve excepciones crudas ni códigos HTTP.
3. La UI (Activity/Fragment): llama al repositorio desde una corrutina, muestra carga, luego datos o error.

Este diseño hace tu código testeable (puedes probar el repositorio sin Android), reutilizable (varias pantallas usan el mismo repositorio) y legible. En los ejemplos paso a paso lo implementaremos así, aunque sea una versión compacta. Cuando en el futuro estudies MVVM con ViewModel y LiveData/StateFlow, verás que encaja perfecto: el ViewModel llama al repositorio y la UI observa al ViewModel.

Autenticación: cómo las APIs saben quién eres

Las APIs públicas de práctica no piden login, pero las reales casi siempre sí. El mecanismo dominante hoy son los tokens: al iniciar sesión (un POST con usuario y contraseña), el servidor te devuelve un token, una cadena larga y opaca. En cada llamada posterior incluyes ese token en la cabecera Authorization, y el servidor sabe quién eres sin que reenvíes tu contraseña.

El flujo completo en una app real combina las dos unidades: el login hace POST, recibe el token, lo guarda en SharedPreferences o DataStore (¿ves cómo se conecta todo?), y cada llamada posterior lo lee de ahí y lo pone en la cabecera. Cuando el usuario cierra sesión, borras el token y las llamadas dejan de autenticarse.

En Retrofit, las cabeceras se ponen de dos formas. Para un token fijo o por llamada, con la anotación `@Header`:

```
@GET("perfil")
suspend fun obtenerPerfil(@Header("Authorization") token: String):
Response<Usuario>
```

Y lo llamarías con `obtenerPerfil("Bearer $token")`. El prefijo "Bearer " es la convención estándar para tokens.

Para algo más profesional (agregar el token a TODAS las llamadas automáticamente), se usa un Interceptor de OkHttp, el motor HTTP que Retrofit usa por debajo. El interceptor revisa cada petición saliente y le agrega la cabecera:

```
class AuthInterceptor(private val proveedorToken: () -> String?) : Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val original = chain.request()
        val token = proveedorToken()
        val request = if (token != null) {
            original.newBuilder()
                .header("Authorization", "Bearer $token")
                .build()
        } else {
            original
        }
        return chain.proceed(request)
    }
}
```

Y lo conectas al construir Retrofit:


```

val cliente = OkHttpClient.Builder()
    .addInterceptor(AuthInterceptor { sesion.obtenerToken() })
    .build()

val retrofit = Retrofit.Builder()
    .baseUrl("https://api.ejemplo.com/")
    .client(cliente)
    .addConverterFactory(GsonConverterFactory.create())
    .build()

```

Los interceptores también sirven para registrar las llamadas en el log durante el desarrollo (HttpLoggingInterceptor), lo cual es invaluable para depurar: ves la URL exacta, las cabeceras y el cuerpo de la respuesta tal cual viajaron. Si algo falla, el log del interceptor te dice si el problema está en lo que enviaste o en lo que respondió el servidor.

Paginación: cuando los datos no caben en una respuesta

Ninguna API sería te devuelve sus 10 millones de registros de una vez. Las APIs pagan: devuelven los resultados en páginas, y tú pides la página que necesitas con parámetros como `?page=2&limit=20` o `?offset=40`. La respuesta suele incluir metadatos: total de elementos, página actual, si hay más páginas.

El patrón en la app es el "scroll infinito": el RecyclerView detecta cuando el usuario llega al final de la lista y pide la siguiente página, agregando los nuevos elementos a los ya mostrados. En Retrofit es simplemente un `@Query` más:

```

@GET("productos")
suspend fun obtenerProductos(
    @Query("page") pagina: Int,
    @Query("limit") limite: Int = 20
): Response<PaginaProductos>

```

Donde `PaginaProductos` es una data class que modela tanto la lista como los metadatos:

```

data class PaginaProductos(
    val productos: List<Producto>,
    val total: Int,
    val pagina: Int,
    @SerializedName("total_paginas") val totalPaginas: Int
)

```

La app guarda la página actual, y cuando necesita más, pide página + 1 hasta llegar a totalPaginas. Este patrón aparece en casi toda app con listas largas: tiendas, redes sociales, noticias.

Buenas prácticas de red que te distinguen

1. Nunca hagas red en el hilo principal. Ya lo dijimos, pero es la regla número uno porque es el error número uno.
2. Pon tiempos de espera (timeouts). Sin timeout, una conexión colgada puede esperar eternamente. Configura `connectTimeout`, `readTimeout` y `writeTimeout` en el `OkHttpClient` (10 a 30 segundos son valores razonables).
3. Reintenta con criterio. Un fallo por timeout a veces se resuelve reintentando una vez; un 401 no se arregla reintentando. No reintentas POSTs que crean recursos sin un mecanismo de idempotencia, o podrías crear duplicados.
4. Cachea con inteligencia. Si los datos cambian poco (catálogos, perfiles), guarda una copia local (Room, Unidad 7) y muestra primero lo guardado mientras actualizas en segundo plano. El usuario ve algo al instante aunque la red tarde. Este patrón "local primero, red después" es el estándar de las apps bien hechas.
5. No expongas claves de API en el código. Las API keys van en archivos de configuración no versionados (`local.properties`, `BuildConfig`), nunca hardcodeadas en el repositorio de git.
6. Valida y sanitiza lo que muestras. Los datos de un servidor son entrada externa: pueden traer HTML, textos larguísimos o caracteres raros. No confíes ciegamente.
7. Maneja el modo sin conexión. Detecta la falta de red y dilo claramente ("Sin conexión a internet") en lugar de mostrar un error genérico o una pantalla vacía. Los usuarios perdonan la falta de red; no perdonan la confusión.
8. Registra los errores para ti, muestra mensajes para el usuario. En el log va el detalle técnico (código, URL, excepción); en la pantalla va algo humano ("No pudimos cargar los productos. Revisa tu conexión e inténtalo de nuevo.").

Con la teoría completa, vamos al código. Usaremos JSONPlaceholder (jsonplaceholder.typicode.com), una API pública gratuita ideal para practicar: no requiere autenticación y tiene endpoints de usuarios, publicaciones y comentarios con datos realistas.

Ejemplos paso a paso

Ejemplo 1: configurar Retrofit y definir la API (GET)

Vamos a consumir JSONPlaceholder, una API pública de práctica. Su endpoint `https://jsonplaceholder.typicode.com/users` devuelve una lista de 10 usuarios en JSON. Nuestro objetivo: traerlos y convertirlos a objetos Kotlin.

Paso 1: dependencias en el build.gradle del módulo app.

```
dependencies {  
    implementation("com.squareup.retrofit2:retrofit:2.9.0")  
    implementation("com.squareup.retrofit2:converter-gson:2.9.0")  
    implementation("com.squareup.okhttp3:logging-interceptor:4.12.0")  
}
```

- retrofit: la biblioteca principal.
- converter-gson: el convertidor que traduce JSON a objetos usando Gson.
- logging-interceptor: para ver las llamadas en el Logcat durante el desarrollo (opcional pero muy recomendado).

Además, necesitas el permiso de internet en el AndroidManifest.xml, sin el cual ninguna llamada de red funciona:

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

Va antes de la etiqueta `application`. Olvidar este permiso es un error clásico: la app falla con un error de red confuso y todo el código está bien.

Paso 2: las data classes que modelan el JSON. Recuerda el JSON de usuario que vimos en la teoría; lo modelamos así:

```

data class Geo(
    val lat: String,
    val lng: String
)

data class Direccion(
    val street: String,
    val city: String,
    val geo: Geo
)

data class Empresa(
    val name: String
)

data class Usuario(
    val id: Int,
    val name: String,
    val username: String,
    val email: String,
    val phone: String,
    val website: String,
    val address: Direccion,
    val company: Empresa
)

```

- Cada objeto anidado del JSON es una data class anidada en Kotlin. Los nombres coinciden con las claves JSON, así que no necesitamos @SerializedName aquí.
- Observa que ignoramos campos que no nos interesan: el JSON trae más datos, pero Gson simplemente ignora lo que no está en la data class. Modela solo lo que vas a usar.

Paso 3: la interface del servicio. Cada método es un endpoint:

```
interface ApiService {

    @GET("users")
    suspend fun obtenerUsuarios(): Response<List<Usuario>>

    @GET("users/{id}")
    suspend fun obtenerUsuario(@Path("id") id: Int): Response<Usuario>

    @GET("users")
    suspend fun buscarUsuarios(@Query("username") username: String):
Response<List<Usuario>>
}
```

- `@GET("users")`: petición GET a la ruta "users" (se suma a la base URL).
- `suspend fun`: las funciones son suspend para llamarlas desde corrutinas; Retrofit se encarga de ejecutar la red en segundo plano.
- `Response<List<Usuario>>`: devolvemos Response para tener acceso al código de estado y manejar errores explícitamente.
- `@Path("id")`: el valor del argumento se inserta en la URL donde está {id}. `obtenerUsuario(3)` llama a `users/3`.
- `@Query("username")`: el argumento se agrega como parámetro de consulta: `users?username=Bret`. JSONPlaceholder filtra por ese campo.

Paso 4: construir Retrofit y crear el servicio. Lo centralizamos en un objeto:

```

object ClienteApi {

    private const val BASE_URL = "https://jsonplaceholder.typicode.com/"

    private val interceptorLog = HttpLoggingInterceptor().apply {
        level = HttpLoggingInterceptor.Level.BODY
    }

    private val clienteHttp = OkHttpClient.Builder()
        .addInterceptor(interceptorLog)
        .connectTimeout(20, TimeUnit.SECONDS)
        .readTimeout(20, TimeUnit.SECONDS)
        .build()

    private val retrofit = Retrofit.Builder()
        .baseUrl(BASE_URL)
        .client(clienteHttp)
        .addConverterFactory(GsonConverterFactory.create())
        .build()

    val servicio: ApiService = retrofit.create(ApiServicio::class.java)
}

```

Explicación línea por línea:

- `BASE\_URL`: la raíz de la API. Debe terminar en / y las rutas de la interface no deben empezar con /.
- `HttpLoggingInterceptor` en nivel BODY: registra en Logcat la petición completa y la respuesta completa. En desarrollo es oro; en producción se baja a NONE o se quita.
- `OkHttpClient.Builder()`: configuramos timeouts de 20 segundos para conexión y lectura, así una red colgada no espera eternamente.
- `Retrofit.Builder()`: unimos base URL, cliente HTTP y el convertidor Gson.
- `retrofit.create(ApiServicio::class.java)`: Retrofit genera la implementación de nuestra interface. Desde aquí, ClienteApi.servicio.obtenerUsuarios() hace la llamada real.

Con esto, la infraestructura está lista. En el siguiente ejemplo la usamos desde una corrutina con manejo de errores profesional.

Ejemplo 2: repositorio con corrutinas y manejo de errores

Ahora la capa que separa la UI de la red: un repositorio que llama a Retrofit dentro de corrutinas, interpreta cada resultado y devuelve un tipo sellado que la UI entiende sin conocer detalles HTTP.

Paso 1: el tipo de resultado. Una sealed class es perfecta porque el when que la consuma queda exhaustivo (el compilador te obliga a manejar todos los casos):

```
sealed class ResultadoApi<out T> {  
    data class Exito<T>(val datos: T) : ResultadoApi<T>()  
    data class Error(val mensaje: String) : ResultadoApi<Nothing>()  
    object Cargando : ResultadoApi<Nothing>()  
}
```

- Exito contiene los datos; Error contiene un mensaje amigable para mostrar; Cargando representa el estado mientras se espera. La UI reacciona a estos tres estados: spinner, lista o mensaje.

Paso 2: el repositorio.

```

class UsuarioRepositorio {

    private val servicio = ClienteApi.servicio

    suspend fun obtenerUsuarios(): ResultadoApi<List<Usuario>> {
        return try {
            val respuesta = servicio.obtenerUsuarios()
            if (respuesta.isSuccessful) {
                val usuarios = respuesta.body()
                if (usuarios != null) {
                    ResultadoApi.Exito(usuarios)
                } else {
                    ResultadoApi.Error("El servidor respondió sin datos")
                }
            } else {
                ResultadoApi.Error(mensajeParaCodigo(respuesta.code()))
            }
        } catch (e: IOException) {
            ResultadoApi.Error("Sin conexión a internet. Revisa tu red e inténtalo de nuevo.")
        } catch (e: HttpException) {
            ResultadoApi.Error(mensajeParaCodigo(e.code()))
        } catch (e: Exception) {
            ResultadoApi.Error("Ocurrió un error inesperado: ${e.message}")
        }
    }

    private fun mensajeParaCodigo(codigo: Int): String {
        return when (codigo) {
            400 -> "La petición no es válida"
            401 -> "Necesitas iniciar sesión"
            403 -> "No tienes permiso para ver esto"
            404 -> "No se encontró la información solicitada"
            429 -> "Demasiadas peticiones, espera un momento"
            in 500..599 -> "El servidor falló, inténtalo más tarde"
            else -> "Error del servidor (código $codigo)"
        }
    }
}

```

Explicación detallada:

- La función es suspend: se llama desde una corrutina y Retrofit suspende sin bloquear el hilo mientras espera la red.

- `respuesta.isSuccessful`: true para códigos 2xx. Si lo es, extraemos `body()`; si el cuerpo es null (raro pero posible), lo tratamos como error con mensaje claro.
- Si no es exitosa, traducimos el código HTTP a un mensaje humano con `mensajeParaCodigo()`. El rango `in 500..599` agrupa todos los errores de servidor.
- `catch (e: IOException)`: problemas de red puros: sin internet, timeout, DNS fallido. Es el catch más común en el mundo real y merece su propio mensaje.
- `catch (e: HttpException)`: respuestas HTTP no exitosas cuando no usas `Response` (aquí lo incluimos por completitud defensiva).
- `catch (e: Exception)`: red de seguridad para lo imprevisto, como un JSON malformado que `Gson` no puede parsear. Nunca dejes una corrutina sin este último catch si su fallo llegaría a la UI.

Paso 3: usarlo desde la Activity con `lifecycleScope`.

```

class UsuariosActivity : AppCompatActivity() {

    private val repositorio = UsuarioRepositorio()

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

        val textoEstado = findViewById<TextView>(R.id.textoEstado)
        val progreso = findViewById<ProgressBar>(R.id.progreso)

        lifecycleScope.launch {
            // Estado: cargando
            progreso.visibility = View.VISIBLE
            textoEstado.text = "Cargando usuarios..."

            when (val resultado = repositorio.obtenerUsuarios()) {
                is ResultadoApi.Exito -> {
                    progreso.visibility = View.GONE
                    val nombres = resultado.datos.joinToString("\n") { it.name }
                    textoEstado.text = nombres
                }
                is ResultadoApi.Error -> {
                    progreso.visibility = View.GONE
                    textoEstado.text = resultado.mensaje
                    Toast.makeText(this@UsuariosActivity, resultado.mensaje,
                        Toast.LENGTH_LONG).show()
                }
                is ResultadoApi.Cargando -> {
                    // No se usa aquí porque controlamos la carga manualmente,
                    // pero existe para pantallas que observan un flujo de estados
                }
            }
        }
    }
}

```

- `lifecycleScope.launch``: la corrutina vive mientras la Activity viva; si el usuario sale, se cancela sola y no intentará tocar vistas destruidas.
- El `when` es exhaustivo gracias a la sealed class: si mañana agregas un estado nuevo, el compilador te avisa en cada `when` que lo usa.
- La UI nunca ve una excepción ni un código HTTP: solo estados limpios. Esa es la marca de una capa de red bien diseñada.

Nota sobre hilos: no especificamos Dispatcher porque las funciones suspend de Retrofit ya hacen el trabajo de red en un hilo de fondo y reanudan en el hilo que llamó (el principal, gracias a lifecycleScope). Tú escribes código lineal y Kotlin se encarga del resto. Esa es la magia de las corrutinas bien integradas.

Ejemplo 3: POST para enviar datos y mostrar todo en un RecyclerView

Leer datos está muy bien, pero las apps reales también envían: publicar un comentario, registrar un usuario, subir una foto. Vamos a ver el POST y luego cerraremos mostrando los datos en un RecyclerView cómo se hace en producción.

Primero, la data class para una publicación y el endpoint POST en la interface:

```
data class Publicacion(  
    val id: Int = 0,  
    val userId: Int,  
    val title: String,  
    val body: String  
)
```

```
@POST("posts")  
suspend fun crearPublicacion(@Body publicacion: Publicacion): Response<Publicacion>
```

- `@Body`: el objeto se serializa a JSON automáticamente con Gson y se envía como cuerpo de la petición. Tú trabajas con objetos Kotlin; Retrofit y Gson hacen la traducción.
- El valor por defecto `id = 0`: al crear, aún no tenemos id; el servidor lo asigna y lo devuelve en la respuesta.

El método en el repositorio:

```

suspend fun crearPublicacion(titulo: String, texto: String):
ResultadoApi<Publicacion> {
    return try {
        val nueva = Publicacion(userId = 1, title = titulo, body = texto)
        val respuesta = servicio.crearPublicacion(nueva)
        if (respuesta.isSuccessful && respuesta.body() != null) {
            ResultadoApi.Exito(respuesta.body()!!)
        } else {
            ResultadoApi.Error(mensajeParaCodigo(respuesta.code()))
        }
    } catch (e: IOException) {
        ResultadoApi.Error("Sin conexión a internet")
    } catch (e: Exception) {
        ResultadoApi.Error("No se pudo publicar: ${e.message}")
    }
}

```

JSONPlaceholder simula la creación y devuelve la publicación con un id asignado (101). En una API real, ese objeto volvería a tu lista local.

Ahora el RecyclerView. Necesitamos tres piezas: el layout del item, el adaptador y la Activity que los une.

El layout del item (item_usuario.xml), una tarjeta sencilla:

```

<com.google.android.material.card.MaterialCardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="16dp">

        <TextView
            android:id="@+id/textoNombre"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="18sp"
            android:textStyle="bold" />

        <TextView
            android:id="@+id/textoCorreo"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="4dp" />

        <TextView
            android:id="@+id/textoCiudad"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="4dp" />

    </LinearLayout>
</com.google.android.material.card.MaterialCardView>

```

El adaptador:

```

class UsuarioAdaptador : RecyclerView.Adapter<UsuarioAdaptador.VistaUsuario>() {

    private var usuarios: List<Usuario> = emptyList()

    fun actualizar(nuevos: List<Usuario>) {
        usuarios = nuevos
        notifyDataSetChanged()
    }

    class VistaUsuario(vista: View) : RecyclerView.ViewHolder(vista) {
        val nombre: TextView = vista.findViewById(R.id.textoNombre)
        val correo: TextView = vista.findViewById(R.id.textoCorreo)
        val ciudad: TextView = vista.findViewById(R.id.textoCiudad)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): VistaUsuario {
        val vista = LayoutInflater.from(parent.context)
            .inflate(R.layout.item_usuario, parent, false)
        return VistaUsuario(vista)
    }

    override fun onBindViewHolder(holder: VistaUsuario, position: Int) {
        val usuario = usuarios[position]
        holder.nombre.text = usuario.name
        holder.correo.text = usuario.email
        holder.ciudad.text = "${usuario.address.city} - ${usuario.address.street}"
    }

    override fun getItemCount(): Int = usuarios.size
}

```

- `actualizar()`: reemplaza la lista y avisa al RecyclerView. En producción se usaría DiffUtil para animar cambios, pero para este ejemplo notifyDataSetChanged() es claro y suficiente.
- `onBindViewHolder`: aquí ocurre el mapeo de objeto a vista, incluyendo el acceso a objetos anidados (usuario.address.city). Gracias a Gson, esos objetos ya vienen contruidos.

Y la Activity completa que une red + corrutinas + RecyclerView + estados:

```

class ListaUsuariosActivity : AppCompatActivity() {

    private val repositorio = UsuarioRepositorio()
    private lateinit var adaptador: UsuarioAdaptador

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

        val recycler = findViewById<RecyclerView>(R.id.recyclerUsuarios)
        val progreso = findViewById<ProgressBar>(R.id.progreso)
        val textoError = findViewById<TextView>(R.id.textoError)
        val botonReintentar = findViewById<Button>(R.id.botonReintentar)

        adaptador = UsuarioAdaptador()
        recycler.layoutManager = LinearLayoutManager(this)
        recycler.adapter = adaptador

        fun cargar() {
            lifecycleScope.launch {
                progreso.visibility = View.VISIBLE
                textoError.visibility = View.GONE
                botonReintentar.visibility = View.GONE

                when (val resultado = repositorio.obtenerUsuarios()) {
                    is ResultadoApi.Exito -> {
                        progreso.visibility = View.GONE
                        adaptador.actualizar(resultado.datos)
                    }
                    is ResultadoApi.Error -> {
                        progreso.visibility = View.GONE
                        textoError.visibility = View.VISIBLE
                        textoError.text = resultado.mensaje
                        botonReintentar.visibility = View.VISIBLE
                    }
                    is ResultadoApi.Cargando -> Unit
                }
            }
        }

        botonReintentar.setOnClickListener { cargar() }
        cargar()
    }
}

```

Este es el esqueleto de casi cualquier pantalla de lista con datos remotos en Android: estados de carga visibles, error con mensaje humano y botón de reintentar, y la lista que se llena cuando los datos llegan. Si dominas este archivo, dominas el 80 por ciento de las pantallas de datos de cualquier app.

El layout `activity_lista_usuarios.xml` contiene el `RecyclerView`, el `ProgressBar` centrado, el `TextView` de error y el botón de reintentar, alternando visibilidad según el estado. Esa alternancia explícita de visibilidades es intencional y didáctica: en arquitecturas más avanzadas lo harías con un `ViewModel` y estados observados, pero el principio es exactamente el mismo.

Ejemplo 4: llamadas en paralelo con `async` y la versión Moshi

Dos técnicas que te faltan para redondear: lanzar varias llamadas a la vez y conocer la alternativa Moshi.

Llamadas en paralelo. Supón que necesitas el usuario Y sus publicaciones al mismo tiempo. Hacerlas en secuencia tarda la suma de ambas; en paralelo tarda lo que tarde la más lenta:

```
@GET("posts")
suspend fun obtenerPublicaciones(@Query("userId") userId: Int):
Response<List<Publicacion>>
```

```
lifecycleScope.launch {
    val repositorio = UsuarioRepositorio()

    // Las dos llamadas se lanzan al mismo tiempo
    val deferredUsuario = async { servicio.obtenerUsuario(1) }
    val deferredPosts = async { servicio.obtenerPublicaciones(1) }

    // await() suspende hasta que ambas terminen
    val respUsuario = deferredUsuario.await()
    val respPosts = deferredPosts.await()

    if (respUsuario.isSuccessful && respPosts.isSuccessful) {
        val usuario = respUsuario.body()
        val posts = respPosts.body()
        // Mostrar usuario con sus publicaciones
    }
}
```


- ``async`` lanza cada llamada inmediatamente sin esperar; ambas vuelan por la red al mismo tiempo.
- ``await()`` recoge cada resultado. Si una falla con excepción, el `await` la relanza y tu `try/catch` la maneja.
- Importante: `async` hereda el alcance de `lifecycleScope`, así que si el usuario sale de la pantalla, ambas se cancelan. Nunca uses `GlobalScope.async` para esto.

La versión Moshi. Si prefieres Moshi sobre Gson, cambian tres cosas: la dependencia, el convertidor y las anotaciones:

```
dependencies {
    implementation("com.squareup.retrofit2:retrofit:2.9.0")
    implementation("com.squareup.retrofit2:converter-moshi:2.9.0")
    implementation("com.squareup.moshi:moshi-kotlin:1.15.0")
}
```

```
val moshi = Moshi.Builder()
    .add(KotlinJsonAdapterFactory())
    .build()

val retrofit = Retrofit.Builder()
    .baseUrl(BASE_URL)
    .addConverterFactory(MoshiConverterFactory.create(moshi))
    .build()
```

Y en las data classes, cuando un campo JSON difiere del nombre Kotlin:

```
@JsonClass(generateAdapter = true)
data class Producto(
    val id: Int,
    @Json(name = "product_name") val nombre: String,
    val price: Double
)
```

- ``KotlinJsonAdapterFactory`` le enseña a Moshi a manejar data classes de Kotlin (nulables, valores por defecto).
- ``@Json(name = "...)`` es el equivalente Moshi de ``@SerializedName`` de Gson.
- Con ``generateAdapter = true``, Moshi genera el adaptador en compilación: más rápido en ejecución y errores detectados antes.

El resto del código (interface, repositorio, corrutinas) es idéntico uses Gson o Moshi. Por eso insistimos: aprende el patrón, no la biblioteca. El patrón es Retrofit + convertidor + corrutinas + repositorio; la biblioteca concreta es un detalle intercambiable.

Práctica

Ejercicio 1: lee el JSON crudo antes de modelarlo

Abre en tu navegador <https://jsonplaceholder.typicode.com/posts/1> y observa el JSON. Luego escribe la data class Kotlin que lo modele y un método Retrofit para obtenerlo.

Solución: el JSON es un objeto con `userId`, `id`, `title` y `body`. La data class:

```
data class Post(  
    val userId: Int,  
    val id: Int,  
    val title: String,  
    val body: String  
)
```

Y el método:

```
@GET("posts/{id}")  
suspend fun obtenerPost(@Path("id") id: Int): Response<Post>
```

La lección: siempre mira el JSON real antes de escribir la data class. Fíjate en si es objeto (`{}`) o arreglo (`[]`), en los nombres exactos de los campos y en los tipos (¿el `id` es número o texto?). Cinco minutos mirando el JSON ahorran una hora depurando un parseo.

Ejercicio 2: agrega un endpoint de comentarios

JSONPlaceholder tiene `GET /posts/{id}/comments` que devuelve los comentarios de una publicación. Agrega el endpoint a la interface, crea la data class `Comentario` con los campos que veas en el JSON real, y escribe la función del repositorio que los obtiene con manejo de errores.

Solución: primero revisas el JSON real en el navegador: cada comentario trae `postId`, `id`, `name`, `email` y `body`. La data class:

```
data class Comentario(
    val postId: Int,
    val id: Int,
    val name: String,
    val email: String,
    val body: String
)
```

El endpoint:

```
@GET("posts/{id}/comments")
suspend fun obtenerComentarios(@Path("id") postId: Int): Response<List<Comentario>>
```

Y en el repositorio, siguiendo el patrón del ejemplo 2:

```
suspend fun obtenerComentarios(postId: Int): ResultadoApi<List<Comentario>> {
    return try {
        val respuesta = servicio.obtenerComentarios(postId)
        if (respuesta.isSuccessful && respuesta.body() != null) {
            ResultadoApi.Exito(respuesta.body()!!)
        } else {
            ResultadoApi.Error(mensajeParaCodigo(respuesta.code()))
        }
    } catch (e: IOException) {
        ResultadoApi.Error("Sin conexión a internet")
    } catch (e: Exception) {
        ResultadoApi.Error("No se pudieron cargar los comentarios")
    }
}
```

Observa que el patrón se repite idéntico: el repositorio es una plantilla que aplicas a cada endpoint. Cuando un patrón se repite tanto, es señal de que lo entendiste bien.

Ejercicio 3: maneja el 404 de un usuario inexistente

Llama a `obtenerUsuario(9999)` (no existe en JSONPlaceholder) y verifica que tu código muestre "No se encontró la información solicitada" en lugar de tronar. Explica qué camino sigue el código.

Solución: el servidor responde con código 404 y cuerpo vacío. En el repositorio, `respuesta.isSuccessful` es false (404 no es 2xx), así que entra a la rama `else` y llama a `mensajeParaCodigo(404)`, que devuelve "No se encontró la información solicitada". Ese

mensaje viaja en `ResultadoApi.Error` hasta la UI, que lo muestra en el `TextView` de error con el botón de reintentar. En ningún momento se lanza una excepción: los códigos HTTP no exitosos NO son excepciones cuando usas `Response`, son respuestas normales que tú interpretas. Esa es exactamente la razón por la que preferimos `Response` sobre el retorno directo.

Ejercicio 4: pantalla de detalle con dos llamadas en paralelo

Crea una pantalla que muestre el nombre de un usuario y, debajo, la lista de sus publicaciones. Usa `async` para lanzar `obtenerUsuario(id)` y `obtenerPublicaciones(userId)` en paralelo, y muestra ambas cuando terminen. Incluye manejo de error si alguna falla.

Solución: el esqueleto con corrutinas:

```
lifecycleScope.launch {
    progreso.visibility = View.VISIBLE
    try {
        val defUsuario = async { servicio.obtenerUsuario(id) }
        val defPosts = async { servicio.obtenerPublicaciones(id) }

        val respU = defUsuario.await()
        val respP = defPosts.await()

        if (respU.isSuccessful && respP.isSuccessful) {
            nombreUsuario.text = respU.body()?.name
            adaptadorPosts.actualizar(respP.body() ?: emptyList())
        } else {
            mostrarError("No se pudo cargar la información")
        }
    } catch (e: IOException) {
        mostrarError("Sin conexión a internet")
    } catch (e: Exception) {
        mostrarError("Ocurrió un error inesperado")
    } finally {
        progreso.visibility = View.GONE
    }
}
```

El `finally` garantiza que el spinner se oculte pase lo que pase, un detalle que los principiantes olvidan y que deja la app "cargando eternamente" tras un error. Las dos llamadas vuelan en paralelo, así que la pantalla tarda lo que tarde la más lenta, no la suma.

Ejercicio 5: caché local primero, red después

Combina las Unidades 7 y 8: diseña (en pseudocódigo o Kotlin) el flujo de una pantalla que muestra la lista de usuarios guardando una copia en Room. Al abrir, muestra inmediatamente los datos guardados (si hay) y en segundo plano actualiza desde la red, refrescando la lista y la copia local.

Solución: el patrón "local primero":

```
lifecycleScope.launch {
    // 1. Mostrar lo que haya en local al instante
    val guardados = dao.obtenerTodos()
    if (guardados.isNotEmpty()) adaptador.actualizar(guardados)

    // 2. Actualizar desde la red en segundo plano
    when (val resultado = repositorio.obtenerUsuarios()) {
        is ResultadoApi.Exito -> {
            adaptador.actualizar(resultado.datos)
            // 3. Refrescar la copia local
            dao.borrarTodos()
            dao.insertarTodos(resultado.datos.map { it.aEntidad() })
        }
        is ResultadoApi.Error -> {
            // Sin red: nos quedamos con los datos locales y avisamos
            if (guardados.isEmpty()) mostrarError(resultado.mensaje)
            else Toast.makeText(this, "Mostrando datos guardados",
                Toast.LENGTH_SHORT).show()
        }
        is ResultadoApi.Cargando -> Unit
    }
}
```

Este es el patrón que usan las apps bien hechas: apertura instantánea, datos frescos cuando hay red, y funcionamiento degradado elegante sin conexión. Necesitas una función de mapeo (aEntidad()) entre el modelo de red y la entidad Room, porque es buena práctica mantenerlos separados: el JSON puede cambiar sin romper tu base local.

Ejercicio 6: publica un comentario (POST con @Body)

Agrega el endpoint POST /comments de JSONPlaceholder, que recibe un comentario (postId, name, email, body) y devuelve el creado con id. Escribe la data class, el endpoint y la llamada desde un formulario sencillo.

Solución:

```
data class ComentarioNuevo(
    val postId: Int,
    val name: String,
    val email: String,
    val body: String
)

@POST("comments")
suspend fun publicarComentario(@Body comentario: ComentarioNuevo):
Response<Comentario>
```

Y la llamada desde el formulario:

```
lifecycleScope.launch {
    val nuevo = ComentarioNuevo(
        postId = 1,
        name = campoNombre.text.toString(),
        email = campoCorreo.text.toString(),
        body = campoTexto.text.toString()
    )
    when (val r = repositorio.publicarComentario(nuevo)) {
        is ResultadoApi.Exito -> {
            Toast.makeText(this, "Comentario publicado con id ${r.datos.id}",
                Toast.LENGTH_SHORT).show()
            limpiarFormulario()
        }
        is ResultadoApi.Error -> Toast.makeText(this, r.mensaje,
            Toast.LENGTH_LONG).show()
        is ResultadoApi.Cargando -> Unit
    }
}
```

Valida los campos antes de enviar (no publiques comentarios vacíos) y deshabilita el botón mientras se envía para evitar doble POST por doble toque, que crearía comentarios duplicados en una API real.

Resumen de la unidad

- Una API es un contrato: URLs públicas que devuelven datos; REST es el estilo de diseño más común, basado en recursos, métodos HTTP y JSON.

- HTTP: la petición lleva método, URL, cabeceras y cuerpo; la respuesta trae un código de estado, cabeceras y cuerpo.
- Memoriza los códigos clave: 200 éxito, 201 creado, 400 petición inválida, 401 sin autenticación, 403 sin permiso, 404 no encontrado, 429 demasiadas peticiones, 500 error del servidor.
- Los métodos HTTP expresan la acción: GET leer, POST crear, PUT actualizar, DELETE borrar; las URLs llevan parámetros con @Query y variables de ruta con @Path.
- JSON es texto estructurado con objetos {} y arreglos []; míralo crudo en el navegador antes de modelarlo y usa tipos nulos para campos que pueden faltar.
- Retrofit convierte tu API en una interface Kotlin: métodos anotados (@GET, @POST, @Body, @Query, @Path) cuyas implementaciones genera la biblioteca.
- Configura Retrofit con base URL, OkHttpClient (timeouts, interceptores) y un convertidor (Gson o Moshi); no olvides el permiso INTERNET en el manifest.
- Gson usa @SerializedName y Moshi usa @Json para mapear nombres de campos JSON a propiedades Kotlin; modela solo los campos que vas a usar.
- Prefiere que los métodos devuelvan Response<T>: así manejas los códigos de estado explícitamente en lugar de recibir excepciones.
- Las corrutinas permiten red sin bloquear la UI: funciones suspend, launch para disparar, async/await para paralelo, lifecycleScope para atarlas a la pantalla.
- Nunca uses GlobalScope en una app; ata las corrutinas al ciclo de vida para que se cancelen solas si el usuario sale.
- Centraliza la red en un repositorio que devuelva un tipo sellado (Exito/Error/Cargando): la UI solo maneja estados limpios, jamás excepciones crudas.
- Maneja IOException (red), códigos HTTP (traducidos a mensajes humanos) y una rama genérica; el spinner siempre se oculta en finally.
- La autenticación con tokens viaja en la cabecera Authorization ("Bearer ..."); usa un interceptor de OkHttpClient para agregarla automáticamente.
- Patrón de pantalla profesional: carga visible, lista con RecyclerView, error con mensaje humano y botón de reintentar; considera caché local (Room) para apertura instantánea y modo sin conexión.

Extra: depuración de red como un profesional

La red falla de formas creativas y cada fallo tiene una firma distinta en el Logcat. Esta sección te enseña a leer esas firmas y a usar las herramientas correctas.

Los errores de red que verás en Logcat (y qué significan)

1. `android.os.NetworkOnMainThreadException`. Hiciste red en el hilo principal. Con Retrofit + suspend esto casi no ocurre porque la suspensión es automática, pero si alguna vez llamas a `execute()` en lugar de una función `suspend`, o usas `URLConnection` a mano en la UI, aquí está. Solución: corrutinas, siempre.
2. `java.net.UnknownHostException`: `Unable to resolve host`. No hay DNS o no hay internet: el teléfono no puede traducir el dominio a una dirección. En el emulador, verifica que tenga conexión; en físico, que el WiFi/datos funcionen. Tu app debe traducirlo a "Sin conexión a internet".
3. `java.net.SocketTimeoutException` / `ConnectTimeoutException`. El servidor tardó más que tu timeout en responder o en aceptar la conexión. Puede ser red lenta o servidor saturado. Aquí un reintento automático una vez suele ser razonable.
4. `javax.net.ssl.SSLHandshakeException`. Problema con el certificado HTTPS del servidor: expirado, autofirmado o con fecha del dispositivo mal configurada. En desarrollo contra servidores de prueba a veces se ve; en producción con APIs públicas reales es raro.
5. `Cleartext HTTP traffic not permitted`. Intentaste una URL `http://` en lugar de `https://`. Desde Android 9, el tráfico sin cifrar está bloqueado por defecto. Solución: usa `https` siempre. Si de verdad necesitas `http` en desarrollo, se configura en el manifest con `usesCleartextTraffic`, pero jamás en producción.
6. `Expected BEGIN_OBJECT but was BEGIN_ARRAY (Gson)`. Tu data class esperaba un objeto { pero el JSON trae un arreglo [. Revisa el JSON crudo: probablemente el endpoint devuelve una lista y tú modelaste un solo objeto, o viceversa.
7. `Expected a string but was NULL`. Un campo que declaraste no nutable vino null o ausente. Solución: hazlo nutable (`String?`) o dale valor por defecto en la data class.

El interceptor de logging: tu rayos X

El `HttpLoggingInterceptor` que configuramos en el ejemplo 1, en nivel `BODY`, imprime en Logcat cada petición y respuesta completas: método, URL exacta con parámetros, cabeceras y cuerpo. Cuando algo falla, este log responde las tres preguntas clave: ¿la URL está bien formada? ¿envié lo que creía enviar? ¿qué respondió exactamente el servidor?

El flujo de depuración profesional es: primero miras el log del interceptor; si la petición sale mal, el bug está en tu interface (@Query, @Path, @Body); si la petición sale bien pero la respuesta no se parsea, el bug está en tus data classes (nombres o tipos de campos); si la respuesta es un error HTTP, el bug está en tu lógica (autenticación, parámetros) o en el servidor.

Advertencia: el nivel BODY registra todo, incluyendo tokens de autenticación. Nunca dejes el interceptor en BODY en una versión de producción; usa Level.NONE o quítalo con una bandera de BuildConfig.DEBUG.

Probar el endpoint antes de programarlo

Regla de oro que ya mencionamos pero vale repetir con el procedimiento exacto:

1. Lee la documentación de la API: base URL, endpoints, parámetros, autenticación, formato de respuesta y límites de uso.
2. Abre el endpoint en el navegador para ver el JSON crudo (solo funciona con GETs sin autenticación).
3. Para POSTs o endpoints con autenticación, usa una herramienta como Postman o la extensión de navegador de tu preferencia: arma la petición, envíala y examina la respuesta.
4. Solo cuando la petición funciona en la herramienta, la traduces a Retrofit. Si algo falla después, sabes que el problema está en tu código Android, no en la API.

Este hábito separa al desarrollador que depura en minutos del que depura en horas.

Ejemplo 5: scroll infinito con paginación

Vamos a implementar la paginación que describimos en la teoría. La API de ejemplo será conceptual (cualquier API con ?page y ?limit sirve):

```
@GET("productos")
suspend fun obtenerProductos(
    @Query("page") pagina: Int,
    @Query("limit") limite: Int = 20
): Response<PaginaProductos>
```

En la Activity, detectamos el final del scroll y pedimos la página siguiente:

```
class ProductosActivity : AppCompatActivity() {
```

```

private val repositorio = ProductoRepositorio()
private lateinit var adaptador: ProductoAdaptador

private var paginaActual = 1
private var totalPaginas = Int.MAX_VALUE
private var cargando = false

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

    val recycler = findViewById<RecyclerView>(R.id.recyclerProductos)
    adaptador = ProductoAdaptador()
    recycler.layoutManager = LinearLayoutManager(this)
    recycler.adapter = adaptador

    recycler.addOnScrollListener(object : RecyclerView.OnScrollListener() {
        override fun onScrolled(rv: RecyclerView, dx: Int, dy: Int) {
            val lm = rv.layoutManager as LinearLayoutManager
            val visibles = lm.childCount
            val total = lm.itemCount
            val primero = lm.findFirstVisibleItemPosition()

            val cercaDelFinal = (visibles + primero) >= total - 5
            if (cercaDelFinal && !cargando && paginaActual < totalPaginas) {
                paginaActual++
                cargarPagina(paginaActual)
            }
        }
    })

    cargarPagina(1)
}

private fun cargarPagina(pagina: Int) {
    cargando = true
    lifecycleScope.launch {
        when (val r = repositorio.obtenerProductos(pagina)) {
            is ResultadoApi.Exito -> {
                totalPaginas = r.datos.totalPaginas
                adaptador.agregar(r.datos.productos)
            }
            is ResultadoApi.Error -> {
                Toast.makeText(this@ProductosActivity, r.mensaje,
                    Toast.LENGTH_SHORT).show()
            }
        }
    }
}

```

```

        paginaActual-- // permite reintentar la página fallida
    }
    is ResultadoApi.Cargando -> Unit
}
cargando = false
}
}
}
}

```

Puntos clave:

- El OnScrollListener calcula si estamos a 5 elementos del final; ese umbral evita esperar hasta el último píxel y da tiempo a que la siguiente página llegue.
- Las banderas cargando y paginaActual < totalPaginas evitan peticiones duplicadas o más allá del final. Sin la bandera cargando, un scroll rápido dispararía la misma página varias veces.
- adaptador.agregar() suma a la lista existente en lugar de reemplazarla (a diferencia del actualizar() del ejemplo 3).
- Si la página falla, decrementamos paginaActual para poder reintentarla; el usuario no pierde la posición del scroll.

Este patrón, con variantes, es el que usan Instagram, TikTok y cualquier app con feed infinito. La diferencia en producción es que se implementa con la biblioteca Paging 3 de Jetpack, que automatiza todo esto; pero entender la versión manual te hace entender qué automatiza Paging.

Ejemplo 6: subir un archivo con @Multipart

Enviar archivos (fotos de perfil, documentos) usa el tipo multipart, donde cada parte puede ser un archivo o un campo de texto:

```

@Multipart
@POST("perfil/foto")
suspend fun subirFoto(
    @Part foto: MultipartBody.Part,
    @Part("descripcion") descripcion: RequestBody
): Response<RespuestaSubida>

```

Y la llamada:

```

fun prepararArchivo(uri: Uri): MultipartBody.Part {
    val archivo = File(obtenerRuta(uri))
    val cuerpo = archivo.asRequestBody("image/jpeg".toMediaType())
    return MultipartBody.Part.createFormData("foto", archivo.name, cuerpo)
}

lifecycleScope.launch {
    val parte = prepararArchivo(uriFoto)
    val desc = "Mi foto de perfil".toRequestBody("text/plain".toMediaType())
    when (val r = repositorio.subirFoto(parte, desc)) {
        is ResultadoApi.Exito -> Toast.makeText(this, "Foto subida",
        Toast.LENGTH_SHORT).show()
        is ResultadoApi.Error -> Toast.makeText(this, r.mensaje,
        Toast.LENGTH_LONG).show()
        is ResultadoApi.Cargando -> Unit
    }
}

```

- @Multipart indica que la petición lleva partes; @Part marca cada una. createFormData recibe el nombre del campo que espera el servidor ("foto"), el nombre del archivo y el cuerpo.
- El MediaType ("image/jpeg") le dice al servidor qué tipo de archivo es. Usar el tipo correcto importa: algunos servidores rechazan tipos inesperados.
- Para archivos grandes, considera mostrar progreso de subida y permitir cancelación; Retrofit/OkHttp lo soportan envolviendo el RequestBody, aunque eso ya es tema avanzado.

Glosario rápido de la unidad

- API: interfaz pública de un servidor; contrato de URLs y formatos para intercambiar datos.
- REST: estilo de diseño de APIs basado en recursos, URLs y métodos HTTP estándar.
- Endpoint: una URL concreta de la API que representa una operación, por ejemplo GET /users.
- JSON: formato de texto ligero para datos estructurados; el idioma de las APIs REST.
- Método HTTP: verbo de la petición: GET (leer), POST (crear), PUT (actualizar), DELETE (borrar).
- Código de estado: número en la respuesta que indica el resultado (2xx éxito, 4xx error del cliente, 5xx error del servidor).

- Query param: par clave=valor en la URL después de ?, usado para filtros y paginación.
- Path param: variable dentro de la ruta de la URL, como el id en /users/42.
- Cabecera (header): metadato de la petición o respuesta, como Authorization o Content-Type.
- Retrofit: biblioteca estándar de Android para consumir APIs REST de forma declarativa.
- Convertidor: componente de Retrofit (Gson/Moshi) que traduce JSON a objetos Kotlin y viceversa.
- Corrutina: mecanismo de Kotlin para trabajo asíncrono que se escribe como código secuencial sin bloquear hilos.
- Función suspend: función que puede pausarse y reanudarse; base de las corrutinas.
- Dispatcher: determina en qué hilo se ejecuta una corrutina (Main para UI, IO para red/disco).
- lifecycleScope: alcance de corrutina ligado a una Activity/Fragment; se cancela solo al salir.
- Repositorio: clase que centraliza el acceso a datos (red, base local) y expone resultados limpios a la UI.
- Token: credencial temporal que autentica las llamadas a una API sin reenviar la contraseña.
- Interceptor: componente de OkHttp que inspecciona o modifica cada petición/respuesta (logging, autenticación).
- Paginación: técnica de dividir resultados grandes en páginas que se piden una por una.
- Multipart: formato de petición para enviar archivos junto con campos de texto.

Con esto cerramos la Unidad 8 y tu parte del libro. Tu app ya no vive encerrada en el teléfono: guarda datos localmente con criterio profesional y conversa con el mundo a través de APIs, con corrutinas que mantienen la interfaz fluida y un manejo de errores que trata bien al usuario. Lo que sigue en tu camino es arquitectura (ViewModels, flujos de datos, inyección de dependencias), pero la base que acabas de construir - persistencia + red - es el núcleo de casi toda aplicación Android que existe.

Manejo avanzado: reintentos, caché HTTP y estrategia offline

Para llevar tu manejo de red al nivel profesional, hay tres técnicas que distinguen una app robusta: reintentar con inteligencia, aprovechar el caché HTTP y diseñar la experiencia sin conexión. Vamos a verlas con código.

Reintentos con espera exponencial

No todos los errores merecen un reintento, pero los transitorios sí: un timeout aislado, un 503 momentáneo, una ráfaga de pérdida de paquetes. La técnica correcta es el backoff exponencial: esperas 1 segundo, luego 2, luego 4. Así no saturas al servidor que ya está sufriendo.

```
suspend fun <T> reintentar(
    maxIntentos: Int = 3,
    esperaInicialMs: Long = 1000,
    bloque: suspend () -> ResultadoApi<T>
): ResultadoApi<T> {
    var espera = esperaInicialMs
    repeat(maxIntentos) { intento ->
        val resultado = bloque()
        if (resultado is ResultadoApi.Exito) return resultado

        val reintentable = resultado is ResultadoApi.Error &&
            (resultado.mensaje.contains("internet") ||
            resultado.mensaje.contains("más tarde"))

        if (!reintentable || intento == maxIntentos - 1) return resultado

        delay(espera)
        espera *= 2
    }
    return ResultadoApi.Error("No se pudo completar la operación")
}
```

- Es una función genérica: sirve para cualquier llamada que devuelva ResultadoApi.
- Solo reintentará errores marcados como reintentables (problemas de red o servidor saturado). Un 404 o un 401 jamás se arreglan reintentando, así que se devuelven de inmediato.
- delay() es la suspensión amable de las corrutinas: no bloquea ningún hilo mientras espera.
- Úsala así: `val resultado = reintentar { repositorio.obtenerUsuarios() }`. Tres líneas en el llamador, robustez real.

Caché HTTP con OkHttp

OkHttp puede cachear respuestas automáticamente siguiendo las cabeceras que envía el servidor (Cache-Control). Para datos que cambian poco, esto significa respuestas instantáneas y menos consumo de datos del usuario:

```
val cache = Cache(File(context.cacheDir, "http_cache"), 10L * 1024 * 1024) // 10 MB

val cliente = OkHttpClient.Builder()
    .cache(cache)
    .addInterceptor { chain ->
        var request = chain.request()
        request = request.newBuilder()
            .header("Cache-Control", "public, max-age=60")
            .build()
        chain.proceed(request)
    }
    .build()
```

Con esto, las respuestas GET se guardan 10 MB en disco y se reutilizan durante 60 segundos sin tocar la red. Para un catálogo o una lista de noticias, la diferencia en velocidad de apertura es notable. El caché HTTP no reemplaza a Room (el caché es opaco y el servidor manda), pero es una primera línea de defensa barata y efectiva.

Diseñando para el modo sin conexión

La mejor app no es la que nunca falla la red, sino la que falla con gracia. La estrategia completa combina lo que ya sabes:

1. Detecta la conectividad antes de la llamada para dar mensajes precisos. Android ofrece `ConnectivityManager` con `NetworkCapabilities`; un helper `isOnline()` evita mostrar "error del servidor" cuando el problema es el WiFi apagado.
2. Muestra datos locales primero (Unidad 7 + ejercicio 5 de esta unidad): la app abre con lo último que guardó, y un indicador sutil dice "actualizando..." mientras la red trabaja.
3. Las acciones del usuario sin red se encolan, no se pierden. El patrón profesional: guardas la acción pendiente en Room con una bandera "sincronizado = false", y cuando vuelve la red, un Worker (`WorkManager`) las envía en orden. Así el usuario publica su comentario en el metro sin señal y se envía solo al salir a la superficie.
4. Sé honesto en la UI: un banner "Sin conexión: mostrando datos guardados" genera más confianza que una pantalla que finge estar al día.

API keys: autenticación simple para servicios públicos

Muchas APIs (clima, mapas, noticias) no usan login de usuario sino una API key: una cadena que identifica tu app. Viaja como query param (?apikey=...) o como cabecera (X-API-Key). En Retrofit:

```
@GET("weather")
suspend fun obtenerClima(
    @Query("q") ciudad: String,
    @Query("appid") apiKey: String = BuildConfig.WEATHER_API_KEY
): Response<Clima>
```

Con un valor por defecto tomado de BuildConfig, cada llamada incluye la key sin que tengas que recordarla. Y recuerda la buena práctica: la key vive en local.properties (no versionado), se inyecta a BuildConfig en el build.gradle, y jamás se hardcodea en el código fuente que subes a git. Una key expuesta en un repositorio público será abusada por bots en cuestión de horas, y la cuenta es tuya.

Lista de verificación antes de publicar una app con red

Cierra la unidad con este checklist, el mismo que usan los equipos profesionales en la revisión previa al lanzamiento:

- Todas las llamadas van en corrutinas, ninguna en el hilo principal.
- Timeouts configurados en el OkHttpClient.
- El interceptor de logging en BODY solo en builds de debug.
- Manejo de los tres estados en cada pantalla: cargando, éxito, error con reintento.
- Mensajes de error humanos, sin códigos HTTP ni stack traces en la UI.
- Permiso INTERNET declarado; nada de tráfico cleartext en producción.
- API keys fuera del código versionado.
- Probado en modo avión: la app informa y degrada con elegancia.
- Probado con red lenta (el emulador permite simularlo): no hay ANRs ni spinners eternos.
- Respuestas grandes paginadas; listas largas con scroll infinito o Paging.
- Datos críticos con copia local para apertura instantánea.

Si tu app pasa esta lista, está lista para usuarios reales. Y con eso, tu parte del libro queda completa.

UNIDAD 9: Permisos y funciones del dispositivo

En las unidades anteriores aprendiste a construir pantallas, navegar entre ellas, cargar datos de internet y guardarlos en el teléfono. Ya sabes hacer una app que se ve bien y funciona. Pero hay un salto importante que separa una app de juguete de una app de verdad: interactuar con el mundo real. Eso significa pedirle al usuario que te deje usar su cámara, su galería de fotos, su ubicación y mandarle notificaciones cuando algo importante pase.

Piensa en las apps que usas todos los días. WhatsApp no sería WhatsApp si no pudiera abrir tu cámara para tomar una foto rápida o entrar a tu galería para mandarle una imagen a tu mamá. Google Maps no serviría de nada si no pudiera saber dónde estás. Uber no funcionaría sin tu ubicación en tiempo real. Y las notificaciones son el puente que mantiene viva la relación entre tu app y el usuario: sin ellas, tu app muere en silencio después de cerrarse, nadie la vuelve a abrir.

En esta unidad vas a dominar todo eso. Son tres clases que se conectan entre sí de forma natural. En la clase 25 vas a entender el sistema de permisos de Android, que es la puerta de entrada a todo lo demás. Android protege al usuario: ninguna app puede acceder a la cámara, los contactos o la ubicación sin pedir permiso primero, y tu trabajo como desarrollador es pedirlo en el momento correcto, con las palabras correctas, y saber qué hacer cuando el usuario te dice que no. Eso último es lo que separa a un desarrollador principiante de uno profesional: el principiante asume que siempre le van a decir que sí; el profesional diseña su app para que siga funcionando aunque le digan que no.

En la clase 26 vas a trabajar con la cámara y la galería. Vas a aprender a lanzar la cámara del sistema para tomar una foto, a recibir esa foto de vuelta en tu app y a mostrarla en pantalla. También vas a aprender a abrir la galería para que el usuario elija una imagen existente. Aquí hay un detalle técnico importante que vas a dominar: las fotos pueden ser enormes, y si las cargas completas en memoria tu app se va a trabar o se va a cerrar con un error de falta de memoria. Vas a aprender a escalarlas antes de mostrarlas.

En la clase 27 vas a entrar al mundo de las notificaciones y la ubicación por GPS. Las notificaciones tienen su propio sistema de canales desde Android 8, y si no creas bien el canal, tus notificaciones simplemente no aparecen y no vas a saber por qué. La ubicación, por su parte, es el permiso más delicado de todos: los usuarios desconfían cuando una app les pide su ubicación, así que vas a aprender a pedirla con contexto y a usarla de forma eficiente sin drenar la batería.

Esta unidad es práctica por naturaleza. Vas a escribir mucho código: launchers de permisos, intents para la cámara, manejo de archivos temporales, creación de canales de notificación,

lectura de coordenadas GPS. Al final vas a poder construir apps que tocan el hardware del teléfono de forma segura y profesional, que es exactamente lo que esperan los usuarios de una app sería en 2026.

Una nota importante sobre las versiones: en este libro usamos Kotlin moderno (1.9+) y las APIs actuales. La forma vieja de pedir permisos con `onRequestPermissionsResult` y `startActivityForResult` ya esta deprecada; nosotros vamos a usar el sistema moderno de `ActivityResultLauncher` con `registerForActivityResult`, que es más limpio, más seguro y es lo que se usa en la industria hoy. Si buscas tutoriales viejos en internet y ves esos metodos, ya sabes: son codigo del pasado.

Explicación detallada

Clase 25: Permisos en tiempo de ejecucion

Imagina que llegas a una casa ajena y, sin decir nada, abres el refrigerador y te sirves un vaso de leche. Eso es exactamente lo que sería una app que accede a tu cámara o tu ubicacion sin avisar. El sistema de permisos de Android existe para que eso no pase: cada app vive en su propia "caja" aislada y solo puede salir de ella con el permiso explicito del usuario.

Hay dos tipos de permisos. Los permisos normales, como el acceso a internet o a la vibracion, se otorgan automaticamente cuando instalas la app; el usuario ni se entera. Los permisos peligrosos, en cambio, protegen datos privados o funciones sensibles del dispositivo: la cámara, la ubicacion, los contactos, el microfono, el almacenamiento, los sensores corporales. Estos se piden en tiempo de ejecucion, es decir, mientras la app esta corriendo, en el momento en que los necesitas.

La regla de oro es pedir el permiso en contexto. Si tu app de recetas pide la ubicacion al abrirse, el usuario se va a asustar y va a decir que no, porque no entiende para que la quieres. Pero si el usuario toca el boton "Buscar recetas cerca de mi" y ahi le pides la ubicacion, la probabilidad de que diga que si se dispara, porque el permiso tiene sentido en ese momento. Google incluso lo exige en sus politicas de Play Store: el permiso debe pedirse en el flujo donde se usa la funcion.

El flujo moderno para pedir un permiso tiene tres piezas. Primera: declarar el permiso en el `AndroidManifest.xml`, porque si no esta declarado el sistema lo ignora y te lo niega en silencio. Segunda: registrar un `ActivityResultLauncher` con `registerForActivityResult` y el contrato `ActivityResultContracts.RequestPermission()`, que es el encargado de mostrar el dialogo del sistema y entregarte la respuesta (true o false). Tercera: lanzar la solicitud con

``launcher.launch(Manifest.permission.CAMERA)`` en el momento adecuado, y reaccionar a la respuesta.

Pero aquí viene la parte que muchos tutoriales se saltan: que hacer cuando el usuario dice que no. Hay tres escenarios. Uno: el usuario dice que no por primera vez. Tu app debe seguir funcionando y, si la función es importante, explicarle por que necesitas el permiso y ofrecerle intentarlo de nuevo. Dos: el usuario dice que no y marca "No volver a preguntar" (o lo niega dos veces, que en Android moderno equivale a un rechazo permanente). En este caso el diálogo del sistema ya no aparece; la única salida es mandarlo a los ajustes de la app con un Intent a ``Settings.ACTION_APPLICATION_DETAILS_SETTINGS`` para que lo active manualmente. Tres: el usuario ya había dado el permiso antes; entonces simplemente usas la función. Tu código debe cubrir los tres casos con elegancia.

Para saber en que caso estas, usas ``shouldShowRequestPermissionRationale()``. Este método te dice si deberías mostrarle al usuario una explicación antes de volver a pedir el permiso. Devuelve true cuando el usuario ya te negó el permiso una vez (pero no de forma permanente): es tu señal de que debes explicarle el por que con tus propias palabras antes de lanzar el diálogo del sistema otra vez. Si devuelve false, puede ser que sea la primera vez que lo pides o que el rechazo sea permanente; la diferencia la marcas tu llevando control o simplemente intentando pedirlo y viendo que el sistema no muestra nada.

Un error común gravísimo es pedir todos los permisos juntos al iniciar la app, como una ametralladora de diálogos. El usuario se siente atacado y niega todo. Otro error común es no declarar el permiso en el manifest y pasar horas depurando por que el launcher siempre devuelve false. Y otro más: asumir que el permiso concedido dura para siempre. En Android 11 en adelante el usuario puede dar un permiso "solo esta vez", y en cualquier momento puede revocarlo desde los ajustes; tu app debe verificar el permiso cada vez que va a usar la función, no solo la primera vez.

Buena práctica profesional: crea una función de ayuda como ``tienePermiso()`` que verifique con ``ContextCompat.checkSelfPermission()`` antes de cada uso, y centraliza la lógica de permisos en un solo lugar de tu Activity o Fragment. Así evitas código repetido y errores.

Clase 26: Cámara y galería

Tomar una foto desde tu app no significa programar una cámara desde cero. Sería una locura: tendrías que manejar el hardware, el enfoque, el flash, la rotación. Lo inteligente es delegar: le pides a la app de cámara del sistema que tome la foto por ti mediante un Intent, y ella te devuelve la imagen. Es como pedirle a un fotógrafo profesional que tome la foto y te la entregue: tú no necesitas saber de fotografía, solo recibir el resultado.

Hay dos formas de tomar una foto. La forma simple usa el contrato `ActivityResultContracts.TakePicturePreview()`, que te devuelve directamente un `Bitmap` pequeño (una miniatura). Es perfecta para fotos de perfil o avatares donde no necesitas alta resolución. La forma completa usa `ActivityResultContracts.TakePicture()`, que guarda la foto en alta resolución en un archivo que tú le indicas mediante un `Uri`. Para esto necesitas un `FileProvider`, que es un componente que genera URIs seguras para compartir archivos con otras apps; sin el, Android te bloquea el acceso al archivo por seguridad y la cámara falla.

El `FileProvider` suena complicado pero es mecánico: lo declaras en el manifest con una autoridad (normalmente `tu.paquete.fileprovider`), creas un archivo XML de rutas (`res/xml/file_paths.xml`) que dice que carpeta puede compartir, y en código generas el Uri con `FileProvider.getUriForFile()`. Una vez que lo haces dos veces, lo haces de memoria.

Elegir una imagen de la galería es más simple: usas el contrato `ActivityResultContracts.GetContent()` con el tipo MIME `"image/*"` y el sistema te muestra el selector de archivos. Tu recibes un `Uri` que apunta a la imagen elegida y la muestras en un `ImageView` con `setImageURI()`. Parece magia, pero hay un detalle: ese Uri es temporal; si necesitas la imagen después de cerrar la app, debes copiarla a tu almacenamiento interno.

Y aquí llegamos al tema crítico: la memoria. Una foto moderna puede pesar 12 megapíxeles, y cargarla completa como `Bitmap` consume decenas de megabytes de RAM. Si tu app carga dos o tres fotos así, el sistema la mata con un `OutOfMemoryError`. La solución es escalar la imagen antes de cargarla: primero lees solo sus dimensiones con `BitmapFactory.Options` y `inJustDecodeBounds = true` (esto no carga la imagen, solo mide), calculas un factor de escala, y luego la cargas ya reducida con `inSampleSize`. Es una técnica estándar que todo desarrollador Android debe dominar. En la práctica, para mostrar en un `ImageView` de 500 píxeles no necesitas la foto completa de 4000 píxeles.

Errores comunes en esta clase: olvidar el permiso de cámara antes de lanzar el Intent (la app se cierra), no configurar el `FileProvider` (la cámara devuelve error), y cargar el `Bitmap` completo sin escalar (la app se traba). Otro error sutil: en Android 13 en adelante, el permiso para leer imágenes ya no es `READ_EXTERNAL_STORAGE` sino `READ_MEDIA_IMAGES`; si pides el viejo en Android 13+, el sistema lo ignora. Siempre verifica la versión con `Build.VERSION.SDK_INT` cuando trabajes con permisos que cambiaron entre versiones.

Clase 27: Notificaciones y ubicación GPS

Las notificaciones son mensajes que tu app le deja al usuario fuera de la app: en la barra de estado, en la pantalla de bloqueo, como banner flotante. Pero desde Android 8 (API 26), todas las notificaciones deben pertenecer a un canal de notificación. Un canal es una categoría con

su propio nombre, descripción, importancia y sonido; el usuario puede silenciar o bloquear canales individuales desde los ajustes. Si intentas mostrar una notificación sin crear su canal primero, en Android 8+ simplemente no aparece y no hay mensaje de error. Es el bug fantasma más común de los principiantes.

Crear un canal es sencillo: construyes un `NotificationChannel` con un ID único, un nombre legible y un nivel de importancia (`IMPORTANCE_DEFAULT`, `IMPORTANCE_HIGH`, etc.), y lo registras con el `NotificationManager`. Esto se hace una sola vez, típicamente al iniciar la app. Luego, para mostrar la notificación, usas `NotificationCompat.Builder` indicando el ID del canal, el icono pequeño (obligatorio), el título y el texto, y la lanzas con `notify()`.

Desde Android 13 (API 33), mostrar notificaciones también requiere un permiso en tiempo de ejecución: `POST_NOTIFICATIONS`. Así que tu app moderna debe pedirlo como cualquier otro permiso peligroso, idealmente la primera vez que va a mostrar una notificación importante, no al abrir la app. Si el usuario lo niega, tu app sigue funcionando; simplemente no muestra notificaciones.

La ubicación por GPS es el permiso más sensible. Hay dos niveles: ubicación aproximada y ubicación precisa (`ACCESS_COARSE_LOCATION` y `ACCESS_FINE_LOCATION`). En Android 10+ existe además el acceso en segundo plano, que requiere un permiso separado y una justificación muy sólida ante Google Play. Para esta clase nos quedamos con la ubicación en primer plano, que cubre el 95 por ciento de los casos.

Para obtener la ubicación se usa la API de Google Play Services: `FusedLocationProviderClient`, que combina GPS, WiFi y redes celulares para darte la mejor ubicación disponible ahorrando batería. Puedes pedir la última ubicación conocida con `getLastLocation` (rápido, puede ser nula o vieja) o suscribirte a actualizaciones periódicas con `requestLocationUpdates` (más preciso, consume más batería). La regla de oro: pide actualizaciones solo mientras las necesitas y cancela con `removeLocationUpdates` cuando termines; una app que deja el GPS encendido en segundo plano drena la batería y el usuario la desinstala.

Un error común es pedir la ubicación sin verificar que el GPS esté activado en el dispositivo; si está apagado, `getLastLocation` devuelve null y muchos principiantes creen que su código está mal. Otro error es no manejar el caso de permiso denegado y dejar la pantalla en blanco. Buena práctica: siempre muestra un estado alternativo ("Activa tu ubicación para ver lugares cercanos") en lugar de una pantalla vacía o un crash.

El ciclo de vida de los permisos: revocacion y "solo esta vez"

Un permiso concedido no es un contrato permanente. El usuario puede entrar a los ajustes de tu app y quitarte el permiso cuando quiera, incluso con la app instalada y en uso. Además, desde Android 11 existe la opcion "Solo esta vez": el permiso dura mientras la app está en uso y se revoca automaticamente cuando se cierra. Y desde Android 11 también hay auto-revoco: si el usuario no abre tu app en meses, el sistema le quita los permisos solo.

Que significa esto para tu codigo? Que nunca debes guardar en una variable algo como ``yaTengoPermiso = true`` y confiar en eso para siempre. La verificacion con ``checkSelfPermission()`` debe hacerse justo antes de usar la funcion, cada vez. Piensa en el permiso como en la llave de un hotel: te la dan al entrar, pero la tienes que mostrar cada vez que quieres usar la alberca, porque te la pueden haber desactivado.

Otro detalle fino: cuando pides un permiso de ubicacion en Android 10+, el dialogo del sistema ofrece "Solo esta vez" como opcion. Tu codigo recibe ``true`` igual que si hubiera sido permanente; no hay forma de distinguirlo desde el callback. Por eso la verificacion previa a cada uso es la unica estrategia segura.

Scoped Storage: guardar fotos donde el usuario las encuentre

Cuando tomas una foto con ``TakePicture()`` y la guardas en ``getExternalFilesDir()``, esa foto vive en una carpeta privada: si el usuario desinstala tu app, la foto se borra. A veces eso es lo que quieres (fotos temporales de perfil). Pero si tu app es una cámara o un editor, el usuario espera encontrar sus fotos en la galeria del teléfono aunque desinstale la app.

Para eso existe MediaStore: el almacen público de medios del sistema. Guardar ahi no requiere permiso de escritura en Android 10+ (scoped storage): el sistema te deja escribir tus propios archivos sin permiso especial. El patron es así:

```
// Guarda un Bitmap en la galeria publica (MediaStore) sin pedir permisos extra.
fun guardarEnGaleria(context: Context, bitmap: Bitmap, nombre: String): Uri? {
    val valores = ContentValues().apply {
        put(MediaStore.Images.Media.DISPLAY_NAME, nombre)
        put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg")
        // En Android 10+, RELATIVE_PATH decide la carpeta publica.
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
            put(MediaStore.Images.Media.RELATIVE_PATH, "Pictures/MiApp")
            put(MediaStore.Images.Media.IS_PENDING, 1)
        }
    }

    val coleccion = MediaStore.Images.Media.getContentUri(MediaStore.VOLUME_EXTERNAL_PRIMARY)
    val uri = context.contentResolver.insert(coleccion, valores) ?: return null
    context.contentResolver.openOutputStream(uri)?.use { salida ->
        bitmap.compress(Bitmap.CompressFormat.JPEG, 90, salida)
    }
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.Q) {
        valores.clear()
        valores.put(MediaStore.Images.Media.IS_PENDING, 0)
        context.contentResolver.update(uri, valores, null, null)
    }
    return uri
}
```

`IS\_PENDING` es un detalle elegante: marca el archivo como "en construccion" para que la galeria no lo muestre a medias mientras lo escribes; al ponerlo en 0 lo liberas. Este es el codigo que usan las apps de cámara serias.

Geocodificacion: de coordenadas a direccion legible

Mostrar "19.4326, -99.1332" no le dice nada al usuario. La geocodificacion inversa convierte coordenadas en una direccion legible ("Av. Juarez 123, Ciudad de Mexico"). Android trae `Geocoder` para eso, pero ojo: puede tardar y puede fallar sin internet, así que siempre va en un hilo secundario:

```

// Convierte coordenadas en direccion. Llamalo desde un hilo secundario.
fun obtenerDireccion(context: Context, lat: Double, lon: Double): String {
    return try {
        val geocoder = Geocoder(context, Locale.getDefault())
        val direcciones = geocoder.getFromLocation(lat, lon, 1)
        if (!direcciones.isNullOrEmpty()) {
            val d = direcciones[0]
            // locality = ciudad, countryName = pais, getAddressLine(0) = direccion
            completa.
            d.getAddressLine(0) ?: "Direccion no disponible"
        } else {
            "Sin resultados para esas coordenadas"
        }
    } catch (e: IOException) {
        "Sin conexion para buscar la direccion"
    }
}

// Uso correcto desde una corrutina:
lifecycleScope.launch(Dispatchers.IO) {
    val direccion = obtenerDireccion(this@MapaActivity, lat, lon)
    withContext(Dispatchers.Main) {
        textViewDireccion.text = direccion
    }
}

```

Nunca llames a `getFromLocation` en el hilo principal: bloquea y tu app se congela. La corrutina con `Dispatchers.IO` es el patron moderno. Y siempre ten un texto alternativo para cuando no hay internet: el usuario en un pueblo sin señal también merece una app que no se trabe.

Niveles de importancia y comportamiento de las notificaciones

El nivel de importancia del canal decide cómo se comporta la notificación, y el usuario puede cambiarlo después. Los niveles que más vas a usar: `IMPORTANCE\_LOW` (aparece en la barra pero sin sonido ni banner: ideal para avisos pasivos como "descarga completada"), `IMPORTANCE\_DEFAULT` (sonido y aparece en la barra: el caballo de batalla), `IMPORTANCE\_HIGH` (banner flotante que interrumpe: solo para cosas urgentes como mensajes o alarmas).

Error común de principiante: poner todo en HIGH "para que el usuario si lo vea". El resultado es el contrario: el usuario se harta, silencia tu canal o desinstala la app. Reserva HIGH para lo

que de verdad lo amerita. Piensa en la importancia como en el tono de voz: si gritas todo el tiempo, nadie te escucha cuando de verdad importa.

Otro detalle: la importancia se fija al crear el canal y el sistema la recuerda. Si en una actualización de tu app cambias la importancia en código, los usuarios que ya tenían el canal creado conservan la anterior. Para cambios de comportamiento, crea un canal nuevo con otro ID.

Buenas prácticas de UX al pedir permisos (lo que Google evalúa)

Google Play revisa como pides los permisos, sobre todo los sensibles como ubicación. Estas son las reglas no escritas que separan una app aprobada de una rechazada o mal calificada:

Uno: pide en contexto, nunca al inicio. El botón "Buscar tiendas cercanas" es el momento de pedir ubicación; la pantalla de bienvenida no.

Dos: explica el beneficio, no la tecnología. "Para mostrarte las tiendas más cercanas" convierte; "Necesito ACCESS_FINE_LOCATION" asusta. El usuario no sabe qué es un permiso técnico, sabe lo que quiere lograr.

Tres: no bloques la app si te dicen que no. La función asociada se desactiva con elegancia y el resto sigue funcionando. Una app que muestra una pantalla negra porque le negaron un permiso es una app mal diseñada.

Cuatro: no pidas permisos que no usas. Si tu app pide la ubicación pero en ningún flujo la usa, Google Play puede rechazar tu publicación. Cada permiso declarado debe tener una función visible que lo justifique.

Cinco: para la ubicación en segundo plano hay reglas extra duras: necesitas justificarlo en la consola de Play con un video mostrando por qué lo necesitas. Si tu app funciona con ubicación solo en primer plano, ni lo pidas en segundo plano.

Notificaciones programadas: que tu app avise aunque esté cerrada

Hasta ahora tus notificaciones se muestran cuando el usuario toca un botón con la app abierta. Pero el caso real más valioso es el contrario: que la app avise sola a las 8 de la mañana aunque lleve días cerrada. Para eso Android ofrece dos herramientas. `AlarmManager` programa un disparo exacto a una hora (ideal para recordatorios puntuales como "toma tu medicamento a las 8:00"). `WorkManager` programa trabajo diferido que el sistema ejecuta cuando convenga (ideal para sincronizaciones periódicas que no necesitan hora exacta).

Para un recordatorio diario con AlarmManager el patrón es este:

```

// Programa una notificacion diaria a la hora indicada.
fun programarRecordatorioDiario(context: Context, hora: Int, minuto: Int) {
    val intent = Intent(context, RecordatorioReceiver::class.java)
    val pending = PendingIntent.getBroadcast(
        context, 0, intent,
        PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE
    )
    val alarmManager = context.getSystemService(Context.ALARM_SERVICE) as
AlarmManager
    val calendario = Calendar.getInstance().apply {
        set(Calendar.HOUR_OF_DAY, hora)
        set(Calendar.MINUTE, minuto)
        set(Calendar.SECOND, 0)
        // Si la hora de hoy ya paso, programa para manana.
        if (timeInMillis <= System.currentTimeMillis()) add(Calendar.DAY_OF_YEAR,
1)
    }
    // setRepeating esta limitado por el sistema en versiones modernas;
    // para precision real se reprograma cada vez que suena.
    alarmManager.setInexactRepeating(
        AlarmManager.RTC_WAKEUP, calendario.timeInMillis,
        AlarmManager.INTERVAL_DAY, pending
    )
}

// El receiver se despierta a la hora programada y muestra la notificacion.
class RecordatorioReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        mostrarNotificacion(context, "Recordatorio", "Es hora de tu actividad
diaria")
    }
}

```

El receiver se declara en el manifest:

```

<receiver android:name=".RecordatorioReceiver" android:exported="false" />

```

Dos advertencias profesionales. Primera: en Android 12+, las alarmas exactas (`setExact`) requieren el permiso `SCHEDULE_EXACT_ALARM`, que el usuario puede revocar; por eso el ejemplo usa `setInexactRepeating`, que no lo necesita y es suficiente para recordatorios diarios. Segunda: si el usuario reinicia el teléfono, las alarmas se borran; las apps serias escuchan `BOOT_COMPLETED` y reprograman. Para sincronización de datos en segundo plano (subir fotos pendientes, actualizar cache), prefiere `WorkManager`, que sobrevive reinicios y respeta el modo de ahorro de batería (Doze).

Caso real guiado: flujo "foto de perfil" en una app profesional

Vamos a juntar todo lo de la unidad en un solo flujo narrado, como lo haría una app real de red social. El usuario toca su avatar. La app verifica el permiso de cámara. Si no lo tiene y es primera vez, lanza el diálogo del sistema directamente. Si el usuario lo niega, la próxima vez que toque el avatar la app muestra su propio diálogo: "Usamos la cámara solo para tu foto de perfil, no guardamos nada en servidores". Si lo niega dos veces, el tercer toque abre un diálogo que dice "Activaste el bloqueo permanente" con botón a ajustes.

Con el permiso concedido, la app ofrece dos opciones: "Tomar foto" o "Elegir de galería". Si elige cámara, se crea el archivo temporal con `FileProvider` y se lanza `TakePicture()`. Al volver, la foto se carga escalada a 800 píxeles, se corrige la rotación con EXIF y se muestra. Luego se guarda la copia en almacenamiento interno (la foto de perfil oficial) y opcionalmente en la galería pública si el usuario lo pidió en configuración.

Cada paso tiene su manejo de error: cámara cancelada (no pasa nada), permiso revocado a mitad del flujo (se verifica de nuevo antes de lanzar), foto corrupta (try/catch al decodificar), sin espacio en disco (`IOException` capturada con mensaje amable). Ningún camino termina en crash. Este nivel de cuidado es lo que hace que una app se sienta "bien hecha": el usuario nunca ve un error técnico, solo mensajes que entiende.

Si implementas este flujo completo por tu cuenta, sin mirar los ejemplos, ya no eres un estudiante de permisos: eres un desarrollador que los domina.

Inyección de dependencias: el eslabón que une las piezas

En los ejemplos de esta clase vas a ver que el `ViewModel` necesita un `Repository`, y el `Repository` necesita el `DAO` y la `API`. ¿Quién crea a quién? Si cada clase crea sus propias dependencias con `=`, terminas con código rígido imposible de probar. La inyección de dependencias (DI) es la técnica de entregar las dependencias desde fuera, normalmente en el constructor.

Para este curso usaremos DI manual (pasar las dependencias por constructor desde la Activity o un contenedor simple), porque no requiere librerías y deja clarísimo el concepto. En proyectos grandes la industria usa Hilt (la librería oficial de Google), que genera ese cableado automáticamente. Aprende primero el manual: cuando uses Hilt después, entenderás que magia está haciendo por ti en vez de usarla a ciegas.

El patrón manual mínimo: una clase `AppContainer` que crea una sola vez la base de datos, el DAO, Retrofit, la API y el Repository, y la Activity le pide el Repository para dárselo al ViewModel mediante un `ViewModelProvider.Factory`. Es un poco de código repetitivo, pero cada pieza queda separada y testeable.

Testing: probar el Repository sin internet ni teléfono

La gran ventaja de la arquitectura con Repository es que puedes probar la lógica sin un teléfono. Un test unitario del Repository usa un DAO falso (un "fake" en memoria) y una API falsa, y verifica que `refrescar()` guarde los datos mapeados correctamente. Con JUnit y corrutinas de prueba (`runTest`), ese test corre en segundos en tu computadora.

No vamos a profundizar en testing en este libro (es tema de un curso avanzado), pero quiero que te lleves la idea: si tu lógica vive en Activities pegada a Retrofit, no hay forma de probarla; si vive en un Repository con dependencias inyectadas, probarla es trivial. La arquitectura no es un lujo académico: es lo que hace tu código profesional.

Versionado semántico: el idioma de las versiones

El `versionName` sigue una convención llamada versionado semántico: `MAYOR.MENOR.PARCHES` (ejemplo: 2.4.1). Subes el PARCHES para correcciones de bugs (2.4.1 a 2.4.2), el MENOR para funciones nuevas compatibles (2.4.0 a 2.5.0) y el MAYOR para cambios grandes o incompatibles (2.0.0 a 3.0.0). Los usuarios y los sistemas lo entienden intuitivamente: un cambio de MAYOR asusta un poco, un PARCHES tranquiliza. Usa este idioma siempre; es una señal de profesionalismo que los reclutadores notan.

Ejemplos paso a paso

Ejemplo 1: Pedir permiso de cámara con el sistema moderno

Este es el patrón completo que vas a usar en casi todos tus proyectos. Primero declara el permiso en el `AndroidManifest.xml`:

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

Esa línea va antes de la etiqueta `<application>`. Sin ella, todo lo demás falla en silencio.

Ahora el código en tu Activity. Vamos línea por línea:

```
class MainActivity : AppCompatActivity() {

    // 1. Registramos el launcher ANTES de usarlo, como propiedad de la clase.
    // El contrato RequestPermission() muestra el dialogo del sistema
    // y nos devuelve true si el usuario acepto, false si lo nego.
    private val permisoCamaraLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { concedido ->
        if (concedido) {
            abrirCamara()
        } else {
            // El usuario dijo que no: NO te quedes callado.
            // Explicale que puede activar el permiso desde ajustes.
            Toast.makeText(
                this,
                "Sin permiso de camara no puedo tomar fotos. Activalo en Ajustes.",
                Toast.LENGTH_LONG
            ).show()
        }
    }

    // 2. Funcion de ayuda: verifica si YA tenemos el permiso.
    // Se debe llamar cada vez, no solo la primera vez, porque el
    // usuario puede revocarlo cuando quiera desde los ajustes.
    private fun tienePermisoCamara(): Boolean {
        return ContextCompat.checkSelfPermission(
            this,
            Manifest.permission.CAMERA
        ) == PackageManager.PERMISSION_GRANTED
    }

    // 3. Esta es la funcion que llama tu boton "Tomar foto".
    private fun onBotonCamaraClick() {
        if (tienePermisoCamara()) {
            // Ya lo tenemos: directo a la funcion.
            abrirCamara()
        } else if (shouldShowRequestPermissionRationale(Manifest.permission.CAMERA)) {
            // El usuario ya nos dijo que no una vez.
            // Le explicamos con nuestras palabras ANTES de insistir.
        }
    }
}
```

```

        mostrarDialogoExplicacion()
    } else {
        // Primera vez, o rechazo permanente: lanzamos el dialogo del sistema.
        permisoCamaraLauncher.launch(Manifest.permission.CAMERA)
    }
}

// 4. Dialogo de explicacion: honesto y corto.
private fun mostrarDialogoExplicacion() {
    AlertDialog.Builder(this)
        .setTitle("Necesito la camara")
        .setMessage("Uso la camara solo para tomar tu foto de perfil. Tus fotos no salen de tu telefono.")
        .setPositiveButton("Entendido") { _, _ ->
            permisoCamaraLauncher.launch(Manifest.permission.CAMERA)
        }
        .setNegativeButton("Ahora no", null)
        .show()
}

private fun abrirCamara() {
    // Aquí iria el codigo de la clase 26 para tomar la foto.
}
}

```

Fijate en el orden de decision dentro de `onBotonCamaraClick`: primero verificas si ya tienes el permiso (el caso feliz), luego si debes dar una explicación (el usuario ya te nego una vez), y al final lanzas la petición. Este orden cubre los tres escenarios sin codigo duplicado.

Ejemplo 2: Mandar al usuario a los ajustes cuando el rechazo es permanente

Cuando el usuario nego el permiso dos veces, el sistema ya no muestra el dialogo. Insistir con `launch()` no sirve de nada. La salida profesional es abrir la pantalla de ajustes de tu app:

```

private fun abrirAjustesDeLaApp() {
    // Intent especial que abre la pantalla de "Informacion de la app"
    // donde el usuario puede activar los permisos manualmente.
    val intent = Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS).apply {
        data = Uri.fromParts("package", packageName, null)
    }
    startActivity(intent)
}
}

```

Línea por línea: `Settings.ACTION_APPLICATION_DETAILS_SETTINGS` es la acción del sistema para abrir los detalles de una app. El `data` con esquema `"package"` y tu `packageName` le dice al sistema QUE app debe mostrar (la tuya). Es un patrón que vas a reutilizar tal cual en todos tus proyectos.

Como sabes que el rechazo es permanente? Una forma práctica: si lanzaste el permiso, el usuario lo negó, y `shouldShowRequestPermissionRationale()` devuelve false después de la negación, entonces ya no te va a mostrar el diálogo. En ese punto muestras un diálogo propio con un botón "Ir a ajustes" que llame a `abrirAjustesDeLaApp()`.

Ejemplo 3: Tomar una foto rápida (miniatura) con TakePicturePreview

Es la forma más simple de obtener una foto. El sistema te devuelve un `Bitmap` pequeño, ideal para avatares:

```

class PerfilActivity : AppCompatActivity() {

    private lateinit var imageViewFoto: ImageView

    // El contrato TakePicturePreview no necesita archivo ni FileProvider:
    // la camara te devuelve directamente el Bitmap de la miniatura.
    private val tomarFotoLauncher = registerForActivityResult(
        ActivityResultContracts.TakePicturePreview()
    ) { bitmap ->
        if (bitmap != null) {
            // Mostramos la foto en el ImageView.
            imageViewFoto.setImageBitmap(bitmap)
        } else {
            // El usuario cancelo la camara: no es un error, no hagas nada
            // dramático.
            Toast.makeText(this, "No se tomo ninguna foto",
                Toast.LENGTH_SHORT).show()
        }
    }

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

        imageViewFoto = findViewById(R.id.imageViewFoto)

        findViewById<Button>(R.id.botonTomarFoto).setOnClickListener {
            // OJO: aqui deberias verificar el permiso de camara primero
            // con el patron del Ejemplo 1, y solo lanzar si lo tienes.
            tomarFotoLauncher.launch()
        }
    }
}

```

Nota importante: `TakePicturePreview()` no recibe parametros, por eso `launch()` va vacío. El Bitmap que devuelve es pequeño (tamaño miniatura), así que no sirve para imprimir ni para fotos de alta calidad, pero es perfecto para una foto de perfil.

Ejemplo 4: Foto en alta resolucion con FileProvider (paso a paso completo)

Cuando necesitas la foto completa, debes darle a la cámara un archivo donde guardarla. Como tu app no puede pasarle una ruta directa a otra app por seguridad, usas un FileProvider. Son cuatro pasos mecánicos.

Paso 1: declara el proveedor en el `AndroidManifest.xml`, dentro de ``:

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="com.lucasegaacademy.miapp.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/file_paths" />
</provider>
```

`android:authorities` debe ser único: usa tu package name más `.fileprovider`.
`android:exported="false"` significa que otras apps no pueden usar tu proveedor libremente, solo con el permiso temporal que les das.

Paso 2: crea el archivo `res/xml/file\_paths.xml` que define que carpetas puedes compartir:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>
    <!-- Permite compartir archivos de la carpeta de imagenes de tu app -->
    <external-files-path name="mis_fotos" path="Pictures" />
</paths>
```

Paso 3: en tu Activity, crea el archivo temporal y genera el Uri seguro:

```

class CamaraActivity : AppCompatActivity() {

    private var uriFotoActual: Uri? = null
    private lateinit var imageViewFoto: ImageView

    private val tomarFotoAltaLauncher = registerForActivityResult(
        ActivityResultContracts.TakePicture()
    ) { exito ->
        if (exito) {
            // La camara guardo la foto en nuestro archivo: la mostramos.
            imageViewFoto.setImageURI(uriFotoActual)
        }
    }

    // Crea un archivo vacio en la carpeta de imagenes de la app
    // y devuelve un Uri seguro que la camara si puede usar.
    private fun crearArchivoParaFoto(): Uri {
        val nombreArchivo = "foto_${System.currentTimeMillis()}.jpg"
        val directorio = getExternalFilesDir(Environment.DIRECTORY_PICTURES)
        val archivo = File(directorio, nombreArchivo)
        return FileProvider.getUriForFile(
            this,
            "com.lucasegaacademy.miapp.fileprovider",
            archivo
        )
    }

    private fun tomarFotoEnAlta() {
        uriFotoActual = crearArchivoParaFoto()
        // Le pasamos el Uri a la camara: ella guarda ahi la foto completa.
        tomarFotoAltaLauncher.launch(uriFotoActual)
    }
}

```

Línea por línea de lo importante: `getExternalFilesDir(Environment.DIRECTORY_PICTURES)` te da una carpeta privada de tu app que no necesita permisos extra de almacenamiento. `FileProvider.getUriForFile()` convierte ese archivo en un Uri con formato `content://` que la cámara puede escribir gracias al `grantUriPermissions`. El contrato `TakePicture()` devuelve un Boolean: true si la foto se guardo bien.

Ejemplo 5: Elegir una imagen de la galeria

```
class GaleriaActivity : AppCompatActivity() {

    private lateinit var imageViewFoto: ImageView

    // GetContent abre el selector de archivos del sistema.
    // El parametro "image/*" filtra para mostrar solo imagenes.
    private val elegirImagenLauncher = registerForActivityResult(
        ActivityResultContracts.GetContent()
    ) { uri ->
        if (uri != null) {
            imageViewFoto.setImageURI(uri)
        }
        // Si uri es null, el usuario cancelo: no pasa nada.
    }

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

        imageViewFoto = findViewById(R.id.imageViewFoto)

        findViewById<Button>(R.id.botonGaleria).setOnClickListener {
            elegirImagenLauncher.launch("image/*")
        }
    }
}
```

Recuerda el detalle de versiones: en Android 13+ necesitas el permiso `READ\_MEDIA\_IMAGES` antes de abrir la galeria en algunos casos; en versiones anteriores era `READ\_EXTERNAL\_STORAGE`. Verificalo con `Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.TIRAMISU`.

Ejemplo 6: Escalar un Bitmap para no saturar la memoria

Esta funcion es oro molido: carga cualquier imagen reducida al tamaño que necesitas:

```

// Carga un Bitmap desde un Uri, escalado para que no exceda
// anchoMaximo x altoMaximo pixeles. Asi evitas OutOfMemoryError.
fun cargarBitmapEscalado(context: Context, uri: Uri, anchoMaximo: Int, altoMaximo:
Int): Bitmap? {
    // Paso 1: medir sin cargar. inJustDecodeBounds = true significa
    // "solo dime cuanto mide, no cargues ni un pixel en memoria".
    val opciones = BitmapFactory.Options().apply { inJustDecodeBounds = true }
    context.contentResolver.openInputStream(uri)?.use { stream ->
        BitmapFactory.decodeStream(stream, null, opciones)
    }

    // Paso 2: calcular el factor de escala (siempre potencia de 2).
    var escala = 1
    while (opciones.outWidth / (escala * 2) >= anchoMaximo &&
        opciones.outHeight / (escala * 2) >= altoMaximo) {
        escala *= 2
    }

    // Paso 3: cargar de verdad, pero ya reducida.
    val opcionesFinales = BitmapFactory.Options().apply { inSampleSize = escala }
    return context.contentResolver.openInputStream(uri)?.use { stream ->
        BitmapFactory.decodeStream(stream, null, opcionesFinales)
    }
}

```

Cómo se usa en la práctica:

```

val bitmapSeguro = cargarBitmapEscalado(this, uriFoto, 800, 800)
imageViewFoto.setImageBitmap(bitmapSeguro)

```

Con esto, una foto de 4000 pixeles se carga como una de 800: ocupa 25 veces menos memoria y en pantalla se ve identica, porque el ImageView de todos modos la iba a mostrar pequeña.

Ejemplo 7: Crear un canal y mostrar una notificacion

El canal se crea una sola vez, tipicamente cuando arranca la app. Un buen lugar es el `onCreate` de tu Activity principal o una clase Application:

```
// ID unico del canal: usalo igual al crear el canal y al notificar.
const val CANAL_ID = "canal_recordatorios"

fun crearCanalNotificacion(context: Context) {
    // Los canales solo existen desde Android 8 (API 26).
    // En versiones anteriores este codigo se ignora sin problema.
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val nombre = "Recordatorios"
        val descripcion = "Avisos de tareas pendientes de la app"
        val importancia = NotificationManager.IMPORTANCE_DEFAULT

        val canal = NotificationChannel(CANAL_ID, nombre, importancia).apply {
            this.description = descripcion
        }

        val manager = context.getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
        manager.createNotificationChannel(canal)
    }
}
```

Y para mostrar la notificacion:

```
fun mostrarNotificacion(context: Context, titulo: String, texto: String) {
    val builder = NotificationCompat.Builder(context, CANAL_ID)
        .setSmallIcon(R.drawable.ic_notificacion) // OBLIGATORIO: sin icono no se
muestra
        .setContentTitle(titulo)
        .setContentText(texto)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT)
        .setAutoCancel(true) // Se quita sola cuando el usuario la toca

    val manager = NotificationManagerCompat.from(context)
    // El ID 1 identifica esta notificacion: si lanzas otra con el mismo
// ID, la reemplaza; con ID distinto, se acumulan.
    if (ActivityCompat.checkSelfPermission(context,
Manifest.permission.POST_NOTIFICATIONS)
        == PackageManager.PERMISSION_GRANTED ||
        Build.VERSION.SDK_INT < Build.VERSION_CODES.TIRAMISU) {
        manager.notify(1, builder.build())
    }
}
```

Detalles que no debes olvidar: `setSmallIcon` es obligatorio; sin el la notificación falla en silencio. El `CANAL\_ID` del Builder debe ser exactamente el mismo que usaste al crear el canal. Y en Android 13+ debes verificar el permiso `POST\_NOTIFICATIONS` antes de llamar a `notify()`, cómo hace el `if` del ejemplo.

Ejemplo 8: Pedir el permiso de notificaciones (Android 13+)

```
class MainActivity : AppCompatActivity() {

    private val permisoNotificacionesLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { concedido ->
        if (concedido) {
            mostrarNotificacion(this, "Bienvenido", "Activaste los avisos de la
app")
        }
        // Si lo niega, no pasa nada grave: la app sigue funcionando sin avisos.
    }

    private fun pedirPermisoNotificacionesSiHaceFalta() {
        // Solo tiene sentido pedirlo en Android 13 o superior.
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            if (ContextCompat.checkSelfPermission(this,
Manifest.permission.POST_NOTIFICATIONS)
                != PackageManager.PERMISSION_GRANTED) {

                permisoNotificacionesLauncher.launch(Manifest.permission.POST_NOTIFICATIONS)
            }
        }
    }
}
```

Buena práctica: no pidas este permiso al abrir la app. Pídelo cuando el usuario active algo que necesite avisos, por ejemplo al activar un recordatorio o al entrar a la pantalla de configuración de notificaciones. El permiso en contexto siempre convierte mejor.

No olvides declararlo en el manifest:

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

Ejemplo 9: Obtener la ubicación con FusedLocationProviderClient

Primero agrega la dependencia de Play Services Location en tu `build.gradle` del modulo app:

```
implementation("com.google.android.gms:play-services-location:21.0.1")
```

Y declara los permisos en el manifest:

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

Ahora el codigo para obtener la última ubicacion conocida:

```
class MapaActivity : AppCompatActivity() {

    private lateinit var fusedClient: FusedLocationProviderClient
    private lateinit var textViewUbicacion: TextView

    private val permisoUbicacionLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { concedido ->
        if (concedido) {
            obtenerUbicacion()
        } else {
            textViewUbicacion.text = "Sin permiso de ubicacion no puedo mostrarte donde estas."
        }
    }

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

        textViewUbicacion = findViewById(R.id.textViewUbicacion)
        // El cliente se obtiene con LocationServices: el combina GPS, WiFi y red.
        fusedClient = LocationServices.getFusedLocationProviderClient(this)

        findViewById<Button>(R.id.botonUbicacion).setOnClickListener {
            if (ContextCompat.checkSelfPermission(this,
                Manifest.permission.ACCESS_FINE_LOCATION)
                == PackageManager.PERMISSION_GRANTED) {
                obtenerUbicacion()
            } else {
                permisoUbicacionLauncher.launch(Manifest.permission.ACCESS_FINE_LOCATION)
            }
        }
    }
}
```

```

    }
}

@SuppressLint("MissingPermission") // Ya verificamos el permiso arriba
private fun obtenerUbicacion() {
    // lastLocation es rapido pero puede devolver null
    // (GPS apagado o sin ubicacion registrada todavia).
    fusedClient.lastLocation.addOnSuccessListener { location ->
        if (location != null) {
            textViewUbicacion.text =
                "Latitud: ${location.latitude}\nLongitud: ${location.longitude}"
        } else {
            textViewUbicacion.text = "No hay ubicacion disponible. Activa el GPS."
        }
    }
}
}
}

```

Puntos clave: siempre maneja el `location != null`; el null es normal y no es un error de tu código. El `@SuppressLint("MissingPermission")` le dice al compilador "ya verifique el permiso, no me reganes"; solo úsalo cuando de verdad ya lo verificaste, nunca para ocultar un descuido.

Para actualizaciones continuas (por ejemplo, una app que rastrea un recorrido), usarías `requestLocationUpdates` con un `LocationRequest` y lo cancelarias en `onPause` con `removeLocationUpdates`. La regla es sagrada: GPS encendido solo mientras se necesita.

Práctica

Ejercicio 1: Permiso de contactos con explicación previa

Enunciado: Crea una Activity con un botón "Ver mis contactos" que pida el permiso `READ\_CONTACTS` siguiendo el patrón completo: verificar si ya se tiene, mostrar diálogo de explicación si el usuario ya lo negó una vez, y lanzar la petición. Si el usuario lo concede, muestra un Toast que diga "Permiso concedido"; si lo niega, muestra un Toast que diga "Sin permiso no puedo mostrar tus contactos".

Solución:

```

class ContactosActivity : AppCompatActivity() {

    private val permisoContactosLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission()
    )
}

```



```

    ) { concedido ->
        if (concedido) {
            Toast.makeText(this, "Permiso concedido", Toast.LENGTH_SHORT).show()
            // Aqui iria el codigo para leer los contactos.
        } else {
            Toast.makeText(
                this,
                "Sin permiso no puedo mostrar tus contactos",
                Toast.LENGTH_LONG
            ).show()
        }
    }
}

private fun tienePermiso(): Boolean {
    return ContextCompat.checkSelfPermission(
        this, Manifest.permission.READ_CONTACTS
    ) == PackageManager.PERMISSION_GRANTED
}

private fun onBotonClick() {
    when {
        tienePermiso() -> {
            Toast.makeText(this, "Permiso concedido", Toast.LENGTH_SHORT).show()
        }
        shouldShowRequestPermissionRationale(Manifest.permission.READ_CONTACTS) ->
    {
        AlertDialog.Builder(this)
            .setTitle("Permiso de contactos")
            .setMessage("Necesito leer tus contactos para mostrarte tu lista
de amigos.")
            .setPositiveButton("Entendido") { _, _ ->
permisoContactosLauncher.launch(Manifest.permission.READ_CONTACTS)
            }
            .setNegativeButton("Cancelar", null)
            .show()
        }
    }
    else -> permisoContactosLauncher.launch(Manifest.permission.READ_CONTACTS)
    }
}
}
}

```

Explicación de la solución: el `when` cubre los tres caminos en orden lógico. No olvides declarar `

Ejercicio 2: Boton que abre los ajustes tras rechazo permanente

Enunciado: Agrega al ejercicio anterior un dialogo que aparezca cuando el usuario nego el permiso de forma permanente (el sistema ya no muestra el dialogo). El dialogo debe tener un boton "Ir a ajustes" que abra la pantalla de informacion de la app.

Solución:

```
private val permisoContactosLauncher = registerForActivityResult(
    ActivityResultContracts.RequestPermission()
) { concedido ->
    if (!concedido) {
        // Despues de negar, si ya no debemos mostrar explicacion,
        // el rechazo es permanente: ofrecemos ir a ajustes.
        if
        (!shouldShowRequestPermissionRationale(Manifest.permission.READ_CONTACTS)) {
            AlertDialog.Builder(this)
                .setTitle("Permiso bloqueado")
                .setMessage("Negaste el permiso dos veces. Activalo manualmente en
los ajustes de la app.")
                .setPositiveButton("Ir a ajustes") { _, _ -> abrirAjustesDeLaApp()
            }
                .setNegativeButton("Ahora no", null)
                .show()
        }
    }
}

private fun abrirAjustesDeLaApp() {
    val intent = Intent(Settings.ACTION_APPLICATION_DETAILS_SETTINGS).apply {
        data = Uri.fromParts("package", packageName, null)
    }
    startActivity(intent)
}
```

Explicación de la solución: la clave está en llamar a `shouldShowRequestPermissionRationale()` Después de la negacion. Si devuelve false en ese momento, significa que el sistema ya no va a mostrar su dialogo, así que la unica via es

ajustes. Este patron es el que usan las apps profesionales y evita que el usuario se quede atorado sin saber que hacer.

Ejercicio 3: Foto de perfil completa (cámara + FileProvider + ImageView)

Enunciado: Implementa una pantalla de perfil con un ImageView circular (puedes usar un ImageView normal) y un boton "Cambiar foto" que: 1) verifique el permiso de cámara, 2) tome la foto en alta resolucion con FileProvider, 3) la muestre escalada a 600x600 para cuidar la memoria.

Solución:

```

class PerfilFotoActivity : AppCompatActivity() {

    private lateinit var imageViewPerfil: ImageView
    private var uriFoto: Uri? = null

    private val permisoLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { concedido ->
        if (concedido) tomarFoto() else
            Toast.makeText(this, "Necesito la camara para tu foto",
                Toast.LENGTH_SHORT).show()
    }

    private val fotoLauncher = registerForActivityResult(
        ActivityResultContracts.TakePicture()
    ) { exito ->
        if (exito && uriFoto != null) {
            // Cargamos escalada a 600x600: cuida la memoria.
            val bitmap = cargarBitmapEscalado(this, uriFoto!!, 600, 600)
            imageViewPerfil.setImageBitmap(bitmap)
        }
    }

    private fun onBotonCambiarFotoClick() {
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
            == PackageManager.PERMISSION_GRANTED) {
            tomarFoto()
        } else {
            permisoLauncher.launch(Manifest.permission.CAMERA)
        }
    }

    private fun tomarFoto() {
        val archivo = File(
            getExternalFilesDir(Environment.DIRECTORY_PICTURES),
            "perfil_${System.currentTimeMillis()}.jpg"
        )
        uriFoto = FileProvider.getUriForFile(
            this, "${packageName}.fileprovider", archivo
        )
        fotoLauncher.launch(uriFoto)
    }
}

```

Explicación de la solución: combina las tres técnicas de la unidad (permiso en tiempo de ejecución, FileProvider y escalado de Bitmap) en un flujo real. Nota el truco ```\${packageName}.fileprovider```: así no escribes el package a mano y funciona aunque cambies el nombre del paquete. Recuerda declarar el provider en el manifest y el archivo `file\_paths.xml`.

Ejercicio 4: Galeria con seleccion multiple simulada y vista previa

Enunciado: Crea una pantalla con un boton "Elegir foto" que abra la galeria y muestre la imagen elegida en un ImageView. Además, guarda una copia de la imagen elegida en el almacenamiento interno de la app para que siga disponible después de cerrar la app.

Solución:

```
class GaleriaGuardarActivity : AppCompatActivity() {

    private lateinit var imageViewFoto: ImageView

    private val galeriaLauncher = registerForActivityResult(
        ActivityResultContracts.GetContent()
    ) { uri ->
        if (uri != null) {
            // Mostramos una version escalada para cuidar la memoria.
            val bitmap = cargarBitmapEscalado(this, uri, 800, 800)
            imageViewFoto.setImageBitmap(bitmap)
            // Guardamos copia permanente en almacenamiento interno.
            guardarCopiaInterna(uri)
            Toast.makeText(this, "Foto guardada en la app",
                Toast.LENGTH_SHORT).show()
        }
    }

    // Copia el contenido del Uri temporal a un archivo propio.
    private fun guardarCopiaInterna(uriOrigen: Uri) {
        val archivoDestino = File(filesDir, "foto_guardada.jpg")
        contentResolver.openInputStream(uriOrigen)?.use { entrada ->
            FileOutputStream(archivoDestino).use { salida ->
                entrada.copyTo(salida)
            }
        }
    }
}
```

Explicación de la solución: el Uri que devuelve la galeria es temporal; si solo lo guardas como texto y cierras la app, después ya no sirve. La copia a `filesDir` (almacenamiento interno privado) garantiza que la imagen viva con tu app. El `use` cierra los streams automaticamente aunque haya un error, evitando fugas de recursos.

Ejercicio 5: Sistema de recordatorios con notificaciones

Enunciado: Implementa un "recordatorio de tomar agua": un boton que, al presionarlo, pide el permiso de notificaciones si hace falta (Android 13+) y luego muestra una notificacion en el canal "Salud" con titulo "Hora de hidratarte" y texto "Toma un vaso de agua". El canal debe crearse al iniciar la Activity.

Solución:

```
class AguaActivity : AppCompatActivity() {

    companion object {
        const val CANAL_SALUD = "canal_salud"
    }

    private val permisoLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { mostrarRecordatorio() } // Lo intente o no, mostramos (si no hay permiso, no sale nada)

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

        findViewById<Button>(R.id.botonAgua).setOnClickListener {
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU &&
                ContextCompat.checkSelfPermission(this,
Manifest.permission.POST_NOTIFICATIONS)
                    != PackageManager.PERMISSION_GRANTED) {
                permisoLauncher.launch(Manifest.permission.POST_NOTIFICATIONS)
            } else {
                mostrarRecordatorio()
            }
        }
    }

    private fun crearCanalSalud() {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
```

```

        val canal = NotificationChannel(
            CANAL_SALUD, "Salud", NotificationManager.IMPORTANCE_DEFAULT
        ).apply { description = "Recordatorios de habitos saludables" }
        val manager = getSystemService(Context.NOTIFICATION_SERVICE) as
NotificationManager
        manager.createNotificationChannel(canal)
    }
}

private fun mostrarRecordatorio() {
    val notificacion = NotificationCompat.Builder(this, CANAL_SALUD)
        .setSmallIcon(R.drawable.ic_agua)
        .setContentTitle("Hora de hidratarte")
        .setContentText("Toma un vaso de agua")
        .setAutoCancel(true)
        .build()
    NotificationManagerCompat.from(this).notify(100, notificacion)
}
}

```

Explicación de la solución: el canal se crea en `onCreate` para garantizar que exista antes de cualquier notificación. El permiso se pide en contexto (al tocar el botón), no al abrir la app. Si el usuario niega el permiso, `notify()` simplemente no muestra nada y la app no se cae: ese es el comportamiento correcto.

Ejercicio 6: Pantalla "lugares cercanos" con ubicación y estado alternativo

Enunciado: Crea una Activity que al abrirse intente obtener la ubicación. Si tiene permiso y hay ubicación, muestra las coordenadas. Si no hay permiso, muestra un botón "Activar ubicación" que lo pide. Si el GPS está apagado (location null), muestra el texto "Activa tu GPS para continuar". Todo sin crashear en ningún caso.

Solución:

```

class CercanosActivity : AppCompatActivity() {

    private lateinit var fusedClient: FusedLocationProviderClient
    private lateinit var textEstado: TextView
    private lateinit var botonActivar: Button

    private val permisoLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission()
    ) { concedido ->
        if (concedido) obtenerUbicacion()
        else {

```

```

        textEstado.text = "Sin ubicacion no puedo mostrarte lugares cercanos."
        botonActivar.visibility = View.VISIBLE
    }
}

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

    textEstado = findViewById(R.id.textEstado)
    botonActivar = findViewById(R.id.botonActivar)
    fusedClient = LocationServices.getFusedLocationProviderClient(this)

    botonActivar.setOnClickListener {
        permisoLauncher.launch(Manifest.permission.ACCESS_FINE_LOCATION)
    }

    // Al abrir: si ya hay permiso, directo a buscar ubicacion.
    if (ContextCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)
        == PackageManager.PERMISSION_GRANTED) {
        obtenerUbicacion()
    } else {
        textEstado.text = "Toca el boton para activar tu ubicacion."
        botonActivar.visibility = View.VISIBLE
    }
}

@SuppressLint("MissingPermission")
private fun obtenerUbicacion() {
    botonActivar.visibility = View.GONE
    textEstado.text = "Buscando tu ubicacion..."
    fusedClient.lastLocation.addOnSuccessListener { location ->
        if (location != null) {
            textEstado.text = "Estas en: ${location.latitude},
${location.longitude}"
        } else {
            // GPS apagado o sin datos: estado alternativo, no crash.
            textEstado.text = "Activa tu GPS para continuar."
            botonActivar.visibility = View.VISIBLE
            botonActivar.text = "Reintentar"
        }
    }
}
}
}

```


Explicación de la solución: cada estado posible tiene su pantalla: cargando, con coordenadas, sin permiso y sin GPS. Ningún camino termina en crash ni en pantalla vacía misteriosa. Este manejo de estados es exactamente lo que evalúa un reclutador cuando revisa tu código: no solo el caso feliz, sino todos los casos.

Ejercicio 9: Guardar la foto tomada en la galería pública

Enunciado: Extiende el ejercicio 3: después de tomar la foto en alta resolución, guarda una copia en la galería pública del teléfono usando MediaStore, para que el usuario la encuentre aunque desinstale la app. Muestra un Toast con "Foto guardada en tu galería".

Solución:

```
private val fotoLauncher = registerForActivityResult(
    ActivityResultContracts.TakePicture()
) { éxito ->
    if (éxito && uriFoto != null) {
        val bitmap = cargarBitmapEscalado(this, uriFoto!!, 1200, 1200)
        imageViewPerfil.setImageBitmap(bitmap)
        // Guardamos copia pública en la galería del sistema.
        val uriPublica = guardarEnGaleria(this, bitmap!!,
            "perfil_${System.currentTimeMillis()}.jpg")
        if (uriPublica != null) {
            Toast.makeText(this, "Foto guardada en tu galería",
                Toast.LENGTH_SHORT).show()
        }
    }
}
```

Explicación de la solución: reusamos `guardarEnGaleria()` del apartado de Scoped Storage y `cargarBitmapEscalado()` del Ejemplo 6. Fíjate cómo las piezas de la unidad se combinan como bloques de construcción: permiso, captura, escalado y almacenamiento público en un solo flujo. En Android 10+ esto no necesita ningún permiso extra de escritura; en Android 9 o menor necesitas `WRITE\_EXTERNAL\_STORAGE`, así que si tu `minSdk` es menor a 29, agrega la verificación de versión.

Ejercicio 10: Mini-app "Diario de viaje"

Enunciado: Crea una Activity que combine todo: un boton "Registrar visita" que pida los permisos de cámara y ubicacion (multiples), tome una foto, obtenga las coordenadas, las convierta a direccion con Geocoder y muestre una notificacion que diga "Visita registrada: [direccion]". Si algun permiso falta, la app debe degradarse con elegancia como aprendiste.

Solución (esqueleto que une las piezas):

```
class DiarioActivity : AppCompatActivity() {

    private var uriFoto: Uri? = null
    private lateinit var fusedClient: FusedLocationProviderClient

    private val permisosLauncher = registerForActivityResult(
        ActivityResultContracts.RequestMultiplePermissions()
    ) { resultados ->
        val camaraOk = resultados[Manifest.permission.CAMERA] == true
        val ubiOk = resultados[Manifest.permission.ACCESS_FINE_LOCATION] == true
        if (camaraOk) {
            tomarFoto() // Al terminar la foto, en el callback, si ubiOk ->
            registrarUbicacion()
            fotoPendienteConUbicacion = ubiOk
        } else {
            Toast.makeText(this, "Sin camara no puedo registrar la visita",
                Toast.LENGTH_SHORT).show()
        }
    }

    private var fotoPendienteConUbicacion = false

    private val fotoLauncher = registerForActivityResult(
        ActivityResultContracts.TakePicture()
    ) { exito ->
        if (!exito || uriFoto == null) return@registerForActivityResult
        if (fotoPendienteConUbicacion) {
            registrarVisitaConUbicacion()
        } else {
            mostrarNotificacion(this, "Visita registrada", "Foto guardada sin
ubicacion")
        }
    }

    @SuppressWarnings("MissingPermission")
    private fun registrarVisitaConUbicacion() {
        fusedClient.lastLocation.addOnSuccessListener { location ->
            if (location != null) {
```

```

        lifecycleScope.launch(Dispatchers.IO) {
            val direccion = obtenerDireccion(this@DiarioActivity,
location.latitude, location.longitude)
            withContext(Dispatchers.Main) {
                mostrarNotificacion(this@DiarioActivity, "Visita registrada",
direccion)
            }
        }
    } else {
        mostrarNotificacion(this, "Visita registrada", "Foto guardada (sin GPS
disponible)")
    }
}
}
}
}

```

Explicación de la solución: este es el ejercicio integrador de la unidad. La bandera `fotoPendienteConUbicacion` coordina dos operaciones asincronas (la foto y el permiso de ubicacion): primero se toma la foto y solo al confirmarla se busca la ubicacion. El Geocoder va en `Dispatchers.IO` porque bloquea. Cada rama del flujo termina en una notificacion informativa: nunca hay silencio misterioso. Si llegaste hasta aquí y entiendes cada línea, dominas la unidad completa.

Ejercicio 11: Checklist de publicación de funciones sensibles

Enunciado (teórico, sin código): Antes de publicar una app que usa cámara, ubicacion y notificaciones, escribe la lista de verificacion que seguirias para no tener sorpresas en produccion ni rechazos en Play Store.

Solución:

1. Cada permiso peligroso esta declarado en el manifest y se pide en tiempo de ejecucion en el contexto donde se usa.
2. La app funciona aunque el usuario niegue cada permiso (degradacion elegante, estados alternativos, sin crashes).
3. El caso de rechazo permanente manda a ajustes con un dialogo que lo explique.
4. Los canales de notificacion se crean al iniciar la app y cada notificacion tiene su icono pequeño.
5. En Android 13+ se pide POST_NOTIFICATIONS antes de notificar.

6. La ubicacion se verifica antes de cada uso (no se cachea el permiso), se maneja el null del GPS apagado y se cancelan las actualizaciones en onPause.
7. Los Bitmaps se escalan antes de cargarse en memoria.
8. Cada permiso tiene una funcion visible en la app que lo justifica (regla de Play Store).
9. Probado en un dispositivo real con Android 13 o superior, negando cada permiso al menos una vez.

Explicación de la solución: este checklist es el que usan los equipos profesionales antes de cada release. Memorízalo: te va a ahorrar rechazos en Play Console y reseñas de una estrella. La mayoría de los bugs de permisos no se descubren en el emulador del desarrollador (donde todo se concede), sino en el teléfono del usuario real (donde todo se niega).

Profundización: técnicas avanzadas y casos reales

Pedir varios permisos a la vez

Hay funciones que necesitan más de un permiso. El ejemplo clásico es una app que toma foto y la guarda con ubicación: necesita cámara y ubicación al mismo tiempo. Para eso existe el contrato `ActivityResultContracts.RequestMultiplePermissions()`, que te devuelve un mapa de permiso a resultado:

```

private val permisosMultiplesLauncher = registerForActivityResult(
    ActivityResultContracts.RequestMultiplePermissions()
) { resultados ->
    // resultados es un Map<String, Boolean>: permiso -> concedido o no.
    val camaraOk = resultados[Manifest.permission.CAMERA] == true
    val ubicacionOk = resultados[Manifest.permission.ACCESS_FINE_LOCATION] == true

    when {
        camaraOk && ubicacionOk -> tomarFotoConUbicacion()
        camaraOk -> {
            // La camara si, la ubicacion no: la foto se toma sin coordenadas.
            Toast.makeText(this, "Foto sin ubicacion: no diste ese permiso",
                Toast.LENGTH_LONG).show()
            tomarFoto()
        }
        else -> Toast.makeText(this, "Sin camara no puedo continuar",
            Toast.LENGTH_SHORT).show()
    }
}

private fun pedirPermisosDeFoto() {
    permisosMultiplesLauncher.launch(
        arrayOf(
            Manifest.permission.CAMERA,
            Manifest.permission.ACCESS_FINE_LOCATION
        )
    )
}

```

La lección profesional aquí es el diseño degradado: tu app no se bloquea porque falte un permiso secundario. La función principal (tomar la foto) sigue trabajando; solo la función extra (geolocalizarla) se desactiva. Diseña cada permiso como un interruptor independiente y tu app será robusta.

Notificaciones enriquecidas: texto largo y botones de acción

Una notificación básica muestra una línea de texto. Pero puedes expandirla con estilos y agregarle botones que ejecutan acciones sin abrir la app. El estilo `BigTextStyle` permite mostrar párrafos completos cuando el usuario expande la notificación:

```

fun mostrarNotificacionRica(context: Context) {
    // Intent que se ejecuta cuando el usuario toca el boton "Marcar leida".
    val intentMarcar = Intent(context, AccionReceiver::class.java).apply {
        action = "ACCION_MARCAR_LEIDA"
    }
    val pendingMarcar = PendingIntent.getBroadcast(
        context, 0, intentMarcar,
        PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE
    )

    val notificacion = NotificationCompat.Builder(context, CANAL_ID)
        .setSmallIcon(R.drawable.ic_mensaje)
        .setContentTitle("Nuevo mensaje de Ana")
        .setContentText("Te mando la lista del super...")
        .setStyle(
            NotificationCompat.BigTextStyle()
                .bigText("Te mando la lista del super: leche, huevos, pan, queso y tortillas. Pasa por la tienda antes de las 6 porfa.")
        )
        .addAction(R.drawable.ic_check, "Marcar leida", pendingMarcar)
        .setAutoCancel(true)
        .build()

    NotificationManagerCompat.from(context).notify(2, notificacion)
}

```

Tres conceptos nuevos aquí. Primero, `BigTextStyle`: el texto corto se muestra colapsado y el largo al expandir. Segundo, `addAction`: agrega un boton a la notificacion. Tercero, el `PendingIntent`: es un permiso para que el sistema ejecute tu codigo después, cuando el usuario toque el boton; nota la bandera `FLAG_IMMUTABLE`, obligatoria en Android 12+ por seguridad (un `PendingIntent` mutable podria ser manipulado por otra app). El `AccionReceiver` sería un `BroadcastReceiver` que marca el mensaje como leído.

Ubicacion en tiempo real: actualizaciones periodicas

`lastLocation` te da una foto fija del pasado. Para una app de recorridos, delivery o ejercicio necesitas la pelicula completa: actualizaciones periodicas. El patron es suscribirse al iniciar y cancelar al pausar:

```

class RecorridoActivity : AppCompatActivity() {

    private lateinit var fusedClient: FusedLocationProviderClient
    private lateinit var locationCallback: LocationCallback

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        fusedClient = LocationServices.getFusedLocationProviderClient(this)

        // Este callback recibe cada actualizacion de ubicacion.
        locationCallback = object : LocationCallback() {
            override fun onLocationResult(result: LocationResult) {
                val ultima = result.lastLocation ?: return
                // Aqui actualizarias el mapa o guardarias el punto del recorrido.
                registrarPunto(ultima.latitude, ultima.longitude)
            }
        }
    }

    @SuppressWarnings("MissingPermission")
    private fun iniciarSeguimiento() {
        val request = LocationRequest.Builder(
            Priority.PRIORITY_HIGH_ACCURACY, 5000L // cada 5 segundos
        ).build()
        fusedClient.requestLocationUpdates(request, locationCallback, mainLooper)
    }

    override fun onPause() {
        super.onPause()
        // SAGRADO: al pausar la pantalla, se apaga el seguimiento.
        // Asi no drenas la bateria cuando el usuario no te esta viendo.
        fusedClient.removeLocationUpdates(locationCallback)
    }

    private fun registrarPunto(lat: Double, lon: Double) {
        // Guardar en lista o base de datos.
    }
}

```

El intervalo de 5 segundos con `PRIORITY\_HIGH\_ACCURACY` es para seguimiento activo (navegacion, ejercicio). Para algo menos crítico usa `PRIORITY\_BALANCED\_POWER\_ACCURACY` con intervalos más largos: el sistema usara WiFi y red en vez del GPS y la bateria te lo agradecera. Elegir la prioridad correcta según el caso de uso es una decision de diseno que distingue al profesional.

El problema de las fotos rotadas: metadatos EXIF

Un bug clásico que vuelve locos a los principiantes: tomas una foto vertical, la muestras en tu app y sale rotada 90 grados. No es tu culpa: la cámara guarda la foto siempre en horizontal y anota la orientación real en los metadatos EXIF de la imagen. Algunos visores leen ese dato y la muestran bien; tu ImageView no siempre lo hace.

La solución es leer el EXIF y rotar el Bitmap tu mismo:

```
fun corregirRotacion(context: Context, uri: Uri, bitmap: Bitmap): Bitmap {
    val input = context.contentResolver.openInputStream(uri) ?: return bitmap
    val exif = ExifInterface(input)
    val orientacion = exif.getAttributeInt(
        ExifInterface.TAG_ORIENTATION, ExifInterface.ORIENTATION_NORMAL
    )
    val angulo = when (orientacion) {
        ExifInterface.ORIENTATION_ROTATE_90 -> 90f
        ExifInterface.ORIENTATION_ROTATE_180 -> 180f
        ExifInterface.ORIENTATION_ROTATE_270 -> 270f
        else -> 0f
    }
    if (angulo == 0f) return bitmap
    val matriz = Matrix().apply { postRotate(angulo) }
    return Bitmap.createBitmap(bitmap, 0, 0, bitmap.width, bitmap.height, matriz,
        true)
}
```

`ExifInterface` viene en la librería `androidx.exifinterface:exifinterface`. Es de esos detalles que nadie te enseña hasta que te topas con el bug en producción; ahora ya lo sabes antes.

Recopilación de errores clásicos (para no tropezar dos veces)

Primero: pedir el permiso pero olvidar declararlo en el manifest. Síntoma: el diálogo nunca aparece o siempre devuelve false. Siempre verifica el manifest primero cuando un permiso "no funciona".

Segundo: lanzar el Intent de cámara sin verificar el permiso de cámara. En algunos dispositivos la cámara del sistema lo pide sola, en otros tu app se cierra. No dependas de la suerte: verifica siempre.

Tercero: no crear el canal de notificación antes de notificar. Síntoma: en Android 8+ no aparece nada y no hay error. Crea el canal al iniciar la app, una sola vez.

Cuarto: olvidar `setSmallIcon` en la notificación. La notificación no se muestra y el logcat apenas lo menciona.

Quinto: cargar el Bitmap completo sin escalar. Síntoma: la app se traba al abrir la segunda o tercera foto y eventualmente muere con `OutOfMemoryError`. Escala siempre.

Sexto: pedir `READ_EXTERNAL_STORAGE` en Android 13+ para leer imágenes. Ese permiso ya no aplica a imágenes; necesitas `READ_MEDIA_IMAGES`. Revisa `Build.VERSION.SDK_INT` cuando el permiso cambie entre versiones.

Septimo: dejar `requestLocationUpdates` activo al salir de la pantalla. Síntoma: la batería se drena y los usuarios te dejan reseñas de una estrella. Cancela siempre en `onPause`.

Octavo: no manejar el `location` null de `lastLocation`. No es un error de tu código: significa GPS apagado o sin datos. Muestra un estado alternativo.

Ejercicio 7: Permisos múltiples con degradación elegante

Enunciado: Implementa la pantalla de "foto con ubicación" del ejemplo de permisos múltiples. Si ambos permisos se conceden, toma la foto y muestra las coordenadas en un Toast. Si solo se concede la cámara, toma la foto sin ubicación y avisa. Si se niega la cámara, no hagas nada más que avisar.

Solución: usa el código del apartado de permisos múltiples tal cual, completando las funciones:

```
private fun tomarFotoConUbicacion() {  
    // 1. Tomamos la foto con el launcher de TakePicture (Ejemplo 4).  
    tomarFoto()  
    // 2. Obtenemos la ubicación y la mostramos.  
    obtenerUbicacion { lat, lon ->  
        Toast.makeText(this, "Foto guardada en $lat, $lon",  
            Toast.LENGTH_LONG).show()  
    }  
}
```

Explicación de la solución: la clave del ejercicio no es el código de la foto ni el del GPS (ya los dominas), sino la lógica del `when` sobre el mapa de resultados. Cada combinación de permisos tiene un camino definido y ninguno deja al usuario en blanco. Esa es la degradación elegante: la app hace lo máximo posible con lo que el usuario le permitió.

Ejercicio 8: Notificación con `BigTextStyle` que abre la app al tocarla

Enunciado: Muestra una notificación con texto largo expandible. Al tocar la notificación (no el botón), debe abrirse tu `MainActivity`. Usa un `PendingIntent` con `FLAG_IMMUTABLE`.

Solución:

```
fun mostrarNotificacionQueAbreApp(context: Context) {
    val intentAbrir = Intent(context, MainActivity::class.java).apply {
        flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
    }
    val pendingAbrir = PendingIntent.getActivity(
        context, 0, intentAbrir,
        PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE
    )

    val notificacion = NotificationCompat.Builder(context, CANAL_ID)
        .setSmallIcon(R.drawable.ic_noticia)
        .setContentTitle("Noticia completa")
        .setContentText("Toca para leer la noticia completa...")
        .setStyle(NotificationCompat.BigTextStyle().bigText(
            "Este es el cuerpo largo de la noticia. Al expandir la notificación el usuario puede leer el primer párrafo sin abrir la app, y si le interesa, toca y entra directo al detalle."
        ))
        .setContentIntent(pendingAbrir) // <-- esto abre la app al tocar
        .setAutoCancel(true)
        .build()

    NotificationManagerCompat.from(context).notify(3, notificacion)
}
```

Explicación de la solución: `setContentIntent` define qué pasa al tocar el cuerpo de la notificación, mientras que `addAction` define botones. Son mecanismos distintos para intenciones distintas: el toque abre el detalle, el botón ejecuta una acción rápida. Las banderas del Intent (`NEW_TASK` y `CLEAR_TASK`) hacen que la app abra limpiamente aunque estuviera cerrada. Y `FLAG_IMMUTABLE` es obligatorio en Android 12+: sin él, la app se cierra al crear el `PendingIntent`.

Glosario de la unidad

Permiso normal: permiso que el sistema otorga automáticamente al instalar la app, sin preguntar al usuario (ejemplo: acceso a internet, vibración).

Permiso peligroso: permiso que protege datos privados o funciones sensibles y debe solicitarse en tiempo de ejecución (ejemplo: cámara, ubicación, contactos).

Tiempo de ejecución: momento en que la app ya está corriendo. Los permisos peligrosos se piden aquí, no en la instalación.

ActivityResultLauncher: componente moderno que lanza una petición al sistema (permiso, cámara, galería) y recibe el resultado en un callback, sin métodos deprecados.

Contrato (ActivityResultContract): define que se pide y que se devuelve. Ejemplos: RequestPermission, RequestMultiplePermissions, TakePicture, TakePicturePreview, GetContent.

shouldShowRequestPermissionRationale(): método que indica si debes explicarle al usuario por que necesitas el permiso antes de volver a pedirlo (devuelve true cuando ya te lo negó una vez sin bloqueo permanente).

Rechazo permanente: cuando el usuario negó el permiso dos veces o marcó "no volver a preguntar"; el diálogo del sistema ya no aparece y la única salida es la pantalla de ajustes.

FileProvider: proveedor de contenido que genera URIs seguras (content://) para compartir archivos privados de tu app con otras apps, como la cámara del sistema.

Uri: identificador universal de un recurso. En Android apunta a archivos, imágenes de la galería o datos del MediaStore.

Bitmap: representación en memoria de una imagen, píxel por píxel. Cargar uno muy grande sin escalar causa OutOfMemoryError.

inSampleSize: factor de reducción (potencia de 2) que se usa con BitmapFactory.Options para cargar una imagen ya reducida y ahorrar memoria.

EXIF: metadatos incrustados en una foto (orientación, fecha, modelo de cámara). La orientación debe leerse para corregir fotos rotadas.

Canal de notificación: categoría obligatoria desde Android 8 a la que pertenece cada notificación; tiene nombre, descripción e importancia configurables por el usuario.

PendingIntent: permiso empaquetado para que el sistema ejecute tu código más tarde (al tocar una notificación o sonar una alarma), con banderas de seguridad como FLAG_IMMUTABLE.

FusedLocationProviderClient: cliente de Play Services que combina GPS, WiFi y red celular para entregar la mejor ubicacion disponible ahorrando bateria.

Geocodificacion inversa: conversion de coordenadas (latitud, longitud) a una direccion legible mediante Geocoder, siempre en hilo secundario.

Scoped Storage: modelo de almacenamiento desde Android 10 donde cada app ve principalmente sus propios archivos; escribir en MediaStore no requiere permiso de escritura.

MediaStore: almacen público del sistema para fotos, videos y audio, visible en la galeria aunque se desinstale tu app.

Resumen de la unidad

En esta unidad aprendiste a conectar tu app con el mundo real. La clase 25 te dio el sistema de permisos en tiempo de ejecucion: declarar en el manifest, pedir con `ActivityResultLauncher` y el contrato `RequestPermission()`, verificar con `ContextCompat.checkSelfPermission()`, explicar con `shouldShowRequestPermissionRationale()` cuando el usuario ya te nego una vez, y mandar a ajustes cuando el rechazo es permanente. La regla de oro quedo clara: pide el permiso en contexto, en el momento en que la funcion lo necesita, nunca en ametralladora al abrir la app.

La clase 26 te enseno a trabajar con imágenes: tomar fotos rapidas con `TakePicturePreview()`, fotos en alta resolucion con `TakePicture()` más `FileProvider` (declarado en el manifest con su archivo `file_paths.xml`), elegir imágenes de la galeria con `GetContent()`, y lo más importante para la estabilidad de tu app, escalar los Bitmaps con `inJustDecodeBounds` e `inSampleSize` antes de cargarlos en memoria. También aprendiste el cambio de Android 13: `READ_MEDIA_IMAGES` reemplaza a `READ_EXTERNAL_STORAGE` para imágenes.

La clase 27 cubrio notificaciones y ubicacion. Para notificaciones: crear el canal una sola vez con `NotificationChannel` (obligatorio desde Android 8), construir con `NotificationCompat.Builder` sin olvidar el icono pequeño, pedir `POST_NOTIFICATIONS` en Android 13+, y usar IDs de notificacion con intencion. Para ubicacion: `FusedLocationProviderClient` con `lastLocation` para lecturas rapidas, manejo del caso null cuando el GPS esta apagado, permisos `ACCESS_FINE_LOCATION` y `ACCESS_COARSE_LOCATION`, y la disciplina de apagar las actualizaciones cuando no se necesitan para no drenar la bateria.

El patron mental que te llevas es el manejo de los tres escenarios de permiso (concedido, negado temporal, negado permanente) y el diseno de estados alternativos para cada funcion del dispositivo. Con esto, tus apps ya no se caen cuando el usuario dice que no: responden con elegancia, explican y ofrecen caminos. Eso es desarrollo profesional.

En la siguiente unidad vas a juntar TODO lo aprendido en el curso: vas a armar una app completa que combine listas, API de internet y base de datos local, y luego la vas a firmar y publicar en Google Play Store como un desarrollador profesional. Es el cierre del curso y tu carta de presentacion al mundo.

Retos extra (tu turno, sin solución escrita)

Reto 1: construye una pantalla de "escaner de documentos" que tome una foto, la muestre escalada y corregida de rotacion, y la guarde en MediaStore dentro de la carpeta "Pictures/MisDocumentos". Pista: combina el Ejemplo 4, la funcion de correccion EXIF y ``guardarEnGaleria()`` cambiando el `RELATIVE_PATH`.

Reto 2: implementa un recordatorio de medicamentos con AlarmManager que suene todos los días a las 9:00 y muestre una notificacion de importancia alta con dos botones: "Tomado" y "Posponer 10 min". Pista: el boton "Posponer" programa otra alarma una sola vez con ``set()`` a 10 minutos en el futuro.

Reto 3: crea una pantalla que muestre tu ubicacion actual como direccion legible (no coordenadas) y un boton "Copiar direccion" que la ponga en el portapapeles con ``ClipboardManager``. Maneja los tres casos: sin permiso, GPS apagado y sin internet para el Geocoder.

Reto 4: disena el flujo de permisos de una app de mensajeria que necesita cámara, microfono y notificaciones. Dibuja en papel los tres caminos de cada permiso (concedido, negado, bloqueo permanente) y escribe que muestra la pantalla en cada combinacion. Si puedes explicar tu diagrama en voz alta sin trabarte, el tema esta dominado.

UNIDAD 10: Proyecto final y publicación en Google Play

Llegaste a la última unidad del curso, y es la más importante de todas. Hasta aquí aprendiste piezas: pantallas, listas, navegacion, internet, bases de datos, permisos, cámara, notificaciones. Un estudiante promedio termina el curso sabiendo las piezas. Un profesional termina sabiendo armarlas en un producto completo y ponerlo en las manos de millones de personas. Esta unidad es ese puente.

En la clase 28 vas a construir una app completa de verdad, no un ejemplo de juguete. La app va a combinar las tres fuentes de datos que dominan el desarrollo móvil moderno: una lista que el usuario ve y toca (RecyclerView), una API de internet de donde llegan los datos frescos (Retrofit) y una base de datos local donde todo se guarda para funcionar sin internet (Room). Este trio tiene un nombre en la industria: arquitectura offline-first con patron Repository. Es lo que usan las apps serias: primero muestran lo que tienen guardado, luego actualizan desde internet en segundo plano. El usuario nunca ve una pantalla vacia con un spinner eterno; siempre ve contenido.

En la clase 29 vas a aprender lo que casi ningún tutorial enseña y que todos los juniors sufren en su primer trabajo: firmar tu app y generar el paquete de publicación. Vas a entender qué es un keystore, por qué perderlo es una tragedia (literalmente no puedes volver a actualizar tu app si lo pierdes), la diferencia entre APK y AAB, como configurar el build de release con su firma, y como activar R8 para ofuscar y reducir tu codigo. Al terminar esta clase vas a tener en tus manos un archivo .aab firmado, listo para subirse a la tienda.

En la clase 30 vas a entrar a Google Play Console, el panel de control de tu vida como desarrollador publicado. Vas a crear tu cuenta de desarrollador, llenar la ficha de tu app (titulo, descripcion, icono, capturas de pantalla, video promocional), responder el cuestionario de clasificacion de contenido, redactar tu politica de privacidad, pasar las revisiones de politicas de Google y lanzar tu app primero en prueba interna y luego en produccion. También vas a aprender que hacer cuando Google rechaza tu app, porque le pasa a todo el mundo la primera vez y no es el fin del mundo: es un tramite con pasos claros.

Una advertencia honesta antes de empezar: publicar una app real cuesta 25 dólares una sola vez (el registro de desarrollador en Play Console) y necesitas una cuenta de Google. Si ahora no puedes pagarlos, no pasa nada: todo lo de esta unidad lo puedes practicar hasta el paso final sin pagar, generando tu AAB firmado y dejando tu ficha lista. El conocimiento queda; el pago llega cuando estes listo.

Al final de esta unidad no solo sabras programar apps: sabras entregarlas. Esa es la diferencia entre alguien que "esta aprendiendo Android" y alguien que "es desarrollador Android". Vamos a cruzar ese puente juntos, paso a paso.

Explicación detallada

Clase 28: La app completa - lista + API + base de datos local

Imagina un restaurante. El comensal (la pantalla, tu UI) no entra a la cocina a buscar su comida; le pide al mesero (el ViewModel), y el mesero le pide a la cocina (el Repository), que decide si sirve del refrigerador (la base de datos local, rápido) o cocina algo fresco (la API de internet). El comensal nunca sabe de dónde vino su plato, solo que llegó rápido y caliente. Esa es la arquitectura que vas a construir.

El patron Repository es una clase que centraliza el acceso a los datos. Tu UI nunca habla directamente con Retrofit ni con Room; habla con el Repository, y el Repository decide la estrategia. La estrategia ganadora se llama offline-first y funciona así: primero lees la base de datos local y muestras esos datos de inmediato (la app se siente instantanea), luego lanzas la petición a la API en segundo plano, y cuando llegan los datos frescos los guardas en la base de datos y la UI se actualiza sola gracias a Flow o LiveData. Si no hay internet, la app sigue mostrando los últimos datos guardados en vez de un error gigante.

Vamos a modelar una app de ejemplo concreta para toda la clase: "MisPelículas", una app que muestra una lista de películas populares desde una API, las guarda en Room, y permite marcar favoritas que persisten aunque no haya internet. Es el proyecto típico de portafolio porque demuestra exactamente lo que piden en las entrevistas de trabajo.

Las piezas y sus responsabilidades:

La entidad de Room (`PeliculaEntity`) es la tabla de la base de datos: id, titulo, descripcion, url de imagen, calificacion y una bandera de favorita. El DAO (`PeliculaDao`) tiene las consultas: obtener todas como Flow, insertar una lista (reemplazando en conflicto), actualizar la bandera de favorita. La base de datos (`AppDatabase`) las junta.

Los modelos de la API (`PeliculaDto`) representan el JSON que devuelve el servidor; pueden tener nombres distintos a la entidad, y una funcion de mapeo los convierte de DTO a entidad. Nunca uses el DTO directamente en la UI: si la API cambia, solo tocas el mapeo.

El servicio de Retrofit (`PeliculaApi`) define el endpoint, por ejemplo `GET /peliculas/populares`. El Repository (`PeliculaRepository`) orquesta: expone `peliculas`:

Flow<List<Pelicula>>` que sale del DAO, y una funcion `suspend fun refrescar()` que llama a la API, mapea y guarda en Room dentro de un try/catch (si falla la red, no pasa nada: los datos locales siguen ahi).

El ViewModel (`PeliculasViewModel`) expone el Flow a la UI y lanza `refrescar()` en `viewModelScope` al iniciar. La Activity observa el Flow con `lifecycleScope` y actualiza el RecyclerView. El adaptador muestra cada pelicula con su imagen (cargada con Coil, la libreria moderna de imágenes) y un boton de favorito que llama al ViewModel, que actualiza Room, que emite el Flow de nuevo, que actualiza la lista. Todo el ciclo es reactivo: cambias un dato en la base y la pantalla se entera sola.

Errores comunes en esta clase: llamar a la API directamente desde la Activity (rompe la arquitectura y es imposible de probar), no manejar el error de red en `refrescar()` (la app se cierra sin internet), usar el DTO de la API como modelo de UI (cuando la API cambia, todo se rompe), y olvidar que Room no permite acceso en el hilo principal (usa corrutinas siempre). Otro clásico: no poner estrategia de conflicto en el insert (`OnConflictStrategy.REPLACE`) y duplicar registros cada vez que se refresca.

Buena práctica: el Repository es el único que decide de dónde vienen los datos; el ViewModel no sabe si hay API o base de datos, solo pide "dame las peliculas". Esa separacion es lo que te permite probar cada pieza por separado y lo que los reclutadores buscan cuando revisan tu GitHub.

Clase 29: Firmar tu app - keystore, release y AAB

Cuando instalas tu app desde Android Studio en modo debug, Android la firma automaticamente con una llave de depuracion generica. Esa firma sirve para probar, pero Google Play no la acepta: cada app publicada debe estar firmada con TU llave privada, unica y secreta. Esa llave vive en un archivo llamado keystore.

Piensa en el keystore como el sello notarial de tu app. El archivo `.jks` (o `.keystore`) contiene tu llave privada protegida por contraseñas: una para el almacen y otra para la llave. Con esa llave firmas cada version que subes, y Google verifica que la firma coincida con la original. Si alguien intenta subir una actualizacion falsa de tu app firmada con otra llave, Google la rechaza. Así se protege a tus usuarios.

La regla más importante de esta clase, grabatela: SI PIERDES TU KEYSTORE, NO PUEDES VOLVER A ACTUALIZAR TU APP. Tendrias que publicar una app nueva con otro nombre de paquete y tus usuarios tendrian que instalarla desde cero, perdiendo sus datos. Por eso el keystore se guarda en al menos dos lugares seguros (tu computadora más un respaldo cifrado en la nube) y sus contraseñas van en un gestor de contraseñas, nunca en el codigo ni en Git.

En serio: si subes tu keystore o sus contraseñas a GitHub, cualquiera puede firmar actualizaciones falsas de tu app.

Hoy Google ofrece "Play App Signing": tu subes tu app firmada con tu llave de subida, y Google la vuelve a firmar con la llave final de distribución que el custodia. Esto te salva si pierdes tu llave de subida (puedes pedir un reseteo), pero no te exime de cuidarla. Vamos a usar este esquema porque es el estándar actual.

APK vs AAB: el APK es el paquete instalable clásico, con todo para todos los dispositivos. El AAB (Android App Bundle) es el formato de publicación moderno: subes un solo archivo y Google genera APKs optimizados para cada dispositivo (solo el idioma, la densidad de pantalla y la arquitectura que cada teléfono necesita). El resultado: descargas más pequeñas y usuarios más felices. Google Play exige AAB para apps nuevas desde 2021; el APK quedo para distribución directa fuera de la tienda.

El build de release se configura en el `build.gradle` del modulo app con dos bloques: `buildTypes { release { ... } }` donde activas `minifyEnabled` (R8 ofusca y reduce el código) y `shrinkResources` (elimina recursos no usados), y `signingConfigs` donde apuntas a tu keystore. Las contraseñas NUNCA van escritas en el build.gradle si el proyecto está en Git: van en variables de entorno o en un archivo `keystore.properties` que agregas al `.gitignore`.

`versionCode` y `versionName`: cada vez que subes una actualización, el `versionCode` (un número entero) debe ser mayor que el anterior; es lo que Google usa para saber que es más nuevo. El `versionName` (texto como "1.2.0") es lo que ven los usuarios. Olvidar subir el versionCode es el error más común al publicar actualizaciones: Play Console te rechaza el paquete sin contemplaciones.

R8 (el ofuscador) merece una mención especial: reduce el tamaño de tu app eliminando código muerto y renombra tus clases a nombres cortos para que nadie pueda leer tu código fácilmente. Pero ojo: si usas reflexión o librerías que la usan (algunas de JSON), necesitas reglas en `proguard-rules.pro` para no romper nada. Prueba tu build de release en un dispositivo real antes de subirlo: un crash que solo pasa en release por culpa de R8 es un clásico doloroso.

Clase 30: Google Play Console - de tu computadora a millones de teléfonos

Play Console (play.google.com/console) es el panel donde nace tu vida pública como desarrollador. El registro cuesta 25 dólares de pago único y requiere verificar tu identidad. Una vez dentro, creas tu primera app eligiendo el nombre, el idioma por defecto y si es app o juego.

La ficha de Play Store es tu aparador: es lo que decide si alguien instala tu app o pasa de largo. Sus piezas: el título (máximo 30 caracteres, con tu palabra clave principal), la descripción corta (80 caracteres: tu pitch de elevator) y la descripción larga (hasta 4000 caracteres donde cuentas beneficios, no características: "organiza tus gastos en segundos" vende más que "usa Room y Retrofit"). El icono (512x512), la imagen destacada (1024x500) y las capturas de pantalla (mínimo 2, se permiten hasta 8 por tipo de dispositivo) son tu publicidad visual: usalas para mostrar la app en acción, con textos breves que expliquen cada pantalla. Puedes agregar un video promocional de YouTube: las fichas con video convierten notablemente mejor.

Las capturas merecen estrategia: no subas pantallazos crudos del emulador. Las apps que destacan usan marcos de teléfono, fondos de color de marca y una frase por captura ("Ve tus gastos de un vistazo", "Funciona sin internet"). Hay herramientas gratuitas para generarlas; lo importante es que cuenten una historia en 3 segundos, porque eso es lo que el usuario les dedica.

La clasificación de contenido es un cuestionario obligatorio (basado en el sistema IARC) donde declaras si tu app tiene violencia, lenguaje fuerte, compras integradas, publicidad, etc. Responde con honestidad: si mientes y Google lo detecta, tu app se suspende. La mayoría de las apps utilitarias quedan en "Para todos" o "Mayores de 10".

La política de privacidad es obligatoria si tu app maneja datos personales (y casi todas lo hacen: hasta un login con email cuenta). Debe estar publicada en una URL pública y enlazada en la ficha. Puedes generarla con plantillas, pero debe describir de verdad que datos recoges y para que. Si tu app pide ubicación, cámara o contactos, la política debe explicarlo; esto conecta directo con la Unidad 9.

El lanzamiento se hace por etapas, y esta es la práctica profesional: primero "Prueba interna" (hasta 100 testers con enlace directo, ideal para que tus amigos la rompan antes que el público), luego "Prueba cerrada" o "abierta" si quieres más gente, y al final "Producción". Incluso en producción puedes hacer lanzamiento gradual (primero al 10 por ciento de usuarios) para detectar crashes antes de llegar a todos. Nunca lances al 100 por ciento una versión que no probó nadie más que tú.

Los rechazos de Google son normales la primera vez. Los motivos más comunes: descripción engañosa, falta de política de privacidad, pedir permisos sensibles sin justificarlos, y crashes en dispositivos de prueba. Cada rechazo viene con el motivo y los pasos para corregirlo: lo corriges, subes un nuevo AAB con versionCode mayor y reenvías. No es un castigo, es un trámite. Los desarrolladores profesionales lo viven como parte del ciclo de release.

Después del lanzamiento empieza la vida real: monitorea los reportes de ANR y crashes en el panel de "Calidad" de la consola, responde las resenas de los usuarios (las respuestas del desarrollador se muestran publicamente y una buena respuesta convierte una resena de 1 estrella en un usuario fiel), y publica actualizaciones con regularidad. Una app abandonada sin actualizar pierde posicion en la tienda.

Ejemplos paso a paso

Ejemplo 1: La capa local - entidad, DAO y base de datos Room

Primero la entidad, que es la tabla `peliculas`:

```
// La entidad es la tabla. Cada propiedad es una columna.
@Entity(tableName = "peliculas")
data class PeliculaEntity(
    @PrimaryKey val id: Int,
    val titulo: String,
    val descripcion: String,
    val urlImagen: String,
    val calificacion: Double,
    val esFavorita: Boolean = false
)
```

El DAO con las operaciones que necesitamos:

```
@Dao
interface PeliculaDao {
    // Flow: la UI se actualiza sola cada vez que cambia la tabla.
    @Query("SELECT * FROM peliculas ORDER BY calificacion DESC")
    fun obtenerTodas(): Flow<List<PeliculaEntity>>

    // REPLACE: si la pelicula ya existe, se actualiza; asi no se duplican al
    refrescar.
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertarTodas(peliculas: List<PeliculaEntity>)

    @Query("UPDATE peliculas SET esFavorita = :favorita WHERE id = :id")
    suspend fun marcarFavorita(id: Int, favorita: Boolean)
}
```

Y la base de datos que las junta:

```
@Database(entities = [PelículaEntity::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun películaDao(): PelículaDao
}
```

Tres líneas de base de datos y ya tienes persistencia reactiva. Nota que `obtenerTodas()` devuelve `Flow`: no es una lista estática, es un río de datos que emite una nueva lista cada vez que la tabla cambia. Esa es la pieza que hace mágica la UI.

Ejemplo 2: La capa de red - DTO, API Retrofit y mapeo

El DTO refleja el JSON del servidor (puede llamarse distinto a tu entidad):

```
// Lo que devuelve la API. Los nombres con SerializedName se adaptan al JSON real.
data class PelículaDto(
    @SerializedName("id") val id: Int,
    @SerializedName("title") val titulo: String,
    @SerializedName("overview") val descripcion: String,
    @SerializedName("poster_path") val rutaImagen: String,
    @SerializedName("vote_average") val calificación: Double
)

// Respuesta paginada típica de APIs reales.
data class RespuestaPelículas(
    @SerializedName("results") val resultados: List<PelículaDto>
)
```

La interfaz de Retrofit:

```
interface PelículaApi {
    @GET("películas/populares")
    suspend fun obtenerPopulares(): RespuestaPelículas
}
```

Y la función de mapeo, el traductor entre los dos mundos:

```
// Convierte lo que viene de internet a lo que guardamos en la base.
// Si la API cambia sus nombres, SOLO se toca esta funcion.
fun PeliculaDto.aEntidad(): PeliculaEntity {
    return PeliculaEntity(
        id = id,
        titulo = titulo,
        descripcion = descripcion,
        urlImagen = "https://image.tmdb.org/t/p/w500$rutaImagen",
        calificacion = calificacion
    )
}
```

La regla de oro: la UI nunca ve un `PeliculaDto`. El DTO vive y muere en la capa de red. Así, si mañana la API renombra `title` a `name`, cambias una línea del mapeo y la app sigue funcionando.

Ejemplo 3: El Repository - el cerebro offline-first

```

class PeliculaRepository(
    private val api: PeliculaApi,
    private val dao: PeliculaDao
) {
    // La UI observa esto: siempre sale de la base local (rapido, funciona sin
    internet).
    val peliculas: Flow<List<PeliculaEntity>> = dao.obtenerTodas()

    // Refresca desde internet en segundo plano. Si falla la red,
    // la excepcion se captura y los datos locales siguen intactos.
    suspend fun refrescar() {
        try {
            val respuesta = api.obtenerPopulares()
            val entidades = respuesta.resultados.map { it.aEntidad() }
            dao.insertarTodas(entidades)
        } catch (e: IOException) {
            // Sin internet: no hacemos nada dramático. Los datos locales siguen
            ahí.
        } catch (e: HttpException) {
            // El servidor respondió con error: igual, seguimos con lo local.
        }
    }

    suspend fun alternarFavorita(pelicula: PeliculaEntity) {
        dao.marcarFavorita(pelicula.id, !pelicula.esFavorita)
    }
}

```

Lee el flujo completo: `peliculas` es un Flow que emite desde Room al instante. `refrescar()` trae lo nuevo de la API, lo mapea y lo guarda; al guardarse, Room emite de nuevo y la UI se actualiza sola. Los dos `catch` son la diferencia entre una app que se cierra sin internet y una que sigue funcionando. Este archivo de 30 líneas es el corazón de la arquitectura moderna de Android.

Ejemplo 4: ViewModel y Activity - la UI reactiva

```

class PeliculasViewModel(private val repository: PeliculaRepository) : ViewModel()
{
    // La UI observa este StateFlow: lista actual de películas.
    val peliculas: StateFlow<List<PeliculaEntity>> = repository.peliculas
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())

    init {
        // Al nacer el ViewModel, refrescamos en segundo plano.
        viewModelScope.launch {
            repository.refrescar()
        }
    }

    fun onFavoritaClick(pelicula: PeliculaEntity) {
        viewModelScope.launch {
            repository.alternarFavorita(pelicula)
        }
    }
}

```

Y en la Activity se observa así:

```

class ListaActivity : AppCompatActivity() {

    private val viewModel: PeliculasViewModel by viewModels {
        // Factory manual: le entregamos el repository ya construido.
        PeliculasViewModelFactory((application as MiApp).repository)
    }

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

        val adaptador = PeliculasAdapter { pelicula ->
            viewModel.onFavoritaClick(pelicula)
        }
        findViewById<RecyclerView>(R.id.recyclerPeliculas).adapter = adaptador

        // Observamos el Flow: cada emision actualiza la lista sola.
        lifecycleScope.launch {
            repeatOnLifecycle(Lifecycle.State.STARTED) {
                viewModel.peliculas.collect { lista ->
                    adaptador.enviarLista(lista)
                }
            }
        }
    }
}

```

`repeatOnLifecycle` es importante: la coleccion solo está activa mientras la pantalla es visible, así no gastas recursos en segundo plano. El adaptador recibe la lambda del click de favorita y la deriva al ViewModel; la Activity nunca toca la base de datos ni la API directamente. Arquitectura limpia de punta a punta.

Ejemplo 5: El contenedor manual - cableando las dependencias

La clase Application crea una sola vez cada pieza y la Activity las pide de ahí:


```

class MiApp : Application() {

    // La base de datos se crea una sola vez para toda la app.
    val database: AppDatabase by lazy {
        Room.databaseBuilder(this, AppDatabase::class.java, "peliculas.db").build()
    }

    // Retrofit se crea una sola vez.
    private val api: PeliculaApi by lazy {
        Retrofit.Builder()
            .baseUrl("https://api.miservidor.com/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(PeliculaApi::class.java)
    }

    // El repository recibe sus dependencias por constructor (inyeccion manual).
    val repository: PeliculaRepository by lazy {
        PeliculaRepository(api, database.peliculaDao())
    }
}

// Factory para que el ViewModel reciba el repository.
class PeliculasViewModelFactory(
    private val repository: PeliculaRepository
) : ViewModelProvider.Factory {
    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return PeliculasViewModel(repository) as T
    }
}

```

No olvides declarar `MiApp` en el manifest: `android:name=".MiApp"` dentro de ``. Con esto, cada pieza se crea una vez (`by lazy`), las dependencias fluyen por constructor y todo es reemplazable en tests. Es DI sin librerías: simple y transparente.

Ejemplo 6: Generar tu keystore (una sola vez en la vida de la app)

Abre la terminal y ejecuta (cambia los datos por los tuyos):

```

keytool -genkeypair -v -keystore mi-app-release.jks \
    -alias mi-app-key -keyalg RSA -keysize 2048 -validity 10000

```

Te pedirá la contraseña del almacén, la de la llave y tus datos (nombre, organización, ciudad, país). La validez de 10000 días (27 años) es la recomendada por Google. El resultado es `mi-app-release.jks`: GUÁRDALO en dos lugares seguros y anota las contraseñas en tu gestor de contraseñas. Este archivo vale más que tu código: sin él no hay actualizaciones.

Ejemplo 7: Configurar el build de release firmado

Primero crea el archivo `keystore.properties` en la raíz del proyecto (junto a `settings.gradle`) y agrégalo al `.gitignore` para que NUNCA se suba a Git:

```
storeFile=mi-app-release.jks
storePassword=TU_CONTRASENA_DEL_ALMACEN
keyAlias=mi-app-key
keyPassword=TU_CONTRASENA_DE_LA_LLAVE
```

Luego en el `build.gradle` del módulo app:

```

android {
    // ...
    defaultConfig {
        applicationId = "com.lucasegaacademy.mispeliculas"
        versionCode = 1
        versionName = "1.0.0"
    }

    // Lee las contraseñas del archivo privado (que no está en Git).
    val keystoreProps = Properties().apply {
        load(rootProject.file("keystore.properties").inputStream())
    }

    signingConfigs {
        create("release") {
            storeFile = file(keystoreProps["storeFile"] as String)
            storePassword = keystoreProps["storePassword"] as String
            keyAlias = keystoreProps["keyAlias"] as String
            keyPassword = keystoreProps["keyPassword"] as String
        }
    }

    buildTypes {
        release {
            signingConfig = signingConfigs.getByName("release")
            isMinifyEnabled = true // R8: ofusca y reduce el código
            isShrinkResources = true // elimina imágenes y layouts sin uso
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }
}

```

Línea por línea de lo crítico: `signingConfigs.create("release")` define cómo firmar; las contraseñas salen del archivo privado, nunca escritas en el gradle. `isMinifyEnabled = true` activa R8 (tu APK/AAB se hace más pequeño y tu código más difícil de leer). `isShrinkResources = true` borra recursos huérfanos. Para generar el paquete: en Android Studio, menú Build > Generate Signed Bundle/APK, o por terminal `./gradlew bundleRelease`. El AAB firmado aparece en `app/build/outputs/bundle/release/`. Ese archivo es el que subes a Play Console.

Verificación antes de subir: instala el AAB en un teléfono real (con `bundletool` o subiéndolo a la pista de prueba interna) y prueba el flujo completo. Un crash que solo existe en release (típico por R8 ofuscando algo que usa reflexión) se detecta aquí, no después de publicar.

Práctica

Ejercicio 1: Agregar búsqueda local a MisPeliculas

Enunciado: Agrega una función de búsqueda por título que filtre las películas guardadas en Room (sin ir a la API). Debes agregar: 1) la consulta al DAO, 2) el método en el Repository, 3) la exposición en el ViewModel con un texto de búsqueda observable, 4) el SearchView en la Activity que actualice ese texto.

Solución:

```

// 1. DAO: consulta con LIKE para búsqueda parcial.
@Query("SELECT * FROM peliculas WHERE titulo LIKE '%' || :texto || '%' ORDER BY
calificacion DESC")
fun buscarPorTitulo(texto: String): Flow<List<PeliculaEntity>>

// 2. Repository: expone la búsqueda.
fun buscar(texto: String): Flow<List<PeliculaEntity>> = dao.buscarPorTitulo(texto)

// 3. ViewModel: el texto de búsqueda es un StateFlow; la lista reacciona con
flatMapLatest.
private val textoBusqueda = MutableStateFlow("")
val peliculas: StateFlow<List<PeliculaEntity>> = textoBusqueda
    .flatMapLatest { texto ->
        if (texto.isBlank()) repository.peliculas else repository.buscar(texto)
    }
    .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), emptyList())

fun onTextoBusquedaCambio(nuevoTexto: String) {
    textoBusqueda.value = nuevoTexto
}

// 4. Activity: el SearchView alimenta al ViewModel.
searchView.setOnQueryTextListener(object : SearchView.OnQueryTextListener {
    override fun onQueryTextChange(nuevo: String): Boolean {
        viewModel.onTextoBusquedaCambio(nuevo)
        return true
    }
    override fun onQueryTextSubmit(query: String) = true
})

```

Explicación de la solución: `flatMapLatest` es el operador clave: cada vez que cambia el texto, cancela la búsqueda anterior y empieza una nueva contra Room. Como todo sale de la base local, la búsqueda es instantánea y funciona sin internet. Fíjate que no se tocó la API ni el mapeo: la arquitectura en capas hace que agregar funciones sea sumar piezas, no reescribir.

Ejercicio 2: Pantalla de detalle con favorita persistente

Enunciado: Al tocar una película de la lista se abre `DetalleActivity` que muestra título, descripción, imagen (con Coil) y calificación, más un botón de favorito que persiste en Room. El detalle debe funcionar sin internet usando los datos locales.

Solución:

```

// DAO: obtener una pelicula por id.
@Query("SELECT * FROM peliculas WHERE id = :id")
fun obtenerPorId(id: Int): Flow<PeliculaEntity?>

// ViewModel del detalle.
class DetalleViewModel(private val repository: PeliculaRepository, id: Int) :
    ViewModel() {
    val pelicula: StateFlow<PeliculaEntity?> = repository.obtenerPorId(id)
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), null)

    fun onFavoritaClick() {
        viewModelScope.launch {
            pelicula.value?.let { repository.alternarFavorita(it) }
        }
    }
}

// Activity: observa y pinta. La imagen se carga con Coil.
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.pelicula.collect { p ->
            p ?: return@collect
            textTitulo.text = p.titulo
            textDescripcion.text = p.descripcion
            textCalificacion.text = "Calificacion: ${p.calificacion}"
            imagePoster.load(p.urlImagen) // Coil: una linea para cargar la imagen
            botonFavorita.text = if (p.esFavorita) "Quitar de favoritas" else
"Marcar favorita"
        }
    }
}

```

Explicación de la solución: el detalle observa `obtenerPorId` como Flow, así que si la favorita cambia (desde esta pantalla o desde la lista), ambas se actualizan solas. Coil (`imageView.load(url)`) maneja descarga, cache y memoria por ti; es el estandar moderno y reemplaza a las soluciones manuales. La Activity recibe el id por Intent extras. Todo funciona sin internet porque los datos salen de Room.

Ejercicio 3: Estrategia de refresco con indicador de "datos viejos"

Enunciado: Mejora el `refrescar()` del Repository para que guarde la fecha del último refresco exitoso en SharedPreferences (o DataStore). La UI debe mostrar "Actualizado hace X minutos" y, si los datos tienen más de 1 hora, refrescar automaticamente al abrir la app.

Solución:

```
class PeliculaRepository(
    private val api: PeliculaApi,
    private val dao: PeliculaDao,
    private val prefs: SharedPreferences
) {
    val peliculas: Flow<List<PeliculaEntity>> = dao.obtenerTodas()

    suspend fun refrescar() {
        try {
            val respuesta = api.obtenerPopulares()
            dao.insertarTodas(respuesta.resultados.map { it.aEntidad() })
            // Solo guardamos la fecha si el refresco fue exitoso.
            prefs.edit().putLong("ultimo_refresco",
                System.currentTimeMillis()).apply()
        } catch (e: IOException) {
            // Sin internet: no tocamos la fecha; los datos siguen marcados como
            viejos.
        }
    }

    fun minutosDesdeUltimoRefresco(): Long {
        val ultimo = prefs.getLong("ultimo_refresco", 0)
        if (ultimo == 0L) return Long.MAX_VALUE // nunca se refresco
        return (System.currentTimeMillis() - ultimo) / 60000
    }
}

// En el ViewModel, al iniciar:
init {
    viewModelScope.launch {
        if (repository.minutosDesdeUltimoRefresco() > 60) {
            repository.refrescar()
        }
    }
}
```

Explicación de la solución: la fecha solo se actualiza en el camino exitoso; si la red falla, los datos conservan su marca de "viejos" y la app lo sabe. `Long.MAX\_VALUE` para "nunca" garantiza que la primera apertura siempre refresque. La UI puede mostrar el texto con `minutosDesdeUltimoRefresco()` formateado ("hace 5 min", "hace 2 h"). Es una mejora pequeña con gran impacto en la experiencia: el usuario siempre sabe que tan frescos están sus datos.

Ejercicio 4: Firma tu app de extremo a extremo

Enunciado: Toma cualquier app que hayas hecho en el curso y dejala lista para publicar: 1) genera su keystore con keytool, 2) crea el `keystore.properties` y agregalo al `.gitignore`, 3) configura el `signingConfigs` y el `buildType release` en el build.gradle, 4) genera el AAB con `./gradlew bundleRelease` y verifica que el archivo exista y pese lo esperado.

Solución (pasos verificables):

1. Genera el keystore: `keytool -genkeypair -v -keystore mi-app.jks -alias mi-key -keyalg RSA -keysize 2048 -validity 10000`. Guarda el archivo FUERA del proyecto (por ejemplo en `~/kestores/`) o, si lo dejas en el proyecto, asegurate de que el `.gitignore` lo excluya (*.jks).
2. Crea `keystore.properties` en la raíz del proyecto con las cuatro líneas (storeFile con RUTA ABSOLUTA al jks, storePassword, keyAlias, keyPassword).
3. Agrega al `.gitignore`: `keystore.properties` y `\*.jks`.
4. Copia el bloque `signingConfigs` y `buildTypes` del Ejemplo 7 a tu build.gradle, ajustando el `applicationId`.
5. Ejecuta `./gradlew bundleRelease` y confirma que existe `app/build/outputs/bundle/release/app-release.aab`.
6. Verificación extra: abre el AAB con `bundletool` o subelo a la pista de "Prueba interna" de Play Console para confirmar que la firma es válida.

Explicación de la solución: el punto crítico es el paso 3: si el properties o el jks llegan a Git, tus secretos quedan expuestos para siempre (aunque los borres después, quedan en el historial). La ruta absoluta en `storeFile` evita que el build falle cuando otro miembro del equipo (o tu futura yo) clone el proyecto en otra carpeta. Este ejercicio, hecho una vez con calma, te ahorra el pánico del "día del lanzamiento".

Ejercicio 5: Redacta la ficha de Play Store de MisPeliculas

Enunciado: Escribe la ficha completa como si fueras a publicarla: título (máximo 30 caracteres), descripción corta (máximo 80 caracteres), descripción larga (mínimo 300 caracteres, enfocada en beneficios), y la lista de 4 capturas de pantalla que subirías describiendo que muestra cada una y que frase llevaría.

Solución de ejemplo:

Título: "MisPeliculas: tu cine en el bolsillo" (34 caracteres: demasiado largo; se recorta a "MisPeliculas: cine en tu bolsillo").

Descripción corta: "Descubre películas populares, guárdalas y márcalas como favoritas sin internet."

Descripción larga: "MisPelículas es tu compañero de cine. Explora las películas más populares del momento con sus calificaciones y sinopsis, y guárdalas en tu lista de favoritas. Como tus datos se guardan en el teléfono, puedes consultar tu colección en el avión, en el metro o donde no haya señal: la app siempre responde al instante. Marca tus favoritas con un toque y ten tu próxima noche de películas siempre lista. Sin cuentas complicadas, sin esperas, sin internet obligatorio: solo tu cine, en tu bolsillo."

Capturas: 1) La lista de películas con posters, frase "Las más populares, al instante". 2) El detalle de una película, frase "Sinopsis y calificación de un vistazo". 3) La búsqueda filtrando, frase "Encuentra tu próxima favorita". 4) La app en modo avión mostrando los datos guardados, frase "Funciona sin internet".

Explicación de la solución: nota que la descripción larga no menciona Room, Retrofit ni Flow: al usuario no le importa tu stack, le importan sus noches de películas. Cada captura cuenta UNA idea con UNA frase; el usuario decide en 3 segundos. La captura 4 es la más vendedora porque muestra el diferenciador (offline-first). Esa es la mentalidad de ficha: beneficios, no características.

Ejercicio 6: Plan de lanzamiento y respuesta a un rechazo

Enunciado (teórico): Escribe tu plan de lanzamiento en 5 pasos usando las pistas de Play Console, y luego responde: Google rechazó tu app con el motivo "La declaración de privacidad no coincide con los permisos solicitados: la app solicita ubicación precisa pero la política no la menciona". ¿Qué haces, paso a paso?

Solución:

Plan de lanzamiento: 1) Subir el AAB a la pista de Prueba interna y compartir el enlace con 10 testers por 1 semana. 2) Corregir los crashes reportados en el panel de Calidad. 3) Pasar a Prueba cerrada con 100 usuarios por 2 semanas para validar en más dispositivos. 4) Lanzar en Producción al 20 por ciento con lanzamiento gradual. 5) Si no hay crashes nuevos en 3 días, subir al 100 por ciento.

Respuesta al rechazo: 1) Leer el motivo completo en Play Console (sección Políticas). 2) Actualizar la política de privacidad agregando un parrafo que explique que datos de ubicacion se recogen, para que se usen y que no se comparten. 3) Verificar que el permiso de ubicacion de verdad se necesite; si no, quitarlo del manifest (menos permisos = menos friccion). 4) Subir un nuevo AAB con `versionCode` incrementado. 5) Reenviar a revision con una nota explicando el cambio. 6) No discutir con el revisor: corregir y reenviar es más rápido.

Explicación de la solución: el lanzamiento gradual existe para que los errores le peguen al 20 por ciento y no al 100 por ciento; usarlo es gratis y te salva la reputacion. Y ante un rechazo, la regla es no tomarlo personal: es un checklist, no un juicio. La mayoría de los rechazos se resuelven en un día siguiendo los pasos que la propia consola te indica.

DataStore: el reemplazo moderno de SharedPreferences

En el ejercicio 3 usamos SharedPreferences para guardar la fecha del último refresco. Funciona, pero Google recomienda hoy DataStore: es asincrono por diseño (no bloquea el hilo principal), es seguro contra corrupcion de datos y se integra con Flow y corrutinas de forma natural. Hay dos sabores: Preferences DataStore (clave-valor, el reemplazo directo de SharedPreferences) y Proto DataStore (objetos tipados). Para este curso basta el primero:

```
// 1. Define el DataStore una sola vez (top-level, fuera de clases).
val Context.misDatos: DataStore<Preferences> by preferencesDataStore(name =
"ajustes")

// 2. Clave tipada para la fecha del ultimo refresco.
val ULTIMO_REFRESCO = longPreferencesKey("ultimo_refresco")

// 3. Guardar (suspend: va en corrutina).
suspend fun guardarUltimoRefresco(context: Context) {
    context.misDatos.edit { prefs ->
        prefs[ULTIMO_REFRESCO] = System.currentTimeMillis()
    }
}

// 4. Leer como Flow: la UI reacciona sola si el valor cambia.
fun flujoUltimoRefresco(context: Context): Flow<Long> {
    return context.misDatos.data.map { prefs ->
        prefs[ULTIMO_REFRESCO] ?: 0L
    }
}
```

La diferencia filosofica con SharedPreferences: nada es instantaneo ni bloqueante; todo es un flujo de datos. Si vienes del mundo del `getLong` directo, al principio parece más código, pero a cambio nunca congelas la pantalla leyendo disco y nunca corrompes el archivo por escribir desde dos hilos a la vez. Para apps nuevas, DataStore es la elección correcta.

Migraciones de Room: cuando tu tabla cambia

La versión 1 de tu base tiene 6 columnas. Un día decides agregar `fechaEstreno`. Si solo cambias la entidad y subes `version = 2`, los usuarios que actualicen pierden... nada, si haces la migración bien; TODO, si la haces mal (Room borra y recrea la tabla si no le das la migración, salvo que configures `fallbackToDestructiveMigration`, que es justamente "borra todo").

La migración es un objeto que dice como pasar de la versión 1 a la 2:

```
val MIGRACION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        // Agregamos la columna nueva con un valor por defecto para las filas
        // existentes.
        db.execSQL("ALTER TABLE peliculas ADD COLUMN fechaEstreno TEXT NOT NULL
        DEFAULT ''")
    }
}

// Y se registra al construir la base:
Room.databaseBuilder(this, AppDatabase::class.java, "peliculas.db")
    .addMigrations(MIGRACION_1_2)
    .build()
```

Reglas de oro: nunca modifiques una migración ya publicada (si te equivocaste, crea la 2 a 3 que lo corrija), prueba las migraciones con datos reales de la versión anterior, y para cambios destructivos avisa al usuario. Room tiene soporte de testing de migraciones (`room-testing`) que verifica el esquema automáticamente: úsalo cuando tu app tenga usuarios de verdad. Perder los datos de tus usuarios en una actualización es la forma más rápida de ganarte resenas de una estrella.

R8 sin miedo: reglas de ProGuard que salvan tu release

R8 ofusca (renombrar clases y metodos) y elimina lo que cree que no se usa. El problema: si tu codigo usa reflexion o serializacion por nombres (como Gson, que busca campos por su nombre original), R8 puede romperlo en release mientras en debug todo funciona. El sintoma es el peor de todos: la app funciona perfecto en tus pruebas y se cierra en manos del usuario.

La solución es el archivo `proguard-rules.pro`, donde le dices a R8 que no toque ciertas cosas:

```
# Conserva tus modelos de datos si usas Gson con reflexion:
-keep class com.lucasegaacademy.mispeliculas.modelo.** { *; }

# Si usas Retrofit, conserva las anotaciones y los metodos de la interfaz:
-keepattributes Signature, InnerClasses, EnclosingMethod
-keepattributes RuntimeVisibleAnnotations, RuntimeVisibleParameterAnnotations

# Conserva los miembros serializados por nombre:
-keepclassmembers class * {
    @com.google.gson.annotations.SerializedName <fields>;
}
```

La estrategia profesional: genera el AAB release, instalalo en un teléfono real y prueba TODOS los flujos (login, lista, detalle, favoritas) antes de subirlo. El 90 por ciento de los equipos descubre sus problemas de R8 en esa prueba de 15 minutos. Los que se la saltan los descubren en las resenas de Play Store.

Checklist final pre-lanzamiento (la lista del profesional)

Antes de subir tu AAB a produccion, recorre esta lista como un piloto antes de despegar. Cada punto viene de un error real que alguien ya cometio:

1. El `versionCode` es mayor que el de la version anterior publicada.
2. El AAB fue generado en modo release, firmado con tu llave de subida, y probado en un dispositivo físico.
3. Probaste los flujos criticos en release: abrir, lista, detalle, favorita, rotar la pantalla, modo avion, negar permisos.
4. Los textos de la app están revisados: sin "lorem ipsum", sin TODOs visibles, sin logs de depuracion que muestren datos sensibles.
5. El icono de la app existe en todas las densidades y se ve bien en el launcher.
6. La ficha de Play Store esta completa: titulo, descripcion corta y larga, icono 512, imagen destacada, al menos 2 capturas por formato, categoría y datos de contacto.

7. La política de privacidad está publicada en una URL y enlazada; menciona cada permiso sensible que la app usa.
8. La clasificación de contenido (IARC) está respondida con honestidad.
9. Si la app usa ubicación en segundo plano, cámara o accesibilidad, la justificación está lista para la revisión de Google.
10. Tienes el keystore respaldado en dos lugares y las contraseñas en tu gestor.
11. Definiste la estrategia de lanzamiento: pista de prueba interna primero, luego producción gradual.
12. Sabes dónde ver los crashes (Play Console > Calidad > Informes de fallos) y tienes un plan para la primera semana post-lanzamiento.

Imprime esta lista y pégala junto a tu monitor. El día del lanzamiento, con los nervios, es cuando más se olvidan las cosas obvias.

Ejercicio 9: Migración real de Room (agregar columna sin perder datos)

Enunciado: A la entidad `PeliculaEntity` agréga el campo `fechaEstreno: String = ""`. Escribe la migración de la versión 1 a la 2, actualiza la anotación `@Database` y registra la migración. Explica qué pasaría si olvidaras la migración y usaras `fallbackToDestructiveMigration()`.

Solución:

```

// 1. Entidad actualizada:
@Entity(tableName = "peliculas")
data class PeliculaEntity(
    @PrimaryKey val id: Int,
    val titulo: String,
    val descripcion: String,
    val urlImagen: String,
    val calificacion: Double,
    val esFavorita: Boolean = false,
    val fechaEstreno: String = "" // <-- columna nueva
)

// 2. Version de la base actualizada:
@Database(entities = [PeliculaEntity::class], version = 2)
abstract class AppDatabase : RoomDatabase() {
    abstract fun peliculaDao(): PeliculaDao
}

// 3. Migracion que conserva los datos:
val MIGRACION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("ALTER TABLE peliculas ADD COLUMN fechaEstreno TEXT NOT NULL
DEFAULT ''")
    }
}

// 4. Registro al construir:
Room.databaseBuilder(contexto, AppDatabase::class.java, "peliculas.db")
    .addMigrations(MIGRACION_1_2)
    .build()

```

Explicación de la solución: el `DEFAULT ""` es obligatorio porque las filas existentes no tienen ese dato y la columna es NOT NULL; sin default, la migracion falla. Si en vez de esto usaras `fallbackToDestructiveMigration()`, Room detectaria que no hay migracion y BORRARIA la tabla para recrearla: tus usuarios perderian todas sus peliculas favoritas al actualizar. Por eso el fallback destructivo solo se usa en desarrollo, nunca en una app con usuarios reales.

Ejercicio 10: Migrar el "último refresco" de SharedPreferences a DataStore

Enunciado: Reescribe el ejercicio 3 usando DataStore en vez de SharedPreferences. El Repository debe exponer `ultimoRefresco: Flow<Long>` y una funcion `suspend fun refrescar()` que guarde la fecha con DataStore. La Activity debe mostrar "Actualizado hace X min" observando el Flow.

Solución:

```
class PeliculaRepository(
    private val api: PeliculaApi,
    private val dao: PeliculaDao,
    private val context: Context
) {
    val peliculas: Flow<List<PeliculaEntity>> = dao.obtenerTodas()
    val ultimoRefresco: Flow<Long> = flujoUltimoRefresco(context)

    suspend fun refrescar() {
        try {
            val respuesta = api.obtenerPopulares()
            dao.insertarTodas(respuesta.resultados.map { it.aEntidad() })
            guardarUltimoRefresco(context) // DataStore: suspend, sin bloqueos
        } catch (e: IOException) {
            // Sin red: no se actualiza la fecha.
        }
    }
}

// Activity: muestra la antigüedad observando el Flow.
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        repository.ultimoRefresco.collect { millis ->
            textActualizado.text = if (millis == 0L) "Aun no se actualiza"
            else "Actualizado hace ${((System.currentTimeMillis() - millis) / 60000)}
min"
        }
    }
}
```

Explicación de la solución: la fecha dejó de ser un valor que "preguntas" para ser un flujo que "observas": si otro proceso actualiza el dato, la pantalla se entera sola. `guardarUltimoRefresco` es suspend y no bloquea; el `edit` de DataStore es transaccional y seguro. Este es el patron moderno completo: UI reactiva de punta a punta, del DAO al DataStore.

El adaptador con DiffUtil: listas que se actualizan sin parpadear

En los ejemplos usamos `adaptador.enviarLista(lista)`. Si adentro haces `notifyDataSetChanged()`, el RecyclerView redibuja TODO cada vez que llega un dato nuevo: se pierde la animación, parpadea y gasta batería. La solución profesional es DiffUtil, que

calcula exactamente qué cambió (que item se agregó, cuál se movió, cuál se actualizó) y anima solo eso:

```
class PeliculasAdapter(  
    private val onFavoritaClick: (PeliculaEntity) -> Unit  
) : ListAdapter<PeliculaEntity, PeliculasAdapter.PeliculaVH>(Comparador) {  
  
    // DiffUtil: define cuando dos items son "el mismo" y cuando "cambio su  
    contenido".  
    object Comparador : DiffUtil.ItemCallback<PeliculaEntity>() {  
        override fun areItemsTheSame(a: PeliculaEntity, b: PeliculaEntity): Boolean  
{  
            return a.id == b.id // mismo id = mismo item, aunque cambien sus datos  
        }  
        override fun areContentsTheSame(a: PeliculaEntity, b: PeliculaEntity):  
Boolean {  
            return a == b // data class: compara todos los campos automaticamente  
        }  
    }  
  
    inner class PeliculaVH(val vista: View) : RecyclerView.ViewHolder(vista) {  
        fun enlazar(pelicula: PeliculaEntity) {  
            vista.findViewById<TextView>(R.id.textTitulo).text = pelicula.titulo  
  
vista.findViewById<ImageView>(R.id.imagePoster).load(pelicula.urlImagen)  
            val boton = vista.findViewById<Button>(R.id.botonFavorita)  
            boton.text = if (pelicula.esFavorita) "En favoritas" else "Agregar"  
            boton.setOnClickListener { onFavoritaClick(pelicula) }  
        }  
    }  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): PeliculaVH {  
        val vista = LayoutInflater.from(parent.context)  
            .inflate(R.layout.item_pelicula, parent, false)  
        return PeliculaVH(vista)  
    }  
  
    override fun onBindViewHolder(holder: PeliculaVH, position: Int) {  
        holder.enlazar(getItem(position))  
    }  
}  
  
// En la Activity, en vez de enviarLista manual: submitList hace la magia.  
adaptador.submitList(lista)
```


`ListAdapter` + `submitList` es el duo moderno: tu solo entregas la lista nueva y el adaptador calcula el diff en un hilo secundario y anima los cambios. Marcar una favorita ahora se ve como un cambio suave en ese item, no como un parpadeo de toda la pantalla. Es un detalle que los usuarios sienten aunque no sepan nombrarlo: "esta app se siente rapida".

Coil a fondo: imágenes sin dolor

Coil (Coroutine Image Loader) es la libreria moderna de carga de imágenes, hecha en Kotlin con corrutinas. Lo básico es `imageView.load(url)`, pero en una app real necesitas más:

```
imageView.load(pelicula.urlImagen) {  
    placeholder(R.drawable.placeholder_cine) // mientras descarga  
    error(R.drawable.error_imagen)          // si la URL falla  
    crossfade(true)                        // transicion suave al aparecer  
    transformations(CircleCropTransformation()) // recorte circular para avatares  
    memoryCachePolicy(CachePolicy.ENABLED) // cache en memoria (por defecto)  
}
```

Tres cosas que Coil hace por ti y que son faciles de arruinar a mano: cancela la descarga si la vista se recicla (sin esto, las imágenes se mezclan al hacer scroll rápido), escala el bitmap al tamaño real del ImageView (sin esto, el OutOfMemoryError de la Unidad 9), y cachea en memoria y disco (sin esto, cada scroll recargaria todo). Si vienes de tutoriales viejos que usan Picasso o Glide, Coil es el equivalente moderno y el recomendado hoy.

Recursos por configuracion: una app que se adapta sola

Android elige recursos automaticamente según el dispositivo: `drawable-hdpi` vs `drawable-xxhdpi` para densidades, `values-es` vs `values-en` para idiomas, `layout-land` para horizontal. Para tu ficha de Play Store esto importa: una app que se ve bien en teléfono y tablet, en espanol e inglés, tiene más mercado.

Lo mínimo profesional: todos tus textos en `res/values/strings.xml` (nunca texto quemado en el layout), con una traduccion al inglés en `res/values-en/strings.xml`. No necesitas traducir perfecto el día uno; pero si tu codigo ya usa `@string/...`, agregar idiomas después es copiar un archivo. Las apps que queman texto en los layouts condenan su internacionalizacion para siempre.

Ejercicio 11: Migra tu adaptador a ListAdapter con DiffUtil

Enunciado: Toma cualquier adaptador de RecyclerView que hayas escrito con `notifyDataSetChanged()` y migralo a `ListAdapter` con `DiffUtil.ItemCallback`, siguiendo el ejemplo de esta sección. Luego verifica visualmente: marca una favorita y observa que solo ese item se anima en vez de parpadear toda la lista.

Solución: sigue el código del apartado DiffUtil paso a paso: 1) cambia la herencia a `ListAdapter<TuModelo, TuViewHolder>(Comparador)`, 2) implementa `areItemsTheSame` con el id único de tu modelo y `areContentsTheSame` con la comparación completa (si tu modelo es data class, `a == b` basta), 3) reemplaza tu método `enviarLista` por `submitList`, 4) en la Activity cambia `adaptador.enviarLista(lista)` por `adaptador.submitList(lista)`.

Explicación de la solución: el error más común es implementar mal `areItemsTheSame` (por ejemplo comparando títulos en vez de ids): si dos películas distintas comparten título, DiffUtil las confunde y la animación sale mal. El id único es sagrado. El segundo error es llamar a `submitList` con la MISMA instancia de lista mutada: DiffUtil compara la lista vieja contra la nueva, y si mutaste la vieja, no detecta cambios. Entrega siempre una lista nueva (Room con Flow ya lo hace por ti).

Cuando Gradle falla: depuración del build sin pánico

Todo desarrollador Android ha visto el muro rojo de Gradle. Los errores más comunes y sus soluciones: "Could not resolve dependency" casi siempre es falta de internet o un repositorio mal escrito en `settings.gradle` (verifica que `google()` y `mavenCentral()` estén declarados). "Duplicate class" significa que dos librerías traen la misma clase: se resuelve excluyendo el módulo duplicado o actualizando versiones. "minSdk incompatible" aparece cuando una librería exige un `minSdk` mayor que el tuyo: o subes tu `minSdk` o buscas una versión vieja de la librería.

El ritual de limpieza cuando nada tiene sentido: Build > Clean Project, luego File > Invalidate Caches y reiniciar. El 80 por ciento de los errores "fantasma" (código que ayer compilaba y hoy no) se arreglan así, porque Gradle cachea cosas que a veces se corrompen. Y si el error menciona tu keystore o el signingConfig, revisa primero que el archivo `keystore.properties` exista y que la ruta del `storeFile` sea correcta: es la causa número uno de builds release fallidos.

Del emulador al dispositivo real: tu matriz de pruebas

El emulador miente por omision: tiene internet perfecto, GPS perfecto, batería infinita y todos los permisos concedidos. Tu app debe probarse en un teléfono real, y no en uno solo. La matriz mínima profesional: un gama baja con Android 10 u 11 (aquí descubres los problemas de memoria y rendimiento), un gama media con Android 13 o 14 (aquí descubres los cambios de permisos modernos) y, si puedes, una tablet (aquí descubres los layouts rotos).

En cada dispositivo, ejecuta el "tour del usuario cruel": instala desde cero, niega todos los permisos, usa la app 5 minutos, activa el modo avión a mitad de una carga, rota la pantalla en cada pantalla, deja la app en segundo plano una hora y vuelve. Cada crash que encuentres tú es una reseña de una estrella que no vas a recibir. Los equipos profesionales automatizan parte de esto, pero el tour manual de 30 minutos antes de cada release no lo reemplaza nada.

Siguientes pasos: adónde ir desde aquí

Este libro te dio los fundamentos sólidos del desarrollo Android nativo con Kotlin. El camino natural después es: primero, Jetpack Compose, el toolkit declarativo moderno de UI que está reemplazando a los layouts XML; con lo que sabes de ViewModel, Flow y corrutinas, Compose se te hará familiar porque usa exactamente los mismos conceptos. Segundo, Hilt para inyección de dependencias sin el código manual del contenedor. Tercero, testing con JUnit, MockK y Espresso: escribir tests es lo que separa al junior del semi-senior en la mayoría de las empresas.

Y lo más importante: publica. Una app publicada con 100 descargas te enseña más que tres cursos más. El portafolio que consigue entrevistas no es el de "se hacer X", es el de "aquí está mi app en Play Store, con estas decisiones técnicas y estos aprendizajes". Cada unidad de este libro fue un escalón; ahora te toca caminar.

Crashlytics: enterate de los crashes antes que tus usuarios

Tu app va a fallar en algún teléfono que nunca probaste: un modelo raro, una versión vieja de Android, una memoria llena. Sin monitoreo, te enteras por una reseña de una estrella tres semanas después. Con Firebase Crashlytics, te enteras en minutos, con el reporte exacto de dónde falló y en cuántos dispositivos.

La integracion es corta: agregas el plugin de Crashlytics en tu build.gradle, inicializas Firebase en tu Application y listo; cada crash no capturado se reporta solo con su stack trace. La práctica profesional es revisar el panel después de cada lanzamiento y definir un umbral: por ejemplo, "si un crash afecta al 1 por ciento de las sesiones, se corrige esta misma semana". Los equipos serios no lanzan la siguiente version con crashes rojos pendientes.

Un detalle importante: Crashlytics también registra "errores no fatales" (excepciones que capturaste pero que indican algo raro, como una API que devolvio un formato inesperado). Usalos para vigilar la salud de tu app sin esperar a que explote. Monitorear no es opcional en produccion: es la diferencia entre reaccionar y adivinar.

Responder resenas: tu marketing gratuito

Las resenas en Play Store no son solo calificacion: son conversacion publica. Cuando respondes una resena (desde Play Console), tu respuesta se muestra junto a ella para que la vean todos los futuros usuarios. Una respuesta empatica a una resena de 1 estrella ("Tienes razon, ese error ya lo corregimos en la version 1.2, gracias por reportarlo") convierte más que diez resenas de 5 estrellas compradas.

Estrategia profesional: responde primero las resenas negativas recientes (son las que más duelen en conversion), luego las preguntas (un usuario con duda es una instalacion a punto de perderse), y agradece las positivas más utiles. Nunca discutas, nunca copies la misma respuesta generica en todas, y nunca pidas "cambianos la calificacion" directamente: informa que el problema se resolvió y deja que el usuario decida. Los usuarios leen las respuestas del desarrollador antes de instalar; es tu aparador de atencion al cliente y es gratis.

Glosario de la unidad

Repository: clase que centraliza el acceso a los datos y decide si vienen de la red, de la base local o de ambas; la UI solo habla con el.

Offline-first: estrategia donde la app muestra primero los datos locales y actualiza desde internet en segundo plano; funciona sin conexion.

DTO (Data Transfer Object): modelo que representa los datos tal como los entrega la API; vive solo en la capa de red.

Mapeo: funcion que convierte un DTO en una entidad de base de datos (o en un modelo de UI); aísla los cambios de la API.

Keystore: archivo que contiene tu llave privada de firma; perderlo impide actualizar tu app. Se respalda y nunca se sube a Git.

Firma digital: sello criptografico que prueba que el paquete lo generaste tu; Google Play la exige para publicar.

APK: paquete instalable clásico de Android, con recursos para todos los dispositivos.

AAB (Android App Bundle): formato de publicación moderno; Google genera desde el APKs optimizados por dispositivo. Obligatorio para apps nuevas en Play.

Build type: variante de compilacion (debug/release); release lleva firma real, ofuscacion y optimizacion.

R8: ofuscador y reductor de codigo que se activa en release con minifyEnabled; achica la app y dificulta la ingenieria inversa.

versionCode: número entero que debe crecer con cada subida a Play; es como Google sabe que una version es más nueva.

versionName: texto de version que ven los usuarios ("1.2.0"); sigue versionado semantico MAYOR.MENOR.PARCHE.

Play Console: panel web donde gestionas tu cuenta de desarrollador, tus apps, sus fichas, lanzamientos y estadísticas.

Ficha de Play Store: página publica de tu app: titulo, descripciones, icono, capturas y video; es tu principal herramienta de marketing.

Pista (track): canal de distribucion en Play Console: prueba interna, cerrada, abierta y produccion; se usan en orden para lanzar con seguridad.

Lanzamiento gradual: publicar primero a un porcentaje de usuarios y ampliar si no hay problemas; práctica profesional estandar.

Profundizacion: arquitectura y tienda a fondo

MVVM a fondo: por que cada capa existe

MVVM (Model-View-ViewModel) no es burocracia: cada capa resuelve un problema real. El Model (Repository + Room + Retrofit) resuelve "de dónde vienen los datos y cómo sobreviven sin internet". El ViewModel resuelve "como sobreviven los datos a la rotacion de pantalla": cuando giras el teléfono, la Activity se destruye y se recrea, pero el ViewModel NO; por eso los

datos no se recargan y el scroll no se pierde. La View (Activity + XML o Compose) resuelve "cómo se ve y cómo se toca", y su única inteligencia es observar y reaccionar.

Si mezclas capas, pagas el precio: lógica de red en la Activity significa que al rotar la pantalla se dispara otra petición y se duplican los datos; acceso a base de datos en el adaptador del RecyclerView significa código imposible de probar. Cada vez que dudes dónde poner una línea de código, pregunta: "esto es dato, es lógica de pantalla o es dibujo?" Dato va al Repository, lógica de pantalla al ViewModel, dibujo a la Activity. Esa pregunta te guía el 95 por ciento de las veces.

Estados de UI con sealed classes: Loading, Exito, Error

En los ejemplos mostramos la lista directamente, pero una app profesional comunica sus estados: "cargando", "contenido", "error" y "vacío". El patrón moderno es una sealed class que representa esos estados y un solo StateFlow en el ViewModel:

```

// Los cuatro estados posibles de cualquier pantalla de datos.
sealed class EstadoUi<out T> {
    data object Cargando : EstadoUi<Nothing>()
    data class Exito<T>(val datos: T) : EstadoUi<T>()
    data class Error(val mensaje: String) : EstadoUi<Nothing>()
    data object Vacio : EstadoUi<Nothing>()
}

class PeliculasViewModel(private val repository: PeliculaRepository) : ViewModel() {
    private val _estado =
MutableStateFlow<EstadoUi<List<PeliculaEntity>>>(EstadoUi.Cargando)
    val estado: StateFlow<EstadoUi<List<PeliculaEntity>>> = _estado

    init {
        viewModelScope.launch {
            // Primero emitimos lo local al instante (offline-first)...
            repository.peliculas.collect { lista ->
                _estado.value = if (lista.isEmpty()) EstadoUi.Vacio
                    else EstadoUi.Exito(lista)
                // ...y en paralelo refrescamos de la red una sola vez.
            }
        }
        viewModelScope.launch {
            _estado.value = EstadoUi.Cargando
            try {
                repository.refrescar()
            } catch (e: Exception) {
                // Si no habia datos locales, mostramos el error;
                // si habia, la lista local ya se esta mostrando.
                if ((_estado.value as? EstadoUi.Exito)?.datos.isNullOrEmpty()) {
                    _estado.value = EstadoUi.Error("Sin conexion. Revisa tu
internet e intenta de nuevo.")
                }
            }
        }
    }
}

```

Y en la Activity, un `when` pinta cada estado:

```
viewModel.estado.collect { estado ->
    when (estado) {
        is EstadoUi.Cargando -> mostrarSpinner()
        is EstadoUi.Exito -> mostrarLista(estado.datos)
        is EstadoUi.Error -> mostrarError(estado.mensaje)
        is EstadoUi.Vacio -> mostrarVacio()
    }
}
```

Este patron elimina los bugs de "pantalla en blanco sin explicación": cada estado tiene su diseño y el usuario siempre sabe qué pasa. Es el estandar en entrevistas técnicas: si muestras este codigo en tu portafolio, hablas el idioma de la industria.

Estrategias de cache: cuando refrescar y cuando no

El ejercicio 3 introdujo el refresco por tiempo (cada hora). Hay más estrategias y cada una sirve a un caso distinto. "Al abrir la app" es simple y funciona para noticias o redes sociales. "Pull-to-refresh" (deslizar hacia abajo) le da el control al usuario y es el patron de Twitter e Instagram: implementalo con `SwipeRefreshLayout` llamando a `repository.refrescar()`. "Por evento" refresca cuando algo cambia (después de crear un elemento, recargas la lista). "Nunca automático" es para datos caros o sensibles (usa solo el boton de actualizar).

La regla de oro del cache: define quien es la fuente de verdad. En offline-first, la fuente de verdad es la base local; la red solo la actualiza. Nunca muestres datos de la red sin guardarlos primero: si la app se cierra a mitad de la respuesta, esos datos se pierden y la proxima apertura muestra algo distinto. Guardar-primero es lo que hace que tu app se sienta solida y predecible.

Play App Signing: tu llave de subida y la llave de Google

Vamos a aclarar el esquema moderno de firmas porque confunde a todos la primera vez. Tu generas un keystore y firmas tu AAB con tu "llave de subida". Subes ese AAB a Play Console. Google verifica tu firma, y luego vuelve a firmar el contenido con LA llave de distribucion, que Google genera y custodia por ti. Los usuarios instalan la version firmada por Google.

Ventajas: si pierdes tu llave de subida, puedes contactar a soporte de Play y pedir un reseteo (con verificacion de identidad); tu app sigue actualizable porque la llave de distribucion nunca se perdio. Además Google optimiza y protege mejor la firma final. Desventaja: ninguna real para la mayoria; es el estandar y Play lo activa por defecto en apps nuevas.

Ojo: esto NO significa que puedas descuidar tu keystore. El reseteo es un proceso manual que tarda días y requiere demostrar que eres el dueño. Guarda tu llave de subida como si fuera la definitiva, porque en la práctica casi lo es.

ASO: que tu app aparezca cuando la buscan

ASO (App Store Optimization) es el SEO de las tiendas de apps. El 70 por ciento de las descargas viene de búsquedas dentro de la tienda, así que tu ficha debe hablar el idioma de lo que la gente busca. El título debe incluir tu palabra clave principal ("MisPelículas" solo no dice nada; "MisPelículas: cartelera y favoritas" sí). La descripción larga debe repetir naturalmente 3 a 5 veces tus palabras clave (cartelera, películas, favoritas, sinopsis, cine) sin sonar a robot: Google penaliza el relleno descargado.

Las métricas que más pesan en el ranking: la tasa de instalación (de cada 100 que ven tu ficha, cuántos instalan: las buenas capturas la suben), la retención (cuántos siguen usando la app a los 7 y 30 días: aquí gana el producto, no el marketing) y las reseñas (cantidad, promedio y que tan recientes son: pide reseñas dentro de la app en el momento feliz, después de una acción exitosa, nunca al abrir). El tamaño de descarga también importa: cada 6 MB extra reducen la conversión; por eso el AAB y R8 no son solo técnica, son marketing.

Monetización: panorama de anuncios y compras integradas

Publicar no es el final del negocio: es el principio. Los tres modelos honestos más comunes: publicidad con AdMob (Google te paga por mostrar anuncios; funciona si tienes volumen de usuarios, miles de sesiones diarias), compras integradas o suscripciones con la Billing Library (quitar anuncios, funciones premium, contenido extra; Google se queda con el 15 por ciento en la mayoría de los casos para desarrolladores pequeños) y modelo freemium (gratis con límites, pago por desbloquear).

Lo que NO debes hacer: pedir dinero por funciones que no entregan valor claro, usar anuncios que tapan botones o aparecen cada 10 segundos (Google penaliza la experiencia y los usuarios te destrozan en reseñas), o vender datos de usuarios. La monetización sostenible nace de una app que la gente ama y por la que pagaría feliz: primero el producto, después el dinero. Este libro te dio el producto; el negocio lo construyes iterando con usuarios reales.

Ejercicio 7: Implementa el patrón EstadoUi en una pantalla real

Enunciado: Toma la `ListaActivity` del Ejemplo 4 y migrala al patrón `EstadoUi` con sus cuatro estados. La pantalla debe mostrar: un ProgressBar mientras carga, la lista con datos, un mensaje de error con botón "Reintentar" si falla sin datos locales, y un mensaje "No hay

películas todavía" si la base está vacía. El botón Reintentar debe llamar a `viewModel.reintentar()`.

Solución (piezas clave):

```
// En el ViewModel agregamos la función de reintento:
fun reintentar() {
    viewModelScope.launch {
        _estado.value = EstadoUi.Cargando
        try {
            repository.refrescar()
        } catch (e: Exception) {
            _estado.value = EstadoUi.Error("Sin conexión. Revisa tu internet e intenta de nuevo.")
        }
    }
}

// En la Activity, el when completo:
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.estado.collect { estado ->
            progressBar.isVisible = estado is EstadoUi.Cargando
            recyclerView.isVisible = estado is EstadoUi.Exito
            layoutError.isVisible = estado is EstadoUi.Error
            layoutVacio.isVisible = estado is EstadoUi.Vacio
            when (estado) {
                is EstadoUi.Exito -> adaptador.enviarLista(estado.datos)
                is EstadoUi.Error -> textError.text = estado.mensaje
                else -> Unit
            }
        }
    }
}

botonReintentar.setOnClickListener { viewModel.reintentar() }
```

Explicación de la solución: cada vista se muestra u oculta según el estado con `isVisible` (de `androidx.core.view`); nunca hay dos estados visibles a la vez, lo que elimina el bug clásico del spinner girando sobre la lista. El `else -> Unit` cubre los objetos `Cargando` y `Vacío` que no necesitan datos extra. Este es el código que separa una pantalla "que funciona" de una pantalla "profesional".

Ejercicio 8: Pull-to-refresh con SwipeRefreshLayout

Enunciado: Envuelve el RecyclerView de la lista en un `SwipeRefreshLayout` para que al deslizar hacia abajo se refresquen los datos desde la API. El indicador debe girar mientras se refresca y detenerse al terminar (con éxito o con error). No debe disparar un refresco si ya hay uno en curso.

Solución:

```
// Layout: SwipeRefreshLayout como padre del RecyclerView.
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout
    android:id="@+id/swipeRefresh"
    ...>
    <androidx.recyclerview.widget.RecyclerView ... />
</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>

// ViewModel: bandera para no duplicar refrescos.
private val _estaRefrescando = MutableStateFlow(false)
val estaRefrescando: StateFlow<Boolean> = _estaRefrescando

fun refrescoManual() {
    if (_estaRefrescando.value) return // ya hay uno en curso: ignoramos
    viewModelScope.launch {
        _estaRefrescando.value = true
        try {
            repository.refrescar()
        } finally {
            _estaRefrescando.value = false // se apaga siempre, falle o no
        }
    }
}

// Activity: conecta el gesto con el ViewModel.
swipeRefresh.setOnRefreshListener { viewModel.refrescoManual() }
lifecycleScope.launch {
    repeatOnLifecycle(Lifecycle.State.STARTED) {
        viewModel.estaRefrescando.collect { girando ->
            swipeRefresh.isRefreshing = girando
        }
    }
}
```

Explicación de la solución: el `finally` garantiza que el indicador se detenga aunque la red falle; sin el, el spinner giraría para siempre tras un error (bug clásico). La bandera

`_estaRefrescando`` evita peticiones duplicadas si el usuario desliza como loco. Nota que el refresco manual reutiliza `repository.refrescar()``: la misma funcion sirve al inicio automático, al pull-to-refresh y al boton de reintentar. Una sola fuente de verdad para actualizar datos.

Resumen de la unidad

Esta unidad te convirtió de programador de piezas en entregador de productos. En la clase 28 construiste la arquitectura completa de una app real: Room como fuente local reactiva con Flow, Retrofit como fuente de red con DTOs y mapeo, el Repository como cerebro offline-first que decide de dónde vienen los datos y captura los errores de red, el ViewModel como puente con la UI y la Activity observando con `repeatOnLifecycle``. Aprendiste que el DTO nunca llega a la UI, que el Repository es el único que conoce las fuentes de datos y que la inyeccion de dependencias (aunque sea manual) es lo que hace tu codigo testeable y profesional.

En la clase 29 dominaste la firma y el build de release: qué es un keystore y por qué perderlo es irreversible, como generarlo con keytool, como guardar sus secretos en `keystore.properties`` fuera de Git, la diferencia entre APK y AAB (y por que Play exige AAB), como configurar `signingConfigs`` y el build type release con R8 y reduccion de recursos, y el duo `versionCode`/`versionName`` con versionado semantico. Terminaste generando tu propio AAB firmado, el artefacto que separa al aficionado del profesional.

En la clase 30 recorriste Google Play Console: el registro de desarrollador, la ficha (titulo, descripciones orientadas a beneficios, icono, capturas que cuentan una historia, video), la clasificacion de contenido honesta, la politica de privacidad obligatoria, el lanzamiento por pistas (interna, cerrada, produccion gradual) y el manejo profesional de los rechazos como tramite, no como tragedia. También aprendiste que el lanzamiento no es el final: monitorear crashes, responder resenas y actualizar con regularidad es lo que mantiene viva una app.

El mensaje de cierre del curso es este: ya sabes construir apps completas, firmarlas y publicarlas. Lo que sigue es práctica deliberada: publica tu primera app, aunque sea pequeña; mide, escucha a tus usuarios, itera. Cada app que publiques te enseña más que diez tutoriales. Bienvenido al oficio de desarrollador Android.

Retos extra (tu turno, sin solución escrita)

Reto 1: agrega a MisPeliculas una pantalla de "Favoritas" que muestre solo las peliculas marcadas, usando una consulta Room con `WHERE esFavorita = 1``. Debe actualizarse sola al marcar/desmarcar desde cualquier pantalla.

Reto 2: implementa paginacion simple: un boton "Cargar más" al final de la lista que pida la siguiente página a la API (`?page=2``) y la agregue a Room sin duplicar. Pista: ``OnConflictStrategy.REPLACE`` ya evita duplicados.

Reto 3: genera el AAB firmado de tu app del curso y subelo a la pista de Prueba interna de Play Console (puedes crear la app en la consola sin pagar hasta el momento del lanzamiento). Invita a un amigo como tester y pidele que te reporte un bug.

Reto 4: escribe la descripcion larga de tu propia app (la que vas a publicar) siguiendo la regla "beneficios, no características", y pidele a dos personas que no son técnicas que te digan si les provoca instalarla. Itera hasta que digan que si.

EXAMEN FINAL

Instrucciones: este examen cubre las 10 unidades del curso. Son 50 preguntas: 30 de opción múltiple, 10 de verdadero/falso y 10 abiertas donde debes escribir código. Responde primero todo sin mirar las soluciones; después revisa la sección "Respuestas y soluciones" y califícate con honestidad. Si sacas menos de 35 correctas, repasa las unidades marcadas antes de publicar tu primera app. Consejo: en las preguntas de código, escribe tu respuesta en Android Studio o en papel antes de ver la solución; el músculo se entrena escribiendo, no leyendo.

Guía de repaso expres: lo esencial de las 10 unidades

Antes de responder, repasa estos puntos clave. Si alguno te suena borroso, vuelve a su unidad antes de seguir.

Unidad 1 (Fundamentos y Kotlin): Android es el sistema operativo móvil más usado del mundo; las apps se escriben en Kotlin y se construyen con Android Studio y Gradle. Repasa: variables ``val`/`var``, funciones, clases, data classes, null safety (`?``, `?:``, `!!``), y que el ``AndroidManifest.xml`` declara tu app ante el sistema. Truco de memoria: ``val`` es como escribir con pluma (no se borra), ``var`` es como lápiz (se puede cambiar). Y el null safety es el superpoder de Kotlin frente a Java: el compilador te obliga a manejar los nulos antes de que se conviertan en crashes en manos del usuario.

Unidad 2 (Layouts): la interfaz se define en XML con vistas organizadas en layouts. Repasa: `LinearLayout` (apila), `ConstraintLayout` (restricciones, diseños planos sin anidar), `FrameLayout` (encima), `dp` vs `px` vs `sp`, ids con ``@+id/``, y que los textos van en ``strings.xml``, nunca quemados. Regla de diseño: si tu layout tiene más de 3 niveles de anidamiento, es hora de pasarlo a `ConstraintLayout`. Y recuerda la triada de unidades: `dp` para tamaños, `sp` para textos (respetar la accesibilidad del usuario), `px` casi nunca en código manual.

Unidad 3 (Activities y ciclo de vida): cada pantalla es una Activity con ciclo de vida: `onCreate` (una vez), `onStart`, `onResume`, `onPause`, `onStop`, `onDestroy`, `onRestart`. Repasa: Intents explícitos (navegar entre tus pantallas con extras) e implícitos (delegar acciones al sistema), y que la rotación destruye y recrea la Activity. Por eso los datos que deben sobrevivir a la rotación viven en el `ViewModel`, no en la Activity. Y los extras del Intent son el "paquete" que viaja entre pantallas: siempre valida que la clave exista al leerlos para evitar nulos sorpresa.

Unidad 4 (Listas con RecyclerView): para listas eficientes se usa `RecyclerView` con `Adapter` y `ViewHolder` (que cachea las referencias a las vistas). Repasa: ``onCreateViewHolder``, ``onBindViewHolder``, ``getItemCount``, y la versión moderna con ``ListAdapter`` + ``DiffUtil`` +

`submitList` para animar solo los cambios. Recuerda el trio de metodos que siempre implementas: `onCreateViewHolder` (infla una vez), `onBindViewHolder` (pinta cada item visible), `getItemCount` (dice cuantos hay). Si tu lista parpadea al actualizar, te faltó `DiffUtil`.

Unidad 5 (Fragments y navegacion): un Fragment es una porcion de UI reutilizable con ciclo de vida propio que vive en una Activity. Repasa: el Navigation Component (grafo, destinos, acciones, NavController) y cuando conviene un Fragment frente a una Activity nueva. Regla práctica: si dos pantallas comparten la misma barra superior o el mismo menú, probablemente son Fragments de una sola Activity; si son flujos totalmente distintos (login vs app principal), son Activities separadas. El grafo de navegacion dibuja ese mapa antes de escribir codigo.

Unidad 6 (Corrutinas): concurrencia ligera en Kotlin. Repasa: `suspend`, `delay` vs `Thread.sleep`, `Dispatchers.Main` (UI) vs `Dispatchers.IO` (red/disco), `viewModelScope` y `lifecycleScope` (se cancelan solos), y `repeatOnLifecycle` para observar Flows solo con la pantalla visible. El error mortal es lanzar corrutinas en `GlobalScope`: sobreviven a tu pantalla y actualizan vistas muertas. Memoriza el duo seguro: `viewModelScope` en ViewModels, `lifecycleScope` en Activities/Fragments. Todo lo demas es caso especial.

Unidad 7 (Red con Retrofit): cliente HTTP declarativo. Repasa: la interfaz con anotaciones (`@GET`, `@POST`, `@Query`, `@Body`, `@Path`), el convertidor Gson, las funciones `suspend`, y el manejo de errores (IOException sin red, HttpException con error del servidor). Patron inolvidable: la interfaz declara, Retrofit implementa, el convertidor traduce, la corrutina ejecuta. Si tu app se cierra sin internet, te faltó el try/catch alrededor de la llamada: la red siempre puede fallar y tu codigo siempre debe tener un plan B.

Unidad 8 (Base local con Room): persistencia con SQLite sin SQL manual. Repasa: `@Entity` (tabla), `@Dao` (consultas), `@Database`, consultas que devuelven `Flow` (reactivas), `@Insert` con estrategia de conflicto, y migraciones al cambiar el esquema. La frase que te salva en entrevistas: "Room es SQLite con superpoderes de compilacion: los errores de SQL se detectan al compilar, no en el teléfono del usuario". Y jamas uses `allowMainThreadQueries` fuera de un prototipo: congela la UI con bases reales.

Unidad 9 (Permisos y dispositivo): los permisos peligrosos se declaran en el manifest y se piden en tiempo de ejecucion con `registerForActivityResult` + `RequestPermission()`. Repasa: los tres escenarios (concedido, negado, bloqueo permanente a ajustes), cámara con `TakePicture` + `FileProvider`, galeria con `GetContent`, escalado de Bitmaps con `inSampleSize`, canales de notificacion obligatorios desde API 26, permiso `POST_NOTIFICATIONS` desde API 33, y ubicacion con `FusedLocationProviderClient`. El mantra de permisos: "declarar, pedir en contexto, verificar siempre, degradar con elegancia". Si memorizas ese orden, el 90 por ciento de los bugs de permisos desaparecen. Y recuerda: el

usuario puede revocar un permiso en cualquier momento, así que `checkSelfPermission` se llama antes de CADA uso, no una vez al instalar.

Unidad 10 (Proyecto final y publicación): arquitectura MVVM offline-first con Repository (Room + Retrofit + Flow), estados de UI con sealed classes, firma con keystore (perderlo = no más actualizaciones), AAB como formato de publicación, `versionCode` creciente, R8 en release, y Play Console (ficha, capturas, clasificacion, politica de privacidad, pistas de lanzamiento gradual). El checklist del día del lanzamiento cabe en una frase: "firmado con mi llave, versionCode mayor, probado en release en un teléfono real, ficha completa y honesta, lanzamiento gradual". Si cumples esos cinco, el resto son detalles.

Los 12 errores que más puntos cuestan (leelos antes de responder)

1. Confundir dp con px: si la pregunta habla de "mismo tamaño físico en cualquier pantalla", es dp. px varia con la densidad.
2. Creer que onCreate se repite: solo se llama una vez al crear la Activity; al volver de segundo plano van onStart/onResume.
3. Hacer red en el hilo principal: siempre Dispatchers.IO para red/disco y Main solo para UI. NetworkOnMainThreadException es la prueba de que lo olvidaste.
4. Pensar que basta declarar el permiso en el manifest: los peligrosos exigen dialogo en tiempo de ejecucion; sin el, el sistema los niega en silencio.
5. Esperar foto en alta resolucion de TakePicturePreview: solo da miniatura; la alta resolucion es TakePicture + FileProvider.
6. Notificar sin canal en API 26+: la notificacion no aparece y no hay error. El canal se crea una sola vez al iniciar la app.
7. Subir la llave de debug a Play: la consola la rechaza; necesitas tu firma release con tu keystore.
8. Olvidar subir el versionCode: cada AAB nuevo necesita un entero mayor; el versionName es solo texto para el usuario.
9. Cargar el Bitmap completo sin inSampleSize: OutOfMemoryError garantizado con fotos reales. Escala siempre.
10. Dejar requestLocationUpdates sin cancelar en onPause: drena la bateria y te llueven las resenas de 1 estrella.
11. Usar el DTO de la API directamente en la UI: cuando la API cambie, todo se rompe; el mapeo a entidad/modelo aísla el cambio.

12. Llamar a Geocoder en el hilo principal: bloquea y congela la app; siempre en Dispatchers.IO con texto alternativo sin internet.

Preguntas

Opcion multiple

PREGUNTA 1: Qué es una Activity en Android?

- a) Un archivo de configuracion del proyecto b) Una pantalla con la que el usuario interactua c) Una libreria de disenio gráfico d) Un permiso del sistema

PREGUNTA 2: Para que sirve el archivo AndroidManifest.xml?

- a) Para escribir el codigo Kotlin de la app b) Para declarar componentes, permisos y características de la app ante el sistema c) Para guardar las imágenes del proyecto d) Para compilar el codigo a bytecode

PREGUNTA 3: Que layout permite posicionar vistas usando restricciones entre ellas y los bordes del contenedor?

- a) LinearLayout b) FrameLayout c) ConstraintLayout d) TableLayout

PREGUNTA 4:Cuál es la diferencia práctica entre dp y px?

- a) Son lo mismo con distinto nombre b) dp es una unidad independiente de la densidad: se ve del mismo tamaño físico en cualquier pantalla; px es un pixel real y varia según la densidad c) px es independiente de la densidad y dp no d) dp solo sirve para textos y px solo para imágenes

PREGUNTA 5: Que metodo del ciclo de vida de la Activity se llama UNA sola vez cuándo la pantalla se crea?

- a) onStart b) onResume c) onCreate d) onRestart

PREGUNTA 6:Cuál es la diferencia entre un Intent explicito y uno implicito?

- a) No hay diferencia, son sinonimos b) El explicito nombra directamente la clase destino; el implicito describe una accion y el sistema elige que app la maneje c) El implicito es más rápido que el explicito d) El explicito solo sirve para abrir paginas web

PREGUNTA 7: En un RecyclerView, que responsabilidad tiene el ViewHolder?

a) Descargar los datos de internet b) Guardar las referencias a las vistas de un item para no buscarlas con findViewById en cada scroll c) Ordenar la lista de datos d) Pedir los permisos de la app

PREGUNTA 8: Para que sirve DiffUtil en un RecyclerView?

a) Para encriptar los datos de la lista b) Para calcular que items cambiaron entre dos listas y animar solo esos cambios en vez de redibujar todo c) Para descargar imágenes más rápido d) Para duplicar la lista en memoria

PREGUNTA 9: Qué es un Fragment?

a) Un archivo de audio b) Una porcion reutilizable de interfaz con su propio ciclo de vida que vive dentro de una Activity c) Un tipo de permiso peligroso d) Una base de datos local

PREGUNTA 10: En el componente de navegacion (Navigation Component), que representa un "destino" (destination)?

a) Una llave del keystore b) Una pantalla (Fragment o Activity) a la que se puede navegar dentro del grafo de navegacion c) Un servidor de internet d) Una tabla de Room

PREGUNTA 11: Qué es una corrutina en Kotlin?

a) Un hilo del sistema operativo b) Una unidad de concurrencia ligera que puede suspenderse y reanudarse sin bloquear el hilo donde corre c) Un tipo de base de datos d) Un layout de Android

PREGUNTA 12: Cuándo debes usar Dispatchers.IO en una corrutina?

a) Para actualizar vistas en pantalla b) Para operaciones de red, disco o base de datos que bloquean c) Nunca, esta deprecado d) Solo para animaciones

PREGUNTA 13: Que ventaja te da viewModelScope frente a GlobalScope?

a) Ninguna, son identicos b) viewModelScope se cancela automaticamente cuando el ViewModel se destruye, evitando fugas de memoria y trabajo inutil c) viewModelScope es más rápido d) GlobalScope no existe en Kotlin

PREGUNTA 14: Que hace la anotacion @GET en Retrofit?

a) Crea una tabla en Room b) Declara que esa funcion hara una petición HTTP GET a la ruta indicada c) Pide un permiso al usuario d) Genera el icono de la app

PREGUNTA 15: Para que sirve GsonConverterFactory en Retrofit?

a) Para firmar el APK b) Para convertir automaticamente el JSON de la respuesta en objetos Kotlin (y viceversa) c) Para comprimir las imágenes d) Para crear canales de notificacion

PREGUNTA 16: En Room, que indica la anotacion @Entity?

a) Que la clase es una tabla de la base de datos b) Que la clase es una pantalla c) Que el metodo hace una petición de red d) Que el permiso es peligroso

PREGUNTA 17: Si un metodo del DAO devuelve Flow<List<Película>>, que ocurre cuándo cambia la tabla?

a) Nada, hay que consultar de nuevo manualmente b) El Flow emite automaticamente la nueva lista y la UI que lo observa se actualiza sola c) La app se cierra d) Se borra la base de datos

PREGUNTA 18:Cuál es un permiso PELIGROSO (requiere solicitud en tiempo de ejecucion)?

a) INTERNET b) VIBRATE c) CAMERA d) ACCESS_NETWORK_STATE

PREGUNTA 19: Que contrato de ActivityResultContracts usas para pedir UN permiso al usuario?

a) TakePicture() b) GetContent() c) RequestPermission() d) StartActivityForResult()

PREGUNTA 20: Que te indica shouldShowRequestPermissionRationale() cuándo devuelve true?

a) Que el permiso ya fue concedido b) Que el usuario ya nego el permiso una vez (sin bloqueo permanente) y conviene explicarle por que lo necesitas antes de volver a pedirlo c) Que debes cerrar la app d) Que el dispositivo no tiene cámara

PREGUNTA 21: Para que sirve FileProvider al tomar una foto en alta resolucion?

a) Para pedir el permiso de internet b) Para generar un Uri seguro (content://) que le permite a la app de cámara guardar la foto en un archivo privado de tu app c) Para comprimir el APK d) Para crear el canal de notificacion

PREGUNTA 22: Que hace inSampleSize al cargar un Bitmap?

a) Aumenta la resolucion de la imagen b) Es un factor de reduccion (potencia de 2) para cargar la imagen ya escalada y ahorrar memoria c) Cambia el formato del archivo d) Pide el permiso de almacenamiento

PREGUNTA 23: Desde que version de Android es OBLIGATORIO asignar cada notificacion a un canal?

a) Android 5 (API 21) b) Android 8 (API 26) c) Android 10 (API 29) d) Nunca es obligatorio

PREGUNTA 24: Desde que version de Android mostrar notificaciones requiere el permiso POST_NOTIFICATIONS en tiempo de ejecucion?

a) Android 8 (API 26) b) Android 10 (API 29) c) Android 13 (API 33) d) Android 6 (API 23)

PREGUNTA 25: Qué es FusedLocationProviderClient?

a) Un layout para mapas b) El cliente de Play Services que combina GPS, WiFi y red celular para entregar la mejor ubicacion ahorrando bateria c) Un permiso del manifest d) Un tipo de notificacion

PREGUNTA 26: En la arquitectura del curso, cuál es la responsabilidad del Repository?

a) Dibujar las vistas en pantalla b) Centralizar el acceso a los datos y decidir si vienen de la API, de la base local o de ambas c) Firmar el APK d) Pedir permisos al usuario

PREGUNTA 27: Que describe la estrategia "offline-first"?

a) La app solo funciona con internet b) La app muestra primero los datos locales guardados y actualiza desde internet en segundo plano c) La app nunca usa base de datos d) La app se publica sin pasar por Play Console

PREGUNTA 28: Que ocurre si pierdes el keystore con el que firmaste tu app publicada?

a) Nada, Google te genera uno nuevo automaticamente sin tramite b) No puedes volver a publicar actualizaciones de esa app; tendrias que publicar una app nueva con otro nombre de paquete c) La app se desinstala de los teléfonos d) Solo pierdes las estadísticas

PREGUNTA 29:Cuál es la diferencia entre APK y AAB?

a) Son lo mismo b) El APK es instalable directamente; el AAB es el formato de publicación con el que Google genera APKs optimizados por dispositivo. Play exige AAB para apps nuevas c) El AAB es para iOS y el APK para Android d) El APK es más moderno que el AAB

PREGUNTA 30: Que diferencia hay entre versionCode y versionName?

a) Son sinonimos b) versionCode es un entero que debe crecer en cada subida (lo usa Google para ordenar versiones); versionName es el texto que ven los usuarios ("1.2.0") c) versionName es el entero y versionCode el texto d) Ninguno de los dos es necesario para publicar

Verdadero o falso

PREGUNTA 31: Anidar muchos layouts unos dentro de otros no afecta el rendimiento de la app. (V/F)

PREGUNTA 32: onCreate() se ejecuta cada vez que la Activity regresa de segundo plano. (V/F)

PREGUNTA 33: Puedes hacer peticiones de red directamente en Dispatchers.Main sin problema. (V/F)

PREGUNTA 34: Retrofit puede usarse sin un convertidor como Gson, recibiendo el cuerpo crudo de la respuesta. (V/F)

PREGUNTA 35: Room permite hacer consultas en el hilo principal por defecto. (V/F)

PREGUNTA 36: Basta con declarar un permiso peligroso en el AndroidManifest.xml; no es necesario pedirlo en tiempo de ejecucion. (V/F)

PREGUNTA 37: ActivityResultContracts.TakePicturePreview() devuelve la foto en maxima resolucion. (V/F)

PREGUNTA 38: En Android 8 o superior, una notificacion sin canal asignado se muestra normalmente. (V/F)

PREGUNTA 39: Puedes publicar en Play Store un APK firmado con la llave de debug de Android Studio. (V/F)

PREGUNTA 40: Cada actualizacion que subas a Play Console debe llevar un versionCode mayor que la anterior. (V/F)

Abiertas con codigo

PREGUNTA 41: Escribe un layout XML con un LinearLayout vertical que contenga un TextView con el texto "Hola, Android" y un Button con el texto "Empezar", ambos centrados horizontalmente. El Button debe tener el id botonEmpezar.

PREGUNTA 42: Escribe el codigo Kotlin para abrir una segunda Activity llamada DetalleActivity desde la actual, pasandole un extra de tipo String con clave "nombre" y valor "Ana".

PREGUNTA 43: Escribe un ViewHolder mínimo de RecyclerView (clase que herede de RecyclerView.ViewHolder) que reciba la vista del item y exponga una funcion enlazar(titulo: String) que ponga ese titulo en un TextView con id textTitulo.

PREGUNTA 44: Escribe una funcion suspend llamada cargarNombres que espere 1 segundo (simulando red) y devuelva una lista de tres nombres: "Ana", "Luis", "Maria".

PREGUNTA 45: Escribe la interfaz Retrofit para un endpoint GET "usuarios" que reciba un parametro de consulta page (entero) y devuelva una lista de objetos UsuarioDto. La funcion debe ser suspend.

PREGUNTA 46: Escribe una entidad Room llamada Tarea con id autogenerado (Int), titulo (String) y completada (Boolean, por defecto false); y su DAO con un metodo que devuelva todas las tareas como Flow ordenadas por id.

PREGUNTA 47: Escribe el registro de un ActivityResultLauncher que pida el permiso de cámara: si se concede llama a tomarFoto(), si se niega muestra un Toast que diga "Sin cámara no puedo continuar".

PREGUNTA 48: Escribe una funcion crearCanal(context: Context) que cree un canal de notificacion con id "canal_tareas", nombre "Tareas" e importancia por defecto, solo en Android 8 o superior.

PREGUNTA 49: Escribe el bloque signingConfigs de tipo release dentro del bloque android del build.gradle, leyendo storeFile, storePassword, keyAlias y keyPassword desde un objeto Properties llamado props.

PREGUNTA 50: Escribe un layout XML para una pantalla de perfil: un ImageView con id imagePerfil (200dp x 200dp) y debajo un Button con id botonFoto y texto "Cambiar foto", todo dentro de un LinearLayout vertical centrado.

Respuestas y soluciones

RESPUESTA 1: b) Una pantalla con la que el usuario interactua. La Activity es el componente base de la interfaz: cada pantalla visible de tu app es (normalmente) una Activity con su ciclo de vida. La opcion a) es incorrecta porque el archivo de configuracion es el AndroidManifest.xml; la c) es incorrecta porque las librerias de diseno (como Material) no son Activities; la d) es incorrecta porque los permisos se declaran en el manifest, no son pantallas.

RESPUESTA 2: b) Para declarar componentes, permisos y características de la app ante el sistema. El manifest es la "carta de presentacion" de tu app ante Android: ahi registras Activities, servicios, receivers, permisos y el icono. La a) es incorrecta porque el codigo Kotlin vive en archivos .kt; la c) porque las imágenes van en res/drawable; la d) porque compilar es trabajo de Gradle, no del manifest.

RESPUESTA 3: c) **ConstraintLayout**. Es el layout moderno que posiciona cada vista con restricciones ("pegado al borde superior del padre", "debajo del boton X"), lo que permite disenos planos sin anidamiento. **LinearLayout** (a) apila en una sola direccion; **FrameLayout** (b) encierra vistas una sobre otra; **TableLayout** (d) organiza en filas y columnas rigidas. Si necesitas un formulario complejo sin anidar layouts, **ConstraintLayout** es la respuesta.

RESPUESTA 4: b) **dp** es una unidad independiente de la densidad: se ve del mismo tamaño físico en cualquier pantalla; **px** es un pixel real y varia según la densidad. Un boton de 100dp mide lo mismo en centímetros en un gama baja y en un gama alta; uno de 100px se veria diminuto en una pantalla densa. La a) es falsa por definicion; la c) invierte los conceptos; la d) es falsa: para textos se recomienda **sp** (que además respeta el tamaño de fuente del usuario), y **dp** sirve para todo lo demas.

RESPUESTA 5: c) **onCreate**. Es donde inflas el layout y configuras la pantalla por primera vez; solo se llama al crear la Activity. **onStart** (a) y **onResume** (b) se llaman también cada vez que la Activity vuelve a ser visible (por ejemplo al regresar de otra app); **onRestart** (d) se llama solo cuando la Activity vuelve desde el estado detenido, antes de **onStart**. Confundir **onCreate** con **onResume** es un error común que causa inicializaciones duplicadas.

RESPUESTA 6: b) El explicito nombra directamente la clase destino; el implicito describe una accion y el sistema elige que app la maneje. Ejemplo explicito: `Intent(this, DetalleActivity::class.java)`. Ejemplo implicito: `Intent(Intent.ACTION_VIEW, uri)` para abrir un enlace, donde el sistema muestra el selector de navegadores. La a) niega una diferencia fundamental; la c) es falsa (la velocidad no depende del tipo); la d) es falsa porque el explicito sirve justo para navegar entre tus propias pantallas.

RESPUESTA 7: b) Guardar las referencias a las vistas de un item para no buscarlas con `findViewById` en cada scroll. Sin **ViewHolder**, cada vez que un item aparece en pantalla el sistema buscara sus vistas desde cero y el scroll se trababa; con **ViewHolder** las vistas se buscan una vez y se reutilizan. La a) es trabajo de **Retrofit**; la c) la hace tu lista o la consulta; la d) es trabajo de los launchers de permisos.

RESPUESTA 8: b) Para calcular que items cambiaron entre dos listas y animar solo esos cambios en vez de redibujar todo. Con `notifyDataSetChanged()` toda la lista parpadea; con **DiffUtil** (via **ListAdapter** y `submitList`) solo el item modificado se anima. La a) es falsa (no encripta nada); la c) es trabajo de **Coil**; la d) es falsa (no duplica, compara).

RESPUESTA 9: b) Una porcion reutilizable de interfaz con su propio ciclo de vida que vive dentro de una Activity. Los **Fragments** permiten pantallas modulares (por ejemplo,

un maestro-detalle en tablet) y son los destinos del Navigation Component. La a) es absurda; la c) confunde con permisos; la d) confunde con Room.

RESPUESTA 10: b) Una pantalla (Fragment o Activity) a la que se puede navegar dentro del grafo de navegacion. El grafo (nav_graph.xml) conecta destinos con acciones, y el NavController ejecuta la navegacion. La a) confunde con firma de apps; la c) con Retrofit; la d) con Room.

RESPUESTA 11: b) Una unidad de concurrencia ligera que puede suspenderse y reanudarse sin bloquear el hilo donde corre. Miles de corrutinas pueden vivir en pocos hilos porque cuando una espera (red, disco), libera el hilo para otras. La a) es falsa: justamente evitan crear hilos del sistema, que son caros; la c) confunde con Room; la d) confunde con layouts.

RESPUESTA 12: b) Para operaciones de red, disco o base de datos que bloquean. Dispatchers.IO esta optimizado para muchas operaciones bloqueantes simultaneas; Dispatchers.Main es para tocar la UI. La a) es al revés: la UI va en Main; la c) es falsa (sigue vigente); la d) es falsa (las animaciones van en el hilo principal).

RESPUESTA 13: b) viewModelScope se cancela automaticamente cuando el ViewModel se destruye, evitando fugas de memoria y trabajo inutil. Si rotas la pantalla con un GlobalScope lanzado, la corrutina vieja sigue corriendo y puede actualizar una pantalla que ya no existe (crash o fuga). La a) niega la diferencia clave; la c) es irrelevante; la d) es falsa (GlobalScope existe, pero casi nunca debes usarlo en apps).

RESPUESTA 14: b) Declara que esa funcion hara una petición HTTP GET a la ruta indicada. Retrofit genera la implementacion por ti a partir de la anotacion. La a) es trabajo de @Entity de Room; la c) es trabajo de los launchers; la d) no tiene sentido.

RESPUESTA 15: b) Para convertir automaticamente el JSON de la respuesta en objetos Kotlin (y viceversa). Sin convertidor recibirias texto crudo y tendrias que parsearlo a mano. La a) es trabajo del keystore; la c) no existe como funcion de Retrofit; la d) es trabajo del NotificationManager.

RESPUESTA 16: a) Que la clase es una tabla de la base de datos. Cada instancia es una fila y cada propiedad una columna. La b) confunde con Activity; la c) con Retrofit; la d) con permisos.

RESPUESTA 17: b) El Flow emite automaticamente la nueva lista y la UI que lo observa se actualiza sola. Es la base de la UI reactiva del curso: guardas en Room y la pantalla se entera sin que pidas nada. La a) describe el comportamiento de una consulta suspendida normal, no de un Flow; la c) y la d) son falsas.

RESPUESTA 18: c) CAMERA. Accede a hardware sensible y datos privados, así que se pide en tiempo de ejecución. INTERNET (a), VIBRATE (b) y ACCESS_NETWORK_STATE (d) son permisos normales: se otorgan solos al instalar. Memoriza la distinción: si protege privacidad o hardware sensible, es peligroso.

RESPUESTA 19: c) RequestPermission(). Muestra el diálogo del sistema y te devuelve true/false en el callback. TakePicture() (a) toma fotos; GetContent() (b) abre el selector de archivos; StartActivityForResult (d) es el mecanismo viejo deprecado que ya no debes usar.

RESPUESTA 20: b) Que el usuario ya negó el permiso una vez (sin bloqueo permanente) y conviene explicarle por qué lo necesitas antes de volver a pedirlo. Es tu señal para mostrar tu propio diálogo educativo antes de lanzar el del sistema. La a) se verifica con checkSelfPermission, no con este método; la c) y la d) no tienen relación.

RESPUESTA 21: b) Para generar un Uri seguro (content://) que le permite a la app de cámara guardar la foto en un archivo privado de tu app. Sin FileProvider, pasarle una ruta directa a otra app está bloqueado por seguridad y la cámara falla. La a) es falsa (internet es permiso normal); la c) es trabajo de R8; la d) es trabajo del NotificationManager.

RESPUESTA 22: b) Es un factor de reducción (potencia de 2) para cargar la imagen ya escalada y ahorrar memoria. Con inSampleSize = 4, una foto de 4000px se carga como de 1000px usando 16 veces menos RAM. La a) invierte el concepto; la c) es falsa (no cambia el formato); la d) es falsa (es técnica de memoria, no de permisos).

RESPUESTA 23: b) Android 8 (API 26). Desde esa versión, notificar sin canal hace que la notificación simplemente no aparezca, sin mensaje de error: el bug fantasma más común de los principiantes. Las demás opciones son versiones incorrectas; la d) es peligrosamente falsa.

RESPUESTA 24: c) Android 13 (API 33). Desde ahí, POST_NOTIFICATIONS se pide en tiempo de ejecución como cualquier permiso peligroso, idealmente en contexto (cuando el usuario activa algo que necesita avisos). En versiones anteriores basta con declarar el canal. La a) es la versión de los canales, no del permiso; la b) y la d) son incorrectas.

RESPUESTA 25: b) El cliente de Play Services que combina GPS, WiFi y red celular para entregar la mejor ubicación ahorrando batería. Es la forma moderna de obtener ubicación; usar el GPS crudo directamente gasta más batería y es más código. La a) confunde con layouts; la c) con el manifest; la d) con notificaciones.

RESPUESTA 26: b) Centralizar el acceso a los datos y decidir si vienen de la API, de la base local o de ambas. La UI nunca habla con Retrofit ni Room directamente: habla con el Repository. La a) es trabajo de la Activity; la c) es trabajo del keystore; la d) es trabajo de los launchers.

RESPUESTA 27: b) La app muestra primero los datos locales guardados y actualiza desde internet en segundo plano. El usuario siempre ve contenido al instante y la app funciona en el metro sin señal. La a) describe lo contrario; la c) niega la base local que es el corazón de la estrategia; la d) no tiene sentido.

RESPUESTA 28: b) No puedes volver a publicar actualizaciones de esa app; tendrías que publicar una app nueva con otro nombre de paquete. Google verifica que cada actualización venga firmada con la misma llave; sin ella no hay forma de demostrar que eres el autor. La a) es falsa (el reseteo de la llave de subida con Play App Signing requiere trámite y no aplica a la llave original sin ese esquema); la c) es falsa (las apps instaladas siguen funcionando); la d) minimiza una tragedia real.

RESPUESTA 29: b) El APK es instalable directamente; el AAB es el formato de publicación con el que Google genera APKs optimizados por dispositivo. Play exige AAB para apps nuevas. La a) niega la diferencia; la c) es falsa (ambos son Android); la d) invierte la modernidad.

RESPUESTA 30: b) `versionCode` es un entero que debe crecer en cada subida (lo usa Google para ordenar versiones); `versionName` es el texto que ven los usuarios ("1.2.0"). Olvidar subir el `versionCode` es el error más común al publicar actualizaciones: Play rechaza el paquete. La a) es falsa; la c) invierte los roles; la d) es falsa (ambos son obligatorios en la práctica).

RESPUESTA 31: FALSO. Cada nivel de anidamiento cuesta medición y dibujado; los layouts muy anidados provocan lentitud visible, sobre todo en listas. Por eso existe `ConstraintLayout`: diseños planos sin anidar. La regla práctica es mantener la jerarquía lo más plana posible. Por que importa: en un teléfono gama baja, un layout anidado 6 niveles dentro de cada ítem de una lista de 100 elementos convierte un scroll fluido en una presentación de diapositivas. El rendimiento de UI es de las primeras cosas que un entrevistador revisa en tu código.

RESPUESTA 32: FALSO. `onCreate()` se llama una sola vez al crear la Activity. Al regresar de segundo plano se llaman `onRestart()`, `onStart()` y `onResume()`, pero no `onCreate()` (salvo que el sistema haya destruido la Activity por falta de memoria). Poner inicializaciones pesadas creyendo que "se refrescan solas" al volver es un error común. Por que importa: si recargas datos en `onResume` sin necesidad, cada vez que el usuario

vuelve de contestar un WhatsApp tu app hace una petición de red nueva: gastas sus datos, su batería y tu servidor. Los datos que deben sobrevivir van en el ViewModel; onResume es para reanudar, no para reconstruir.

RESPUESTA 33: FALSO. La red en el hilo principal congela la interfaz y Android lanza `NetworkOnMainThreadException`. La red va en `Dispatchers.IO` (o en un hilo secundario) y la actualización de la UI en `Dispatchers.Main`. Es una de las reglas inquebrantables del desarrollo Android. Por que importa: el hilo principal tiene 16 milisegundos para dibujar cada frame; una petición de red tarda cientos. Bloquearlo congela la app y Android muestra el diálogo "la app no responde" (ANR). Las corrutinas con `Dispatchers.IO` existen justo para que nunca tengas que pensar en hilos manualmente para esto.

RESPUESTA 34: VERDADERO. Sin convertidor, Retrofit te entrega el cuerpo crudo (`ResponseBody` como texto) y tú lo parseas a mano. `GsonConverterFactory` es una comodidad, no un requisito. Saber esto te salva cuando una API devuelve un formato raro que ningún convertidor maneja bien. Por que importa: no todo JSON es un objeto limpio; a veces la API devuelve un string plano, un número suelto o un formato propietario. En esos casos pides `ResponseBody`, lees el texto crudo y lo parseas a mano. Conocer la capa sin convertidor te hace entender que Retrofit no es magia: es HTTP bien organizado.

RESPUESTA 35: FALSO. Room bloquea el acceso en el hilo principal por defecto (lanza excepción) para obligarte a usar corrutinas o hilos secundarios. Existe un modo que lo permite (`allowMainThreadQueries`) pero es solo para prototipos rápidos, nunca para producción. Por que importa: una consulta que tarda 200ms en tu teléfono de gama alta tarda 2 segundos en un gama baja con la base llena. Room te obliga a usar corrutinas por tu bien: es el framework cuidandote de ti mismo. El día que veas un ANR por base de datos, ya es tarde.

RESPUESTA 36: FALSO. Declararlo en el manifest es necesario pero no suficiente: los permisos peligrosos DEBEN solicitarse en tiempo de ejecución con el diálogo del sistema. Si solo lo declaras, el sistema lo niega en silencio y tu función no trabaja. Es el error número uno de los principiantes con permisos. Por que importa: el manifest es la declaración ("esta app PUEDE usar la cámara") y el diálogo en tiempo de ejecución es la autorización ("el usuario ME DEJA usarla ahora"). Sin la segunda, la primera no sirve de nada. Google Play incluso revisa que pidas los permisos en el contexto donde se usan; pedirlos todos al abrir la app es motivo de mala calificación.

RESPUESTA 37: FALSO. `TakePicturePreview()` devuelve solo una miniatura (`Bitmap` pequeño), útil para avatares. Para alta resolución se usa `TakePicture()` con un Uri de

FileProvider. Esperar calidad fotografica del Preview es un malentendido común. Por que importa: si disenas una pantalla de "documentos escaneados" con **TakePicturePreview**, el texto saldra borroso e ilegible. La eleccion del contrato es una decision de producto: Preview para avatares rapidos, **TakePicture + FileProvider** cuando la calidad importa. Elegir mal aquí arruina la funcion completa.

RESPUESTA 38: FALSO. En Android 8+ la notificacion sin canal no se muestra y no hay error visible. El canal debe crearse antes de notificar, tipicamente al iniciar la app. Es el bug fantasma que hace perder horas a los principiantes. Por que importa: no hay excepcion, no hay log, no hay pista; simplemente "no pasa nada". La primera vez que te ocurre juras que tu codigo está mal y reescribes todo, cuando lo único que faltaba era crear el canal al iniciar la app. Ahora ya lo sabes antes de que te pase: ese es el valor de este libro.

RESPUESTA 39: FALSO. Play Store rechaza los paquetes firmados con la llave de debug; exige tu firma de release generada con tu keystore. La llave de debug es compartida y publica por diseno: cualquiera podria firmar actualizaciones falsas con ella. Por que importa: la firma es la identidad de tu app ante millones de teléfonos. Android confia en que "misma firma = mismo autor" para permitir actualizaciones sin reinstalar. Si esa confianza se rompe, se rompe la seguridad de tus usuarios. Por eso tu keystore vale más que tu codigo.

RESPUESTA 40: VERDADERO. El **versionCode** es un entero monotono creciente que Google usa para saber que una version es más nueva que otra. Si subes un AAB con el mismo **versionCode**, la consola lo rechaza. El **versionName** ("1.2.0") es solo cosmetico para los usuarios. Por que importa: el día del lanzamiento, con los nervios, es facilisimo olvidar subir el **versionCode** y que Play Console rechace tu AAB a las 2 de la mañana. Los equipos profesionales automatizan esto en su CI; tu, por ahora, ponlo en tu checklist pre-lanzamiento de la Unidad 10 y revisalo dos veces.

RESPUESTA 41: Solución:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hola, Android"
        android:layout_gravity="center_horizontal" />

    <Button
        android:id="@+id/botonEmpezar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Empezar"
        android:layout_gravity="center_horizontal" />
</LinearLayout>

```

Explicación: `orientation="vertical"` apila los hijos uno debajo del otro; `gravity="center"` en el padre centra el contenido en ambas direcciones, y `layout\_gravity="center\_horizontal"` en cada hijo garantiza el centrado horizontal aunque el padre cambie. El id se declara con `@+id/` (el `+` lo crea). Errores comunes: olvidar el namespace `xmlns:android` (el layout no compila) o usar `match\_parent` en el ancho de los hijos (ocuparían todo el ancho y el centrado no se notaría). Puntos clave que debes dominar: la diferencia entre `gravity` (centra el CONTENIDO dentro de la vista) y `layout\_gravity` (centra la VISTA dentro de su padre) es una de las confusiones más comunes y aparece en entrevistas; `wrap\_content` ajusta al contenido y `match\_parent` llena al padre. Si cambias `orientation` a horizontal, los hijos se ponen lado a lado: ese único atributo define el flujo completo del layout.

RESPUESTA 42: Solución:

```

val intent = Intent(this, DetalleActivity::class.java).apply {
    putExtra("nombre", "Ana")
}
startActivity(intent)

```

Explicación: es un Intent explícito porque nombra la clase destino (`DetalleActivity::class.java`). `putExtra` guarda el dato con su clave; en la segunda Activity se lee con `intent.getStringExtra("nombre")`. El `apply` es azúcar sintáctica de Kotlin para configurar el

objeto al crearlo. No olvides declarar `DetalleActivity` en el AndroidManifest.xml o la app se cerrara con ActivityNotFoundException. Puntos clave: `this` como primer parametro es el contexto de origen; `::class.java` obtiene la clase Java que Android necesita; los extras viajan como pares clave-valor y la clave debe ser IDENTICA al leer (un typo y recibes null). Para objetos complejos entre Activities se usa Parcelable, pero para Strings e Ints, `putExtra` directo basta.

RESPUESTA 43: Solución:

```
class TituloViewHolder(vistaItem: View) : RecyclerView.ViewHolder(vistaItem) {
    private val textTitulo: TextView = vistaItem.findViewById(R.id.textTitulo)

    fun enlazar(titulo: String) {
        textTitulo.text = titulo
    }
}
```

Explicación: el ViewHolder guarda la referencia al TextView UNA vez en el constructor (esa es su razon de existir: evitar findViewById repetidos durante el scroll) y la funcion `enlazar` solo actualiza el contenido. Hereda de `RecyclerView.ViewHolder` pasando la vista del item al padre. En el adaptador, `onCreateViewHolder` crea el holder y `onBindViewHolder` llama a `enlazar`. Puntos clave: el `findViewById` va en el constructor del holder (una sola vez por item visible, no por cada scroll); `enlazar` solo asigna datos, nunca busca vistas. Si tu lista tiene 1000 elementos pero solo 8 caben en pantalla, solo se crean ~10 holders que se reciclan: por eso se llama RecyclerView. Entender este reciclaje es entender por que las listas son fluidas.

RESPUESTA 44: Solución:

```
suspend fun cargarNombres(): List<String> {
    delay(1000) // simula la espera de red sin bloquear el hilo
    return listOf("Ana", "Luis", "Maria")
}
```

Explicación: `suspend` permite usar `delay()`, que suspende la corrutina sin bloquear el hilo (a diferencia de `Thread.sleep()`, que lo congelaria). Esta funcion debe llamarse desde una corrutina o desde otra funcion suspend. Es el patron base para simular red en pruebas y entender la concurrencia antes de conectar una API real. Puntos clave: `delay` es la version suspendida de dormir: no bloquea el hilo, así que otras corrutinas siguen trabajando durante ese segundo; `Thread.sleep` haria lo contrario y congelaria todo. Una funcion `suspend` solo puede llamarse desde una corrutina o desde otra `suspend`: esa regla del compilador es la que mantiene ordenado el mundo asincrono.

RESPUESTA 45: Solución:

```
interface UsuarioApi {  
    @GET("usuarios")  
    suspend fun obtenerUsuarios(@Query("page") page: Int): List<UsuarioDto>  
}
```

Explicación: `@GET("usuarios")` define el endpoint relativo (se suma al baseUrl de Retrofit); `@Query("page")` agrega `?page=N` automáticamente a la URL; `suspend` permite llamarla desde corrutinas sin callbacks. Retrofit genera la implementación: tu solo declaras la interfaz. Si la API devolviera un objeto envolvente en vez de lista directa, el tipo de retorno sería ese objeto (como `RespuestaPeliculas` en la Unidad 10). Puntos clave: `@Query` construye los parametros `?page=2` automáticamente con codificación URL incluida; `@Path` reemplaza segmentos de la ruta (`usuarios/{id}`); `@Body` envía un objeto como JSON en POST/PUT. La interfaz es solo la declaración: Retrofit genera la implementación con magia de proxies en tiempo de ejecución, por eso nunca escribes el cuerpo de estos métodos.

RESPUESTA 46: Solución:

```
@Entity(tableName = "tareass")  
data class Tarea(  
    @PrimaryKey(autoGenerate = true) val id: Int = 0,  
    val titulo: String,  
    val completada: Boolean = false  
)  
  
@Dao  
interface TareaDao {  
    @Query("SELECT * FROM tareass ORDER BY id ASC")  
    fun obtenerTodas(): Flow<List<Tarea>>  
}
```

Explicación: `@Entity` convierte la clase en tabla; `@PrimaryKey(autoGenerate = true)` hace que Room asigne el id solo (por eso el valor por defecto 0); `completada` con default false permite crear `Tarea(titulo = "Comprar pan")` sin más. El DAO devuelve `Flow` para que la UI reaccione a los cambios automáticamente, como aprendiste en la Unidad 10. Recuerda registrar la entidad en la clase `@Database`. Puntos clave: `autoGenerate = true` delega los ids a Room (empiezan en 1 y crecen solos); el valor por defecto `false` en `completada` te deja crear tareas con solo el titulo. El `Flow` del DAO es la pieza reactiva: cada insert, update o delete re-emite la lista y la UI se actualiza sin que pidas nada. Sin Flow tendrías que recargar manualmente después de cada cambio.

RESPUESTA 47: Solución:

```
private val permisoCamaraLauncher = registerForActivityResult(
    ActivityResultContracts.RequestPermission()
) { concedido ->
    if (concedido) {
        tomarFoto()
    } else {
        Toast.makeText(this, "Sin camara no puedo continuar",
            Toast.LENGTH_LONG).show()
    }
}

// Para lanzarlo (verificando primero si ya se tiene):
// permisoCamaraLauncher.launch(Manifest.permission.CAMERA)
```

Explicación: el contrato `RequestPermission()` muestra el dialogo del sistema y entrega el Boolean en el callback. El launcher se registra como propiedad de la Activity (antes de usarse) y se lanza con el nombre del permiso. No olvides declarar `<uses-permission android:name="android.permission.CAMERA" />` en el manifest: sin eso, el callback siempre devuelve false. Este es el patron moderno que reemplaza al deprecado `onRequestPermissionsResult`. Puntos clave: el launcher se declara como propiedad (el registro debe ocurrir antes de que la Activity se cree del todo); el callback recibe un Boolean simple, sin codigos de request que comparar. En codigo real, antes del `launch` verificarias con `checkSelfPermission` y, si el usuario ya nego una vez, mostrarias tu dialogo educativo con `shouldShowRequestPermissionRationale`: el flujo completo de tres escenarios de la Unidad 9.

RESPUESTA 48: Solución:

```
fun crearCanal(context: Context) {
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val canal = NotificationChannel(
            "canal_tareas",
            "Tareas",
            NotificationManager.IMPORTANCE_DEFAULT
        )
        val manager = context.getSystemService(Context.NOTIFICATION_SERVICE) as
            NotificationManager
        manager.createNotificationChannel(canal)
    }
}
```


Explicación: el canal se crea una sola vez (llamadas repetidas con el mismo id no duplican nada). La verificación de versión es obligatoria porque `NotificationChannel` no existe antes de API 26 y la app se cerraría en dispositivos viejos. El id "canal_tareas" debe coincidir exactamente con el que uses en el `NotificationCompat.Builder` al notificar. Sin este paso, en Android 8+ tus notificaciones no aparecen. Puntos clave: el `if` de versión protege a los dispositivos viejos donde `NotificationChannel` no existe (sin él, crash en API < 26); la importancia se fija al crear el canal y el usuario puede cambiarla después en ajustes, así que elige con criterio (`DEFAULT` para lo normal, `HIGH` solo para lo urgente). Llamar a `createNotificationChannel` dos veces con el mismo id no duplica nada: es seguro llamarlo en cada inicio.

RESPUESTA 49: Solución:

```
signingConfigs {
    create("release") {
        storeFile = file(props["storeFile"] as String)
        storePassword = props["storePassword"] as String
        keyAlias = props["keyAlias"] as String
        keyPassword = props["keyPassword"] as String
    }
}
```

Explicación: este bloque va dentro de `android { ... }` y define cómo firmar el build release. Las contraseñas salen del objeto `Properties` (cargado desde `keystore.properties`, que está en el `.gitignore`), nunca escritas directamente en el gradle. Después, en `buildTypes { release { signingConfig = signingConfigs.getByName("release") ... } }` se asocia esta firma al build. Es la configuración que genera el AAB firmado que subes a Play Console. Puntos clave: `create("release")` define la configuración de firma; `props` se carga desde `keystore.properties`, un archivo que vive FUERA de Git (agregado al `.gitignore`) porque contiene secretos. Si escribes las contraseñas directo en el build.gradle y subes el proyecto a GitHub, cualquiera podría firmar actualizaciones de tu app: es un error de seguridad grave, no un detalle menor.

RESPUESTA 50: Solución:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:gravity="center">

    <ImageView
        android:id="@+id/imagePerfil"
        android:layout_width="200dp"
        android:layout_height="200dp"
        android:contentDescription="Foto de perfil" />

    <Button
        android:id="@+id/botonFoto"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Cambiar foto"
        android:layout_marginTop="16dp" />
</LinearLayout>

```

Explicación: el `LinearLayout` vertical con `gravity="center"` centra todo el conjunto; el `ImageView` tiene tamaño fijo en dp (independiente de la densidad, como exige la pregunta 4); el `Button` va debajo por el orden vertical y se separa con `layout_marginTop`. El `contentDescription` es buena práctica de accesibilidad: los lectores de pantalla lo usan para describir la imagen a usuarios con discapacidad visual. Con este layout y el código de la Unidad 9 (permiso + `TakePicture` + `FileProvider`) tienes la pantalla de perfil completa. Puntos clave: `200dp` es tamaño físico consistente en cualquier densidad (repaso directo de la pregunta 4); `contentDescription` no es decoración: es lo que leen los lectores de pantalla a usuarios con discapacidad visual, y Google lo evalúa en accesibilidad; `layout_marginTop` separa sin anidar layouts extra. Este ejercicio conecta las unidades 2 y 9: el layout dibuja y el código le da vida.

Fin de tu parte del libro "Curso Completo de Desarrollo Android": Unidades 9 y 10 más el Examen Final. A publicar se ha dicho.

Como calificarte: tabla de puntajes y plan de repaso

Cada pregunta vale 2 puntos: 100 puntos en total. En las abiertas de código date 2 puntos si tu solución compila en tu cabeza sin errores graves (un nombre de variable distinto no resta; un concepto equivocado sí), 1 punto si la idea es correcta pero el código tiene fallos, 0 si no supiste por dónde empezar.

De 90 a 100: nivel profesional junior listo para publicar. Tu siguiente paso es terminar tu app y subirla a la pista de prueba interna de Play Console esta misma semana. No esperes a sentirte "totalmente listo": la publicación es parte del aprendizaje.

De 70 a 88: base sólida con huecos puntuales. Identifica las unidades donde fallaste más (usa la guía de repaso) y vuelve a sus secciones de práctica. Haz los ejercicios que fallaste sin mirar la solución. En una semana repite el examen.

De 50 a 68: conceptos mezclados. No avances a publicar todavía: relee las unidades 6 a 10 completas, que son donde se concentran los temas que más pesan en el trabajo real (corrutinas, Retrofit, Room, permisos y publicación). Rehaz los ejemplos paso a paso escribiendo el código tu mismo.

Menos de 50: vuelve al principio sin vergüenza. Relee el libro completo a ritmo de una unidad por día, haciendo TODOS los ejercicios de práctica. La programación se aprende con las manos, no con los ojos: si solo leíste sin escribir código, ese es el ajuste que necesitas.

Regla de oro del repaso: por cada pregunta que fallaste, escribe en un cuaderno el concepto en tus propias palabras y un ejemplo de código mínimo. Si puedes explicárselo a alguien más sin trabarte, el concepto es tuyo.

Plan de 7 días si quedaste abajo de 70: día 1-2, unidades 1 a 3 (bases y pantallas); día 3, unidades 4 y 5 (listas y navegación); día 4, unidad 6 (corrutinas, la más densa); día 5, unidades 7 y 8 (red y base local); día 6, unidad 9 (permisos y dispositivo); día 7, unidad 10 (firma y publicación) y repetir el examen. Una unidad por día, con código escrito a mano, no solo leído.

El examen no mide memoria: mide si puedes construir y publicar una app real. Esa es la meta, y ya tienes el mapa completo en este libro.

Felicidades por llegar hasta aquí. La mayoría de la gente abandona los cursos a la mitad; tu terminaste. Ahora publica tu app.

Despedida: lo que sigue es tuyo

Si llegaste hasta aquí, hiciste lo que pocos hacen: terminaste. Empezaste sin saber que era una Activity y ahora sabes pedir permisos en tiempo de ejecución, firmar un AAB y publicar en Play Store. Ese camino, de cero a publicado, es el que separa a los que "quieren aprender a programar" de los que programan.

Tres consejos para el camino que sigue. Primero: publica algo esta semana, aunque sea pequeño e imperfecto. Una app real con 10 usuarios te enseña más que otro curso completo. Segundo: lee código ajeno. Entra a GitHub, busca proyectos Android con muchas estrellas y estudia como organizan sus carpetas, como nombran sus funciones, como manejan los errores. Tercero: ensena. Explicarle a alguien más cómo funciona un ViewModel te obliga a entenderlo de verdad; cada duda que te pregunten es un hueco tuyo que se tapa.

El ecosistema Android seguirá cambiando: Compose crecerá, nuevas APIs de permisos llegarán, Play Console pedirá más requisitos. No te asustes: con los fundamentos de este libro (ciclos de vida, concurrencia, arquitectura en capas y publicación) puedes aprender cualquier novedad en días, porque todas se construyen sobre lo mismo.

Gracias por confiar en Lucas EGA Academy para este viaje. Nos vemos en Play Store: cuando publiques tu app, será la prueba de que este libro cumplió su promesa. A construir.

Curso Completo de Android

Bienvenido al "Curso Completo de Desarrollo Android" de Lucas EGA Academy. Si abriste este libro, probablemente ya pasó por tu cabeza la misma idea que pasa por la cabeza de miles de personas cada semana: "quiero crear mi propia aplicación para celular". Tal vez tienes una idea de negocio, tal vez quieres conseguir un mejor empleo, tal vez simplemente te da curiosidad saber cómo se construye eso que usas todos los días con el pulgar. Cualquiera que sea tu motivo, este libro está escrito para llevarte desde cero hasta el punto en que puedas construir, probar y publicar aplicaciones reales para Android.

lucasegaacademy.com

Colibrí Editorial EGA