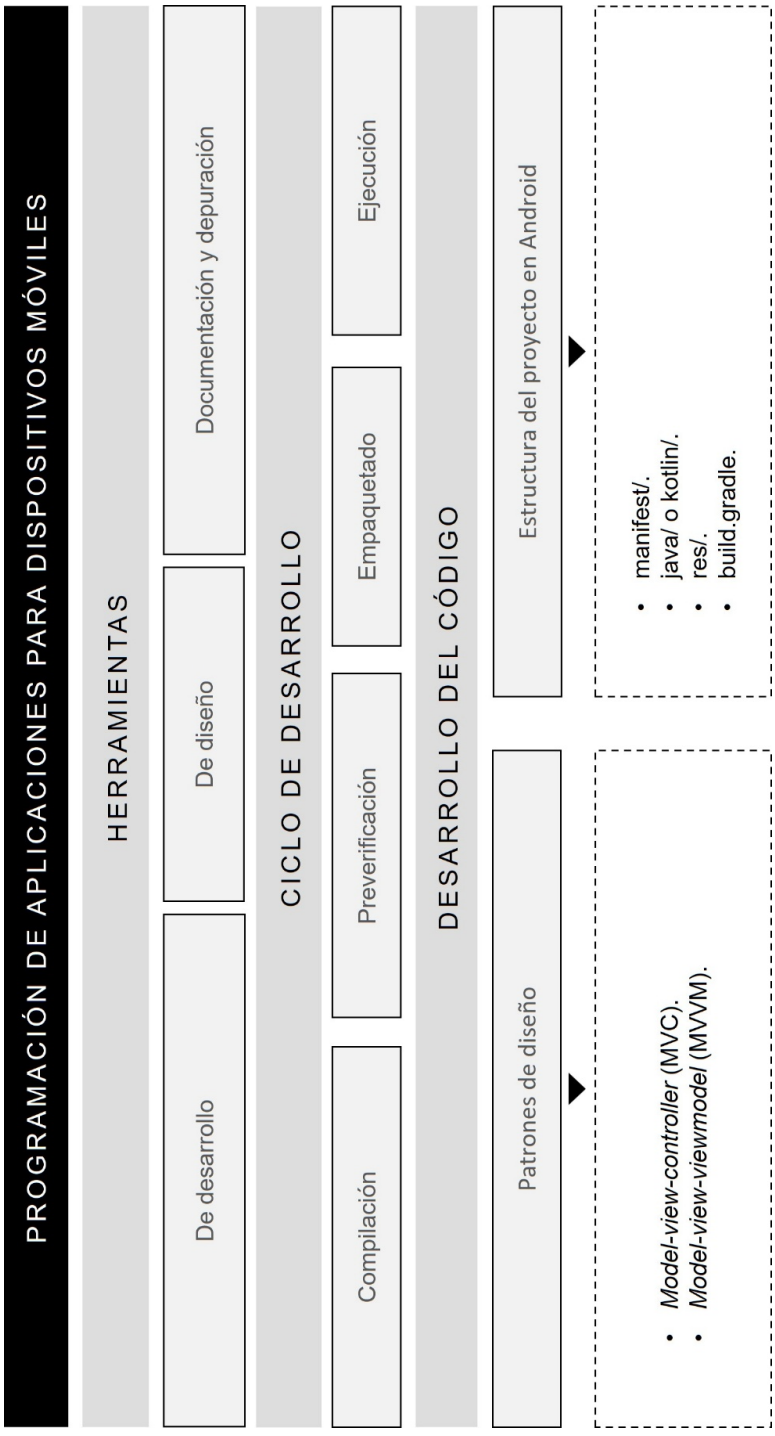
Programación Multimedia y Dispositivos Móviles

Tema 3. Programación de aplicaciones para dispositivos móviles

Índice


PROGRAMACIÓN DE APLICACIONES PARA DISPOSITIVOS MÓVILES
HERRAMIENTAS
De desarrollo
De diseño
Documentación y depuración
CICLO DE DESARROLLO
Compilación
Preverificación
Empaquetado
Ejecución
DESARROLLO DEL CÓDIGO
Patrones de diseño
Model-view-controller (MVC).
Model-view-viewmodel (MVVM).
Estructura del proyecto en Android
manifest/.
java/ o kotlin/.
res/.
build.gradle.

3.1. Introducción y objetivos

En el transcurso de este tema se realizará una **introducción** al **desarrollo** de **aplicaciones móviles** con **Android Studio.** Se analizarán sus **características** y **componentes principales,** la **estructura** y las **fases** de los **proyectos,** así como las **herramientas más relevantes** dentro del programa.

Durante el desarrollo de este tema se deben conseguir los siguientes objetivos:

- Comprender los conceptos básicos relacionados con las aplicaciones en dispositivos móviles.
- Entender la estructura de un proyecto Android.
- Conocer las fases de construcción de un proyecto Android.
- Comprender el desarrollo de las interfaces gráficas en Android.
- Conocer los sistemas de documentación y depuración en un proyecto Android.

3.2. Herramientas y fases de construcción

El desarrollo de aplicaciones móviles para Android es un **proceso complejo** que involucra una **combinación** de **herramientas** de **desarrollo** y un **flujo** de **trabajo** bien estructurado con **varias fases** claramente diferenciadas.

Herramientas de desarrollo para Android

Existen numerosas herramientas que las personas desarrolladoras de aplicaciones Android pueden utilizar a lo largo del ciclo de vida del desarrollo de *software*. Android Studio proporciona algunas esenciales para **escribir, depurar, probar** y **construirlas,** entre las que se incluyen estas:

- Editor de código con autocompletado.
- Emuladores de dispositivos. Permite probar la aplicación en varios dispositivos y versiones de Android sin necesidad de *hardware* físico.
- Herramientas de depuración, como Logcat, que muestra *logs* en tiempo real, y de inspección de memoria y unidad central de procesamiento (CPU).
- Sistema de compilación Gradle. Gestiona la construcción, configuración y las dependencias del proyecto.
- Kit de desarrollo de *software* (SDK) de Android. Incluye bibliotecas, herramientas de línea de comandos, depuradores y ejemplos de código necesarios para desarrollar aplicaciones Android. Algunas de las **utilidades** que provee son estas:
  - Android *debug bridge* (ADB). Permite la interacción con dispositivos o emuladores Android desde la terminal, ejecutar comandos remotos, instalar aplicaciones, etc.
  - Android *virtual device* (AVD). Gestiona los dispositivos virtuales que se utilizan en el emulador de Android.

Material de estudio

- Kotlin. Es el lenguaje de programación oficial para el desarrollo de Android (además de Java). Se prefiere debido a su concisión, seguridad e interoperabilidad con Java. La mayoría de las aplicaciones modernas están desarrolladas o migrando a dicho lenguaje.

- Jetpack Compose. Es un *framework* declarativo moderno diseñado por Google para simplificar y mejorar el desarrollo de interfaces de usuario (UI) en Android. Es parte de la familia Jetpack de herramientas, bibliotecas y componentes que ayudan a quienes desarrollan a crear aplicaciones Android con mayor rapidez, de manera eficiente y con un código más limpio.

- Java. Aunque Kotlin ha ganado popularidad, sigue siendo ampliamente utilizado y es uno de los lenguajes más estables a la par que maduros para Android. Un gran número de bibliotecas, marcos y herramientas están escritas en este lenguaje.

- Firebase. Es una plataforma ofrecida por Google que proporciona una variedad de servicios *backend,* como bases de datos en tiempo real, autenticación, almacenamiento de archivos, análisis y notificaciones *push*. Ayuda a las personas desarrolladoras a enfocarse más en la lógica de la aplicación que en la infraestructura.

- Git. Herramienta de control de versiones esencial para el desarrollo colaborativo. Repositorios tales como GitHub, GitLab o Bitbucket permiten a los equipos compartir y gestionar versiones de su código.

- Retrofit. Biblioteca popular para manejar solicitudes de protocolo de transferencia de hipertexto (HTTP) que facilita la integración de la interfaz de programación de aplicaciones (API) RESTful en ellas. Esta, junto con otras bibliotecas —como Gson o Moshi—, convierte automáticamente las respuestas JSON en objetos Kotlin o Java.

- Glide/Picasso. Se trata de bibliotecas populares para cargar y manejar imágenes dentro de las aplicaciones Android. Son útiles para gestionar la caché, la decodificación y el manejo de imágenes desde un localizador de recursos uniforme (URL) o el almacenamiento local.
- Dagger/Hilt:
  - Dagger. Herramienta para la inyección de dependencias *(dependency injection)* que permite que las aplicaciones sigan el principio de inversión de control (IoC), lo que hace que el código sea más modular y fácil de probar.
  - Hilt. Solución optimizada para Android.
- JUnit y Espresso. Son dos herramientas fundamentales para la creación de pruebas en Android.
- Junit. Es el marco estándar a fin de crear pruebas unitarias.
- Espresso. Es un *framework* para pruebas de UI, utilizado para escribir y ejecutarlas en la aplicación de manera automática.

Herramientas de diseño para Android

En Android Studio las herramientas de diseño juegan un **papel crucial** para **crear UI atractivas** y **funcionales** de manera **eficiente.** Además, permiten a las personas desarrolladoras **visualizar, editar** y **probar** los **diseños** de la interfaz *(layouts),* algo que asegura que la UI se **adapta** a **diferentes dispositivos** y **tamaños** de **pantalla.**

Layout Editor (Editor de diseño)

Es una de las **herramientas más utilizadas** para diseñar la UI en Android Studio. Permite **editar** de manera **visual** los **archivos XML** que definen los *layouts,* lo que facilita la creación de interfaces **sin** necesidad de **escribir todo** el **código** manualmente.

Presenta las siguientes **características:**

- Modo de diseño visual. Permite tanto arrastrar como soltar componentes (botones, textos, imágenes, etc.) desde la paleta al *layout,* organizarlos y ver cómo se verá la UI en tiempo real.
- Modo de diseño XML. Ofrece la posibilidad de alternar entre la vista visual y el código XML del *layout,* algo que permite editar los atributos y las propiedades manualmente.
- Vista dividida. Hace posible ver de forma simultánea el diseño visual y el XML en una vista dividida, lo que facilita la edición del código mientras se ven los cambios reflejados en el diseño.
- Propiedades dinámicas. A medida que se selecciona un componente de la interfaz, en el panel de Propiedades se muestran todos sus atributos (p. ej., ancho, alto, margen, color, etc.), que se pueden modificar directamente.

En la sección de A fondo encontrarás documentación oficial sobre esta herramienta.

Palette (Paleta)

Se trata de una herramienta que **contiene todos** los **componentes** de la **UI** que puedes usar en el diseño del *layout.* Se agrupan por **categorías,** como botones, *widgets* de texto, contenedores, *layouts* y otros componentes avanzados.

Component Tree (Árbol de componentes)

Es una herramienta que **muestra** la **jerarquía** de **vistas** y ***layouts*** dentro del diseño de la interfaz. Permite **ver** cómo están **organizados** los **componentes** y **seleccionar** rápidamente **cualquiera** para **editarlo** o **modificar** su estructura.

Cuenta con esta **características:**

- Jerarquía visual. Muestra la estructura del diseño en forma de árbol, con *layouts* que contienen vistas y otros componentes. Puedes expandir y colapsar nodos para explorar la jerarquía de la interfaz.
- Fácil navegación. Hace posible seleccionar cualquier componente en el árbol, que se resaltará automáticamente en el *layout,* lo que facilita la edición directa de sus propiedades.
- Reorganización de vistas. Puedes arrastrar y soltar componentes en el árbol para reordenarlos en el *layout,* lo cual es especialmente útil en aquellos complejos.

ConstraintLayout y el Editor de restricciones

ConstraintLayout es el **contenedor** de *layout* **más avanzado** en Android. Permite crear **interfaces complejas** y **responsivas** utilizando un **sistema** de **restricciones** que **definen** la **posición** de los elementos en relación con otros componentes o con el contenedor principal.

Preview (Previsualización)

Preview o Ventana de previsualización te permite ver la **manera** en la que se **verá** la UI en varios **dispositivos, tamaños** de **pantalla** y varias **orientaciones** (vertical u horizontal) en **tiempo real** mientras editas el *layout.*

Sus **características** son las siguientes:

- Simulación de dispositivos. Se pueden seleccionar diferentes, como teléfonos, tabletas o pantallas plegables, para ver cómo se ajustará la interfaz en cada uno de ellos.

- Múltiples resoluciones. La previsualización permite probar la UI en varias resoluciones, densidades de pantalla (DPI) y configuraciones de orientación.
- Temas y estilos. También es posible previsualizar cómo se verá la interfaz con diferentes temas de la aplicación (p. ej., claro u oscuro).

Theme Editor (Editor de temas)

Es una herramienta que **facilita** tanto la **creación** como **personalización** de los **temas** y **estilos visuales** de la aplicación. Los temas en Android **controlan** la **apariencia** de los componentes de la interfaz (p. ej., colores, tipografías, fondos, etc.).

Layout Inspector (Inspector de layout)

Se trata de una herramienta que hace posible **inspeccionar** en **detalle** la UI de una aplicación mientras está en **ejecución** tanto en un **dispositivo físico** como en un **emulador.** Es especialmente útil para **depurar problemas** de **diseño.**

Motion Editor (Editor de MotionLayout)

Es una **herramienta visual** para **crear animaciones complejas** y **transiciones** entre **diferentes estados** de la interfaz utilizando MotionLayout . Facilita la creación de animaciones **sin** necesidad de **escribirlas manualmente** en **XML.**

Sus **funciones** se presentan a continuación:

- Diseño visual de animaciones. Permite definir y ajustar animaciones de transición entre diferentes estados de un componente (p. ej., mover un botón de una posición a otra, cambiar su tamaño o animar una imagen).

- *Timeline* de animaciones. Proporciona una línea de tiempo que permite ver y ajustar visualmente tanto su duración como su comportamiento.
- Interfaz basada en estados. Puedes definir varios "estados" de un componente, y MotionLayout se encargará de animar la transición entre estos estados.

Vector Asset Studio

Permite **importar** y **crear gráficos vectoriales** en formato **SVG** o VectorDrawable, que son **escalables sin perder calidad** y **ocupan menos espacio** que los rasterizados.

Font Studio

Faculta la **integración sencilla** de **fuentes personalizadas** en la aplicación. Se pueden importar nuevas desde **archivos externos** o desde la biblioteca de fuentes de **Google Fonts.**

Fases de construcción de aplicaciones móviles en Android

El proceso de desarrollo de una aplicación Android generalmente sigue un ciclo de vida estructurado. Cada una de sus fases es crítica para garantizar que la aplicación final sea de alta calidad, funcional y esté alineada con las expectativas de la persona usuaria.

Las **fases clave** incluyen las siguientes:

- Planificación y requerimientos:
  - Objetivos del proyecto. Se determinan los requisitos de la aplicación, sus funcionalidades principales y los problemas que resolverá.
  - Investigación de mercado. Se analiza la competencia, las necesidades de la persona usuaria y las posibles características diferenciadoras que pueden ofrecerse.
  - Selección de tecnologías más adecuadas para el desarrollo. Esto puede implicar decidir entre Kotlin o Java, Firebase o un *backend* personalizado, así como qué bibliotecas utilizar.
- Diseño de la aplicación:
  - Diseño de la arquitectura. Por ejemplo, seguir patrones tales como *model-view-controller* (MVC), *model-view-viewmodel* (MVVM) o *model-view-presenter* (MVP) es crucial para asegurar la escalabilidad y el mantenimiento del código.
  - Diseño UI/UX. Las personas diseñadoras crean la UI y el diseño de la experiencia de usuario (UX). Se utilizan utilizar herramientas tales como Figma o Sketch para los prototipos de diseño; luego, estos se traducen a componentes Android (p. ej., Views, Fragments y Activities).
  - Creación de *mockups* y *wireframes*. Son bocetos o prototipos de la aplicación que ayudan a visualizar la interfaz y el flujo de la aplicación antes de comenzar el desarrollo.
- Desarrollo. Esta es la fase principal en la que se escribe el código y se integran las funcionalidades. Se divide en varias **subetapas:**
  - Estructuración del proyecto. Se crea la estructura base, los paquetes y los módulos de la aplicación.

- Desarrollo *frontend*. Implica crear la UI en XML o con Jetpack Compose. Se implementan los componentes visuales y se gestionan los eventos del usuario.
- Desarrollo *backend*. Se implementa la lógica de la aplicación y se integra con bases de datos, API y otras funciones (p. ej., autenticación, gestión de datos locales, notificaciones, etc.).
- Pruebas. Mientras se desarrolla se suelen escribir pruebas unitarias con JUnit para asegurar que las clases y los métodos individuales funcionan correctamente. También se pueden crear pruebas automatizadas para UI con Espresso o Robolectric.

▸ Pruebas y *debugging*. Hay **tres tipos:**

- De calidad. Se realizan pruebas de integración y regresión para asegurarse de que las nuevas funcionalidades no rompen las existentes.
- En dispositivos reales. Es importante probar la aplicación en múltiples dispositivos y versiones de Android debido a la fragmentación del ecosistema. Aunque los emuladores son útiles, los dispositivos físicos son esenciales a fin de verificar el rendimiento real.
- De usabilidad. Se realiza una evaluación de la UX final con la aplicación. Esto incluye la facilidad de navegación, el tiempo de respuesta y el diseño de la interfaz.

- Despliegue y publicación:
  - Optimización. Se revisa la aplicación para garantizar un rendimiento óptimo ajustando aspectos tales como la gestión de la memoria, el uso de red o el tiempo de carga.
  - Firmado de la aplicación. Antes de lanzarla es necesario firmarla con una clave privada; es una medida de seguridad requerida por Google Play Store.
  - Publicación en Google Play Store. El proceso de despliegue incluye la creación de una cuenta de desarrollador, subir el paquete de aplicación Android (APK) o Android *app bundle* (AAB), escribir descripciones, agregar capturas de pantalla, especificar la segmentación de la audiencia y seleccionar opciones de precios.
- Mantenimiento y actualizaciones:
  - Corrección de errores. Después del lanzamiento es probable que se descubran errores que no se encontraron durante las pruebas. Se emiten actualizaciones para corregirlos.
  - Actualizaciones de funcionalidad. A medida que las personas usuarias proporcionan retroalimentación o evolucionan los requisitos del negocio, se añaden nuevas características o mejoras a la aplicación.
  - Mantenimiento de compatibilidad. Según se lanzan nuevas versiones de Android, la aplicación debe mantenerse compatible, lo que puede requerir ajustes o actualizaciones de código.

- Monitoreo y mejora continua:

  - Análisis de uso. Herramientas tales como Firebase o Google Analytics se utilizan para monitorizar el comportamiento de las personas usuarias dentro de la aplicación, lo que ayuda a entender cómo interactúan y qué áreas necesitan mejoras.

  - Optimización continua. En función de los datos obtenidos, se optimiza su rendimiento, usabilidad y la retención de personas usuarias mediante nuevas actualizaciones.

En la sección de A fondo localizarás un artículo sobre el *testing* en Android.

3.3. Desarrollo del código

El desarrollo de código para un proyecto Android en Android Studio sigue un **proceso estructurado** que **combina** una serie de **elementos:**

- Escritura de código para la lógica de la aplicación (en Kotlin o Java).
- Definición de la UI (en XML).
- Integración con herramientas y bibliotecas que permiten manejar el ciclo de vida, las interacciones, así como las funcionalidades de la aplicación.

Android Studio soporta la implementación de arquitecturas (p. ej., MVVM) y el uso de bibliotecas de Jetpack, como LiveData y Room , que ayudan a mantener una estructura más organizada y escalable.

El **patrón MVVM** de **arquitectura** se utiliza mucho porque **separa claramente** la **lógica** de la UI (View) de la del **negocio** (Model), mientras que el ViewModel actúa como **intermediario** que gestiona la **lógica** y los **datos** de la **UI.**

- ViewModel . Maneja los datos de la UI y los cambios en el estado.
- LiveData . Es un componente observable que actualiza la interfaz de manera automática cuando cambian los datos.
- Repository . Es una capa que maneja el acceso a datos, ya sea de la base de datos o de la red.

Estructura de un proyecto Android

Al crear un proyecto en Android Studio, este **genera** un **conjunto** de **directorios** y **archivos** que forman la **base** del **proyecto;** su estructura básica se divide en **módulos** y el más común es el módulo app/ .

El directorio app/ es el módulo principal de la aplicación, que contiene todo lo relacionado con la lógica del código, los recursos y la configuración específica del módulo de la aplicación Android. Este es el que se compila y ejecuta en un dispositivo o emulador.

Su **estructura interna** es esta:

- manifest/ .
- java/ o kotlin/ .
- res/ .
- build.gradle (nivel de módulo).

manifest/ (AndroidManifest de la aplicación)

Dentro de este directorio está el archivo **AndroidManifest.xml,** que es **crucial** para **definir aspectos importantes** de la aplicación. Aquí se **declaran** estos **elementos:**

- Componentes (p. ej., Activities , Services , BroadcastReceivers).
- Permisos requeridos (como acceso a internet, cámara, etc.).
- Otras configuraciones, como el tema de la aplicación o la configuración de las *intents*.

A continuación, se muestra un ejemplo de un archivo AndroidManifest.xml donde se declara la actividad principal de la aplicación (MainActivity), así como su icono, nombre y otros aspectos importantes.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.myapp">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.MyApp">
        <activity android:name=".MainActivity">
            <intent-filter>
              <action android:name="android.intent.action.MAIN" />
              <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
     </application>
</manifest>
```

java/ o kotlin/ (código fuente)

Este directorio contiene el **código fuente** de la **aplicación,** ya sea en **Java** o en **Kotlin,** dependiendo del lenguaje que estés utilizando. En esta carpeta se **organizan** los **paquetes** y las **clases** que definen la **lógica** de la aplicación. Se recomienda organizar el código en **paquetes** para seguir el **patrón** de **arquitectura** usado (p. ej., *view, model, viewmodel* si sigues el patrón MVVM).

res/

Almacena todos los **recursos no** de **código** que la aplicación necesita, donde se incluyen los *layouts* de la interfaz, las imágenes, cadenas de texto, los colores, estilos, etc. Android **gestiona automáticamente** los recursos para diferentes dispositivos y configuraciones (p. ej., tamaños de pantalla, densidades o idiomas). Sus **subdirectorios comunes** son los siguientes:

- layout/ . Contiene los archivos XML que definen los *layouts* (UI) (p. ej., activity_main.xml, donde se definen los componentes visuales que conforman una pantalla).
- drawable/ . Alberga imágenes y gráficos vectoriales (p. ej., archivos .png, .jpg, o VectorDrawable [.xml]).
- values/ . Contiene archivos XML que definen valores estáticos, como cadenas de texto (strings.xml), colores (colors.xml), estilos (styles.xml) y dimensiones (dimens.xml).
- mipmap/ . Alberga los iconos de la aplicación en diferentes densidades de pantalla (mdpi, hdpi, xhdpi, etc.).
- menu/ . Define los menús de la aplicación a través de archivos XML (menu.xml).

build.gradle

Los archivos build.gradle son *scripts* de **configuración** del **sistema** de **construcción** Gradle. En un proyecto Android típico hay **dos tipos,** a saber, uno a nivel del proyecto, y otro, a nivel del módulo app :

- build.gradle (nivel de proyecto). Define las configuraciones globales y dependencias compartidas entre módulos:

```
buildscript {
    repositories {
        google()
        mavenCentral()
    }
    dependencies {
        classpath "com.android.tools.build:gradle:8.0.0"
    }
}
```

- build.gradle (nivel de módulo app). Configura los detalles específicos de la aplicación, como el SDK, las versiones, dependencias y tareas de compilación:

```
android {
    compileSdkVersion 33
    defaultConfig {
        applicationId "com.example.myapp"
```

```
        minSdkVersion 21
        targetSdkVersion 33
        versionCode 1
        versionName "1.0"
    }
    buildTypes {
        release {
            minifyEnabled false
        }
    }
}
dependencies {
    implementation "androidx.core:core-ktx:1.8.0"
    implementation "com.google.android.material:material:1.5.0"
}
```

Aquí se configuran las **dependencias,** como las bibliotecas de Android Jetpack, además de las configuraciones del **SDK** y las **versiones** de la **aplicación.**

Otros archivos y directorios

Se pueden resaltar **tres:**

- settings.gradle . Lista los módulos que forman parte del proyecto; por lo general, tiene una entrada que apunta al módulo app .
- gradle.properties . Configura las propiedades globales de Gradle, como las optimizaciones de compilación o configuraciones específicas del entorno de construcción.
- local.properties . Contiene las configuraciones específicas del entorno local (p. ej., la ubicación del SDK de Android en tu máquina).

Módulos adicionales

En proyectos grandes es común que haya **múltiples módulos** dentro de un proyecto Android. Cada uno puede **representar** una **biblioteca** o una **parte independiente** de la aplicación (p. ej., un módulo de red, de base de datos, etc.). Se distinguen **dos tipos:**

- Módulo de aplicación (app/). Es el principal que contiene el código de la aplicación.
- Módulos de bibliotecas (library/). Permiten reutilizar código en diferentes partes de la aplicación, o incluso compartirlo entre proyectos.

3.4. Compilación, preverificación, empaquetado y ejecución

En el desarrollo de aplicaciones Android en Android Studio el proceso de construcción *(build)* y ejecución de una aplicación incluye **varias fases clave,** a saber, **compilación, preverificación, empaquetado** y **ejecución.** Estas aseguran que su **código fuente** se transforma en un **archivo instalable** (APK o AAB) que puede **ejecutarse** en **dispositivos Android.** Android Studio gestiona este ciclo de vida utilizando **Gradle,** que es la herramienta de automatización de construcción. A continuación, se explica cada una de estas fases.

Compilación

Es la fase en la que el **código fuente** (generalmente, escrito en Java o Kotlin) se **convierte** en un **formato** que pueda **entender** la **máquina virtual** (VM) Android (Android Runtime [ART]). Este proceso incluye varios **subprocesos,** como la compilación del código fuente y de recursos, y la generación de archivos intermedios. Seguidamente, se detallan las **subfases** del proceso de compilación.

Compilación del código fuente (Java/Kotlin)

- Los archivos .java y .kt se compilan en *bytecode,* que puede ejecutarse en la VM de Java (JVM). Para Kotlin se utiliza su compilador, mientras que en Java se usa el compilador de Java.
- Android no ejecuta directamente el *bytecode* de JVM, sino que lo convierte en un formato más optimizado para dispositivos Android, llamado Dalvik *executable* (DEX).

Generación de *bytecode* DEX

- Una vez que el código se ha compilado en *bytecode* de la JVM, se convierte en archivos .dex, que son específicos del entorno Android y están optimizados para ejecutarse en la VM Dalvik o ART. Las herramientas de Android utilizan DX o D8 para convertir el *bytecode* en archivos DEX.

- Los archivos DEX son los que finalmente se ejecutan en el dispositivo Android. Cabe resaltar que, a partir de Android 5.0, la ART optimiza el código DEX para mejorar el rendimiento utilizando un proceso llamado *ahead-of-time* (AOT).

Compilación de recursos

- Android no solo compila el código fuente, sino también los recursos (*layouts,* imágenes, *strings,* etc.). En esta fase los archivos XML, como los *layouts* y las definiciones de actividades, se procesan y se generan los archivos binarios correspondientes.

- Android Asset Packaging Tool (AAPT). Es la herramienta que se utiliza para compilar y empaquetar todos los recursos. Se encarga de convertir los archivos XML en un formato binario y comprimir tanto imágenes como otros recursos.

- Esta fase también incluye la generación del archivo R.java o su equivalente en Kotlin, que actúa como un índice de todos los recursos de la aplicación, lo que permite a las personas desarrolladoras acceder a ellos en el código.

Herramientas utilizadas

Se pueden señalar **tres:**

- Gradle. Es el motor que organiza todas estas tareas de compilación. Define la estructura del proyecto, las dependencias y las configuraciones de compilación.

- Compilador de Kotlin/Java. Convierte el código fuente en *bytecode* para JVM.
- D8/R8. Herramientas para la optimización y conversión de *bytecode* a DEX.

Preverificación

Se trata de una fase en la que se **verifica** que el **código generado** es **válido antes** de **empaquetarlo** en un **APK** o **AAB.** Aunque no es un proceso separado y definido en todos los sistemas, en el contexto de Android se refiere a la **validación** de **varias reglas** para asegurar que tanto el **código** como los **recursos** generados **cumplen** con las **especificaciones** del **sistema** Android.

Las **verificaciones típicas** son tres:

- De *bytecode*. Se asegura de que el generado es válido para el entorno de ejecución de Android (ART o Dalvik). Esto puede incluir la validación del formato DEX, lo que garantiza que no haya errores en su transformación.
- De referencias. En esta fase se confirma que todas las referencias a métodos, clases y recursos son correctas, y que no se accede a bibliotecas o API incompatibles con la versión objetivo de Android.
- De permisos. Se verifica que la aplicación tiene declarados correctamente los permisos necesarios en el archivo AndroidManifest.xml.

Empaquetado

Es la fase en la que todo el **código compilado,** los **archivos DEX,** los **recursos** y las **dependencias** se **combinan** en un **archivo ejecutable** que se puede **instalar** en un dispositivo Android. Hay **dos formatos principales:**

- APK. Es el tradicional para distribuir aplicaciones Android. Contiene todos los archivos necesarios para su instalación y ejecución en un dispositivo.

- AAB. Se trata del más reciente y optimiza el tamaño de la aplicación. Permite que Google Play Store genere APK específicos para cada dispositivo, lo que reduce el tamaño de descarga y la cantidad de recursos innecesarios en el dispositivo final.

Sus **pasos** de detallan a continuación (véase la Tabla 1):

Paso	Descripción
Firma de la aplicación	▸ Android requiere que **todas** las **aplicaciones** estén **firmadas digitalmente** antes de que se puedan instalar en un dispositivo. La firma **garantiza** su **autenticidad** e **integridad.** ▸ Durante el desarrollo **Android Studio** utiliza una **clave** de **depuración** para firmarla de forma automática. Sin embargo, a fin de publicar una en **Google Play Store** se requiere hacerlo con una **clave** de **producción.**
Generación de AndroidManifest	El archivo AndroidManifest.xml se procesa durante esta fase. Define las **configuraciones principales** de la **aplicación,** como los permisos necesarios, las actividades principales y los servicios. **Gradle** se asegura de que contenga toda la **información correcta** y esté **optimizado.**
Compresión y empaquetado de archivos	▸ Todos los **archivos DEX,** los **recursos compilados** (imágenes, XML binarios, etc.), y las **bibliotecas** (JAR o AAR) se empaquetan juntos en el APK o AAB. ▸ AAPT empaqueta tanto los recursos como los archivos compilados, y finalmente crea el **APK.** ▸ En el caso de AAB, este proceso genera un **archivo modular** que contiene todos los recursos **optimizados** para diferentes dispositivos.

Tabla 1. Pasos del empaquetado. Fuente: elaboración propia.

Ejecución

Una vez que se ha generado el APK o AAB, la fase de ejecución se encarga tanto de **instalar** la **aplicación** en un **dispositivo físico** o **emulador** como de **lanzarla** (véase la Tabla 2).

Paso	Descripción
Instalación	▸ En **Android Studio** la instalación de la aplicación en el dispositivo es **automática** cuando se presiona el botón **Run.** Esto utiliza **ADB,** una herramienta que permite comunicarse con dispositivos Android conectados por USB o red. ▸ ADB instala el **APK** en el dispositivo o emulador y, opcionalmente, inicia la aplicación una vez que se completa la instalación.
Inicio de la aplicación	▸ Android Studio envía un **comando** a través de **ADB** para **iniciar** la **actividad principal,** definida en el archivo AndroidManifest.xml. ▸ Durante la ejecución Android Studio se puede conectar al proceso de la aplicación para proporcionar **herramientas** de **depuración** en **tiempo real,** como el depurador, Logcat e inspección de memoria.
Depuración en tiempo real	En la ejecución Android Studio permite realizar una **depuración** en **vivo** del código utilizando herramientas (p. ej., *breakpoints,* Logcat y el Android Profiler) para **monitorear** el **rendimiento** de la aplicación.

Tabla 2. Detalles del proceso de ejecución. Fuente: elaboración propia.

Resumen del ciclo de construcción de una aplicación Android

El ciclo de construcción de una aplicación Android se presenta a continuación (véase la Tabla 3). El uso de **Gradle** como herramienta de compilación **automatiza** todo este **proceso** y **proporciona configuraciones específicas** para **diferentes entornos** (depuración, liberación, variantes de la aplicación, etc.), lo que **facilita** tanto el **manejo** como la **personalización** del **flujo** de **trabajo** de construcción en Android Studio.

CICLO DE CONSTRUCCIÓN DE UNA APLICACIÓN ANDROID

1	**Compilación**	El código fuente se transforma en **bytecode JVM**, y luego, en archivos **DEX específicos** de **Android.** Los recursos también se compilan y se integran en el paquete final.
2	**Preverificación**	Se **valida** del **código generado** y los **recursos,** lo que asegura que cumplen con los requisitos de Android.
3	**Empaquetado**	El **código compilado,** los **recursos** y las **dependencias** se empaquetan en un archivo **APK** o **AAB firmado digitalmente** para garantizar la seguridad.
4	**Ejecución**	El **APK** se **instala** y **ejecuta** en un dispositivo Android o emulador, algo que permite la **depuración** en **tiempo real.**

Tabla 3. Ciclo de construcción de una aplicación Android. Fuente: elaboración propia.

3.5. Interfaces de usuario. Clases asociadas

Las UI en un proyecto Android juegan un papel fundamental, ya que **determinan** la **manera** en la que las **personas usuarias interactúan** con la aplicación. En Android su diseño se basa en una **combinación** de ***layouts,*** **vistas** y **componentes** de IU **definidos** en los **archivos XML** o programáticamente en el **código Kotlin/Java** con **Jetpack Compose.**

Estas interfaces están ligadas a clases asociadas, que permiten gestionar la lógica y el comportamiento de los elementos de la interfaz.

Layouts (diseños) en Android

Los *layouts* son la **estructura básica** de las **interfaces** en Android y definen la **forma** de **organizar visualmente** los **componentes** dentro de la pantalla. Son **contenedores** que puede albergar otros *layouts* y vistas, organizando su disposición y comportamiento.

Los **tipos comunes** son estos:

- LinearLayout . Organiza los elementos en una sola fila (horizontal) o columna (vertical). Es útil para interfaces simples que requieren una alineación ordenada de componentes.
- RelativeLayout . Permite organizar componentes relativos entre sí o al contenedor principal. Es flexible para posicionar elementos con relación a otros, pero puede volverse complejo cuando hay muchos componentes.
- ConstraintLayout . Es el más avanzado y flexible, que permite alinear vistas utilizando restricciones *(constraints)*. Es ideal para crear interfaces complejas con un rendimiento optimizado, ya que minimiza la cantidad de anidamientos entre vistas.

- FrameLayout . Es un *layout* simple que contiene una sola vista, pero las adicionales se superponen unas sobre otras. Resulta útil en situaciones donde las vistas deben aparecer una encima de otra.

- RecyclerView . Si bien no es un *layout* en sí, es una vista muy usada para mostrar listas grandes de datos de forma eficiente mediante el uso de un patrón de reciclado de vistas.

Clases relacionadas con la UI

En un proyecto Android las clases relacionadas con la UI se encargan de **controlar** la **interacción, gestionar** la **lógica** y **actualizar** los **elementos visuales.** Las más importantes son Activity , Fragment , así como los componentes View y ViewModel .

Activity

Representa una **pantalla completa** en una aplicación Android. Es una de las clases principales que **controla** la **UI** y **gestiona** el **ciclo** de **vida** de los **componentes** de la pantalla.

Conviene **tener** en **cuenta** lo siguiente:

- Cada actividad tiene un archivo de *layout* asociado, que define la UI, y la actividad se encarga de manejar las interacciones de la persona usuaria con los componentes de esa pantalla.

- La clase Activity en sí misma extiende de Context , lo que le permite interactuar con el sistema operativo (SO) para cosas tales como iniciar nuevas actividades o acceder a recursos.

A continuación, se muestra un ejemplo básico de una Activity :

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_main)

       val button: Button = findViewById(R.id.button)

        button.setOnClickListener {

            // Acción cuando el botón es presionado

            Toast.makeText(this, "Botón presionado",
Toast.LENGTH_SHORT).show()

        }

    }

}
```

Fragment

Es un **componente reutilizable** de la UI que **representa** una **parte** de la **interfaz dentro** de una **actividad.** Permite **dividir** una en **componentes más pequeños** y **modulares,** lo que facilita el manejo de pantallas complejas o la reutilización de la UI. Presenta estas **características:**

- Tiene su propio ciclo de vida y su propia lógica de UI, lo que hace posible que diferentes pantallas se gestionen dentro de una única actividad.
- Es útil en interfaces dinámicas y pantallas que deben cambiar sin necesidad de recrear toda la actividad.

View y Custom Views

La clase `View` es la **base** de **todos** los **componentes visuales** y cada uno de estos (p. ej., botones, textos o imágenes) **hereda** de `View`. También es posible **crear vistas personalizadas** (`Custom Views`) al extender la clase `View` o subclases existentes, así como **definir comportamientos** o **diseños personalizados.**

Aquí se puede ver un ejemplo básico de `Custom View`:

```
class MyCustomView(context: Context, attrs: AttributeSet?) : View(context, attrs) {

    override fun onDraw(canvas: Canvas) {

        super.onDraw(canvas)

        // Dibujar contenido personalizado

        canvas.drawColor(Color.RED)

    }

}
```

ViewModel

En el patrón MVVM —comúnmente utilizado en aplicaciones Android con Jetpack—, el `ViewModel` actúa como **intermediario** entre la **UI** (Vista) y los **datos** (Modelo). Además, **mantiene** el **estado** de la **UI** a través de **cambios** de **configuración,** como la rotación de la pantalla, y **gestiona** la **lógica** de **negocio.**

Resulta particularmente útil para evitar que las actividades o los *fragments* manejen de forma directa los datos, lo que promueve una separación más limpia de las responsabilidades.

En este ejemplo el ViewModel mantiene un contador que se puede actualizar y observar desde la actividad o el *fragment:*

```
class MyViewModel : ViewModel() {

    val counter = MutableLiveData<Int>()

    fun increment() {

        counter.value = (counter.value ?: 0) + 1

    }

}
```

LiveData

Es una clase que permite **manejar datos** que pueden **observarse.** Los componentes de la UI, como una actividad o un *fragment*, tienen la capacidad de **suscribirse** a los **cambios** en un LiveData, y esta **actualizará** la **UI automáticamente** cuando los datos cambien.

3.6. Depuración y documentación

La depuración es una **parte crucial** del desarrollo, ya que permite **identificar** a la par que **corregir errores** en el **código** y **mejorar** el **rendimiento general** de la **aplicación.** Android Studio proporciona varias **herramientas integradas** que ayudan a las personas desarrolladoras a encontrar y solucionar errores de manera eficiente.

Herramientas integradas

Logcat

Logcat es una **herramienta** de **registro** en **tiempo real** que muestra mensajes generados por el sistema Android y las aplicaciones en ejecución. A través de ella, quienes desarrollan pueden **ver errores, advertencias** y **mensajes personalizados,** que **ayudan** a **identificar problemas.** Se pueden **filtrar** por **nivel** de **severidad** (*info, warning* o *error*), por **nombre** del **paquete** o por **palabras clave** específicas.

Los mensajes de Logcat se generan utilizando la **clase** `Log` en el **código Java** o **Kotlin,** que permite escribirlos en **diferentes niveles,** a saber, Log.d() (debug) , Log.e() (error) , Log.w() (warning) , entre otros.

```
Log.d("MainActivity", "This is a debug message")
```

```
Log.e("MainActivity", "An error occurred", exception)
```

Debugger (depurador)

El depurador de Android Studio permite **pausar** la **ejecución** de la **aplicación, inspeccionar** el **estado** de las **variables** y los **objetos,** y **avanzar paso** a **paso** a través del **código.**

Sus **características clave** incluyen las siguientes:

- *Breakpoints* (puntos de interrupción). Permiten detener la ejecución del programa en una línea específica para inspeccionar el estado de las variables o ver qué camino está tomando el código.
- *Step into, step over* y *step out*. Estas opciones hacen posible avanzar paso a paso dentro de las funciones, saltar sobre ellas o salir de una a fin de inspeccionar el flujo de ejecución.
- Evaluar expresiones. Se pueden evaluar variables o expresiones directamente desde el depurador para ver su valor en tiempo real.
- Depuración condicional. Los *breakpoints* condicionales permiten detener el programa solo si se cumplen ciertas condiciones, como cuando una variable tiene un valor específico.
- Depuración remota. Android Studio faculta la depuración de aplicaciones que se ejecutan en dispositivos físicos conectados o en emuladores, e incluso depurar dispositivos de forma remota conectándose a ellos a través de ADB.

Ejemplo de uso básico del *debugger*

- Coloca un *breakpoint* haciendo clic en el margen izquierdo de una línea de código.
- Inicia la aplicación en modo Depuración clicando en el botón Debug en lugar de Run.
- Cuando la ejecución se detenga en el *breakpoint* puedes inspeccionar el estado de las variables, avanzar en el código y modificar el estado directamente si es necesario.

Android Profiler

Se trata de un **conjunto** de **herramientas avanzadas** de **análisis** de **rendimiento** que permiten a quienes desarrollan tanto **medir** como **optimizar** la **utilización** de **recursos** (p. ej., CPU, memoria, red y energía):

- CPU Profiler. Mide el uso de la CPU y permite visualizar los métodos que están consumiendo más recursos. Posibilita hacer un perfil de la ejecución del código, detectar cuellos de botella y optimizar el rendimiento.
- Memory Profiler. Ayuda a rastrear el uso de la memoria en tiempo real, detectando fugas de memoria *(memory leaks)* y gestionando el uso de objetos. Permite realizar un análisis de *heap* para ver qué objetos están en uso y cuáles ha recolectado el Garbage Collector.
- Network Profiler. Mide el uso de la red, monitorea solicitudes y respuestas HTTP, y verifica el tamaño y la latencia de los paquetes de datos. Esto es esencial para identificar problemas relacionados con la eficiencia de las conexiones de red.
- Energy Profiler. Permite analizar la forma en la que la aplicación consume energía, lo cual es vital a fin de mejorar la duración de la batería en los dispositivos.

Inspección de layout

Aquí se destacan **dos:**

- Layout Inspector. Es una herramienta que permite inspeccionar visualmente la jerarquía de vistas de una aplicación en ejecución. Ayuda a ver cómo se estructuran los elementos visuales y permite ajustar sus propiedades. Es útil cuando se trata de depurar problemas de diseño (p. ej., vistas superpuestas, márgenes incorrectos o elementos fuera de lugar).

- Layout Validation. Permite visualizar la manera en la que se ve la UI en diferentes dispositivos y tamaños de pantalla, lo que ayuda a detectar problemas de compatibilidad entre distintos tamaños o densidades de pantalla.

Emulador de Android

El emulador de Android en Android Studio permite **ejecutar** a la par que **depurar aplicaciones** en un **entorno virtual sin** la necesidad de **usar** un **dispositivo físico,** algo útil para probarlas en diferentes versiones de Android y tamaños de pantalla. El emulador incluye soporte para herramientas, como Logcat y el depurador.

Crashlytics (Firebase)

Crashlytics, parte de Firebase, es una herramienta útil para **monitorear** y **rastrear fallos** en la **aplicación** una vez que está en **producción.** Proporciona información detallada sobre los errores que ocurren en los dispositivos de quienes la usan, lo que permite a las personas desarrolladoras tanto **diagnosticar** como **solucionar problemas** en las **versiones lanzadas**.

Lint (herramienta de análisis estático)

Android Studio tiene integrada Lint, una herramienta de análisis estático que **analiza** el **código fuente** para **detectar errores potenciales** y **problemas** de **rendimiento**, así como de **accesibilidad.** Puede identificar **patrones peligrosos** o **problemáticos** antes de ejecutar la aplicación, algo que ayuda a las personas desarrolladoras a mantener un código limpio y eficiente.

Documentación en proyectos Android

La documentación es esencial para **asegurar** que el **código** sea **comprensible** y **fácil** de **mantener** a lo largo del tiempo tanto para la propia persona desarrolladora como para otros miembros del equipo.

Comentarios en el código

Hay **dos tipos:**

- Básicos. Son anotaciones que explican partes específicas del código. Se utilizan para hacer que este sea más legible y comprensible, especialmente en casos donde la lógica es compleja.
- De bloque. Se utilizan con el objetivo de explicar partes más largas del código o de hacer anotaciones de alto nivel.

Kotlin Documentation (KDoc)

KDoc es el **sistema estándar** para **escribir documentación** en proyectos Kotlin. Se parece mucho a **Javadoc** (usado en Java), pero presenta algunas diferencias específicas para su lenguaje. Permite generar documentación a partir de **comentarios estructurados,** lo que hace que sea fácil de leer y comprender.

Los comentarios de KDoc se colocan encima de clases, métodos y propiedades, y pueden incluir descripciones, detalles de parámetros, así como tipos de retorno.

A continuación, se muestra un ejemplo:

```
/**
* Calcula el área de un rectángulo dado su ancho y su altura.
*
* @param width El ancho del rectángulo
* @param height La altura del rectángulo
```

```
* @return El área calculada del rectángulo

*/

fun calculateArea(width: Int, height: Int): Int {

    return width * height

}
```

Las **etiquetas comunes** en KDoc incluyen estas:

- @param . Para describir los parámetros de la función.
- @return . Describe lo que la función devuelve.
- @throws . Sirve para documentar las excepciones que una función puede lanzar.

Documentación en la Wiki o README

En el caso de **proyectos más grandes,** es común mantener la **documentación adicional fuera** del **código,** como en un **archivo README.md** o en una **Wiki** dentro del repositorio de control de versiones (GitHub o GitLab). Esta puede **incluir:**

- Instrucciones de configuración (p. ej., cómo clonar, configurar y ejecutar el proyecto localmente).
- Guías de arquitectura. Explicaciones de la manera en la que se estructura la aplicación (p. ej., usando patrones, como MVVM o MVP).
- Dependencias. Listado de bibliotecas utilizadas y su propósito.
- Guías para contribución. Normas a fin de colaborar en el proyecto, que incluyen estándares de código, revisión de *pull requests,* etc.

Javadoc

Aunque KDoc es la herramienta de documentación nativa para Kotlin, Javadoc sigue siendo **útil** en **proyectos Java** o en aquellos que **mezclan ambos.** Javadoc genera automáticamente la **documentación HTML** a partir de **comentarios** en el **código.**

En la sección de A fondo encontrarás documentación sobre las opciones para desarrolladores de Android Studio.

3.7. Referencias bibliográficas

Anuar-UNIR. (2024a). *PMDM_2024/Tema 3/Entrenamiento_1*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%203/Entrenamiento_1

Anuar-UNIR. (2024b). *PMDM_2024/Tema 3/Entrenamiento_2*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%203/Entrenamiento_2

Anuar-UNIR. (2024c). *PMDM_2024/Tema 3/Entrenamiento_3*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%203/Entrenamiento_3

Anuar-UNIR. (2024d). *PMDM_2024/Tema 3/Entrenamiento_4*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%203/Entrenamiento_4

Anuar-UNIR. (2024e). *PMDM_2024/Tema 3/Entrenamiento_5*. https://github.com/Anuar-UNIR/PMDM_2024-2025/tree/main/Tema%203/Entrenamiento_5

Equipo de capacitación de Google Developers. (2023, octubre 19). *Cómo cambiar el tema de una app*. https://developer.android.com/codelabs/basic-android-kotlin-training-change-app-theme?hl=es-419#0

Layout Editor

Android Developers. (2024, diciembre 20). *Cómo desarrollar una IU con Views*. https://developer.android.com/studio/write/layout-editor?hl=es-419

Documentación oficial sobre el Layout Editor, herramienta oficial de Android Studio para el diseño de interfaces.

Testing en Android

AristiDevs. (2022, febrero 18). Testing en Android. *Curso Kotlin para Android*. https://cursokotlin.com/testing-en-android-test-unitarios/

El *testing* en Android es un aspecto crucial del desarrollo de aplicaciones, ya que garantiza su calidad, estabilidad y funcionalidad antes del lanzamiento. A medida que las aplicaciones móviles crecen en complejidad, las pruebas se vuelven fundamentales para asegurar que funcionan correctamente en una amplia variedad de dispositivos, versiones de Android y situaciones del mundo real.

Opciones para desarrolladores en dispositivos Android

Android Developers. (2025, abril 17). *Cómo configurar las opciones para desarrolladores en el dispositivo*. https://developer.android.com/studio/debug/dev-options?hl=es-419

Las Opciones para desarrolladores en dispositivos Android son un conjunto de configuraciones avanzadas que permiten a quienes desarrollan acceder a funcionalidades y herramientas necesarias para probar, depurar y optimizar sus aplicaciones. Estas opciones se encuentran desactivadas de manera predeterminada en la mayoría de los dispositivos, ya que están diseñadas principalmente para tareas técnicas durante el desarrollo de *software*.

Entrenamiento 1

- Desarrolla una **aplicación** en **Android Studio** con **Kotlin** que disponga de un TexView y un **botón.** Cada vez que se pulsa, el TextView debe **indicar** las **pulsaciones** que se han **realizado.**
- El desarrollo paso a paso es este:
  - Crea proyecto Android de Activity vacía con el diseño basado en Views .
  - Usa ConstraintLayout .
  - Coloca el Texview centrado y en la parte superior.
  - Define un texto inicial.
  - Define un margen superior para el TextView .
  - Define un ID para el TextView .
  - Coloca un Button en el centro de la pantalla y define tu texto (p. ej., ¡Púlsame!).
  - Cambia el color del fondo.
  - Define un ID para el Button .
  - Define en la MainActivity una variable pulsaciones .
  - Captura el evento Click del botón.
  - Muestra en el TextView el número de pulsaciones acumuladas.
- La solución se encuentra en Anuar-UNIR (2024a).

Entrenamiento 2

- Desarrolla una **aplicación** en **Android Studio** con **Kotlin** que **simule** la **vista** de un ***log in*** (véase la Figura 1).

Figura 1. Vista de *log in*. Fuente: elaboración propia.

- El desarrollo paso a paso es el siguiente:
  - Crea un proyecto Android de Activity vacía con el diseño basado en `Views`.
  - Usa `ConstraintLayout` para todos los componentes, a saber, `View`, `TextView` (LOG IN), `TextView` (Nombre de usuario), `EditTextView` (para introducir el correo electrónico), `TextView` (Contraseña), `EditTextView` (para meter la contraseña) y `Button`.
  - Personaliza los colores.
  - Usa los ficheros colors.xml y strings.xml.
  - Define en la `MainActivity` una variable `pulsaciones`.
  - Captura el evento `Click` del botón y muestra la información del correo electrónico y la contraseña.
- La solución se encuentra en Anuar-UNIR (2024b).

Entrenamiento 3

- A partir de la **aplicación** del **Entrenamiento 2, añade** una **actividad** que se **cargue después** de pulsar el **botón** de ***log in.***
- El desarrollo paso a paso es este (véase la Figura 2):
  - Genera una nueva Empty Activity.
  - Añade un `TextView` .
  - Genera un objeto de `Intent()` .
  - Añade los parámetros para pasar a la otra `Activity` .
  - Lee los parámetros.
  - Muestra los valores del *log in* en el `TextView` .

- La solución se encuentra en Anuar-UNIR (2024c).

Figura 2. Resultado del desarrollo de la actividad. Fuente: elaboración propia.

Entrenamiento 4

- Desarrolla una **aplicación** Android para **practicar** con los **estilos** y **temas** en **Android Studio.**
- El desarrollo paso a paso se encuentra en Equipo de capacitación de Google Developers (2013).
- La solución se encuentra en Anuar-UNIR (2024d).

Entrenamiento 5

- Desarrolla una **aplicación Android libre** con todo lo visto hasta ahora donde uses **diferentes componentes** y experimentes con sus **atributos.**
- El desarrollo paso a paso es el siguiente. Se trata de un proyecto libre, pero la aplicación debería tener estos **elementos:**
  - Uso de temas y estilos.
  - Botones, textViews , EditText o cardviews .
  - Uso de ConstraintLayout .
- La solución se encuentra en Anuar-UNIR (2024e).