

Curso desarrollo de aplicaciones móviles Android Trainee

Módulo 3: Desarrollo de aplicaciones móviles Android Java

MANUAL DEL PARTICIPANTE

CONTENIDO

Unidad 1.....	4
UML.....	4
Casos de Uso	6
Diagramas de casos de uso	8
Diagramas de clase.....	11
Unidad 2.....	14
El SDK de Android.....	14
Primeros Pasos	19
Como ejecutar una app en un dispositivo.....	24
Como hacer debug	25
Log de sistema.....	28
Breakpoints	29
Unidad 3.....	31
Componentes de un proyecto Android	31
Fragment	32
Layout / Diseño	33
Constraint Layout	40
App Bar.....	42
Listas	44
Tarjetas.....	45
Selection controls.....	47
Dialogs.....	48
Bottom Navigation	50
TextFields.....	53
Progress Indicators	53
Interacciones de usuario	54
Clase R y pipeline de assets.....	57
Vistas y listeners	58
Obtener input de usuario.....	61
Unidad 4.....	63
Más componentes de un proyecto Android.....	63
Principales funciones de los scripts de compilación	65

Los Assets de un proyecto Android	70
Layouts y tamaño	74
Unidad 5	76
Fundamentos de GIT	76
Comandos básicos de Git	78
Fundamentos de GitHub	81
Git y Android Studio	87
Unidad 6	91
Ciclo de vida de Android	91
Intents	91
Adaptadores Android	96
El ciclo de vida de Android	99
Patrón MVP	106
Callbacks	108
Unidad 7	109
Testing y Android	109
Test unitario usando Junit y Mockito	111
Test de integración usando espresso	112
Otras alternativas para el testing	118
Unidad 8	118

UNIDAD 1

UML

UML es la sigla en inglés del Lenguaje Universal de Modelado (Unified Modeling Language), este es un lenguaje gráfico para el modelamiento y desarrollo de sistemas. Entrega ventajas para el modelamiento y visualización de todas las fases de un proyecto de desarrollo de software, desde el análisis de requerimientos hasta la construcción y desarrollo.

UML se generó basado en los trabajos de Booch, Rumbaugh y Jacobson, a pesar de ser una idea no antigua, sus bases se han utilizado para diseñar y especificar sistemas de software desde principios de los años 90s, lográndose la unificación de varias notaciones a mediados de la década del 90. En 1997 se logró el desarrollo de un metamodelo común para el desarrollo de sistemas de software gracias a la participación de la Object Management Group (OMG), generándose su primera especificación UML 1.0, rápidamente y gracias a la participación de múltiples actores se logró a finales del 97 la generación de una versión refinada, denominada UML 1.1, aceptada por la OMG. Actualmente es esta organización la que se encarga de la continua evolución del lenguaje.

UML ha sido ampliamente adoptado por la industria, transformándose prácticamente en el estándar para el modelamiento visual de sistemas, siendo incorporado en una gran cantidad de herramientas de uso profesional, tales como Visual Studio.

¿Qué es UML?

La idea base de UML es capturar los detalles significativos sobre un sistema de manera tal que el problema que resuelve sea fácilmente comprendido, permitiendo su implementación y desarrollo.

UML es una notación rica en detalles, no solo provee posibilidades de notación para elementos básicos, sino que también incluye formas de expresar relaciones complejas entre las piezas de un sistema. Se permite la notación de relaciones estáticas o dinámicas

- Relaciones estáticas se relacionan con los aspectos estructurales de un sistema. Relaciones de herencia entre clases, interfaces implementadas por una clase y dependencia de otras clases son ejemplos.
- Relaciones dinámicas se relacionan con el comportamiento de un sistema y su ejecución a través del tiempo. El intercambio de mensajes entre un grupo de clases para cumplir con alguna responsabilidad y el flujo de control dentro de un sistema son capturados en el contexto de relaciones dinámicas que existen en un sistema

Existen varios tipos de diagramas UML organizados en áreas de foco específicos llamadas vistas.

Dentro de UML encontramos varios tipos de diagramas, algunos de estos son:

- Diagramas de casos de uso: un diagrama de casos de uso muestra casos de uso, actores y sus relaciones. Estos capturan requerimientos precisos para el sistema desde una perspectiva del usuario.
- Diagrama de clases: un diagrama de clases muestra las relaciones estáticas que existen entre un grupo de clases e interfaces en el sistema. Algunos tipos de relaciones típicas son herencia, agregación y dependencia
- Diagrama de estado: los diagramas de estado son excelentes para capturar el comportamiento dinámico de un sistema. Son particularmente aplicables a sistemas basados en eventos.
- Diagrama de actividad: un diagrama de actividad es una extensión del diagrama de estado y similar en concepto a un diagrama de flujo. Un diagrama de actividad permite el modelamiento del comportamiento del sistema en términos de interacción de distintos objetos. Estos diagramas se utilizan comúnmente para modelar flujos de trabajo (workflows).
- Diagrama de interacción: un diagrama de interacción se utiliza para modelar el comportamiento dinámico de un sistema, existen dos tipos de diagramas de interacción:
 - Diagrama de secuencia: utilizado para modelar el intercambio de mensajes entre objetos en el sistema, también capturando el orden relativo al tiempo de estos mensajes.
 - Diagrama de colaboración: el intercambio de mensajes se captura en el contexto de la estructura general de relaciones entre objetos.
 - Ambos diagramas son equivalentes y es posible convertir de uno al otro fácilmente. Los diagramas de interacción comúnmente se utilizan para modelar el flujo de control en un caso de uso y para describir como objetos interactúan durante la ejecución de esta operación.
- Diagrama de componentes: un componente representa la manifestación física de una parte de un sistema, tal como un archivo, un ejecutable, etc. Un diagrama de componentes ilustra las relaciones y dependencias entre los componentes que forman parte de un sistema. Un componente normalmente se puede formar de una o más clases, o incluso subsistemas.

- Diagrama de despliegue: un diagrama de despliegue muestra la arquitectura de un sistema desde la perspectiva de sus nodos, procesadores y relaciones entre ellos. Uno o más componentes típicamente pueden ser parte de un nodo

¿Por qué usar UML?

Cualquier programador con un nivel básico de conocimientos debería ser capaz de construir piezas de software que cumplan su objetivo. Construir software de nivel empresarial, que sea escalable, mantenible y evolucione es una tarea mucho más exigente, actualmente los sistemas evolucionan cada vez más rápido y nuevas tecnologías aparecen de forma veloz, se hace aún más importante tomar en consideración una mirada a largo plazo donde se entienda la necesidad de mantener, escalar y evolucionar el sistema que se está creando.

Es posible crear sistemas sin tener grandes consideraciones ingenieriles, tareas simples de desarrollo pueden ser llevadas con relativa baja complejidad, pero tarde o temprano es posible que el sistema no pueda escalar de acuerdo a la demanda requerida. Esto se debe a que el sistema probablemente no fue diseñado para evolucionar fácilmente en función de nuevos requerimientos.

Casos de Uso

Los casos de uso son una idea que está presente desde los 80s, inicialmente discutidos por Ivar Jacobsen y luego parte de UML. Los casos de uso también son parte importante del Proceso Racional Unificado (RUP, del inglés Rational Unified Process), siendo Rational hoy en día parte de IBM. Los casos de uso tienen tanto una representación en texto como varias notaciones gráficas y son generalmente utilizados en la construcción de sistemas orientados a objetos, aunque también pueden ser utilizados en la construcción de sistemas con otros paradigmas.

Un caso de uso describe funciones de un sistema de software desde la perspectiva de un usuario o actor. Los casos de uso pueden ser descritos con diferentes niveles de detalle, desde un resumen breve o casual hasta un completo detalle, dependiendo del caso.

Un caso de uso incluye otros tópicos adicionales además de los actores, tales como precondiciones o postcondiciones.

Un caso de uso se puede relacionar con una historia de usuario, ambos tienen una perspectiva similar pero el caso de uso cuenta con un lenguaje más formal y comúnmente pueden ser más extensos que una historia de usuario.

Algunos autores han identificado críticas la utilización de casos de uso como especificación pues se argumenta que no tienen formas de tratar requerimientos no funcionales, tales

como seguridad o calidad. Esta crítica podría ser aplicada válidamente a casi cualquier método de diseño, además, al momento de confeccionar casos de uso, el autor podría tomar las precauciones de añadir definiciones sobre requerimientos no funcionales, de manera formal o informal.

ID: UC.12

Nombre: Realizar Comentario

Actores: Usuario logeado

Descripción: Un usuario realiza un comentario en una publicación

Prioridad: Alta

Flujo:

1. Usuario: El usuario hace click en el cuadro de texto denominado “comentario”
2. Sistema: El sistema presenta la interfaz WYSIWYG para permitir el ingreso de un comentario con formato gráfico
3. Usuario: El usuario ingresa el texto de su comentario y hace click en el botón Publicar
4. Sistema: El sistema presenta el texto formateado en la sección “comentarios” de la publicación actual
5. ...

Identificación de casos de uso

La identificación de un caso de uso comienza por el análisis de las funcionalidades de negocio del sistema, es importante identificar cuáles son las funciones que el sistema permite y que el usuario utilizará para completar una tarea específica. En ocasiones es fácil confundir algunas funcionalidades del sistema con casos de uso, por ejemplo, es común ver el caso de uso “realizar login”, sin embargo esta funcionalidad del sistema no cumple una misión de negocios, corresponde a una actividad que se realiza como condición para poder utilizar las funcionalidades clave del sistema y por lo tanto no debería ser considerada como un caso de uso per se. Sin embargo, no es incorrecto la confección de un caso de uso específico para ser utilizado como referencia en otros. Algunos ejemplos de

casos de uso son: editar publicación, imprimir documento, realizar transferencia, descargar archivo, realizar búsqueda, etc.

Un caso de uso corresponde a un artefacto de análisis, consistente de un relato sobre la forma en que un usuario interactúa con el sistema, detallando acciones realizadas por el usuario y la reacción por parte del sistema. Debe ser construido con un lenguaje formal que permite comprender totalmente todos los pasos que un usuario debe realizar para conseguir utilizar una funcionalidad específica. El caso de uso debe contar como mínimo con un identificador único, título, actores, descripción y detalle del flujo. Los casos de uso pueden ser clasificados de acuerdo con su importancia o recurrencia como:

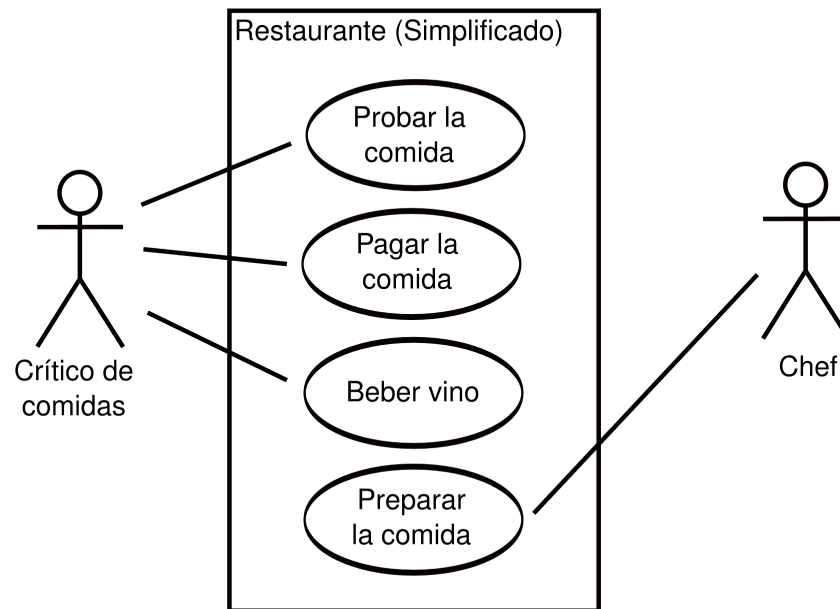
- Primarios: casos de uso que representan los procesos más comunes e importantes del sistema
- Secundario: casos de uso que representan procesos menos comunes o raros.
- Opcional: casos de uso que representan procesos que podrían no abordarse.

En la confección de casos de uso podemos tener un diferente nivel de detalle, variando de acuerdo con su público objetivo o *stakeholders*. Podemos clasificar los casos de uso según la profundidad de su detalle contenido en:

- Resumidos o de “alto nivel”: un caso de uso que no incluye todo el detalle y generalmente se presenta como un resumen en pocos párrafos de como se lleva a cabo la acción por parte del usuario. Generalmente tiene como objetivo presentar la información a un nivel gerencial o no técnico.
- Extendidos: un caso de uso que incluye un nivel de detalle mucho más completo, generalmente este es el tipo de caso de uso que debería ser construido y utilizado como artefacto de especificación a ser utilizado por los desarrolladores o el resto del equipo técnico del proyecto.

Diagramas de casos de uso

Existe una notación gráfica conocido como diagrama de casos de uso, este gráfico se utiliza para especificar la comunicación y comportamiento de un sistema mediante la interacción con otros usuarios y otros sistemas.



Dentro de los elementos de un diagrama de casos de uso encontramos:

- Actores: representación de una entidad o entidades que realizan cierta acción en el sistema, comúnmente se trata de un usuario o rol de usuario.
- Caso de uso: es la representación visual de un caso de uso, corresponde la funcionalidad de negocio en un sistema
- Límite de sistema: define el alcance de lo que el sistema será o incluirá

Ref: <https://www.developer.com/design/article.php/2109801>

Diagramas de secuencia

El diagrama de secuencia es un tipo de diagrama usado para modelar interacción entre objetos en un sistema según UML.

Un diagrama de secuencia muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada caso de uso. A menudo es útil para complementar a un diagrama de clases, pues el diagrama de secuencia se podría describir de manera informal como "el diagrama de clases en movimiento", por lo que ambos deben estar relacionados entre sí (mismas clases, métodos, atributos...). Mientras que el diagrama de casos de uso permite el modelado de una vista business del escenario, el diagrama de secuencia contiene detalles de implementación del escenario, incluyendo los

objetos y clases que se usan para implementar el escenario y mensajes intercambiados entre los objetos;

Típicamente se examina la descripción de un caso de uso para determinar qué objetos son necesarios para la implementación del escenario. Si se dispone de la descripción de cada caso de uso como una secuencia de varios pasos, entonces se puede "caminar sobre" esos pasos para descubrir qué objetos son necesarios para que se puedan seguir los pasos. Un diagrama de secuencia muestra los objetos que intervienen en el escenario con líneas discontinuas verticales, y los mensajes pasados entre los objetos como flechas horizontales.

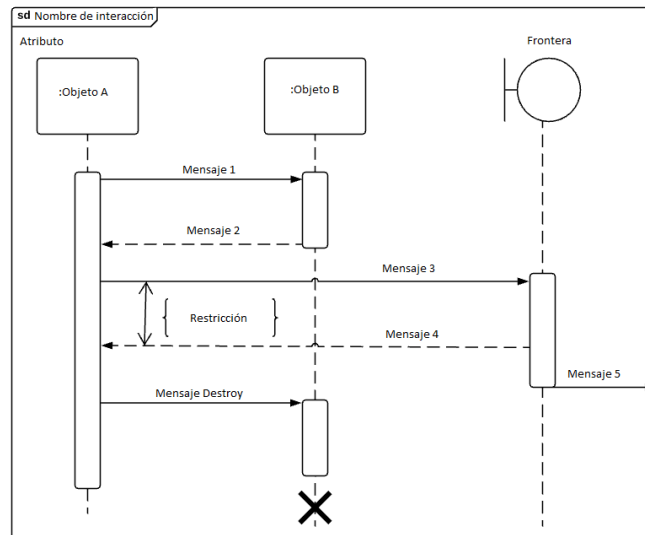
Tipos de mensajes

Existen dos tipos de mensajes: síncronos y asíncronos. Los mensajes síncronos se corresponden con llamadas a métodos del objeto que recibe el mensaje. El objeto que envía el mensaje queda bloqueado hasta que termina la llamada. Este tipo de mensajes se representan con flechas con la punta rellena. Los mensajes asíncronos terminan inmediatamente, y crean un nuevo hilo de ejecución dentro de la secuencia. Se representan con flechas con la punta hueca.

También se representa la respuesta a un mensaje con una flecha discontinua.

Pueden ser usados en dos formas

- De instancia: describe un escenario específico (un escenario es una instancia de la ejecución de un caso de uso).
- Genérico: describe la interacción para un caso de uso. Utiliza ramificaciones ("branches"), condiciones y bucles. A veces se puede usar para usted realizar trabajos



Diagramas de clase

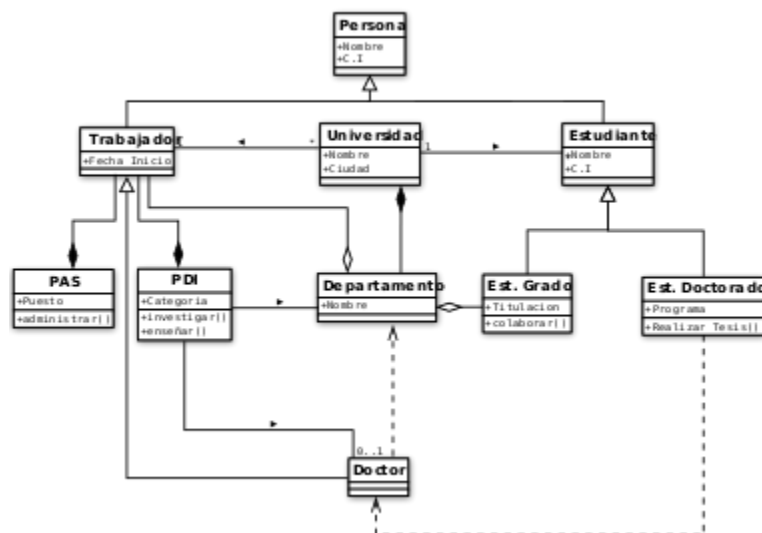
En UML, un diagrama de clases es un artefacto que comunica la estructura estática que describe un sistema, mostrando sus clases, atributos, operaciones o métodos, y las relaciones entre objetos. Este diagrama corresponde a uno de los principales artefactos de especificación de un sistema orientado a objetos, se utiliza para comunicar conceptualmente o también, gracias al uso de algunas herramientas, se puede utilizar para la generación de código ejecutable.

UML especifica dos tipos de ámbitos para los miembros: instancias y clasificadores y estos últimos se representan con nombres subrayados.

- Los miembros clasificadores se denotan comúnmente como “estáticos” en muchos lenguajes de programación. Su ámbito es la propia clase.
 - Los valores de los atributos son los mismos en todas las instancias
 - La invocación de métodos no afecta al estado de las instancias
- Los miembros instancias tienen como ámbito una instancia específica.
 - Los valores de los atributos pueden variar entre instancias
 - La invocación de métodos puede afectar al estado de las instancias (es decir, cambiar el valor de sus atributos)

Para indicar que un miembro posee un ámbito de clasificador, hay que subrayar su nombre. De lo contrario, se asume por defecto que tendrá ámbito de instancia.

Diagrama de Clases



Relaciones

Una relación es un término general que abarca los tipos específicos de conexiones lógicas que se pueden encontrar en los diagramas de clases y objetos. UML presenta las siguientes relaciones:

Enlace: Un enlace es la relación más básica entre objetos.

Asociación: Una asociación representa a una familia de enlaces. Una asociación binaria (entre dos clases) normalmente se representa con una línea continua. Una misma asociación puede relacionar cualquier número de clases. Una asociación que relacione tres clases se llama asociación ternaria.

A una asociación se le puede asignar un nombre, y en sus extremos se puede hacer indicaciones, como el rol que desempeña la asociación, los nombres de las clases relacionadas, su multiplicidad, su visibilidad, y otras propiedades.

Hay cuatro tipos diferentes de asociación: bidireccional, unidireccional, agregación (en la que se incluye la composición) y reflexiva. Las asociaciones unidireccional y bidireccional son las más comunes.

Por ejemplo, una clase vuelo se asocia con una clase avión de forma bidireccional. La asociación representa la relación estática que comparten los objetos de ambas clases.

Agregación: La agregación o agrupación es una variante de la relación de asociación “tiene un”: la agregación es más específica que la asociación. Se trata de una asociación que representa una relación de tipo parte-todo o parte-de.

Como se puede ver en la imagen del ejemplo (en inglés), un Profesor 'tiene una' clase a la que enseña.

Al ser un tipo de asociación, una agregación puede tener un nombre y las mismas indicaciones en los extremos de la línea. Sin embargo, una agregación no puede incluir más de dos clases; debe ser una asociación binaria.

Una agregación se puede dar cuando una clase es una colección o un contenedor de otras clases, pero a su vez, el tiempo de vida de las clases contenidas no tienen una dependencia fuerte del tiempo de vida de la clase contenedora (de el todo). Es decir, el contenido de la clase contenedora no se destruye automáticamente cuando desaparece dicha clase.

En UML, se representa gráficamente con un rombo hueco junto a la clase contenedora con una línea que lo conecta a la clase contenida. Todo este conjunto es, semánticamente, un objeto extendido que es tratado como una única unidad en muchas operaciones, aunque físicamente está hecho de varios objetos más pequeños.

Otras características:

- El diagrama de clases puede tener como ejemplo: una clase que sería un objeto o persona misma en la cual se especifica cada acción y especificación.
- Propiedades de objetos que tienen propiedades y/u operaciones que contienen un contexto y un dominio, los primeros dos ejemplos son clases de datos y el tercero clase de lógica de negocio, dependiendo de quién diseñe el sistema se pueden unir los datos con las operaciones
- El diagrama de clases incluye mucha más información como la relación entre un objeto y otro, la herencia de propiedades de otro objeto, conjuntos de operaciones/propiedades que son implementadas para una interfaz gráfica.
- Presenta las clases del sistema con sus relaciones estructurales y de herencia.
- El diagrama de clases es la base para elaborar una arquitectura MVC o MVP.

UNIDAD 2

Java es uno de los lenguajes de programación más importantes en el desarrollo móvil y es comúnmente utilizada para el desarrollo de aplicaciones Android. Los dispositivos Android no corren archivos .class y .jar como en otras plataformas Java, en su lugar utilizan sus propios formatos optimizados para código compilado, esto implica que no podemos utilizar un ambiente de desarrollo Java cualquiera para desarrollar aplicaciones y requerimos de herramientas especiales para convertir el código compilado a formato Android, para su despliegue y para realizar *debugging*.

El SDK de Android

Android cuenta con su propio SDK, o kit de desarrollo de software por sus siglas en inglés (Software Development Kit), este contiene todas las librerías y herramientas que necesitamos para desarrollar aplicaciones Android.

El Android SDK incluye:

- SDK para cada versión de Android
- SDK Tools: herramientas para el *debuggeo* y *testing*, junto con otras utilidades. Incluye características específicas para cada plataforma.
- Aplicaciones de ejemplo: se incluyen ejemplos que ayudan a comprender la utilización de las librerías incluidas.
- Documentación: se incluye la documentación en su versión específica para cada versión, esto permite la consulta de esta de manera offline.

- Android Support: APIs adicionales que no están disponibles en la plataforma estándar

El ambiente de desarrollo

Para el desarrollo de aplicaciones Android se puede utilizar variadas herramientas, en este manual nos centraremos en Android Studio, una de las herramientas más utilizadas para el desarrollo Android que incluye todas las capacidades necesarias para el desarrollo de aplicaciones de alto impacto. Android Studio incluye varias herramientas de utilidad, destacando la incorporación de plantillas de trabajo que facilitan la creación de proyectos.

Instalar JAVA

Android Studio es un ambiente que funciona sobre Java, necesitamos en primer lugar asegurarnos de que el ambiente Java correcto se encuentre instalado en nuestro equipo de desarrollo.

Primero debemos asegurarnos de cumplir con los requerimientos de Android Studio, al momento de escribir este manual, la versión estable corresponde a la 4.1.2 y sus requerimientos mínimos para Windows, Mac y Linux se detallan a continuación:

Requerimientos mínimos Windows:

- Windows 7/8/10 (64 Bits)
- 4 GB ram, 8 GB recomendado
- 2 GB de espacio disponible como mínimo, 4 o más GB recomendados
- 1280 x 800 de resolución mínima de pantalla

Requerimientos mínimos Mac:

- Mac OSX 10.10 (Yosemite) o superior
- 4 GB ram mínimo, 8 GB recomendado
- 2 GB de espacio disponible como mínimo, 4 o más GB recomendados
- 1280 x 800 de resolución mínima de pantalla

Requerimientos mínimos Linux:

- Escritorio basado en GNOME o KDE
- Distribución de 64 bit capaz de ejecutar programas de 32 bit
- Librería GNU C (glibc) 2.19 o superior

- 4 GB ram mínimo, 8 GB recomendado
- 2 GB de espacio disponible mínimo, 4 o más GB Recomendados
- 1280 x 800 de resolución mínima de pantalla

Es necesaria la instalación de JRE o JDK de acuerdo a las necesidades del desarrollador y de acuerdo a la versión de Android Studio a utilizar, antes de proceder a la siguiente sección asegúrese de descargar e instalar el JRE o JDK indicado en la documentación de Android Studio

<https://java.com/es/download/help/develop.html>

Descargar e instalar Android Studio

Una vez instalado Java proceda a descargar e instalar Android Studio.

<https://developer.android.com/studio/>

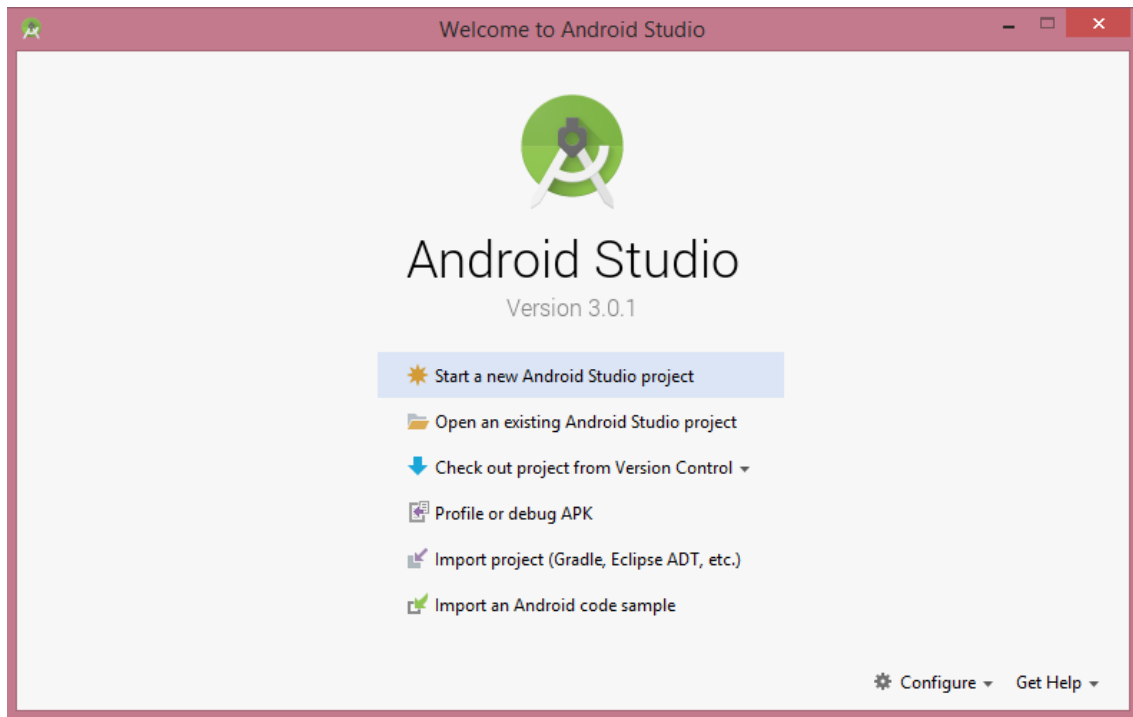
A continuación se guiará la instalación de Android Studio en Windows, para otras plataformas por favor seguir las instrucciones indicadas en el sitio de descarga.

1. Si descargaste un archivo .exe (recomendado), haz doble clic en él para iniciarlo.

Si descargaste un archivo .zip, extráelo y copia la carpeta android-studio en la carpeta Archivos de programa. A continuación, abre la carpeta android-studio > bin y, luego, inicia studio64.exe (para máquinas de 64 bits) o studio.exe (para las de 32 bits).

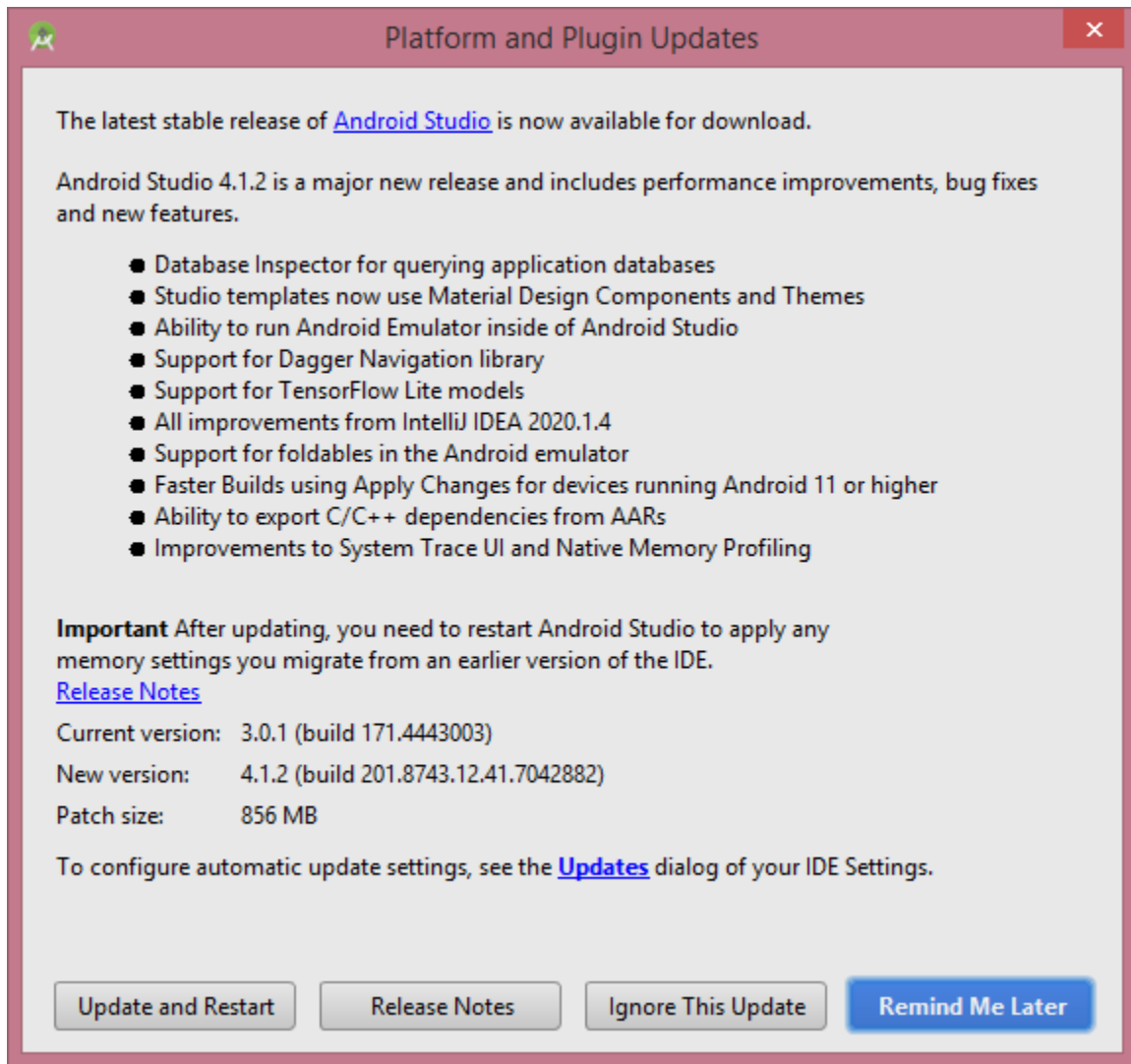
2. Sigue los pasos del asistente de configuración en Android Studio y asegúrate de instalar los paquetes de SDK que recomiende.

Cuando haya herramientas nuevas y otras API disponibles, Android Studio te lo informará por medio de una ventana emergente. También puedes buscar actualizaciones si haces clic en Help > Check for Update.



Es importante mantener actualizado el ambiente de desarrollo, una vez instalado Android Studio es fácil mantener al día el IDE (Ambiente Integrado de Desarrollo, de sus siglas en inglés Integrated Development Environment) y las herramientas de Android SDK con actualizaciones automáticas y Android SDK manager.

Android Studio te notifica con un cuadro de diálogo pequeño cuando hay una actualización disponible para el IDE. También puedes buscar actualizaciones de forma manual si haces clic en Help > Check for Update (en Mac, Android Studio > Check for Updates).



¿Qué es un IDE?

Android Studio es un IDE, existen muchas alternativas en el mercado de IDE para diferentes ambientes de desarrollo, algunos de los más reconocidos son:

- Visual Studio: para el desarrollo utilizando Stack .NET
- Eclipse IDE: una plataforma muy amplia y completa pero nos referimos específicamente a las herramientas de desarrollo para JAVA, tanto en sus versiones estándar como Enterprise.
- IntelliJ Idea: alternativa para el desarrollo de sistemas Java
- Netbeans: ambiente muy utilizado para el desarrollo de sistemas de nivel Enterprise en Java

Es posible desarrollar sin la utilización de un IDE, pero el desarrollo se vuelve una tarea tediosa y muchas veces complicada. Un IDE nos permite contar con un ambiente de herramientas y utilidades que nos entregan amplias facilidades al momento de desarrollar, son de especial utilidad las capacidades de **depuración** que nos permiten encontrar defectos o simplemente apoyarnos al momento de desarrollar funcionalidades nuevas. Podemos pensar en un IDE como la “navaja suiza” del programador.

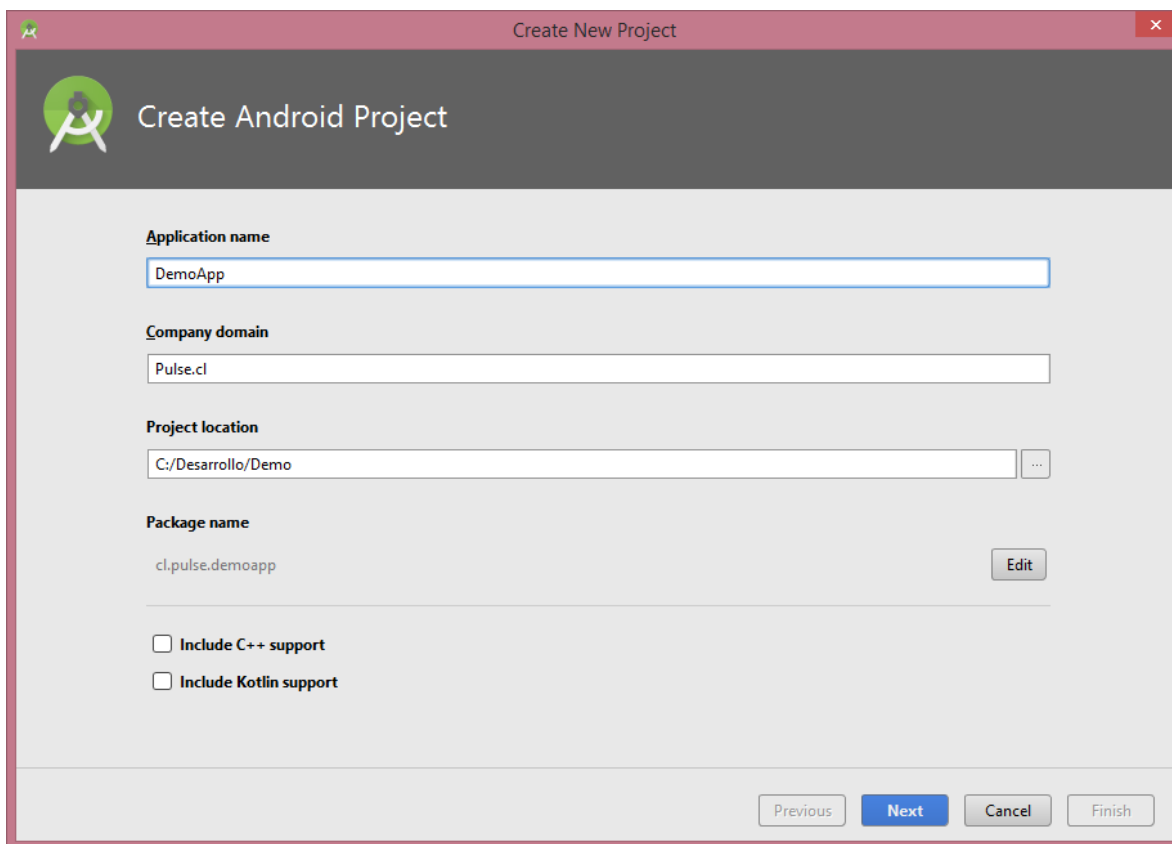
Primeros Pasos

Ahora que el ambiente de desarrollo se encuentra instalado, procederemos a construir una simple aplicación Android de ejemplo.

Para crear una nueva aplicación debemos crear un nuevo proyecto, con Android Studio abierto sigue las siguientes instrucciones.

Crear un nuevo proyecto

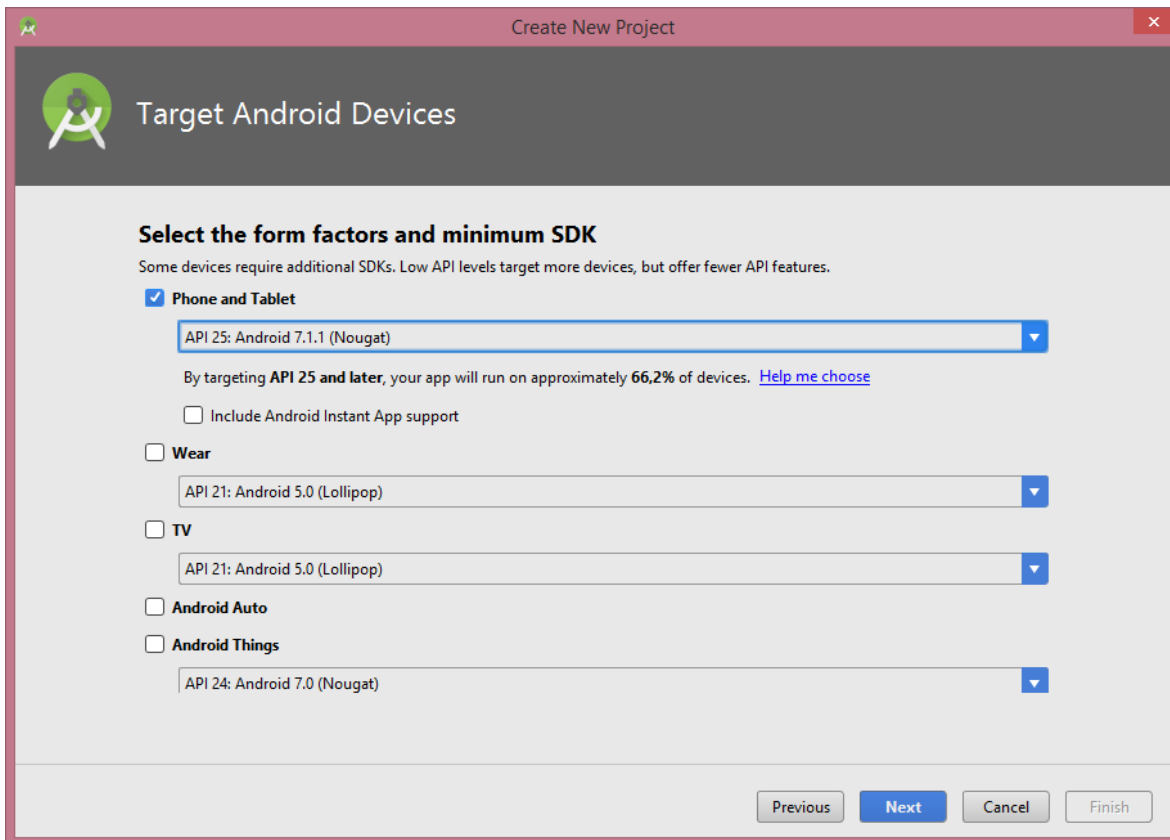
La pantalla inicial de Android Studio nos entrega varias opciones para trabajar, queremos crear un nuevo proyecto por lo tanto debemos dar click en la opción “Crear nuevo proyecto Android”.



The screenshot shows the 'Create New Project' dialog box in Android Studio. The dialog has a title bar with the text 'Create New Project' and a close button. The main area is titled 'Create Android Project' and contains several input fields and checkboxes. The 'Application name' field is filled with 'DemoApp'. The 'Company domain' field is filled with 'Pulse.cl'. The 'Project location' field is filled with 'C:/Desarrollo/Demo' and has a browse button (...). The 'Package name' field is filled with 'cl.pulse.demoapp' and has an 'Edit' button. There are two checkboxes: 'Include C++ support' and 'Include Kotlin support', both of which are unchecked. At the bottom right, there are four buttons: 'Previous', 'Next' (highlighted in blue), 'Cancel', and 'Finish'.

Ingresa el nombre del proyecto y el resto de los datos solicitados, acepta la ubicación por defecto y da click al botón siguiente.

A continuación, se debe seleccionar el nivel de API que se utilizará, el nivel de API es un número que va aumentando con cada versión nueva de API disponible. Recomendamos para este manual utilizar API 25: Android 7.1.1 (Nougat). Da click en el botón continuar.



The screenshot shows the 'Create New Project' dialog in Android Studio, specifically the 'Target Android Devices' screen. The title bar says 'Create New Project'. The main heading is 'Target Android Devices'. Below this, there's a section titled 'Select the form factors and minimum SDK' with a subtitle: 'Some devices require additional SDKs. Low API levels target more devices, but offer fewer API features.'

The 'Phone and Tablet' option is selected with a checked checkbox. Below it, a dropdown menu shows 'API 25: Android 7.1.1 (Nougat)'. A note states: 'By targeting **API 25 and later**, your app will run on approximately **66,2%** of devices. [Help me choose](#)'. There is also an unchecked checkbox for 'Include Android Instant App support'.

Other form factors are listed with unchecked checkboxes: 'Wear' (dropdown shows 'API 21: Android 5.0 (Lollipop)'), 'TV' (dropdown shows 'API 21: Android 5.0 (Lollipop)'), 'Android Auto', and 'Android Things' (dropdown shows 'API 24: Android 7.0 (Nougat)').

At the bottom, there are four buttons: 'Previous' (disabled), 'Next' (active), 'Cancel' (disabled), and 'Finish' (disabled).

Notar que al seleccionar el SDK mínimo para el proyecto, el IDE nos indica el nivel de porcentaje de dispositivos Android en los que será posible ejecutar la aplicación. Utilizar las últimas versiones disponibles no necesariamente significa poder contar con compatibilidad para más dispositivos, generalmente las últimas versiones de Android SDK toman un tiempo para ser adoptadas por los usuarios y es común trabajar con versiones un poco más viejas.

Nombres internos, etiquetas y números de compilación

Las versiones en desarrollo de Android se organizan en familias con nombres internos que van en orden alfabético y se inspiran en golosinas.

Nombre Interno	Versión	Nivel de API
Android11	11	Nivel de API 30
Android10	10	Nivel de API 29
Pie	9	Nivel de API 28
Oreo	8.1.0	Nivel de API 27
Oreo	8.0.0	Nivel de API 26
Nougat	7.1	Nivel de API 25
Nougat	7.0	Nivel de API 24
Marshmallow	6.0	Nivel de API 23
Lollipop	5.1	Nivel de API 22
Lollipop	5.0	Nivel de API 21
KitKat	4.4	Nivel de API 19
Jelly Bean	4.3.x	Nivel de API 18
Jelly Bean	4.2.x	Nivel de API 17
Jelly Bean	4.1.x	Nivel de API 16
Ice Cream Sandwich	4.0.3 – 4.0.4	Nivel de API 15
Ice Cream Sandwich	4.0.1 – 4.0.2	Nivel de API 14
Honeycomb	3.2.x	Nivel de API 13
Honeycomb	3.1	Nivel de API 12
Honeycomb	3.0	Nivel de API 11
Gingerbread	2.3.3 - 2.3.7	Nivel de API 10
Gingerbread	2.3 - 2.3.2	Nivel de API 9
Froyo	2.2.x	Nivel de API 8
Eclair	2.1	Nivel de API 7
Eclair	2.0.1	Nivel de API 6
Eclair	2.0	Nivel de API 5
Donut	1.6	Nivel de API 4
Cupcake	1.5	Nivel de API 3

(sin nombre interno)	1.1	Nivel de API 2
(sin nombre interno)	1.0	Nivel de API 1

A continuación, debemos crear una **activity** para el proyecto. Toda aplicación Android es una colección de pantallas, cada pantalla se compone de una activity y un layout.

Una **activity** es una acción sencilla y definida que un usuario puede hacer. Podemos tener una activity para componer un correo electrónico, tomar una foto o encontrar un contacto. Las activities generalmente se asocian con una pantalla y están escritas en Java.

Un **layout** describe la apariencia de una pantalla. Los layouts se escriben en archivos XML que indican a Android como se organizan los diferentes elementos.

A continuación, detallaremos como las activities y layouts funcionan en conjunto para crear una interfaz de usuario:

1. El dispositivo lanza la aplicación y crea una instancia de un activity
2. La activity especifica un layout
3. La activity le indica a Android que debe desplegar el layout
4. El usuario interactúa con el layout desplegado en pantalla
5. La activity responde a las interacciones ejecutando el código de la aplicación
6. La activity refresca la pantalla
7. El usuario puede ver el resultado en pantalla

Crear un **activity**

La próxima pantalla entrega una serie de plantillas que se pueden utilizar para crear una activity y layout, es común reutilizar alguna de estas opciones. Para este manual utilizaremos la activity vacía pues crearemos todo a mano.

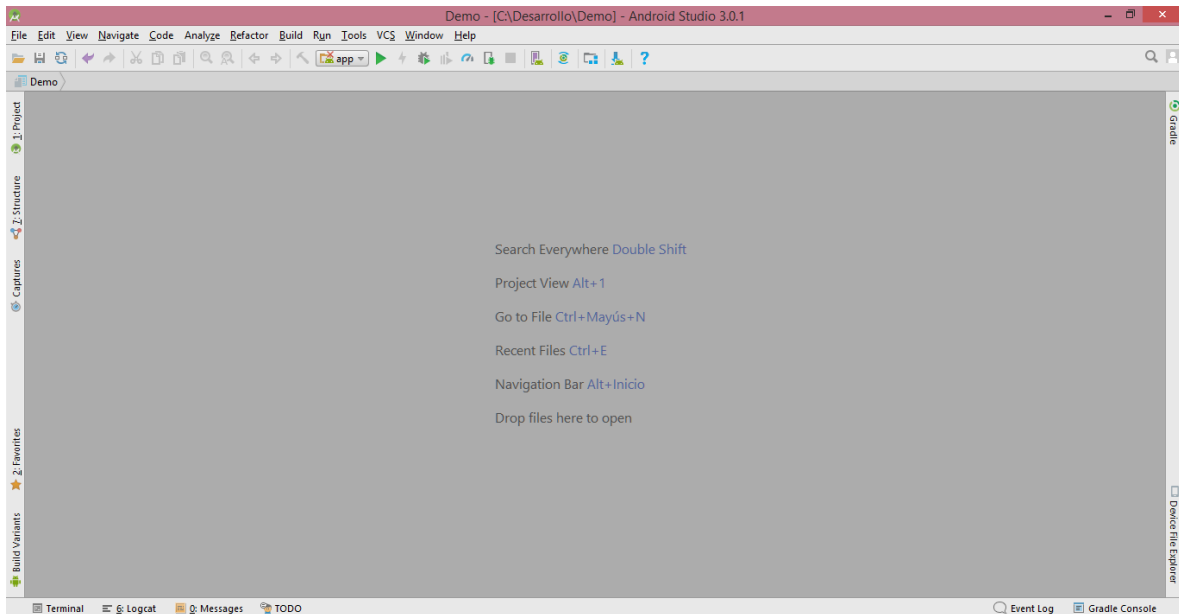
Android Studio continuará con el *wizard* de creación de activity, se consultará al usuario el nombre del activity y layout a utilizar, también se solicitará el título de la pantalla y el nombre para el recurso de menú. Ingrese “MainActivity” para el nombre y “activity_main” para el nombre del layout. La activity es una clase Java y el layout es un archivo XML, dado los nombres ingresados, Android Studio creará los archivos *MainActivity.java* y *activity_main.xml*.

Al dar click al botón “Finalizar”, Android Studio creará el proyecto y preparará todo para que podamos continuar con el desarrollo de nuestra aplicación.

Hasta el momento hemos realizado las siguientes acciones:

- El wizard de Android Studio ha creado un proyecto y configurado una App de acuerdo con lo que hemos detallado.
- Se ha creado un activity y un layout basados en una plantilla básica.

La plantilla de proyecto se encuentra lista y disponible para que continuemos con el desarrollo, podemos ahora



Como ejecutar una app en un dispositivo

Cuando compilas una app de Android, es importante probar siempre la app en un dispositivo real antes de lanzarla. A continuación, se describe cómo configurar el entorno de desarrollo y un dispositivo Android para prueba y depuración a través de una conexión Android Debug Bridge (adb).

Antes de iniciar la depuración en el dispositivo, debemos hacer lo siguiente:

1. En el dispositivo, abre la app de Configuración, selecciona Opciones para desarrolladores y luego habilita Depuración por USB. (puedes encontrar fácilmente las instrucciones para habilitar el “modo desarrollador” usando tu buscador favorito)
2. Configura el sistema para que detecte el dispositivo.
 - a. Sistema operativo Chrome: No se requiere ninguna configuración adicional.
 - b. macOS: No se requiere ninguna configuración adicional.
 - c. Ubuntu Linux: Hay dos cosas que se deben configurar correctamente: una es que cada usuario que quiera usar adb debe estar en el grupo plugdev y la otra es que el sistema tenga instaladas reglas udev que cubran el dispositivo.
 - d. Windows: Instala un controlador USB para adb (ver <https://developer.android.com/studio/run/oem-usb>)

Cuando esté todo listo y el dispositivo esté conectado mediante USB, dar clic en Run en Android Studio para compilar y ejecutar la app en el dispositivo.

También podemos usar adb para ejecutar comandos, de la siguiente manera:

- Verifica que el dispositivo esté conectado; para ello, ejecuta el comando `adb devices` desde el directorio `android_sdk/platform-tools/`. Si está conectado, verás el dispositivo en la lista.
- Ejecutar cualquier comando de adb con la marca `-d` para seleccionar el dispositivo como destino.

Si se siguió las instrucciones, Android Studio se encargará de compilar y empaquetar la aplicación en un archivo APK de acuerdo con las opciones seleccionadas y realizar la instalación de este en el dispositivo conectado, al cabo de unos segundos se ejecutará automáticamente el APK en el dispositivo y permitirá también acceder al modo debug, permitiendo al desarrollador obtener información en línea sobre la ejecución de la app.

En caso de encontrar dificultades, por favor consultar la documentación oficial donde se detallan múltiples posibles escenarios de error y como solucionarlos

<https://developer.android.com/studio/run/device>

Como hacer debug

Android Studio proporciona un depurador que te permite realizar las siguientes acciones y más:

- Seleccionar un dispositivo en el cual depurarás tu app
- Establecer interrupciones en tu código Java, Kotlin y C/C++
- Examinar variables y evaluar expresiones en el tiempo de ejecución

Antes de comenzar a depurar, debemos preparar nuestro ambiente de la siguiente manera:

- Habilitar la depuración en el dispositivo: Si estás usando el emulador, esta opción estará activada de forma predeterminada. Sin embargo, en el caso de un dispositivo conectado, deberás habilitar la depuración en las opciones para desarrolladores del dispositivo.
- Ejecuta una variante de compilación depurable: Debes usar una variante de compilación que incluya `debuggable true` en la configuración de compilación. Por lo general, simplemente puedes seleccionar la variante "debug" predeterminada que se incluye en cada proyecto de Android Studio (aunque no esté visible en el archivo `build.gradle`). Sin embargo, si defines nuevos tipos de compilación que deberían ser depurables, debes agregar "debuggable true" al tipo de compilación de la siguiente manera:

```
android {  
    buildTypes {  
        customDebugType {  
            debuggable true  
            ...  
        }  
    }  
}
```

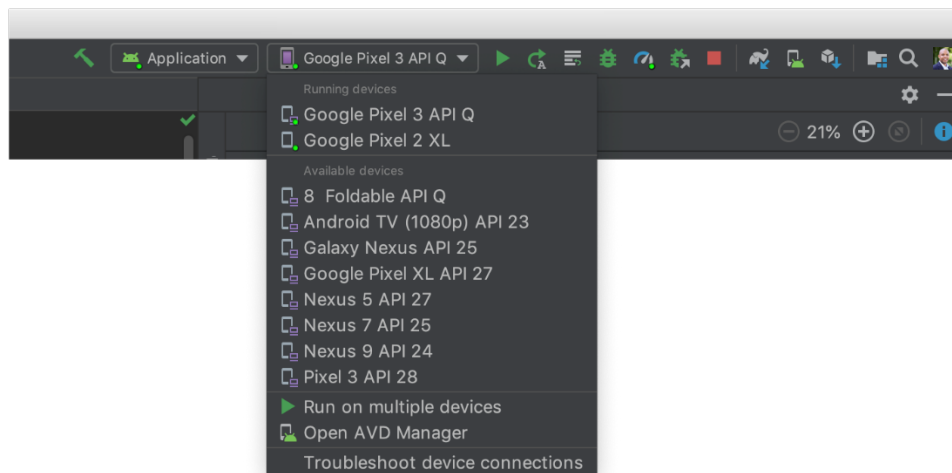
```
}  
}
```

Esta propiedad también se aplica a los módulos con código C o C++ (ya no se usa la propiedad `jniDebuggable`).

Si la app depende de un módulo de biblioteca que también quieres depurar, se debe empaquetar esa biblioteca con `debuggable true` de manera que conserve sus símbolos de depuración. Para garantizar que las variantes depurables del proyecto de tu app reciban la variante depurable de un módulo de biblioteca, asegúrate de publicar versiones no predeterminadas de tu biblioteca.

Puedes iniciar una sesión de depuración de la siguiente manera:

- Establece algunas interrupciones en el código de la app (breakpoints)
- En la barra de herramientas, selecciona el dispositivo en el que depurarás la app en el menú desplegable del dispositivo de destino.

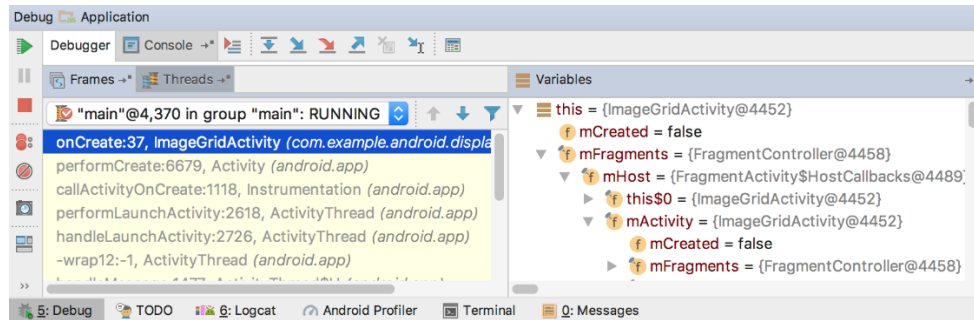


- En la barra de herramientas, haz clic en Debug

Si se muestra el diálogo "switch from Run to Debug", significa que tu app ya se está ejecutando en el dispositivo y se reiniciará para comenzar con la depuración. Si prefieres conservar la misma instancia de la app en ejecución, haz clic en Cancel Debug y, en su lugar, adjunta el depurador a la app en ejecución.

Android Studio compila un APK, lo firma con una clave de depuración, lo instala en el dispositivo seleccionado y lo ejecuta. Si agregas código C y C++ a tu proyecto, Android Studio también ejecutará el depurador LLDB en la ventana Debug para depurar tu código nativo.

- Si la ventana Debug no está abierta, selecciona View > Tool Windows > Debug (o haz clic en Debug en la barra de ventanas de herramientas) y, luego, haz clic en la pestaña Debugger



Si tu app ya se está ejecutando en el dispositivo, puedes comenzar con la depuración sin reiniciarla de la siguiente manera:

- Haz clic en Attach debugger to Android process .
- En el diálogo Choose Process, selecciona el proceso al que deseas adjuntar el depurador.

Si estás usando un emulador o un dispositivo con derechos de administrador, puedes marcar la opción Show all processes para ver todos los procesos.

En el menú desplegable Use Android Debugger Settings, puedes seleccionar una configuración de ejecución y depuración existente. (En proyectos con código C y C++, esto te permite reutilizar los directorios de símbolos y los comandos de arranque y posconexión de LLDB de una configuración existente). Si no tienes una configuración de ejecución y depuración, selecciona Create New. De esta forma, se habilita el menú desplegable Debug Type, donde puedes seleccionar un tipo de depuración diferente. De forma predeterminada, Android Studio usa el tipo de depuración Auto para seleccionar la mejor opción de depuración en función de si tus proyectos incluyen código Java o C/C++.

- Haz clic en OK.

Log de sistema

Para escribir mensajes de registro en tu código, usamos la clase Log. Estos mensajes te ayudan a comprender el flujo de ejecución mediante la recopilación de los resultados de depuración del sistema mientras interactúas con tu app. Además, pueden indicarte cuál es la parte de tu app que falló. Para obtener más información sobre el registro, consulta [Cómo escribir y ver registros](#).

En el siguiente ejemplo, se muestra cómo podemos agregar mensajes de registro para determinar si hay información disponible sobre el estado anterior cuando inicias tu actividad:

```
import android.util.Log;

...

public class MyActivity extends Activity {
    private static final String TAG =
MyActivity.class.getSimpleName();

    ...

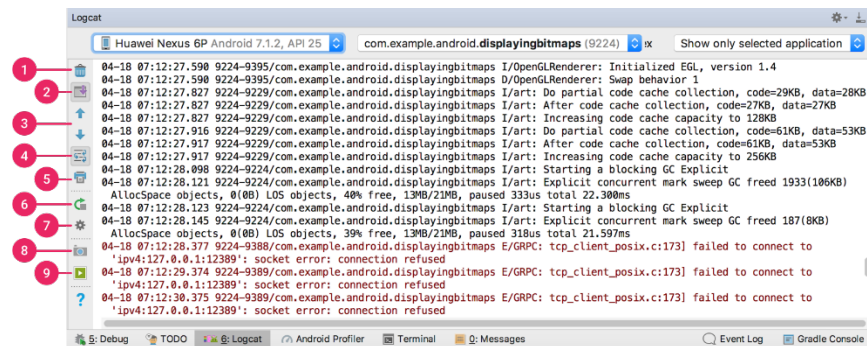
    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        if (savedInstanceState != null) {
            Log.d(TAG, "onCreate() Restoring previous state");
            /* restore state */
        } else {
            Log.d(TAG, "onCreate() No saved state available");
            /* initialize app */
        }
    }
}
```

Durante el desarrollo, el código también puede detectar excepciones y escribir el seguimiento de pila en el registro del sistema:

```
void someOtherMethod() {
    try {
        ...
    } catch (SomeException e) {
        Log.d(TAG, "someOtherMethod()", e);
    }
}
```

Podemos ver y filtrar la depuración y otros mensajes del sistema en la ventana Logcat. Por ejemplo, puedes ver mensajes cuando se produce la recolección de elementos no utilizados, o bien mensajes que agregas a tu app, con la clase Log.

Para usar Logcat, debes iniciar la depuración y seleccionar la pestaña Logcat en la barra de herramientas inferior.

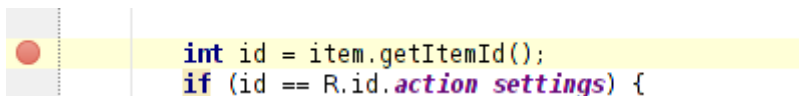


Breakpoints

Android Studio admite varios tipos de interrupciones que activan diferentes acciones de depuración. El tipo más común es una interrupción de línea que detiene la ejecución de tu app en una línea de código especificada. Mientras la app está detenida, puedes examinar variables, evaluar expresiones y, luego, continuar la ejecución línea por línea para determinar las causas de los errores en el tiempo de ejecución.

Para agregar una interrupción de línea, haz lo siguiente:

1. Localiza la línea de código en la que desees detener la ejecución, luego haz clic en el margen izquierdo de esta o coloca el símbolo de intercalación en la línea y presiona Control + F8 (en Mac, Command + F8).
2. Si tu app ya se está ejecutando, no necesitas actualizarla para agregar la interrupción. Simplemente, haz clic en Attach debugger to Android process . De lo contrario, inicia la depuración haciendo clic en Debug



Cuando la ejecución del código llega a la interrupción, Android Studio detiene la ejecución de tu app. Luego, puedes usar las herramientas de la pestaña Debugger para identificar el estado de la app:

- Para examinar el árbol de objetos de una variable, expándelo en la vista Variables. Si no ves la vista Variables, haz clic en Restore Variables View.
- Para evaluar una expresión en el punto de ejecución actual, haz clic en Evaluate Expression
- Para avanzar a la siguiente línea del código (sin ingresar a un método), haz clic en Step Over

- Para avanzar a la primera línea dentro de una llamada a un método, haz clic en Step Into
- Para avanzar a la siguiente línea fuera del método actual, haz clic en Step Out
- Para continuar ejecutando la app normalmente, haz clic en Resume Program

Si tienes experiencia previa con otros ambientes de desarrollo, notarás que las opciones de control y seguimiento del debug son similares a muchos ambientes comúnmente utilizados.

Para más información sobre el uso y configuración de las herramientas de debug, por favor consulta la documentación oficial:

<https://developer.android.com/studio/debug#java>

UNIDAD 3

Componentes de un proyecto Android

Las pantallas de un proyecto Android

Algunos de los componentes en Android que definen visibilidad, es decir, se traducen en representación de interfaz de usuario son listados a continuación:

Activity

La clase Activity es un componente clave de una app para Android, y la forma en que se inician y se crean las actividades es una parte fundamental del modelo de aplicación de la plataforma. A diferencia de los paradigmas de programación en los que las apps se inician con un método main(), el sistema Android inicia el código en una instancia de Activity invocando métodos de devolución de llamada específicos que corresponden a etapas específicas de su ciclo de vida.

La experiencia con las app para dispositivos móviles es diferente de la versión de escritorio, ya que la interacción del usuario con la app no siempre comienza en el mismo lugar. En este caso, no hay un lugar específico desde donde el usuario comienza su actividad. Por ejemplo, al abrir una app de correo electrónico desde la pantalla principal, es posible que veas una lista de correos electrónicos. Por el contrario, si usas una app de redes sociales que luego inicia tu app de correo electrónico, es posible que accedas directamente a la pantalla de la app de correo electrónico para redactar uno.

La clase Activity está diseñada para facilitar este paradigma. Cuando una app invoca a otra, la app que realiza la llamada invoca una activity en la otra, en lugar de a la app en sí. De esta manera, la actividad sirve como el punto de entrada para la interacción de una app con el usuario. Implementas una actividad como una subclase de la clase Activity.

Tal como lo estudiamos en la unidad anterior, una actividad permite a la ventana en la que la app dibuja su interfaz (UI). Por lo general, esta ventana llena la pantalla, pero puede ser más pequeña y flotar sobre otras ventanas. Generalmente, una actividad implementa una pantalla en una app. Por ejemplo, una actividad de una app puede implementar una pantalla Preferencias mientras otra implementa una pantalla Seleccionar foto.

La mayoría de las apps contienen varias pantallas, lo cual significa que incluyen varias actividades. Por lo general, una actividad en una app se especifica como la actividad principal (MainActivity en nuestro ejemplo), que es la primera pantalla que aparece cuando el usuario inicia la app. Luego, cada actividad puede iniciar otra actividad a fin de realizar diferentes acciones. Por ejemplo, la actividad principal de una app de correo electrónico simple podría proporcionar una pantalla en la que se muestra una casilla de correo electrónico. A partir de aquí, la actividad principal podría iniciar otras actividades que proporcionan pantallas para tareas como redactar correos y abrir correos electrónicos individuales.

Si bien las actividades trabajan en conjunto a fin de crear una experiencia del usuario coherente en una app, cada actividad se relaciona vagamente con otras actividades; por lo general, hay una pequeña cantidad de dependencias entre las actividades de una app. De hecho, estas suelen iniciar actividades que pertenecen a otras apps. Por ejemplo, una app de navegador podría iniciar la acción de "Compartir actividad" de una app de redes sociales.

Para utilizar una activity en una app, es necesario registrar información sobre cada activity en un archivo denominado Manifest (o manifiesto) y administrar los ciclos de vida de las actividades de manera apropiada (más sobre esto en la unidad 6)

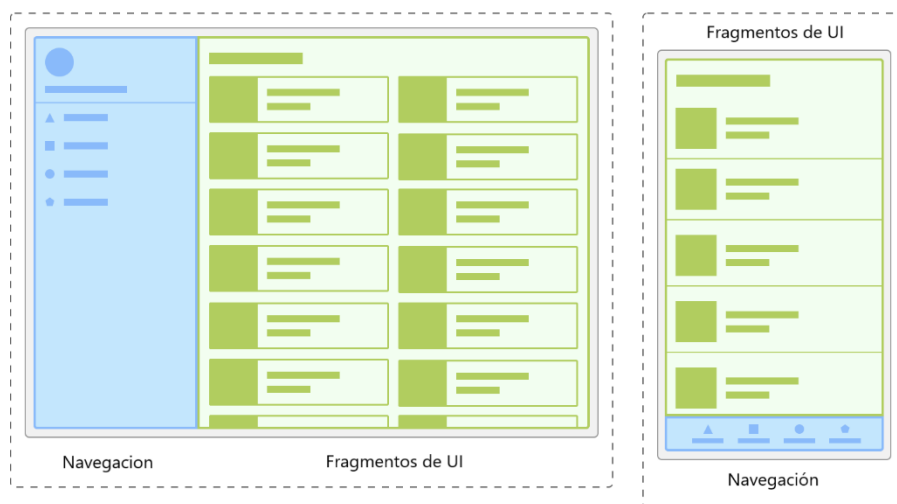
Fragment

Un Fragment representa una parte reutilizable de la IU. Un fragmento define y administra su propio diseño (layout), tiene su propio ciclo de vida y puede administrar sus propios eventos de entrada. Los fragmentos no pueden existir por sí solos, sino que deben estar alojados por una activity u otro fragmento. La jerarquía de vistas del fragmento forma parte de la jerarquía de vistas del host o está conectada a ella.

Nota: Los archivos de diseño (layout) de los fragmentos se ven exactamente iguales que los layouts presentados en la siguiente sección.

Los fragmentos introducen la modularidad y la capacidad de reutilización en la IU de una actividad, ya que te permiten dividir la IU en fragmentos separados. Las actividades son un lugar ideal para colocar elementos globales en la interfaz de usuario de tu app, como un panel lateral de navegación. En cambio, los fragmentos son más adecuados para definir y administrar la IU de una sola pantalla o de una parte de ella.

Tomemos como ejemplo una app que responda a varios tamaños de pantalla. En pantallas más grandes, la app debe mostrar un panel lateral de navegación estático y una lista en un diseño de cuadrícula. En pantallas más pequeñas, debe mostrar una barra de navegación inferior y una lista en un diseño lineal. Administrar todas esas variaciones en la actividad puede resultar complejo. Separar los elementos de navegación del contenido puede hacer que ese proceso sea más fácil de manejar. La actividad se encarga de mostrar la IU de navegación correcta, mientras que el fragmento muestra la lista con el diseño adecuado.



Dividir la IU en fragmentos facilita la modificación de la apariencia de la actividad durante el tiempo de ejecución. Mientras la actividad está en el estado de ciclo de vida STARTED o en uno

superior, podemos agregar, reemplazar o quitar fragmentos. Puedes mantener un registro de esos cambios en una pila de actividades administrada por la actividad, lo que permite que se reviertan los cambios.

Podemos usar varias instancias de la misma clase de fragmento dentro de la misma actividad, en varias actividades o incluso como elemento secundario de otro fragmento. Teniendo esto en consideración, solo debemos proporcionar un fragmento con la lógica necesaria para administrar su propia IU. Debemos evitar que un fragmento dependa de otro, así como la manipulación del fragmento desde otro.

Layout / Diseño

Un layout define la estructura de una interfaz de usuario en tu aplicación, por ejemplo, en una actividad. Todos los elementos del layout se crean usando una jerarquía de objetos View y ViewGroup. Una View suele mostrar un elemento que el usuario puede ver y con el que puede interactuar. En cambio, un ViewGroup es un contenedor invisible que define la estructura de diseño de View y otros objetos ViewGroup.

Los objetos View se denominan "widgets" y pueden ser una de muchas subclases, como Button o TextView. Los objetos ViewGroup se denominan "diseños" y pueden ser uno de los muchos tipos que proporcionan una estructura de diseño diferente, como LinearLayout o ConstraintLayout .

Podemos declarar un layout de dos formas:

- Declarar los elementos de UI en el XML: Android proporciona un vocabulario XML simple que coincide con las clases y subclases de vistas, como las que se usan para widgets y diseños. Podemos definir como se verá el layout escribiendo la definición de estos elementos en los archivos XML. Existe en Android Studio también un editor gráfico que permite usar una herramienta "drag and drop" para la creación de los elementos de diseño.
- Crear una instancia de elementos de diseño en tiempo de ejecución: La aplicación puede crear los elementos gráficos en tiempo de ejecución, es decir, al estar funcionando la app. Los elementos pueden ser instanciados y sus características y comportamiento establecidos usando el lenguaje de programación Java y usando la API.

Declarar la IU en XML permite separar la presentación de la app del código que controla su comportamiento. El uso de archivos XML también facilita la creación de distintos diseños para diferentes tamaños de pantalla y orientaciones.

Al usar vocabulario XML de Android, se pueden crear rápidamente diseños de IU y de los elementos de pantalla que contienen, de la misma manera que creas páginas web en HTML, con una serie de elementos anidados.

Cada archivo de diseño debe contener exactamente un elemento raíz, que debe ser un objeto View o ViewGroup. Una vez que definido el elemento raíz, podemos agregar widgets u objetos de diseño adicionales como elementos secundarios para crear gradualmente una jerarquía de vistas que defina el diseño. Por ejemplo, a continuación, se muestra un diseño XML que usa un LinearLayout vertical para incluir una TextView y un Button

```
<?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" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a TextView" />
    <Button android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, I am a Button" />
</LinearLayout>
```

Después de declarar el diseño en XML, guardamos el archivo con la extensión .xml en el directorio res/layout/ de tu proyecto de Android para que pueda compilarse correctamente.

Cuando compilamos la aplicación, cada archivo XML de diseño se compila en un recurso View. Debemos cargar el recurso de diseño desde el código de la aplicación, en la implementación de devolución de llamada Activity.onCreate(). Para eso, llamar a setContentView() pasando la referencia a tu recurso de diseño en forma de R.layout.layout_file_name. Por ejemplo, si el diseño XML se guarda como main_layout.xml, los cargaremos para una actividad de la siguiente manera:

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main_layout);
}
```

Atributos

Cada objeto View y ViewGroup admite su propia variedad de atributos XML. Algunos atributos son específicos de un objeto View (por ejemplo, TextView admite el atributo textSize), aunque estos atributos también son heredados por cualquier objeto View que pueda extender esta clase.

Algunos son comunes para todos los objetos View, porque se heredan de la clase raíz View (como el atributo id). Además, otros atributos se consideran "parámetros de diseño", que son atributos que describen ciertas orientaciones de diseño de View, tal como lo define el objeto ViewGroup superior de ese objeto.

Cualquier objeto View puede tener un ID entero asociado para identificarse de forma única dentro del árbol. Cuando se compila la aplicación, se hace referencia a este ID como un número entero, pero normalmente se asigna el ID en el archivo XML de diseño como una string del atributo id. Este es un atributo XML común para todos los objetos View (definido por la clase View) y lo utilizarás muy a menudo. La sintaxis de un ID dentro de una etiqueta XML es la siguiente:

```
android:id="@+id/my_button"
```

El símbolo arroba (@) al comienzo de la string indica que el analizador de XML debe analizar y expandir el resto de la string de ID, y luego identificarla como un recurso de ID. El símbolo más (+) significa que es un nuevo nombre de recurso que se debe crear y agregar a nuestros recursos (en el archivo R.java). El framework de Android ofrece otros recursos de ID. Al hacer referencia a un ID de recurso de Android, no necesitamos el símbolo más, pero debemos agregar el espacio de nombres de paquete android de la siguiente manera:

```
android:id="@android:id/empty"
```

Con el espacio de nombres de paquete Android establecido, se hace referencia a un ID de la clase de recursos android.R, en lugar de la clase de recursos local.

Para crear vistas y hacer referencia a ellas desde la aplicación, puedes seguir este patrón común:

1. Definir una vista o un widget en el archivo de diseño y asignarle un ID único:

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

2. Luego, crear una instancia del objeto View y capturarla desde el diseño (generalmente en el método onCreate()):

```
Button myButton = (Button) findViewById(R.id.my_button);
```

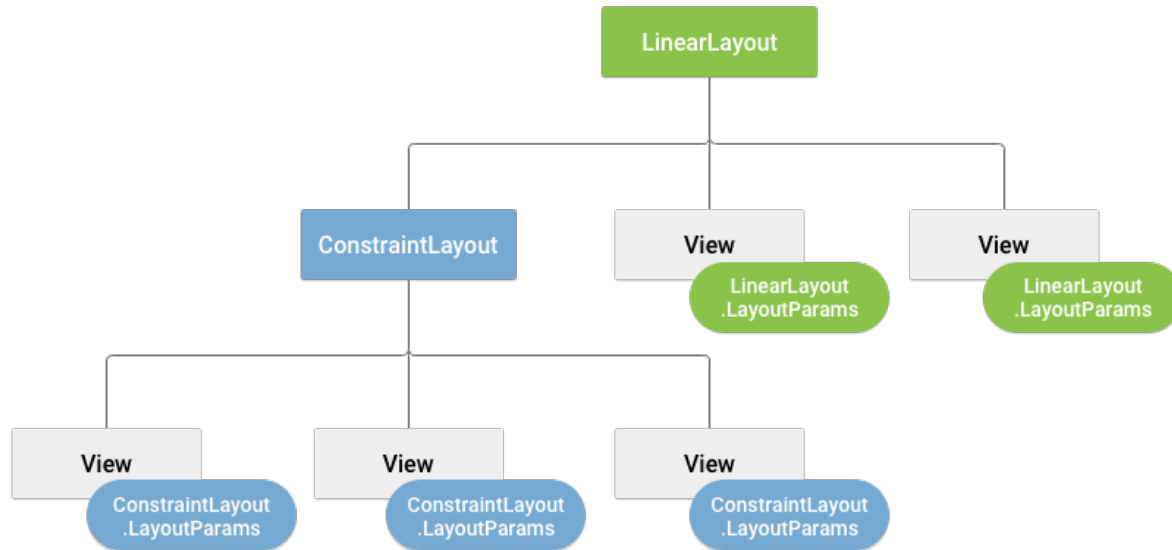
Definir ID para objetos View es importante cuando se crea un RelativeLayout. En un diseño relativo, las vistas del mismo nivel pueden definir su diseño en función de otra vista del mismo nivel, que se identifica con un ID único.

No es necesario que un ID sea único en todo el árbol, pero debe ser único dentro de la parte del árbol en la que estamos buscando (que comúnmente puede ser el árbol completo, por lo que es mejor que, en lo posible, sea siempre único).

Parámetros

Los atributos de diseño XML denominados *layout_something* definen parámetros de diseño para el objeto View que son adecuados para el objeto ViewGroup en el que reside.

Cada clase ViewGroup implementa una clase anidada que extiende ViewGroup.LayoutParams. Esta subclase contiene tipos de propiedad que definen el tamaño y la posición de cada vista secundaria, según resulte apropiado para el grupo de vistas. Como se muestra en la Figura 2, el grupo de vistas superior define parámetros de diseño para cada vista secundaria (incluido el grupo de vistas secundario).



Debemos tener en cuenta que cada subclase LayoutParams tiene su propia sintaxis para configurar valores. Cada elemento secundario debe definir LayoutParams adecuados para su elemento superior, aunque también puede definir diferentes LayoutParams para sus propios elementos secundarios.

Todos los grupos de vistas incluyen un ancho y una altura (*layout_width* y *layout_height*), y cada vista debe definirlos. Muchos LayoutParams también incluyen márgenes y bordes opcionales.

Podemos especificar el ancho y la altura con medidas exactas, aunque no es recomendable hacerlo frecuentemente. Generalmente, se usa una de estas constantes para establecer el ancho o la altura:

- `Wrap_content`: le indica a tu vista que modifique su tamaño conforme a las dimensiones que requiere su contenido
- `Match_parent`: le indica a tu vista que se agrande tanto como lo permita su grupo de vistas superior.

En general, no se recomienda especificar el ancho ni la altura de un layout con unidades absolutas como píxeles. En cambio, el uso de medidas relativas, como unidades de píxeles independientes de la densidad (dp), `wrap_content` o `match_parent`, es un mejor enfoque, ya que ayuda a garantizar que la app se visualice correctamente en distintos tamaños de pantalla de dispositivos.

Posición

La geometría de una vista es la de un rectángulo. Una vista tiene una ubicación, expresada como un par de coordenadas izquierda y superior, y dos dimensiones, expresadas como un ancho y una altura. La unidad para la ubicación y las dimensiones es el píxel.

Es posible recuperar la ubicación de una vista al invocar los métodos `getLeft()` y `getTop()`. El primero muestra la coordenada izquierda, o X, del rectángulo que representa la vista. El segundo muestra la coordenada superior, o Y, del rectángulo que representa la vista. Ambos métodos muestran la ubicación de la vista respecto al elemento superior. Por ejemplo, cuando `getLeft()` muestra 20, significa que la vista se encuentra a 20 píxeles a la derecha del borde izquierdo de su elemento superior directo.

Además, se ofrecen varios métodos convenientes para evitar cálculos innecesarios: `getRight()` y `getBottom()`. Estos métodos muestran las coordenadas de los bordes inferior y derecho del rectángulo que representa la vista. Por ejemplo, llamar a `getRight()` es similar al siguiente cálculo: `getLeft() + getWidth()`.

Layout comunes

Cada subclase de la clase `ViewGroup` proporciona una manera única de mostrar las vistas que anidas en ella. A continuación, se muestran algunos de los tipos de diseño más comunes integrados en la plataforma Android.

- **Layout Lineal**: Un layout que organiza sus elementos secundarios en una sola fila horizontal o vertical. Si la longitud de la ventana supera la de la pantalla, crea una barra de desplazamiento. Podemos especificar la dirección del diseño con el atributo *`android:orientation`*.



Todos los elementos secundarios de un LinearLayout se apilan uno detrás de otro, por lo cual una lista vertical solo tendrá un elemento secundario por fila, independientemente del ancho que tengan, y una lista horizontal solo tendrá la altura de una fila (la altura del elemento secundario más alto, más el padding). Un LinearLayout respeta los márgenes entre los elementos secundarios y la gravedad (alineación a la derecha, centrada o a la izquierda) de cada elemento secundario.

- Layout Relativo: permite especificar la ubicación de los objetos secundarios en función de ellos mismos (el objeto secundario A a la izquierda del objeto secundario B) o en función del elemento primario (alineado con la parte superior del elemento primario).

RelativeLayout es un grupo de vistas que muestra vistas secundarias en posiciones relativas. La posición de cada vista puede especificarse como relativa a elementos del mismo nivel (como a la izquierda de otra vista o por debajo de ella) o en posiciones relativas al área RelativeLayout superior (como alineada a la parte inferior, izquierda o central)



RelativeLayout es una utilidad muy eficaz para diseñar una interfaz de usuario porque puede eliminar grupos de vistas anidados y conservar la estabilidad de la jerarquía de diseño, lo que mejora el rendimiento. Comúnmente el usar muchos LinearLayout anidados implica que quizás puedan reemplazarlos por un grupo RelativeLayout individual.

RelativeLayout permite que vistas secundarias especifiquen su posición relativa a la vista superior o entre sí (especificada por ID), a continuación algunos ejemplos:

- android:layout_alignParentTop: si el valor es "true", el borde superior de esta vista coincidirá con el del elemento superior

- `android:layout_centerVertical`: si el valor es "true", centra este elemento secundario en posición vertical dentro de su elemento superior
- `android:layout_below`: posiciona el borde superior de esta vista debajo de la vista especificada con un ID de recurso
- `android:layout_toRightOf`: posiciona el borde izquierdo de esta vista a la derecha de la vista especificada con un ID de recurso

A continuación, se incluye un ejemplo de archivo layout:

```
<?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="match_parent"
    android:paddingLeft="16dp"
    android:paddingRight="16dp" >
    <EditText
        android:id="@+id/name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/reminder" />
    <Spinner
        android:id="@+id/dates"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentLeft="true"
        android:layout_toLeftOf="@+id/times" />
    <Spinner
        android:id="@+id/times"
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/name"
        android:layout_alignParentRight="true" />
    <Button
        android:layout_width="96dp"
        android:layout_height="wrap_content"
        android:layout_below="@id/times"
        android:layout_alignParentRight="true"
        android:text="@string/done" />
</RelativeLayout>
```

Observar en este ejemplo como se ha establecido los diferentes objetos view y la forma como se relacionan entre sí.

Constraint Layout

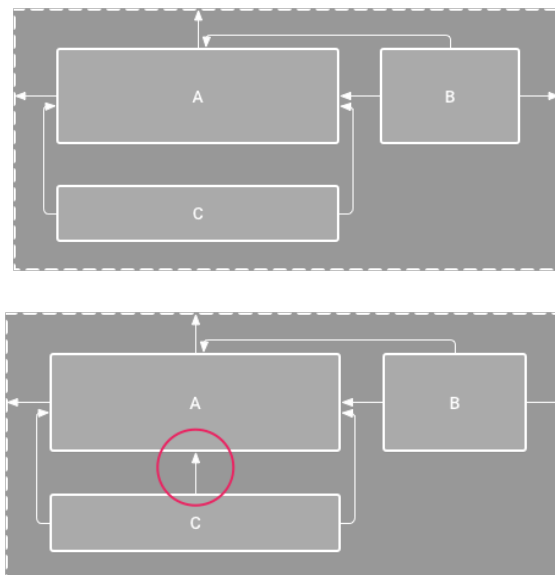
ConstraintLayout permite crear diseños grandes y complejos con una jerarquía de vistas plana (sin grupos de vistas anidadas). Es similar a RelativeLayout en cuanto a que se presentan todas las vistas de acuerdo con las relaciones entre las vistas del mismo nivel y el diseño de nivel superior, pero es más flexible que RelativeLayout y más fácil de usar con el editor de diseño de Android Studio.

Toda la potencia de ConstraintLayout está disponible directamente desde las herramientas visuales del editor de diseño, ya que la API de diseño y el editor de diseño se crearon específicamente para funcionar en conjunto. Así, puedes crear todo tu diseño con ConstraintLayout arrastrando y soltando elementos, en lugar de editar el XML.

Para definir la posición de una vista en ConstraintLayout, debemos agregar al menos una restricción horizontal y una vertical. Cada restricción representa una conexión o alineación con otra vista, el diseño de nivel superior o una guía invisible. Cada restricción define la posición de la vista a lo largo del eje vertical u horizontal, por lo que cada vista debe tener un mínimo de una restricción para cada eje, aunque generalmente se necesitan más.

Al soltar una vista en el editor de diseño, esta permanece donde la dejas, incluso si no tiene restricciones. Sin embargo, esto solo sirve para facilitar la edición. Si una vista no tiene restricciones cuando ejecutas su diseño en un dispositivo, se abre en la posición [0,0] (la esquina superior izquierda).

En la siguiente figura, el diseño se ve bien en el editor, pero no hay restricción vertical en la vista C. Cuando este diseño se abre en un dispositivo, la vista C se alinea horizontalmente con los bordes izquierdo y derecho de la vista A, pero aparece en la parte superior de la pantalla porque no tiene restricción vertical.



Para corregir esto se debe incorporar una restricción vertical en la vista C, tal como se ve en la figura anterior.

Si bien la ausencia de restricciones no causa errores de compilación, el editor de diseño mostrará un error en la barra de herramientas. Para ver los errores y otras advertencias, haz clic en Show Warnings and Errors. Para evitar que falten restricciones, el editor de diseño puede agregar restricciones automáticamente con las funciones Infer Constraints y Autoconnect.

Para usar ConstraintLayout en un proyecto, debemos seguir estos pasos:

1. Asegurarnos de tener el repositorio maven.google.com declarado en el archivo build.gradle de nivel de módulo

```
repositories {  
    google()  
}
```

2. Agregar la biblioteca como una dependencia en el mismo archivo build.gradle, como se muestra en el siguiente ejemplo:

```
dependencies {  
    implementation "androidx.constraintlayout:constraintlayout:  
2.0.2"  
}
```

3. En la barra de herramientas o notificación de sincronización, dar clic en “Sync Project with Gradle Files”.

Principales componentes de interfaz para aplicaciones nativas Android (Material Design)

A continuación, detallamos algunos de los principales componentes que deberíamos conocer para comenzar a desarrollar, nos enfocaremos en Material Design. Para mayor información se recomienda consultar la documentación disponible en:

<https://developer.android.com/guide>

Material Design es una guía completa para el diseño visual, interactivo y de movimiento en plataformas y dispositivos. A fin de usar material design en nuestras apps para Android, es necesario seguir las pautas definidas en su especificación y usar los nuevos componentes y estilos disponibles en la biblioteca de compatibilidad de material design.

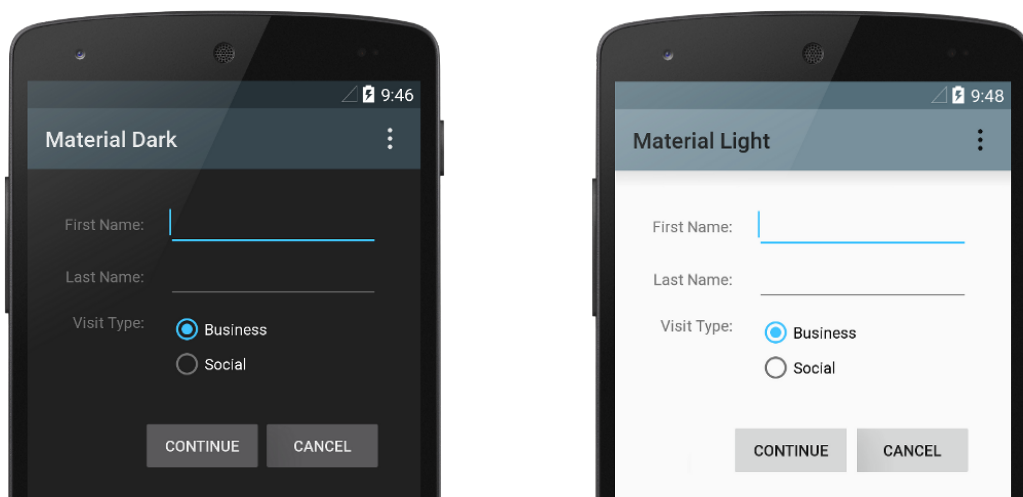
Para utilizar algunos de los elementos de Material Design debemos incluir la dependencia en nuestra aplicación en el archivo build.gradle

```
implementation 'com.google.android.material:material:1.2.1'
```

Android ofrece las siguientes funciones para ayudar a crear apps de material design:

- Un tema de app de material design para diseñar con widgets de IU
- Widgets para vistas complejas, como listas y tarjetas
- Nuevas API para sombras y animaciones personalizadas

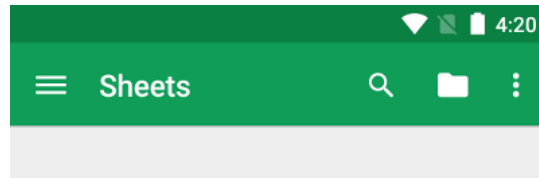
Para aprovechar las características de los materiales, como el estilo de los widgets de IU estándar, y a fin de optimizar la definición de estilo de las app, podemos aplicar un tema basado en material a tu app, por ejemplo:



App Bar

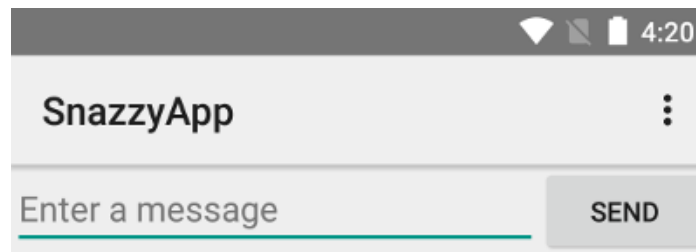
La barra de la app, también conocida como barra de acciones, es uno de los elementos de diseño más importantes de las actividades de una app, ya que proporciona una estructura visual y elementos interactivos que los usuarios conocen. El uso de esta barra permite la alineación de la app con otras apps para Android. De esta manera, los usuarios pueden comprender fácilmente cómo operar la aplicación y tener una buena experiencia. Las principales funciones de la barra de la app son las siguientes:

- Un espacio exclusivo para proporcionarle una identidad a una app y para indicar la ubicación del usuario en ella
- Acceso a acciones importantes de forma predecible; por ejemplo, la búsqueda
- Compatibilidad con navegación y cambios de vistas (con pestañas o listas desplegables)



Esta clase describe la manera de usar el widget `AppBar` de la biblioteca de compatibilidad v7 `appcompat` como una barra de la app. Existen otras maneras de implementar una barra de la app (por ejemplo, algunos temas establecen un elemento `ActionBar` como barra de la app de forma predeterminada), pero el uso del widget `AppBar` `appcompat` facilita la configuración de una barra de la app que funcione en la gran mayoría de los dispositivos y también te permite personalizar tu barra de la app posteriormente, a medida que desarrolles tu app.

Para configurar la app bar, en su forma más básica, la barra de acciones muestra el título de la actividad de un lado y un menú ampliado del otro. Incluso en esta forma sencilla, la barra de app proporciona información útil a los usuarios y ayuda a dar una apariencia coherente a las apps de Android.



A partir de Android 3.0 (API nivel 11), todas las actividades que usan el tema predeterminado tienen un elemento `ActionBar` como barra de la app. No obstante, se fueron agregando funciones de la barra de la app gradualmente al elemento `ActionBar` nativo en diferentes versiones de Android. Como consecuencia, el elemento `ActionBar` nativo se comporta de manera diferente según la versión del sistema Android que use un dispositivo. Por el contrario, las funciones más recientes se agregan a la versión de `AppBar` de la biblioteca de compatibilidad y están disponibles en todos los dispositivos que puedan usar la biblioteca de compatibilidad.

Por este motivo, debemos usar la clase `AppBar` de la biblioteca de compatibilidad para implementar las barras de la app de nuestras actividades. El uso de la barra de herramientas de la biblioteca de compatibilidad ayuda a garantizar que nuestras apps tenga un comportamiento coherente en la mayor cantidad de dispositivos posible. Por ejemplo, el widget `AppBar` proporciona una experiencia de material design en dispositivos con Android 2.1 (API nivel 7) o versiones posteriores, pero la barra de acciones nativa no admite material design a menos que el dispositivo tenga Android 5.0 (API nivel 21) o versiones posteriores.

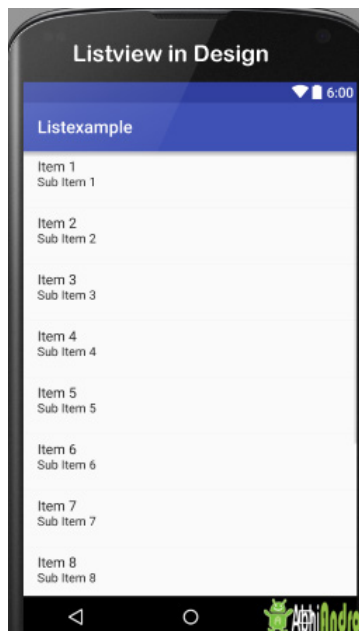
Listas

Un ListView en Android es un objeto vista que nos permite mostrar una lista de elementos desplazables. Nos ayuda a desplegar datos en pantalla en la forma de una lista desplazable. Los usuarios pueden seleccionar cualquier ítem desde la lista haciendo click en el.

Las vistas de lista ListView son ampliamente usadas en las aplicaciones Android, un ejemplo común es una lista de contactos donde tenemos una lista de información de personas y sus números de teléfonos.

A continuación, un ejemplo de código XML para dibujar una listView:

```
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/simpleListView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:context="abhiandroid.com.listexample.MainActivity">
</ListView>
```



Tarjetas

Con frecuencia, las apps deben mostrar datos en contenedores con un estilo similar. Estos contenedores se usan a menudo en listas para retener la información de cada elemento. El sistema proporciona la API de CardView como una manera sencilla de mostrar información en tarjetas que tienen un aspecto coherente en la plataforma. Estas tarjetas tienen una elevación predeterminada por encima del grupo de vistas que las contiene, de modo que el sistema dibuja sombras debajo de ellas. Las tarjetas ofrecen una forma simple de incluir un grupo de vistas y, al mismo tiempo, proporcionar un estilo coherente para el contenedor.

El widget CardView forma parte de AndroidX. Para usarlo en el proyecto, agregamos la siguiente dependencia al archivo build.gradle del módulo de tu app:

```
dependencies {
    implementation "androidx.cardview:cardview:1.0.0"
}
```

Para usar `CardView`, necesitamos agregarlo a tu archivo de diseño. Úsalo como grupo de vistas para incluir otras vistas. En este ejemplo, `CardView` incluye un objeto `TextView` individual que muestra algo de información al usuario.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    ... >
    <!-- A CardView that contains a TextView -->
    <androidx.cardview.widget.CardView
        xmlns:card_view="http://schemas.android.com/apk/res-auto"
        android:id="@+id/card_view"
        android:layout_gravity="center"
        android:layout_width="200dp"
        android:layout_height="200dp"
        card_view:cardCornerRadius="4dp">

        <TextView
            android:id="@+id/info_text"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </androidx.cardview.widget.CardView>
</LinearLayout>
```

Las tarjetas se dibujan en la pantalla con una elevación predeterminada, y esto hace que el sistema dibuje una sombra debajo de ellas. Podemos proporcionar una elevación personalizada para una tarjeta con el atributo `card_view:cardElevation`. De esa manera, se dibujará una sombra más pronunciada si eliges una elevación más grande, y una sombra más suave si optas por una elevación más baja. `CardView` usa la elevación real y sombras dinámicas de Android 5.0 (Nivel de API 21) y versiones posteriores, y regresa a la implementación de una sombra programática en las versiones anteriores.

Podemos usar las siguientes propiedades para personalizar la apariencia del widget CardView:

- Para definir el radio de la esquina de tus diseños, usa el atributo `card_view:cardCornerRadius`.
- Para definir el radio de la esquina de tu código, usa el método `CardView.setRadius`.
- Para definir el color de fondo de una tarjeta, usa el atributo `card_view:cardBackgroundColor`.

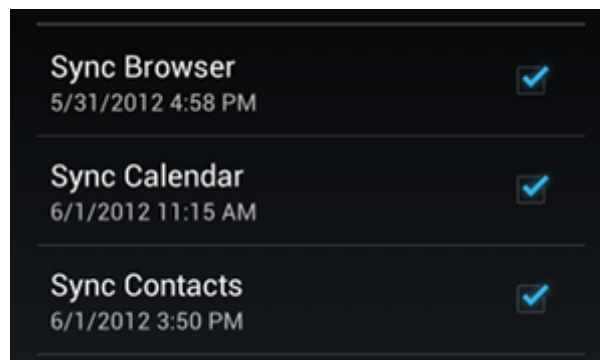
Para más información, por favor consultar la documentación oficial en:

<https://developer.android.com/reference/androidx/cardview/widget/CardView?hl=es>

Selection controls

Material design permite el uso de tres tipos de controles de selección (Selection Controls): Checkboxes para selección múltiples, Radio Buttons para selección única, Switches para selección si/no.

1. Checkboxes permiten la selección al usuario de uno o más opciones desde un conjunto. Típicamente se presenta en una lista vertical.



Para crear un opciones checkbox, crear un elemento *checkbox* en el layout. Ya que cada checkbox permite la elección de múltiples elementos, cada checkbox se maneja independiente y de manera separada, se hace necesario registrar un *listener* por cada uno.

Cuando el usuario selecciona un checkbox este recibe una llamada a su evento *on-click*. Para definir el manejador (handler) del checkbox, agregar el atributo `android:onClick` al elemento en el layout xml. Debe existir un método con este nombre en el activity que se ejecuta.

Ejemplo:

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

```

        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <CheckBox android:id="@+id/checkbox_meat"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/meat"
            android:onClick="onCheckboxClicked"/>
        <CheckBox android:id="@+id/checkbox_cheese"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/cheese"
            android:onClick="onCheckboxClicked"/>
    </LinearLayout>

```

Y en la activity:

```

public void onCheckboxClicked(View view) {

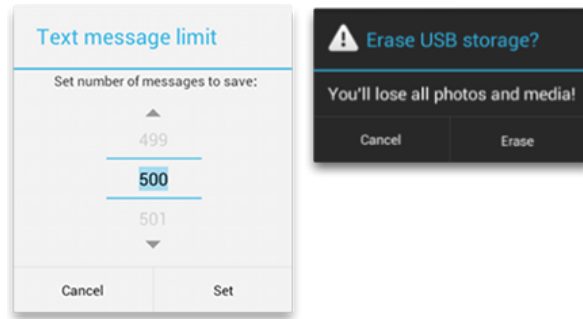
    boolean checked = ((CheckBox) view).isChecked();

    switch(view.getId()) {
        case R.id.checkbox_meat:
            if (checked)
                // hacer algo
            else
                // hacer otra cosa
            break;
        case R.id.checkbox_cheese:
            if (checked)
                // algo más
            else
                // otra cosa más
            break;
    }
}

```

Dialogs

Un diálogo es una ventana pequeña que le indica al usuario que debe tomar una decisión o ingresar información adicional. Un diálogo no ocupa toda la pantalla y, generalmente, se usa para eventos modales que requieren que los usuarios realicen alguna acción para poder continuar.



La clase Dialog es la clase de base para los diálogos, pero debemos evitar crear instancias de Dialog directamente. En su lugar, usamos una de las siguientes subclases

- AlertDialog: un diálogo que puede mostrar un título, hasta tres botones, una lista de elementos seleccionables o un diseño personalizado.
- DatePickerDialog o TimePickerDialog: un diálogo con una IU predefinida que le permite al usuario seleccionar una fecha o una hora

Estas clases definen el estilo y la estructura de un diálogo, pero debemos utilizar un DialogFragment como contenedor. La clase DialogFragment proporciona todos los controles que necesitamos para crear un diálogo y gestionar su apariencia, en lugar de llamar a los métodos del objeto Dialog.

El uso de DialogFragment para administrar el diálogo garantiza que aborde correctamente eventos de ciclo de vida, por ejemplo, cuando el usuario presiona el botón Atrás o gira la pantalla. La clase DialogFragment también permite reutilizar la IU del diálogo como componente integrable a una IU más grande, al igual que un Fragment tradicional (por ejemplo, cuando necesitamos que la IU del diálogo se vea diferente en pantallas grandes y pequeñas)

Por ejemplo, a continuación, hay un AlertDialog básico administrado dentro de un DialogFragment:

```
public class FireMissilesDialogFragment extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // usar la clase Builder
        AlertDialog.Builder builder = new
        AlertDialog.Builder(getActivity());
        builder.setMessage(R.string.dialog_fire_missiles)
            .setPositiveButton(R.string.fire, new
        DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id)
            {
                // Hacer algo
            }
        })
    }
}
```

```

        .setNegativeButton(R.string.cancel, new
DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int id)
{
                // El usuario canceló el dialogo
            }
        });
        return builder.create();
    }
}

```

Bottom Navigation

Bottom Navigation es un elemento similar al App Bar ya mencionado, corresponde a una vista que se agrega al layout para lograr un elemento similar pero, como el nombre lo indica, generar una menú la sección inferior del diseño.

El contenido del menú puede ser poblado especificando un *archivo de recurso*. Por ejemplo:

```

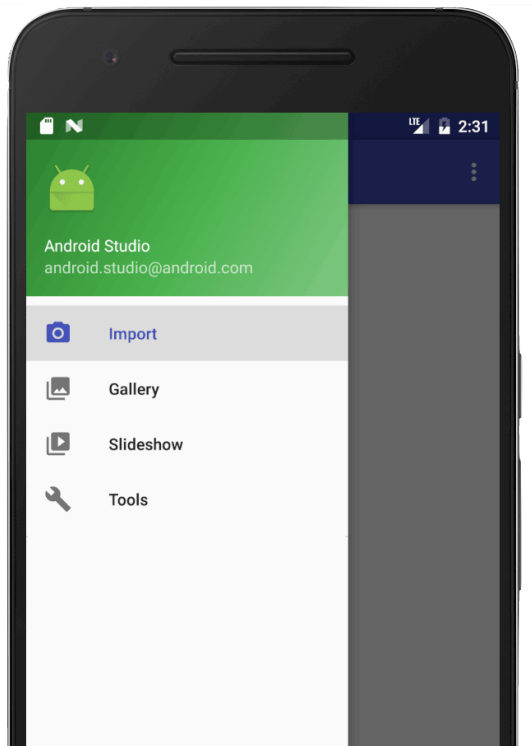
layout resource file:
<com.google.android.material.bottomnavigation.BottomNavigationView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res/res-auto"
    android:id="@+id/navigation"
    android:layout_width="match_parent"
    android:layout_height="56dp"
    android:layout_gravity="start"
    app:menu="@menu/my_navigation_items" />

res/menu/my_navigation_items.xml:
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/action_search"
        android:title="@string/menu_search"
        android:icon="@drawable/ic_search" />
    <item android:id="@+id/action_settings"
        android:title="@string/menu_settings"
        android:icon="@drawable/ic_add" />
    <item android:id="@+id/action_navigation"
        android:title="@string/menu_navigation"
        android:icon="@drawable/ic_action_navigation_menu" />
</menu>

```

Navigation Drawer

Navigation Drawer es un elemento de interfaz definido por Material Design consistente en el típico menú lateral deslizante desde la izquierda y utilizado ampliamente como el principal elemento de navegación de las aplicaciones móviles. Suele encontrarse en la pantalla principal de la app y contar con un botón para desplegarlo aunque también puede ser abierto deslizando el contenido de la pantalla desde el extremo izquierdo. A veces se suele complementar con el menú inferior Bottom Navigation.



Por ejemplo, en el siguiente diseño, se utiliza un DrawerLayout con dos vistas secundarias: un NavHostFragment que abarca el contenido principal y un NavigationView para el contenido del panel lateral de navegación

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Use DrawerLayout as root container for activity -->
<androidx.drawerlayout.widget.DrawerLayout xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">

    <!-- Layout to contain contents of main body of screen (drawer
will slide over this) -->
    <androidx.fragment.app.FragmentContainerView
```

```

        android:name="androidx.navigation.fragment.NavHostFragment"
        android:id="@+id/nav_host_fragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:defaultNavHost="true"
        app:navGraph="@navigation/nav_graph" />

        <!-- Container for contents of drawer - use NavigationView to make
configuration easier -->
        <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true" />

</androidx.drawerlayout.widget.DrawerLayout>

```

Luego, conectando el DrawerLayout al gráfico de navegación. Con ese fin, lo pasamos a AppBarConfiguration, como se muestra en el siguiente ejemplo:

```

AppBarConfiguration appBarConfiguration =
    new AppBarConfiguration.Builder(navController.getGraph())
        .setDrawerLayout(drawerLayout)
        .build();

```

Luego, en la clase de actividad principal, llamar a `setupWithNavController()` desde el método `onCreate()` de la actividad principal, como se muestra a continuación:

```

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

    ...

    NavHostFragment navHostFragment =
supportFragmentManager.findFragmentById(R.id.nav_host_fragment);
    NavController navController = navHostFragment.getNavController();
    NavigationView navView = findViewById(R.id.nav_view);
    NavigationUI.setupWithNavController(navView, navController);
}

```

TextFields

TextFields nos permite establecer campos de estilo formulario para que el usuario ingrese texto. Un TextField se compone de un TextInputLayout y un TextInputEditText como hijo. Es posible utilizar EditText, pero TextInputEditText provee garantías de accesibilidad para el ingreso de texto y mayor control sobre los aspectos visuales.

A continuación, se incluye el XML para establecer un TextField de ejemplo:

```
<com.google.android.material.textfield.TextInputLayout  
    android:id="@+id/textField"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:hint="@string/label">  
  
    <com.google.android.material.textfield.TextInputEditText  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
    />  
  
</com.google.android.material.textfield.TextInputLayout>
```

Progress Indicators

Corresponde a una vista que permite informar o indicar al usuario sobre el estado de una operación en ejecución, tal como cargar una app, enviar un formulario o actualizar cambios. Comunican el estado de una app e indican acciones disponibles, tal como la posibilidad de navegar a otras secciones de la app, fuera de la pantalla actual.

Existen algunas alternativas que podemos utilizar, dentro de las cuales se incluyen:

- Lineal y circular: barra recta o círculo, ambos con animaciones que indican el estado “cargando”
- Determinado o indeterminado: dependiendo de si la animación corresponderá o no con el estado de porcentaje de avance de la operación en curso. Determinado anima el elemento para corresponder exactamente con el % de avance. Indeterminado anima el elemento independiente del estado de avance, mostrando la animación como un loop.

A continuación, se incluye un ejemplo de como agregar un indicador determinado a un diseño:

```

<!-- Linear progress indicator -->
<com.google.android.material.progressindicator.LinearProgressIndicator
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
<!-- Circular progress indicator -->
<com.google.android.material.progressindicator.CircularProgressIndicator
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

```

Para un indicador indeterminado, el código es:

```

<!-- Linear progress indicator -->
<com.google.android.material.progressindicator.LinearProgressIndicator
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:indeterminate="true" />
<!-- Circular progress indicator -->
<com.google.android.material.progressindicator.CircularProgressIndicator
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:indeterminate="true" />

```

Interacciones de usuario

A continuación, detallaremos algunos de los elementos que permiten interacción con el usuario.

View Bindings

La vinculación de vista (View Binding) es una función que permite escribir más fácilmente código que interactúa con las vistas. Una vez que la vinculación de vista está habilitada en un módulo, genera una clase de vinculación para cada archivo de diseño XML presente en ese módulo. Una instancia de una clase de vinculación contiene referencias directas a todas las vistas que tienen un ID en el diseño correspondiente.

En la mayoría de los casos, la vinculación de vistas reemplaza a `findViewById`.

La vinculación de vista se habilita módulo por módulo. Para habilitar la vinculación de vista en un módulo, agregamos el elemento `viewBinding` a su archivo `build.gradle`, como se muestra en el siguiente ejemplo:

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

Si deseamos que se ignore un archivo de diseño mientras se generan clases de vinculación, agregamos el atributo `tools:viewBindingIgnore="true"` a la vista raíz de ese archivo:

```
<LinearLayout  
    ...  
    tools:viewBindingIgnore="true" >  
    ...  
</LinearLayout>
```

Si se habilita la vinculación de vista para un módulo, se genera una clase de vinculación para cada archivo de diseño XML que contiene el módulo. Cada clase de vinculación contiene referencias a la vista raíz y a todas las vistas que tienen un ID. El nombre de la clase de vinculación se genera convirtiendo el nombre del archivo XML según la convención de mayúsculas y minúsculas, y agregando la palabra "Binding" al final.

Por ejemplo, en un archivo de diseño llamado `result_profile.xml`:

```
<LinearLayout ... >  
    <TextView android:id="@+id/name" />  
    <ImageView android:cropToPadding="true" />  
    <Button android:id="@+id/button"  
        android:background="@drawable/rounded_button" />  
</LinearLayout>
```

La clase de vinculación generada se llama `ResultProfileBinding`. Esta clase tiene dos campos: un `TextView` llamado `name` y un `Button` llamado `button`. El campo `ImageView` del diseño no tiene ningún ID, por lo que no se hace referencia a él en la clase de vinculación.

Cada clase de vinculación también incluye un método `getRoot()`, que proporciona una referencia directa para la vista raíz del archivo de diseño correspondiente. En este ejemplo, el método `getRoot()` de la clase `ResultProfileBinding` muestra la vista raíz `LinearLayout`

A fin de configurar una instancia de la clase de vinculación para usarla con una actividad, realiza los siguientes pasos en el método `onCreate()` de la actividad:

1. Llamar al método `inflate()` estático incluido en la clase de vinculación generada. Esto crea una instancia de la clase de vinculación para la actividad que se usará.
2. Para obtener una referencia a la vista raíz, llama al método `getRoot()`
3. Pasar la vista raíz a `setContentView()` para que sea la vista activa en la pantalla.

```
private ResultProfileBinding binding;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = ResultProfileBinding.inflate(getLayoutInflater());
    View view = binding.getRoot();
    setContentView(view);
}
```

Ahora podemos usar la instancia de la clase de vinculación para hacer referencia a cualquiera de las vistas:

```
binding.getName().setText(viewModel.getName());
binding.button.setOnClickListener(new View.OnClickListener() {
    viewModel.userClicked()
});
```

Nota: las clases de vinculación también pueden ser utilizadas con fragmentos, para mayor información por favor consultar la documentación de Android.

La vinculación de vistas tiene ventajas importantes frente al uso de `findViewById`:

- Seguridad Nula: debido a que la vinculación de vista crea referencias directas a las vistas, no hay riesgo de una excepción de puntero nulo debido a un ID de vista no válido. Además, cuando una vista solo está presente en algunas configuraciones de un diseño, el campo que contiene su referencia en la clase de vinculación se marca con `Nullable`. Es común que con el uso de `findViewById` se pueda producir errores de tipo `NullPointerException` que se evitan con View Bindings.

- Seguridad de tipos: Los campos de cada clase de vinculación tienen tipos que coinciden con las vistas a las que hacen referencia en el archivo XML. Esto significa que no hay riesgo de una excepción de transmisión de clase.

Estas diferencias significan que las incompatibilidades entre el diseño y el código harán que falle la compilación durante el momento de compilación en lugar de hacerlo en el tiempo de ejecución.

Clase R y pipeline de assets

Existe una clase especial en Android, denominada clase R. R.java es una clase dinámicamente generada, creada durante el proceso de compilación para identificar dinámicamente todos los assets del proyecto (ej: cadenas de texto, elementos visuales, etc)

Una vez definidos los recursos para una aplicación, podemos acceder a ellos mediante los ID que se generan en la clase R del proyecto. La organización de los assets debería ser realizada en subcarpetas específicas dentro la carpeta “res” del proyecto, por ejemplo:

```
MyProject/  
  src/  
    MainActivity.java  
  res/  
    drawable/  
      graphic.png  
    layout/  
      main.xml  
      info.xml  
    mipmap/  
      icon.png  
    values/  
      strings.xml
```

Una vez establecido un recurso en a aplicación, podemos hacer referencia a el usando su ID. Todos los IDs del proyecto se definen en la clase R, automáticamente generada. Para identificar un recurso se requiere conocer:

- El tipo de recurso: por ejemplo string, drawable y layout.
- El nombre del recurso: definido por ejemplo en el XML por android:name

Para acceder al recurso se hace uno de la clase R, tal como se muestra a continuación para un recurso “hola” que corresponde a un string del proyecto.

En Java:

```
R.string.hola
```

En XML

```
@string/hola
```

Podemos acceder a un recurso al pasar su ID como parámetro. Por ejemplo, podemos establecer que una vista `ImageView` utilice un recurso desde `res/drawable/myimage.png` usando el método `setImageResource()`:

```
ImageView imageView = (ImageView) findViewById(R.id.myimageview);  
imageView.setImageResource(R.drawable.myimage);
```

Vistas y listeners

En Android, existe más de una forma de interceptar los eventos desde una interacción del usuario con una aplicación. Al considerar los eventos dentro de la interfaz de usuario, el enfoque consiste en capturar los eventos desde el objeto de vista específico con el que interactúa el usuario. La clase `View` proporciona los medios para hacerlo.

Un objeto de escucha de eventos (listener) es una interfaz de la clase `View` que contiene un solo método de devolución de llamada (callback). El framework de Android llamará a estos métodos cuando la vista con la que se haya registrado el objeto de escucha se active por la interacción del usuario con el elemento de la IU.

En las interfaces de los objetos de escucha de eventos, se incluyen los siguientes métodos de devolución de llamada:

- `onClick()`: de `View.OnClickListener`. Se llama a este método cuando el usuario toca el elemento (en el modo táctil) o se centra en el elemento con las teclas de navegación o la bola de seguimiento y presiona la tecla "Intro" adecuada o la bola de seguimiento.
- `onLongClick()`: de `View.OnLongClickListener`. Se llama a este método cuando el usuario mantiene presionado el elemento (en el modo táctil) o se centra en el elemento con las teclas de navegación o la bola de seguimiento y mantiene presionada la tecla "Intro" adecuada o la bola de seguimiento (durante un segundo).
- `onFocusChange()`: de `View.OnFocusChangeListener`. Se llama a este método cuando el usuario navega hacia el elemento o sale de él utilizando las teclas de navegación o la bola de seguimiento.
- `onKey()`: de `View.OnKeyListener`. Se llama a este método cuando el usuario se centra en el elemento y presiona o suelta una tecla de hardware del dispositivo.
- `onTouch()`: de `View.OnTouchListener`. Se llama a este método cuando el usuario realiza una acción calificada como evento táctil, por ejemplo, presionar, soltar o cualquier gesto de movimiento en la pantalla (dentro de los límites del elemento).

- `onCreateContextMenu()`: de `View.OnCreateContextMenuListener`. Se llama a este método cuando se crea un menú contextual (como resultado de un "clic largo" sostenido). Consulta la explicación sobre menús contextuales en la guía para desarrolladores Menús.

En el siguiente ejemplo, se muestra cómo registrar un objeto de escucha de clic para un botón.

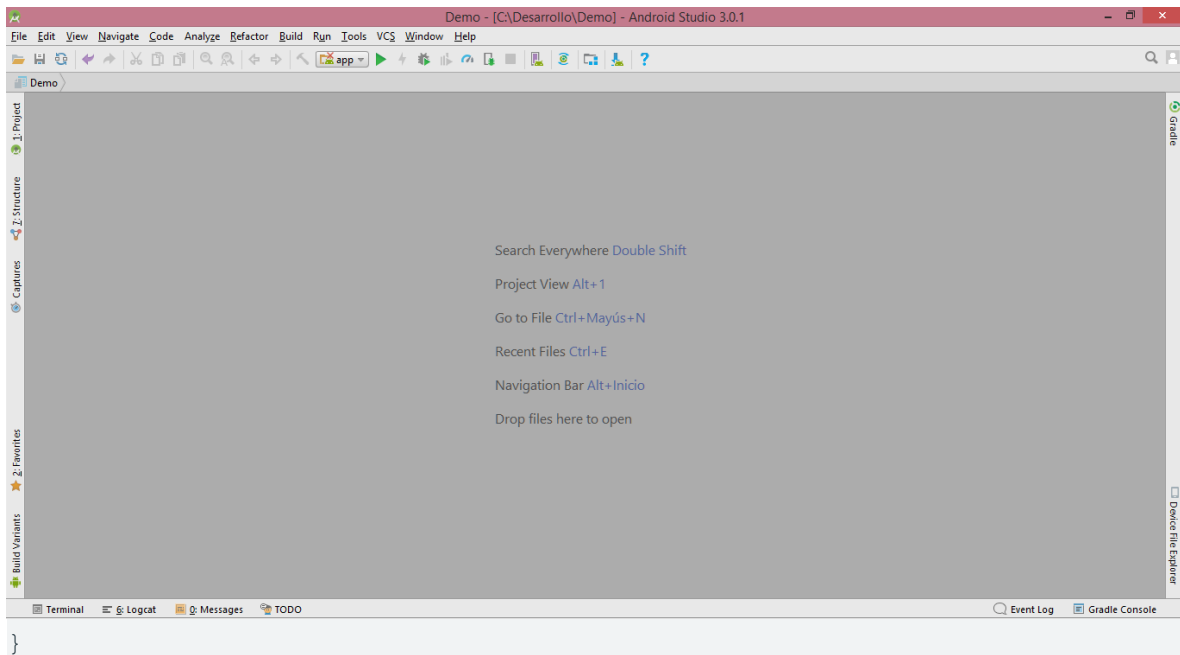
```
// Create an anonymous implementation of OnClickListener
private OnClickListener corkyListener = new OnClickListener() {
    public void onClick(View v) {
        // do something when the button is clicked
    }
};

protected void onCreate(Bundle savedInstanceState) {
    ...
    // Capture our button from layout
    Button button = (Button)findViewById(R.id.corky);
    // Register the onClick listener with the implementation above
    button.setOnClickListener(corkyListener);
    ...
}
```

Puede ser conveniente implementar `OnClickListener` como parte de una actividad. Esto evitará la carga extra de la clase y la asignación de objetos. Por ejemplo:

```
public class ExampleActivity extends Activity implements
OnClickListener {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        Button button = (Button)findViewById(R.id.corky);
        button.setOnClickListener(this);
    }

    // Implement the OnClickListener callback
    public void onClick(View v) {
        // do something when the button is clicked
    }
    ...
}
```



Para más información sobre listeners, por favor consultar la documentación de Android.

Change listeners

Podemos usar objetos de escucha de cambios (ChangeListeners) para recibir notificaciones cuando un archivo específico o carpeta ha cambiado sus contenidos o metadatos. Un objeto de escucha de cambios implementa la interface `OnChangeListener` para el `ChangeEvent` y recibe una llamada de regreso (CallBack) desde el dispositivo.

Podemos agregar un objeto de escucha de cambios al llamar al método `DriveResourceClient.addChangeListener`, pasando el archivo o carpeta de la cual queremos obtener sus cambios y notificaciones. Los objetos de escucha solo se mantienen activos durante la duración de la conexión con el dispositivo o hasta la llamada al método `removeChangeListener`.

En el siguiente ejemplo se incluye código que muestra como agregar un objeto de escucha de cambios para un archivo:

```
getDriveResourceClient()  
    .addChangeListener(file, changeListener)  
    .addOnSuccessListener(this, listenerToken ->  
mChangeListenerToken = listenerToken);
```

para eliminar el objeto de escucha de cambios:

```
getDriveResourceClient().removeChangeListener(mChangeListenerToken);
```

Adicionalmente a agregar el objeto de escucha, debemos implementar el callback con su llamada al método correspondiente. En el ejemplo a continuación se incluye una llamada a un callback que imprime los datos recibidos en un archivo log:

```
/**
 * A listener to handle file change events.
 */
final private OnChangeListener changeListener = new OnChangeListener()
{
    @Override
    public void onChange(ChangeEvent event) {
        mLogTextView.append(getString(R.string.change_event, event));
    }
};
```

Es posible realizar otras operaciones como el cambio de suscripciones, para mayor información por favor consultar la documentación de Android.

Obtener input de usuario

Hemos estudiado como utilizar objetos de escucha, tanto de eventos propios de las vistas particulares como para escucha de cambios en dispositivos. Existen variadas formas de obtener input de usuario en Android que van desde simples acciones con vistas hasta complejas interacciones, tales como gestos táctiles o capacidades del hardware específico (ej: acelerómetro). Uno de los más básicos y comunes corresponde al obtener input desde el teclado del dispositivo, a continuación, se incluye detalles de su utilización:

Entrada desde teclado

Cada campo de texto espera un cierto tipo de entrada de texto, como una dirección de correo electrónico, un número de teléfono o simplemente texto sin formato. Por lo tanto, es importante que especifiquemos el tipo de entrada para cada campo de texto de la app, de manera que el sistema muestre el método de entrada de software apropiado (como un teclado en pantalla).

Además del tipo de botones disponibles con un método de entrada, debemos especificar comportamientos como si el método de entrada proporciona sugerencias ortográficas, escribe mayúsculas en oraciones nuevas y reemplaza el botón de retorno de carro por un botón de acción como Listo o Siguiente.

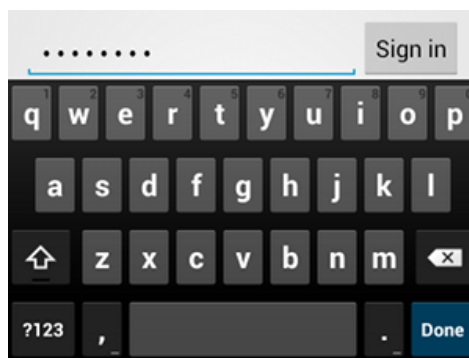
Para definir el elemento de entrada de teclado usamos la vista EditText. Siempre debemos declarar el método de entrada para los campos de texto agregando el atributo android:inputType al elemento <EditText>

Por ejemplo, si deseamos usar un método de entrada para ingresar un número de teléfono, usa el valor "phone".



```
<EditText
    android:id="@+id/phone"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/phone_hint"
    android:inputType="phone" />
```

En cambio, si el campo de texto es de una contraseña, usamos el valor "textPassword" para que el campo de texto oculte la entrada del usuario:



```
<EditText
    android:id="@+id/password"
    android:hint="@string/password_hint"
    android:inputType="textPassword"
    ... />
```

Hay varios valores posibles documentados con el atributo android:inputType y se pueden combinar algunos de los valores para especificar la apariencia del método de entrada y comportamientos adicionales.

Existen muchas otras formas de obtener interacción de parte del usuario, para mayor información y referencia, por favor consultar la documentación específica en:

<https://developer.android.com/guide/input>

UNIDAD 4

Más componentes de un proyecto Android

A continuación, detallaremos algunos de los componentes más importantes de una aplicación Android.

Manifest

Todos los proyectos de apps deben tener un archivo AndroidManifest.xml (con ese mismo nombre) en la raíz de la fuente del proyecto. El archivo de manifiesto describe información esencial de la aplicación para las herramientas de creación de Android, el sistema operativo Android y Google Play. Entre muchas otras cosas, el archivo de manifiesto debe declarar lo siguiente:

- El nombre del paquete de la aplicación que normalmente coincide con el espacio de nombres del código
- Los componentes de la aplicación, que incluyen todas las actividades, servicios, receptores de emisiones y proveedores de contenido.
- Los permisos que necesita la aplicación para acceder a las partes protegidas del sistema o a otras aplicaciones
- Las funciones de hardware y software que requiere la aplicación afectan a los dispositivos que pueden instalar la aplicación desde Google Play

Al usar Android Studio para crear un proyecto, el IDE creará automáticamente un archivo manifest y la mayoría de los elementos esenciales de este se irán agregando a medida que se compila la app.

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
```

```
        android:theme="@style/AppTheme" />
</manifest>
```

Recursos de la app

Ya hemos mencionado anteriormente los assets en la sección dedicada a la clase R, todo proyecto Android seguramente necesitará de recursos, tales como imágenes, definiciones de cadenas de texto, etc.

Se recomienda establecer cada tipo de recurso en una subcarpeta específica dentro de la ruta “res”, tal como se detalla a continuación:

```
MyProject/
  src/
    MainActivity.java
  res/
    drawable/
      graphic.png
    layout/
      main.xml
      info.xml
    mipmap/
      icon.png
    values/
      strings.xml
```

Como podemos ver en el ejemplo, la carpeta “res” incluye todos los recursos: de imágenes, de layouts, para íconos (mipmap) y para cadenas, a continuación, se incluye el detalle de la convención de ubicación para assets.

- animator/ y anim/: archivos XML que definen animaciones
- color/: archivos XML que definen una lista de colores a usar en la app
- drawable/: archivos de mapa de bits (imágenes) o archivos XML que se compilan para crear elementos gráficos
- mipmap/: archivos “dibujables” para las diferentes densidades de “launcher”
- layout/: archivos XML que definen el diseño
- menú/: archivos XML que definen menús
- raw/: archivos generales en su forma original, se accede a ellos usando la clase R, ej: R.raw.nombredearchivo
- values/: archivos XML que contienen valores simples a ser usados, tales como cadenas de texto, números o colores

- `xml/`: archivos XML adicionales que pueden ser utilizados llamando a `Resources.getXML()`
- `font/`: archivos de definición de tipos de letras, tales como `.ttf`, `.otf` o `.ttc`

Los recursos que guardemos en los subdirectorios definidos anteriormente son recursos "predeterminados". Es decir, son los recursos que definen el diseño y el contenido predeterminados de la aplicación. Sin embargo, cada tipo de dispositivo con tecnología Android puede necesitar recursos diferentes. Por ejemplo, si un dispositivo tiene una pantalla de un tamaño mayor que el habitual, debemos proporcionar recursos de diseño diferentes que aprovechen ese espacio adicional en la pantalla. O bien, si un dispositivo tiene una configuración de idioma diferente, debemos proporcionar distintos recursos de strings que traduzcan el texto en su interfaz de usuario. Para proporcionar estos recursos diferentes a distintas configuraciones de dispositivos, debemos proporcionar recursos alternativos, además de tus recursos predeterminados.

Para más información, por favor consultar la documentación de Android:

<https://developer.android.com/guide/topics/resources/providing-resources>

Principales funciones de los scripts de compilación

Gradle del proyecto, Gradle del module app, Conocer las build variants por defecto (debug, release, test y android test), Dependencias, Code version y version name

Archivos Gradle

El sistema de compilación de Android compila recursos y código fuente de la app, y los empaqueta en APK que podamos probar, implementar, firmar y distribuir. Android Studio usa Gradle, un paquete de herramientas de compilación avanzadas, para automatizar y administrar el proceso de compilación, y al mismo tiempo definir configuraciones de compilación personalizadas y flexibles. Cada configuración de compilación puede definir su propio conjunto de código y recursos, y reutilizar las partes comunes a todas las versiones de la app. El complemento de Android para Gradle funciona con el paquete de herramientas de compilación a fin de proporcionar procesos y ajustes configurables específicos para la compilación y prueba de apps para Android.

Gradle y el complemento de Android se ejecutan independientemente de Android Studio. Por lo tanto, podemos compilar apps para Android desde Android Studio, la línea de comandos máquina o en máquinas en las que no esté instalado Android Studio (como servidores de integración continua). Si no usas Android Studio, puedes aprender a compilar y ejecutar tu app desde la línea de comandos. El resultado de la compilación es el mismo si compilamos un proyecto desde la línea de comandos en una máquina remota o con Android Studio.

Archivo de configuración de Gradle

El archivo `settings.gradle`, ubicado en el directorio raíz del proyecto, indica a Gradle los módulos que debe incluir al compilar tu app. Para la mayoría de los proyectos, el archivo es sencillo y solo incluye lo siguiente:

```
include ':app'
```

No obstante, los proyectos con varios módulos deben especificar cada módulo que formará parte de la compilación final.

Archivo de compilación de nivel superior

El archivo `build.gradle` de nivel superior, ubicado en el directorio raíz del proyecto, define configuraciones de compilación que se aplican a todos los módulos de tu proyecto. De forma predeterminada, el archivo de nivel superior usa el bloque `buildscript` para definir los repositorios y las dependencias de Gradle comunes a todos los módulos del proyecto. En el siguiente código de ejemplo, se describen las configuraciones predeterminadas y los elementos de DSL que puedes encontrar en el archivo `build.gradle` de nivel superior después de crear un proyecto nuevo.

Cómo configurar propiedades de todo un proyecto

Para los proyectos que incluyen varios módulos, puede ser útil definir propiedades en el nivel del proyecto y compartirlas en todos los módulos. Puedes hacerlo agregando propiedades adicionales al bloque `ext` en el archivo `build.gradle` de nivel superior.

```
buildscript {...}

allprojects {...}

// This block encapsulates custom properties and makes them available
// to all
// modules in the project.
ext {
    // The following are only a few examples of the types of
    // properties you can define.
    compileSdkVersion = 28
    // You can also create properties to specify versions for
    // dependencies.
    // Having consistent versions between modules can avoid conflicts
    // with behavior.
    supportLibVersion = "28.0.0"
    ...
}
...
```

Para acceder a estas propiedades desde un módulo del mismo proyecto, usamos la siguiente sintaxis en el archivo `build.gradle` del módulo (obtén más información sobre este archivo en la sección que aparece más abajo).

```

android {
    // Use the following syntax to access properties you defined at the
    project level:
    // rootProject.ext.property_name
    compileSdkVersion rootProject.ext.compileSdkVersion
    ...
}
...
dependencies {
    implementation "com.android.support:appcompat-v7:${
{rootProject.ext.supportLibVersion}"
    ...
}

```

Archivo de compilación de nivel de módulo

El archivo build.gradle de nivel de módulo, ubicado en cada directorio project/module/, permite configurar ajustes de compilación para el módulo específico en el que se encuentra. La configuración de esos ajustes de compilación te permite proporcionar opciones de empaquetado personalizadas, como tipos de compilación y variantes de productos adicionales, y anular las configuraciones en el manifiesto main/ de la app o en el archivo build.gradle de nivel superior.

Archivos de propiedades de Gradle

Gradle también incluye dos archivos de propiedades, ubicados en el directorio raíz del proyecto, que podemos usar para especificar ajustes en el paquete de herramientas de compilación de Gradle:

gradle.properties

Aquí podemos configurar ajustes de Gradle para todo el proyecto, como el tamaño máximo del montón de daemon de Gradle. Para obtener más información, consulta Entorno de compilación.

local.properties

Configura las propiedades del entorno local para el sistema de compilación, incluidas las siguientes:

- ndk.dir: Ruta de acceso al NDK. Esta propiedad dejó de estar disponible. Se instalarán las versiones descargadas del NDK en el directorio ndk, dentro del directorio del SDK de Android.

- sdk.dir: Ruta de acceso al SDK.
- cmake.dir: Ruta de acceso a CMake.
- ndk.symlinkdir: En Android Studio 3.5+, crea un symlink al NDK que puede ser más corto que la ruta del NDK instalado.

Build Variants por defecto

Dentro de la configuración del proyecto podremos encontrar algunos “build types” por defecto:

```
buildTypes {
    debug {
    }
    release{
    }
}
```

Estos build types se estudiarán en mayor detalle en la unidad 7, a modo introductorio podemos indicar que esta funcionalidad nos permite obtener distintas versiones del mismo código fuente, agregando pequeñas diferencias que puedan ser necesarias entre estas versiones.

Estos build types permiten la preparación de tu aplicación para el lanzamiento es un proceso de varios pasos que implica las siguientes tareas:

- Configurar la app para el lanzamiento: Como mínimo, debemos quitar las llamadas de Log y el atributo android:debuggable del archivo de manifiesto. También debemos proporcionar valores para los atributos android:versionCode y android:versionName, que se encuentran en el elemento <manifest>. Asimismo, es posible que debamos configurar varios ajustes para cumplir con los requisitos de Google Play o adaptar cualquier método que uses para lanzar tu aplicación.

Si usamos los archivos de compilación de Gradle, podemos emplear el tipo de compilación de lanzamiento (release) a fin de configurar ajustes de compilación para la versión publicada de tu app.

- Compilar y firmar una versión de actualización de la app: Podemos usar los archivos de compilación de Gradle con el tipo de compilación release a fin de compilar y firmar una versión de actualización de la app
- Probar la versión de lanzamiento de la app: Antes de distribuir la app, debemos probar por completo la versión de actualización en al menos un teléfono celular y una tablet de destino.

Estos build types nos permiten generar fácilmente versiones de la app con las características necesarias, tanto para debug (pruebas) como para reléase (lanzamiento), siendo una utilidad muy práctica que podemos usar.

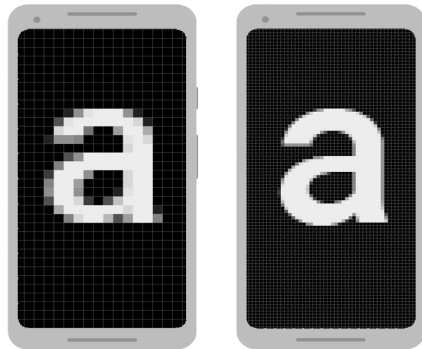
Los Assets de un proyecto Android

Qué pasa si se crea una carpeta erróneamente. Importar recursos en formato rasterizado. Importar recursos en formato vectorizado.

Unidades de medida

Los dispositivos Android no solo vienen con pantallas de diferentes tamaños (teléfonos celulares, tablets, TVs, etc.), sino que sus pantallas también tienen píxeles de distintos tamaños. Es decir, hay dispositivos que tienen 160 píxeles por pulgada cuadrada y otros que adaptan 480 píxeles en el mismo espacio. Si no tienes en cuenta estas variaciones de densidad de píxeles, es posible que el sistema escale las imágenes (y se vean borrosas) o que estas se muestren en el tamaño incorrecto.

Lo primero que debemos evitar es usar píxeles para definir distancias o tamaños. Definir dimensiones con píxeles es un problema, ya que las diferentes pantallas tienen distintas densidades de píxeles. Por lo tanto, la misma cantidad de píxeles puede corresponder a diferentes tamaños físicos en distintos dispositivos.



Para que el tamaño de la IU se mantenga visible en pantallas con distintas densidades, debemos diseñar la IU con píxeles independientes de la densidad (**dp**) como unidad de medida. Un dp es una unidad de píxel virtual que prácticamente equivale a un píxel en una pantalla de densidad media (160 dpi; la densidad "de referencia"). Android traduce este valor a la cantidad apropiada de píxeles reales para cada densidad.

Por ejemplo, observa los dos dispositivos de la figura anterior. Si definiéramos una vista de "100 px" de ancho, parecerá mucho más grande en el dispositivo de la izquierda. Por lo tanto, debemos usar "100 dp" para asegurarnos de que aparezca en ambas pantallas.

Sin embargo, al definir tamaños de texto, debemos usar píxeles escalables (**sp**) como unidades (pero nunca para los tamaños del diseño). De manera predeterminada, la unidad de sp tiene el mismo tamaño que la de dp, pero se modifica en función del tamaño de texto que prefiere el usuario.

Por ejemplo, para especificar el espaciado entre dos vistas, usamos dp:

```
<Button android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
android:text="@string/clickme"  
android:layout_marginTop="20dp" />
```

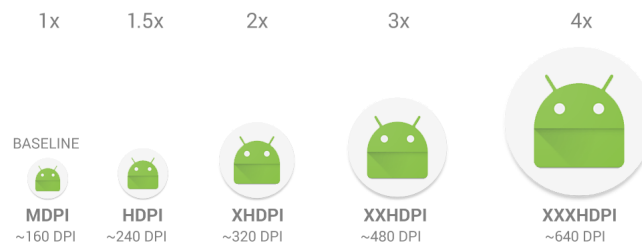
Para especificar el tamaño de texto, siempre utilizamos sp:

```
<TextView android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textSize="20sp" />
```

En algunos casos, necesitaremos expresar las dimensiones en dp y convertirlas a píxeles. La conversión de unidades dp a píxeles de pantalla es simple:

$$px = dp * (dpi / 160)$$

Para proporcionar gráficos de buena calidad en dispositivos con diferentes densidades de píxeles, debemos brindar varias versiones de cada mapa de bits en la app (una para cada intervalo de densidad) en la resolución correspondiente. De lo contrario, Android deberá escalar el mapa de bits para que ocupe el mismo espacio visible en cada pantalla, lo que generará artefactos de escalamiento como imágenes borrosas (o “píxeleadas”).



Hay muchos intervalos de densidad disponibles para usar en apps. A continuación, se describen los diferentes calificadores de configuración disponibles y a qué tipos de pantallas se aplican:

- **Ldpi:** Recursos para pantallas de densidad baja (ldpi) (120 dpi)
- **Mdpi:** Recursos para pantallas de densidad media (mdpi) (~160 dpi; esta es la densidad de referencia)
- **Hdpi:** Recursos para pantallas de densidad alta (hdpi) (~240 dpi)
- **Xhdpi:** Recursos para pantallas de densidad muy alta (xhdpi) (~320 dpi)
- **Xxhdpi:** Recursos para pantallas de densidad muy, muy alta (xxhdpi) (~480 dpi)
- **Xxxhdpi:** Recursos para usos de densidad extremadamente alta (xxxhdpi) (~640 dpi)
- **Nodpi:** Recursos para todas las densidades. Estos son recursos independientes de la densidad. El sistema no escala recursos etiquetados con este calificador, independientemente de la densidad de la pantalla actual.

- Tvdpi: Recursos para pantallas entre mdpi y hdpi; de aproximadamente 213 dpi. Este no se considera un grupo de densidad "principal". Se usa más que nada para televisiones, y la mayoría de las apps no lo necesitarán ya que con recursos mdpi y hdpi será suficiente

Si deseamos crear elementos de diseño alternativos de mapa de bits para diferentes densidades, debemos seguir la proporción de escalamiento 3:4:6:8:12:16 entre las seis densidades principales. Por ejemplo, si tenemos un elemento de diseño de mapa de bits de 48 x 48 píxeles para pantallas de densidad media, los tamaños diferentes deben ser los siguientes:

- 36 x 36 (0.75x) para densidad baja (ldpi)
- 48 x 48 (referencia de 1.0x) para densidad media (mdpi)
- 72 x 72 (1.5x) para densidad alta (hdpi)
- 96 x 96 (2.0x) para densidad muy alta (xhdpi)
- 144 x 144 (3.0x) para densidad muy, muy alta (xxhdpi)
- 192 x 192 (4.0x) para densidad extremadamente alta (xxxhdpi)

Luego, colocar los archivos de imagen generados en el subdirectorío correspondiente en *res/*, y el sistema elegirá el correcto automáticamente, en función de la densidad de píxeles del dispositivo en el que se ejecuta la app:

```
res/
    drawable-xxxhdpi/
        awesome-image.png
    drawable-xxhdpi/
        awesome-image.png
    drawable-xhdpi/
        awesome-image.png
    drawable-hdpi/
        awesome-image.png
    drawable-mdpi/
        awesome-image.png
```

De esta forma, cada vez que se haga referencia a `@drawable/awesomeimage`, el sistema seleccionará el mapa de bits adecuado en función de los dpi de la pantalla. Si no proporcionas un recurso específico de densidad para ese valor, el sistema seleccionará la mejor opción y la escalará para que se ajuste a la pantalla.

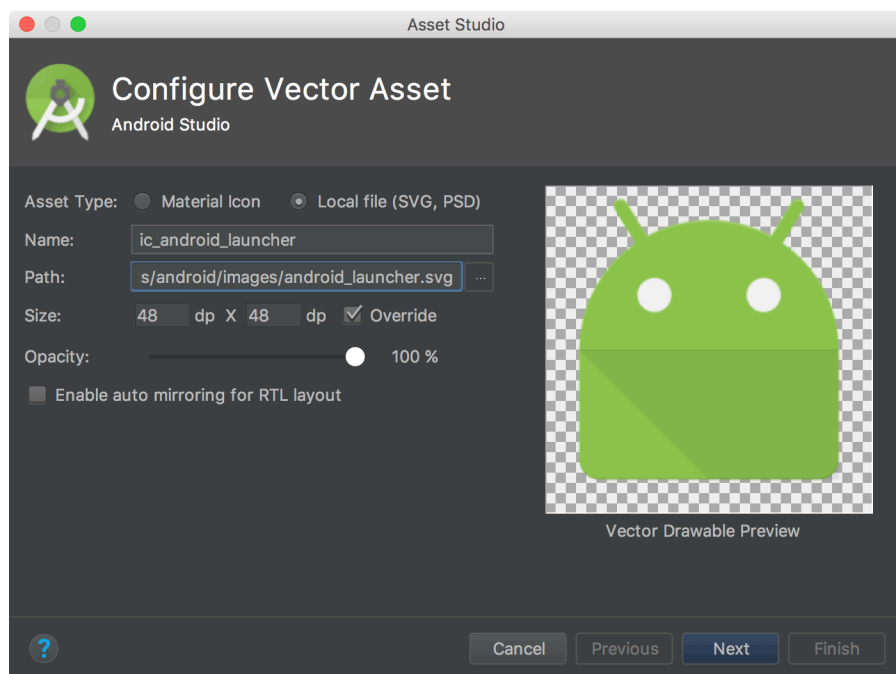
Gráficos Vectoriales

Como alternativa a la creación de varias versiones específicas de densidad de una imagen, podemos crear imágenes con gráficos vectoriales. Estos crean una imagen mediante XML para definir rutas y colores, en lugar de usar mapas de bits de píxeles. Los gráficos vectoriales pueden escalarse a cualquier tamaño sin problemas de escalamiento, aunque suelen funcionar mejor para ilustraciones como íconos, no así para fotos.

Los gráficos vectoriales suelen proporcionarse como un archivo SVG (gráficos vectoriales escalables), pero Android no admite este formato, por lo que debemos convertirlos al formato de elemento de diseño vectorial de Android.

Podemos convertir un archivo SVG a un elemento de diseño vectorial de Android Studio fácilmente con Vector Asset Studio de la siguiente manera:

1. En la ventana Proyecto, haz clic con el botón derecho en el directorio res y selecciona Nuevo > Elemento vectorial.
2. Selecciona Archivo local (SVG, PSD).
3. Ubica el archivo que quieras importar y realiza las modificaciones necesarias



Es posible que aparezcan algunos errores en la ventana de Asset Studio que indiquen que algunas propiedades del archivo no son compatibles con los elementos de diseño vectoriales. Sin embargo, estos no impedirán que realices la importación, ya que las propiedades no compatibles simplemente se ignoran.

4. Haz clic en Next.
5. En la pantalla siguiente, confirma el conjunto de orígenes en donde quieras ubicar el archivo de tu proyecto y haz clic en Finish

Como un elemento de diseño vectorial puede usarse en todas las densidades de píxeles, este archivo se ubica en el directorio de elementos de diseño predeterminado (no necesitas usar directorios específicos de densidad)

```
res/  
    drawable/  
        ic_android_launcher.xml
```

Nota: Es importante probar la app en varios dispositivos con diferentes densidades de píxeles, para asegurarnos de que la IU se escale correctamente. Es fácil probarla en un dispositivo físico, pero también podemos hacerlo en Android Emulator si no hay acceso a dispositivos físicos de todas las densidades de píxeles.

Es posible también probarla en dispositivos físicos, si no es posible comprarlos, podemos usar Firebase Test Lab para acceder a dispositivos en el centro de datos de Google.

Layouts y tamaño

Los dispositivos Android vienen de todas las formas y tamaños, por lo que el diseño de tu app debe ser flexible. Es decir que, en lugar de definir un diseño con dimensiones rígidas para un tamaño de pantalla y una relación de aspecto determinados, tu diseño debería ajustarse a pantallas de diferentes tamaños y orientaciones.

La compatibilidad con muchas pantallas permite que la app se encuentre disponible para la mayor cantidad posible de usuarios con dispositivos diferentes, mediante un único APK. Además, hacer que tu app funcione en distintos tamaños de pantalla garantiza poder procesar cambios en la configuración de la ventana del dispositivo, como cuando el usuario habilita el modo multiventana.

Sin embargo, debes tener en cuenta que adaptar tu app a diferentes tamaños de pantalla no necesariamente hace que sea compatible con todos los factores de forma de Android. Deberás realizar pasos adicionales para agregar compatibilidad con dispositivos Android Wear, Android TV, Android Auto y el Sistema operativo Chrome.

Independientemente del perfil de hardware con el que quieras brindar compatibilidad primero, debes crear un diseño que sea receptivo incluso para las variaciones más leves de tamaño de pantalla.

Cómo usar ConstraintLayout

La mejor manera de crear un diseño reactivo para diferentes tamaños de pantalla es usar ConstraintLayout como el diseño base en tu IU. ConstraintLayout te permite especificar la posición y el tamaño de cada vista según las relaciones espaciales con otras vistas en el diseño. De esta manera, todas las vistas pueden moverse y expandirse a la vez, a medida que cambia el tamaño de la pantalla.

La forma más fácil de compilar un diseño con ConstraintLayout es usar el editor de diseño de Android Studio, ya que te permite arrastrar nuevas vistas al diseño, asociar sus restricciones a la vista principal y a otras vistas relacionadas, y editar las propiedades de la vista, todo sin la necesidad de editar ningún archivo XML manualmente

Aunque ConstraintLayout no resolverá todas las situaciones de diseño (especialmente en el caso de las listas cargadas de forma dinámica, donde deberás usar RecyclerView), no importa cuál sea el diseño que utilices, siempre debes evitar los tamaños de diseño hard-coded (consulta la siguiente sección).

Para asegurarte de que tu diseño sea flexible y se adapte a diferentes tamaños de pantalla, debes usar "wrap_content" y "match_parent" para el ancho y la altura de la mayoría de los componentes de la vista, en lugar de los tamaños hard-coded.

"wrap_content" le indica a la vista que establezca el tamaño necesario para ajustar el contenido dentro de esa vista.

"match_parent" hace que la vista se expanda lo más posible dentro de la vista principal.

Por ejemplo:

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/lorem_ipsum" />
```

Aunque el diseño real de esta vista depende de otros atributos en su vista principal y cualquier vista relacionada, esta TextView intenta establecer el ancho para rellenar todo el espacio disponible (match_parent) y configurar la altura para que sea exactamente igual al espacio requerido para la longitud del texto (wrap_content). Esto permite que la vista se adapte a diferentes tamaños de pantalla y diferentes longitudes de texto.



Si usas un LinearLayout, también puedes expandir las vistas secundarias con ponderación de diseño para que cada vista rellene el espacio restante proporcionalmente a su valor de ponderación. Sin embargo, el uso de ponderaciones en un LinearLayout anidado requiere que el sistema realice varios pases de diseño para determinar el tamaño de cada vista, lo que ralentiza el rendimiento de la IU. Afortunadamente, ConstraintLayout puede lograr casi todos los diseños

posibles con LinearLayout sin afectar el rendimiento, por lo que debes intentar convertir tu diseño a ConstraintLayout. Luego, puedes definir los diseños ponderados mediante cadenas de restricciones.

Para más información, por favor consultar la documentación oficial en:

<https://developer.android.com/training/multiscreen/screensizes?hl=es-419>

UNIDAD 5

Fundamentos de GIT

Al trabajar en equipo en desarrollo de sistemas se hace necesario e imperante contar con alguna herramienta que permita mantener centralizado el código fuente del software, eventualmente los trabajos realizados por cada desarrollador deben juntarse y mezclarse con los avances de otros. En tiempos posteriores a la aparición de los sistemas de control de versiones, esta tarea debía ser realizada de forma manual o apoyándose en herramientas no óptimas, como servidores de FTP. Al no contar con herramientas adecuadas era común generar problemas como pérdidas de código, imposibilidad de mezclar cambios, etc. Estos problemas se resuelven a utilizar estas herramientas para control de versiones diseñadas y pensadas en centralizar el trabajo, siendo parte integral de las herramientas ampliamente usadas en el desarrollo de sistemas.

Un sistema de control de versiones (VCS de sus siglas en inglés), mantiene la historia de cambios a medida que las personas y equipos colaboran en proyectos en conjunto. A medida que el proyecto evoluciona, los equipos pueden correr pruebas, corregir defectos y contribuir con nuevo código con la confianza de que cada versión puede ser recuperada en cualquier momento que sea necesario. Los desarrolladores pueden revisar la historia para averiguar:

- ¿Qué cambios se han hecho?
- ¿Quién hizo los cambios?
- ¿Cuándo se hicieron estos cambios?
- ¿Por qué se hicieron los cambios?

Git es un ejemplo de un sistema de control de versiones distribuido, comúnmente usado para el desarrollo de software de código libre y comercial. Git permite total acceso a cada fila, rama, y cada iteración de un proyecto, permitiendo a los usuarios acceder a un completo historial de todos los cambios. A diferencia de algunos sistemas de control de versiones anteriormente populares, Git no requiere de una conexión constante a un repositorio central. Los desarrolladores pueden trabajar y colaborar desde cualquier lugar en cualquier hora.

Sin un control de versiones los desarrolladores están obligados a realizar tareas redundantes, mantener planificaciones más lentas e incluso mantener múltiples copias de un proyecto. Para eliminar trabajo innecesario Git y otros sistemas de control de versiones, entregan a cada

desarrollador una vista consistente de un proyecto, destacando el trabajo en progreso. Permitir ver la historia transparente de cambios, quien los hizo, y como estos han contribuido al desarrollo de un proyecto, ayuda a los miembros del equipo a mantenerse alineados en su trabajo.

Repositorio

Un repositorio corresponde a la colección completa de archivos y carpetas asociadas a un proyecto, junto con la historia de versiones de cada archivo. El historial de un archivo aparece en un conjunto de “tomas instantáneas” de tiempo llamadas “commit”, estos existen relacionados y pueden ser organizados en diferentes líneas de desarrollo llamadas “ramas”. Debido a que Git es descentralizado, los repositorios son unidades auto contenidas y cualquiera que tenga una copia de este puede acceder a todo el código fuente y todo su historial. Usando línea de comandos u otras interfaces de fácil uso (clientes externos), un repositorio Git permite: interacción con su historia, clonación, manejo de ramas (branches), realizar “commits”, mezclar código, comparar cambios entre versiones, y mucho más.

Instalación

Git puede ser instalado en muchos sistemas operativos, como Windows, Mac y Linux. Comúnmente Git viene instalado por defecto en sistemas Mac OSX y muchas distribuciones de Linux. En este manual nos centraremos en la utilización de Git en sistemas Windows.

Para revisar si tenemos instalado Git, abrir una ventana de consola y ejecutar el siguiente comando:

```
git versión
```

Este comando entregará el número de versión instalada o indicará que git es un comando desconocido en caso contrario.

Para instalar Git en Windows, seguir las siguientes instrucciones:

1. Navegar a <https://gitforwindows.org/> y descargar la última versión.
2. Una vez descargado, ejecutar el instalador y seguir las instrucciones del wizard hasta completar la instalación.
3. Abrir una ventana de consola y ejecutar el comando “git versión”, si se entrega el número de versión, significa que la instalación se ha completado exitosamente.

Nota: Se recomienda la revisión de alternativas de uso de Git tales como TortoiseGit u otros clientes gráficos que resuelven la necesidad de memorizar comandos que en ocasiones puede ser complejos.

Comandos básicos de Git

Para utilizar Git, los desarrolladores usan comandos específicos para crear, cambios y combinar código. Estos comandos pueden ser ejecutados directamente desde líneas de comandos o utilizando aplicaciones externas, a continuación, algunos de los comandos más utilizados:

- `git init` inicializa un nuevo repositorio Git en blanco y comienza el seguimiento del directorio. Este comando agrega una subcarpeta escondida en el directorio actual que mantiene los datos estructurales internos requeridos por el controlador de versiones.
- `git clone` crea una copia local del proyecto que ya existe remotamente (en el repositorio). La clonación incluye todos los archivos, carpetas, ramas e historia. En la práctica se utiliza el comando `git clone` incluyendo el "path" del repositorio a clonar

`git clone /path/repositorio`

- `git add` reserva los cambios. Git mantiene rastro de los cambios que realiza el desarrollador, pero es necesario mantener un "stash" o reserva y mantener el estado de los cambios en una "instantánea" para incluirlos en la historia del proyecto. Este comando realiza una reserva, todos los cambios que se mantienen en la reserva "stash" serán parte de la historia del proyecto.
- `git commit` guarda una toma instantánea de la historia del proyecto y completa el proceso de rastrear los cambios. Commit es como tomar una foto del estado del proyecto. Todo lo que se haya "reservado" con `git add` será parte de la toma "instantánea" realizada. En la práctica se realiza el commit incluyendo un mensaje de la siguiente forma:

`git commit -m "mensaje detallando lo realizado en mi commit"`

- `git status` muestra el estado de cambios como no-rastreados, modificados o reservados.
- `git branch` muestra las ramas sobre las cuales se está trabajando localmente.

- `git checkout` crea una rama o permite realizar un cambio a alguna de las ramas referenciadas localmente, por ejemplo, el siguiente comando “descarga” la rama *feature* y permite al desarrollador trabajar en esta:

git checkout feature

Otro uso común del comando *checkout* es el permitirnos reemplazar los cambios locales, en caso de que hagamos algo mal es posible recuperar el estado anterior, para ello usamos el comando

Git checkout -- <filename>

Este comando reemplaza los cambios en la carpeta de trabajo con el último contenido de HEAD. Los cambios que ya han sido agregados al, así como también los nuevos archivos, se mantendrán sin cambios.

- `git reset` si deseamos deshacer todos los cambios locales y commits, podemos traer la última versión del servidor y apuntar a la copia local principal. Este comando debe ser utilizado con mucho cuidado pues una vez ejecutado no tenemos forma de recuperar el posible trabajo perdido, a continuación, un ejemplo de su uso

git reset --hard origin/master

- `git merge` combina líneas de desarrollo. Este comando típicamente se usa para combinar cambios que fueron realizados en diferentes ramas. Por ejemplo, un desarrollador puede mezclar cuando desea combinar cambios desde la rama en la que se encuentra trabajando hacia la rama principal de desarrollo del proyecto (generalmente la rama “Master”).
- `git rebase` es un comando para rescritura del historial, permite mover o combinar una secuencia de “commits” a un nuevo commit base, podemos pensar que rebase “comprime” todos los cambios en uno solo para luego integrarlo a la rama objetivo. A diferencia de *merge*, rebase “aplana” el historial al transferir el trabajo completo de una rama a otra.

- `git pull` actualiza la línea local de desarrollo con modificaciones desde su contraparte remota. Los desarrolladores usan este comando si un compañero de equipo ha realizado cambios a una rama remota y desean que estos cambios se reflejen en su ambiente de desarrollo local.
- `git push` actualiza el repositorio remoto con los cambios realizados localmente a una rama. En la práctica se indican detalles del repositorio y la rama, por ejemplo:

git push origin master

- `git mv` permite renombrar archivos y carpetas en el actual repositorio. Por ejemplo:

git mv original.txt nuevo.txt

Conflictos

Los sistemas de control de versiones consisten en gestionar contribuciones entre múltiples autores distribuidos. A veces múltiples desarrolladores podrían intentar editar el mismo contenido. Si el desarrollador A intenta editar código que el desarrollador B está editando, podría producirse un conflicto. Para evitar la ocurrencia de conflictos, los desarrolladores trabajarán en ramas aisladas separadas. La función principal del comando git merge es combinar ramas separadas (como ya se explicó) y resolver las ediciones conflictivas.

Normalmente los conflictos surgen cuando dos personas han cambiado las mismas líneas de un archivo o si un desarrollador ha eliminado un archivo mientras otro lo estaba modificando. En estos casos, Git no puede determinar automáticamente qué es correcto. Los conflictos solo afectan al desarrollador que realiza la fusión, el resto del equipo no se entera del conflicto. Git marcará el archivo como que tiene un conflicto y detendrá el proceso de fusión. Entonces el desarrollador es el responsable de resolver el conflicto.

Al producirse un conflicto, Git generará un resultado descriptivo para indicarnos que se ha producido un CONFLICTO. Podemos obtener más información ejecutando el comando git status. Cuando un archivo presenta conflictos, Git añadirá líneas de texto adicionales para indicar las secciones donde estos se producen, detallando específicamente las diferencias entre HEAD (original) y los cambios en la nueva rama, por ejemplo:

- <<<<<< HEAD
- =====
- >>>>>> new_branch_to_merge_later

Considera estas nuevas líneas como "divisorias de conflictos". La línea ===== es el "centro" del conflicto. Todo el contenido entre el centro y la línea <<<<<< HEAD es contenido que existe en la rama actual maestra a la que apunta la referencia HEAD. De manera alternativa, todo el contenido entre el centro y >>>>>> new_branch_to_merge_later es contenido que está presente en nuestra rama de fusión.

La forma más directa de resolver un conflicto de fusión es editar el archivo conflictivo. Abre el archivo merge.txt en tu editor favorito. La comunicación en el equipo es importante para que el desarrollador a cargo de la resolución tenga la información necesaria para poder tomar la decisión correcta al resolver el conflicto a mano.

Una vez que el archivo se ha editado, utiliza git add <nombredearchivo> para organizar el nuevo contenido fusionado, para finalizar la resolución creamos un nuevo "commit" que confirmará los cambios, cerrando el flujo de resolución de conflictos.

Ignore

Para Git, todos los archivos en la copia local se mantienen en uno de 3 estados "rastreado" (tracked), "no rastreado" (untracked) o **ignorado**. Git puede ignorar archivos siempre y cuando se le especifique así. Los archivos ignorados son usualmente artefactos de compilación u otros archivos generados por los ambientes de desarrollo que pueden ser derivados del código fuente pero no forman parte del proyecto.

Los archivos ignorados se detallan en un archivo especial llamado .gitignore que se mantiene en la carpeta raíz del repositorio. No existen comandos específicos para ignorar, en su lugar, el desarrollador debe mantener el archivo .gitignore haciendo uso de editores de texto o de las posibles herramientas que incluya su entorno de desarrollo o IDE. Existen varias formas de agregar detalles al archivo de ignorados, por ejemplo:

**.log : ignora todos los archivos que tengan extensión .log*

Para más información sobre gitignore y patrones a utilizar, por favor consultar la documentación en:

<https://www.atlassian.com/git/tutorials/saving-changes/gitignore>

Fundamentos de GitHub

GitHub es una plataforma de hospedaje para la colaboración y control de versiones. Nos permite trabajar y colaborar en proyectos desde cualquier lugar. Usa el sistema de control de versiones Git

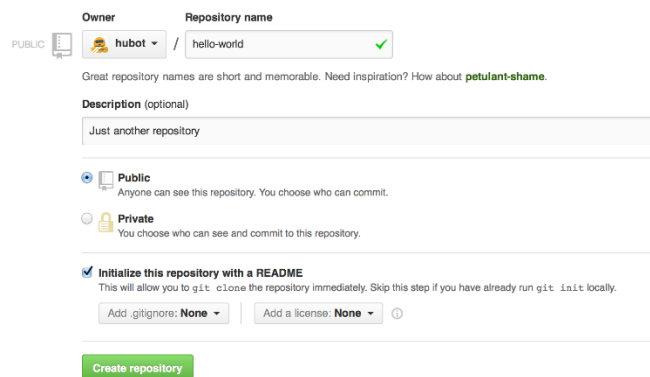
y se utiliza principalmente para la creación de código fuente. Algunas de sus principales características incluyen:

- Wiki para cada proyecto
- Página web para cada proyecto
- Gráfico para ver cómo los desarrolladores trabajan en sus repositorios y bifurcaciones del proyecto.
- Funcionalidades como si se tratase de una red social, por ejemplo, seguidores.
- Herramienta para trabajo colaborativo entre programadores.
- Gestor de proyectos de estilo Kanban.
- Actions herramientas de CI
- Codespaces un IDE en la nube para los repositorios.
- Colaboración para todos.

Crear un repositorio

Un repositorio es usualmente utilizado para organizar un proyecto individual. Estos pueden contener carpetas y archivos, imágenes, videos, planillas o cualquier otra cosa que el proyecto necesite. Para crear un repositorio en GitHub, siga los siguientes pasos:

1. En la esquina superior derecha, dar click al ícono “+” y seleccionar “New Repository” (nota: la UI de GitHub se presenta en inglés).
2. Dar nombre al repositorio, por ejemplo “hello-world”
3. Escribir una descripción breve
4. Seleccionar “Initialize this repository with a README.”
5. Dar click al botón “Create repository”



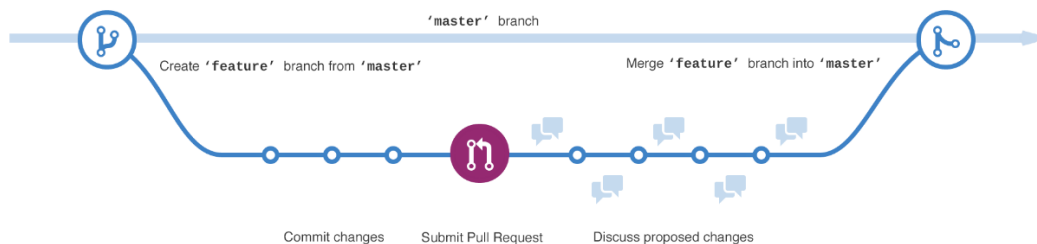
The screenshot shows the GitHub 'Create new repository' form. At the top, there's a 'PUBLIC' label and a 'Repository name' field containing 'hello-world' with a green checkmark. Below this is a note: 'Great repository names are short and memorable. Need inspiration? How about [petulant-shame](#).' The 'Description (optional)' field contains 'Just another repository'. There are two radio buttons for visibility: 'Public' (selected) and 'Private'. Below these is a section 'Initialize this repository with a README' which is checked. At the bottom, there are dropdowns for 'Add .gitignore: None' and 'Add a license: None', followed by a green 'Create repository' button.

Crear una rama

El trabajar en ramas es la forma de mantener diferentes versiones de un repositorio al mismo tiempo. La idea de ramas en GitHub es equivalente a trabajar con ramas desde la línea de comandos usando los comandos de Git.

Por defecto nuestro repositorio tendrá una rama inicial llamada “main”, esta es considerada la rama principal del proyecto. Usamos ramas para experimentar y hacer cambios al código fuente antes de llevarlo a la rama “main”. Tal como se explicó anteriormente, deberíamos crear ramas para realizar cambios al código debido a la necesidad de corregir un defecto, agregar nuevas funcionalidades o experimentar con cambios, asegurándonos de que los cambios realizados no generen un daño a la rama principal.

Al crear una rama desde la rama principal “main”, estamos generando una copia instantánea de esta, tal como se encontraba en el momento del tiempo que se crea la rama. Si alguien más hace cambios a la rama “main” mientras nos encontramos trabajando en nuestra rama, es posible recuperar esos cambios y llevarlos a la nueva rama, ejecutando una acción de “pull”.

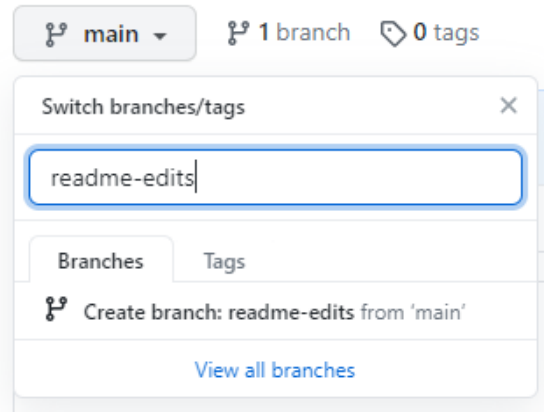


En la imagen podemos ver:

- Una rama principal “main”
- Una nueva rama llamada “feature”, generada desde la rama principal
- Todo el historial por el que pasa la rama “feature”, desde su creación hasta su incorporación a “main”

Para crear una rama en GitHub debemos:

- Acceder a nuestro repositorio “hello-world”
- Dar click en el menú desplegable en la sección superior que dice “main”
- Escribir en el cuadro de texto un nombre para la nueva rama, por ejemplo “readme-edits”
- Seleccionar “Create Branch” o presionar *Enter*



Al crear la nueva rama, nuestro proyecto ahora tiene dos ramas activas “main” y “readme-edits”. Ambas se ven exactamente iguales por ahora.

Realizar cambios

Vamos a realizar cambios a la rama “readme-edits”, los cambios en GitHub se conocen como “commits” (igual que en Git). Cada commit tiene asociado un mensaje o descripción indicado por el desarrollador. Para realizar cambios:

1. Dar click en el ícono editar (un ícono como lápiz) del archivo README.me
2. GitHub despliega el editor de texto, escribir nuevo texto al archivo, como por ejemplo “new changes here”
3. Escribir un mensaje de commit que describa los cambios
4. Dar click al botón “commit changes”

Commit changes

Create README.md

Descripción del commit

☒ Commit directly to the `readme-edits` branch.

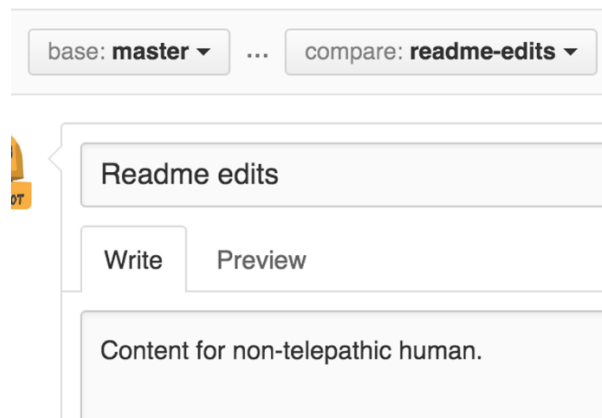
☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

Los cambios realizados al archivo README fueron guardados en el *commit*, estando presente en la rama “readme-edits”, esto significa que la versión del archivo README de la rama principal “main” no tiene estos cambios.

Para llevar los cambios desde la rama “readme-edits” a la rama “main”, usaremos un *pull request*. Un pull request abre una propuesta de cambios y solicita que alguien revise los cambios y mueva la contribución realizada, mezclándola a un Branch. Un pull request muestra diferencias (diff) del contenido entre ambas ramas. Los cambios, adiciones y sustracciones se muestran en verde y rojo.

Al realiza un commit podemos inmediatamente abrir un nuevo pull request. Para abrir un nuevo pull request:

1. Dar click al tab “Pull Request”
2. Dar click al botón “créate pull request”
3. En la sección superior, en el menú de “compare”, seleccionar la rama nueva, en nuestro ejemplo “readme-edits”
4. GitHub actualizará su UI y mostrará las diferencias entre ambas ramas, detallando a nivel de archivos
5. Si estamos satisfechos con los cambios, dar click en el botón “Create pull request”.
6. Escribir un titulo para el pull request y un pequeña descripción.
7. Dar click al botón “Create pull request”



El paso final para llevar los cambios a la rama main corresponde a “mezclar” (merge) la rama, para esto realizamos un “merge”.

1. Dar click en el botón “Merge pull request
2. Dar click a “confirm merge”
3. Ahora podemos borrar la rama ya que sus cambios han sido incorporados a la rama “main”. Dar click al botón “Delete Branch”

Con esta acción hemos completado un flujo de trabajo simple en GitHub, mezclando las ramas hemos logrado que los cambios realizados a “readme-edits” ahora se encuentren en la rama

“main”, manteniendo un orden y generando información de historial del proyecto muy útil para el desarrollo del proyecto.

Clonación de repositorios

No siempre trabajaremos con repositorios propios, a menudo debemos unirnos a un equipo de trabajo, lo que significa también unirnos a su proyecto en GitHub. Para comenzar a trabajar en un repositorio que ya existe, propio o ajeno, debemos hacer una *clonación*. Podemos crear un repositorio para mantener una copia local en nuestro ambiente de desarrollo y luego mantener ambos sincronizados.

Para clonar un repositorio

1. Ir a la página principal del repositorio
2. Sobre la lista de archivos dar click en el botón “Code”
3. Dar click en el botón “clipboard”, esto copiará una cadena de texto que incluye los detalles del repositorio a clonar.
4. Abrir Git bash (o cualquier cliente de Git utilizado) y pegar la URL copiada
- 5.

```
$ git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY
```

6. Dar enter para clonar el repositorio

```
$ git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY
> Cloning into `Spoon-Knife`...
> remote: Counting objects: 10, done.
> remote: Compressing objects: 100% (8/8), done.
> remote: Total 10 (delta 1), reused 10 (delta 1)
> Unpacking objects: 100% (10/10), done.
```

GitHub Markdown

En GitHub existe una forma de dar estilo al texto que ingresamos en la Web. Podemos controlar la forma que se ve un documento, formateando palabras en negritas o cursivas o creando listas entre otras. Markdown es básicamente texto simple que incluye algunos caracteres adicionales como # o *.

Por ejemplo:

Es muy fácil hacer que palabras sean en **negrita** y otras palabras en *cursiva* con Markdown.

Es muy fácil hacer que palabras sean en **negrita** y otras en *cursiva* con Markdown.

Para más información sobre como utilizar Markdown y su sintaxis, por favor consultar:

<https://guides.github.com/features/mastering-markdown/>

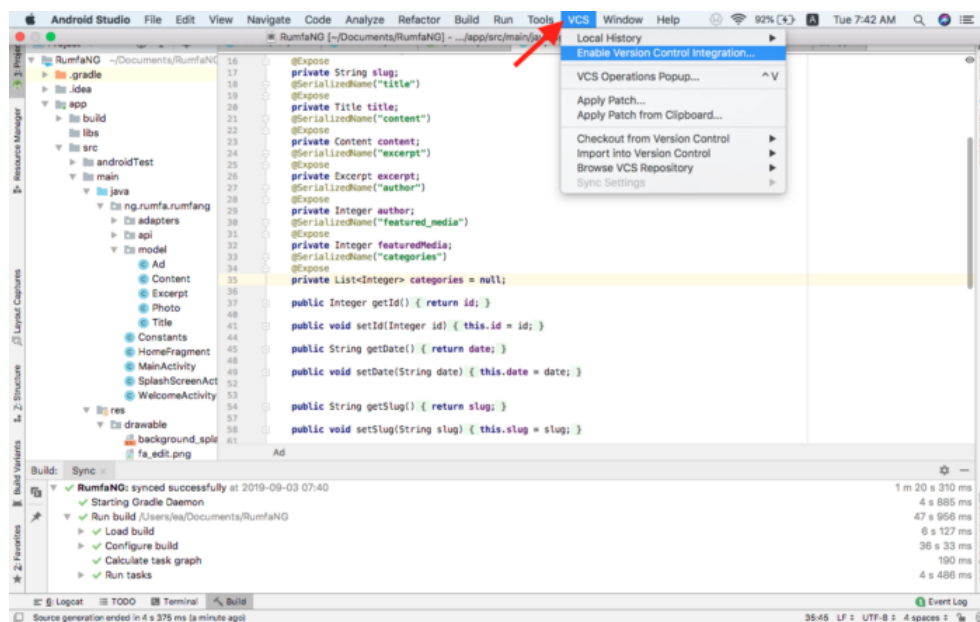
Git y Android Studio

Temas: Configuración del VCS en Android Studio, Iniciar el repositorio usando Android Studio, Listas VCS en Android Studio vs .gitignore, Commit y push usando Android Studio, Branches y merge usando Android Studio

Android Studio admite diferentes sistemas de control de versión (VCS), incluidos Git, GitHub, CVS, Mercurial, Subversion y Google Cloud Source Repositories.

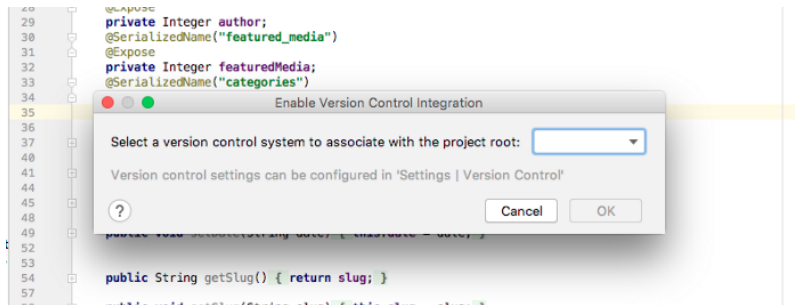
Después de importar tu app a Android Studio o crear un nuevo proyecto, usa las opciones del menú del VCS de Android Studio a fin de habilitar la compatibilidad con VCS para el sistema de control de versión deseado, crear un repositorio, importar los archivos nuevos al control de versión y realizar otras operaciones de control de versión:

En el menú del VCS de Android Studio, haz clic en “Enable Version Control Integration”.



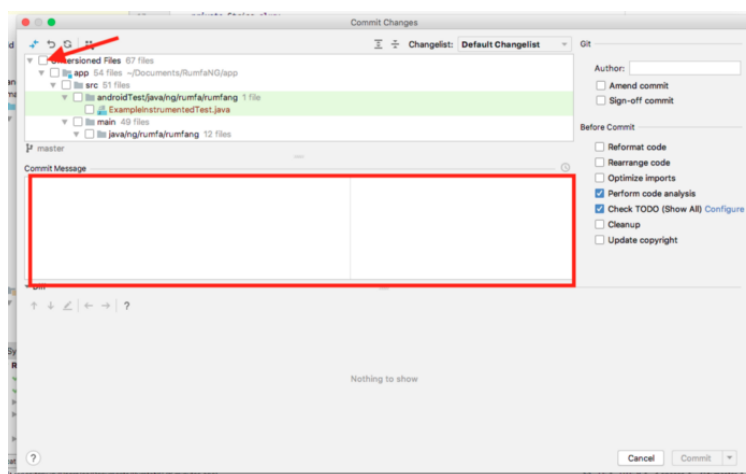
En el menú desplegable, selecciona un sistema de control de versión para asociarlo con la raíz del proyecto y, luego, haz clic en OK.

En el menú del VCS se mostrarán diversas opciones de control de versión según el sistema que hayas seleccionado.



Con el control de versiones ya activado, podemos realizar las acciones típicas para gestionar nuestro código. Podemos revisar nuestros cambios y ver que algunos nombres de archivo ahora tienen color rojo.

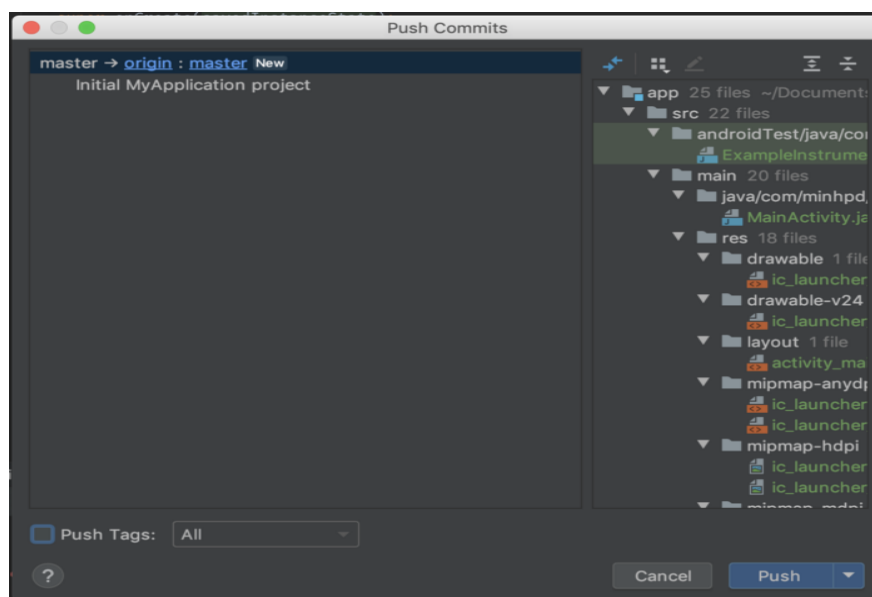
Abriendo el menú de VCS podemos seleccionar VCS -> commit. Android Studio presentará la siguiente ventana



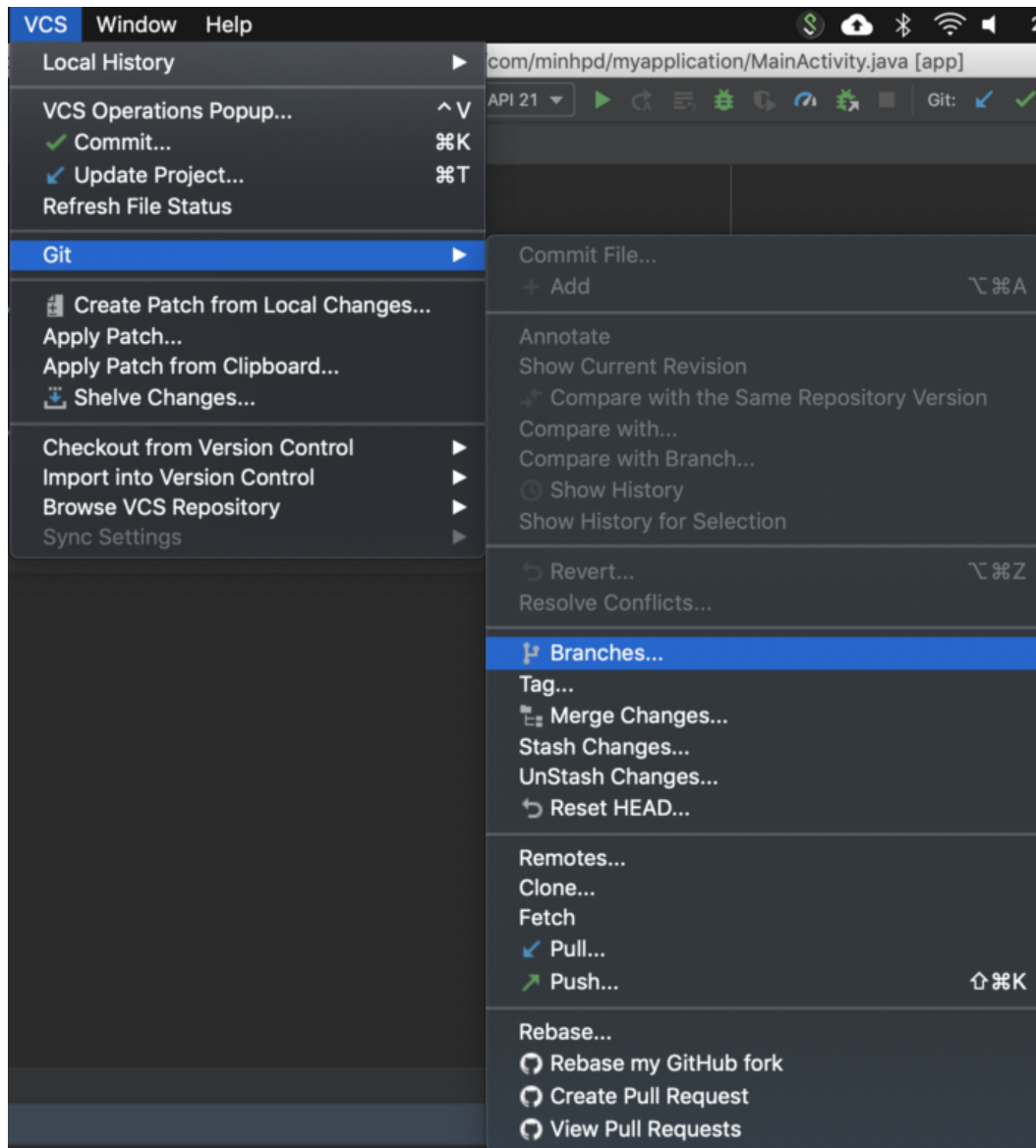
Android Studio nos presenta un listado de archivos no versionados con un check junto a cada archivo y un check para seleccionar todos los archivos. Para realizar el commit debemos agregar un mensaje descriptivo (al igual que en Git directo o GitHub) y dar click al botón “Commit”.

Con la configuración por defecto, Android Studio ejecutará análisis de código y luego realizará el commit. Podemos escoger revisar el análisis de código o podemos simplemente ignorar esto para continuar con el commit. Si todo ha funcionado bien, Android Studio nos indicará con un mensaje indicando que el commit ha sido exitoso.

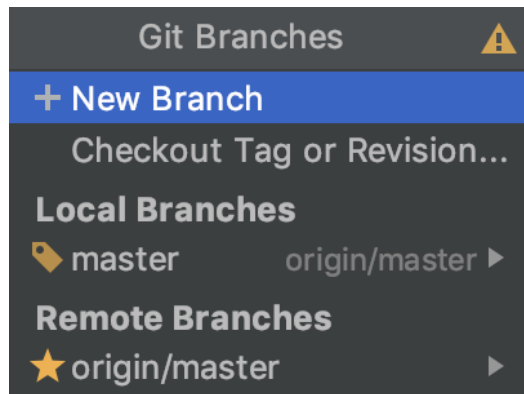
Ahora podemos “empujar” (push) nuestros cambios, para esto seleccionar en el menú VCS -> Git -> Push. Android Studio mostrará una ventana que despliega el commit a ser enviado al repositorio en la cual podemos confirmar el push.



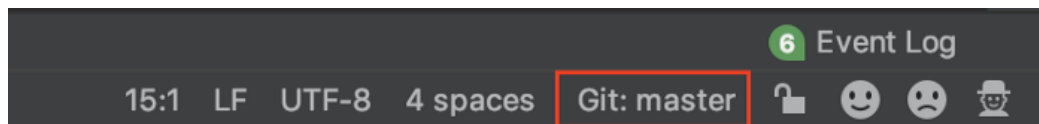
Android Studio también nos permite trabajar con branches (ramas) en sus opciones de VCS, en las opciones de menú ir a VCS -> Git -> Branches.



Esta opción nos, dentro de otras, nos permite crear nuevos branches (ramas). Muestra también todos las ramas locales y remotas (en el repositorio).



Podemos ver la rama actual de trabajo en Android Studio, la información se encuentra en la sección inferior derecha.



Para más detalles sobre la utilización de las herramientas VCS de Android Studio, por favor consultar la documentación oficial:

<https://developer.android.com/studio/intro?hl=es-419>

UNIDAD 6

Ciclo de vida de Android

Hemos estudiado algunos puntos importantes sobre cómo crear apps, ya conocimos uno de los elementos centrales del desarrollo Android, como lo son las activities, lógicamente la mayoría de las aplicaciones necesitan trabajar con más de una activity y necesitan mantener un control sobre como la UI va interactuando con el usuario.

Intents

Una **Intent** es un objeto de mensajería que podemos usar para solicitar una acción de otro componente de una app. Si bien las intents facilitan la comunicación entre componentes de varias formas, existen tres casos de uso principales:

- Iniciar una activity

Una Activity representa una única pantalla en una aplicación. Podemos iniciar una nueva instancia de una Activity pasando una Intent a `startActivity()`. La Intent describe la actividad que se debe iniciar y contiene los datos necesarios para ello.

Si deseamos recibir un resultado de la actividad cuando finalice, llamamos a `startActivityForResult()`. La actividad recibe el resultado como un objeto Intent separado en la devolución de llamada de `onActivityResult()` de la actividad.

- Iniciar un servicio

Un Service es un componente que realiza operaciones en segundo plano sin una interfaz de usuario. Con Android 5.0 (nivel de API 21) y versiones posteriores, podemos iniciar un servicio con `JobScheduler`. Para obtener más información acerca de `JobScheduler`, consulta su API en la documentación oficial.

En las versiones anteriores a Android 5.0 (nivel de API 21), podemos iniciar un servicio usando métodos de la clase `Service`. Podemos iniciar un servicio para realizar una operación única (como descargar un archivo) pasando una Intent a `startService()`. La Intent describe el servicio que se debe iniciar y contiene los datos necesarios para ello.

Si el servicio está diseñado con una interfaz cliente-servidor, podemos establecer un enlace con el servicio de otro componente pasando una Intent a `bindService()`. Para obtener más información, consulta la documentación oficial.

- Transmitir una emisión

Una emisión es un aviso que cualquier aplicación puede recibir. El sistema transmite varias emisiones de eventos, como cuando se inicia el sistema o comienza a cargarse el dispositivo. Podemos transmitir una emisión a otras apps pasando una Intent a `sendBroadcast()` o `sendOrderedBroadcast()`.

Existen dos tipos de intents:

- Las intents explícitas especifican qué aplicación las administrará, ya sea incluyendo el nombre del paquete de la app de destino o el nombre de clase del componente completamente calificado. Normalmente, el usuario usa una intent explícita para iniciar un componente en su propia aplicación porque conoce el nombre de clase de la actividad o el servicio que desea iniciar. Por ejemplo, podemos utilizarla para iniciar una actividad nueva en respuesta a una acción del usuario o iniciar un servicio para descargar un archivo en segundo plano.

- Las intents implícitas no nombran el componente específico, pero, en cambio, declaran una acción general para realizar, lo cual permite que un componente de otra aplicación la maneje. Por ejemplo, si deseamos mostrar al usuario una ubicación en un mapa, podemos usar una intent implícita para solicitar que otra aplicación apta muestre una ubicación específica en un mapa.

Cuando usamos una intent implícita, el sistema Android busca el componente apropiado para iniciar comparando el contenido de la intent con los filtros de intents declarados en el archivo de manifiesto (Manifest.xml) de otras aplicaciones en el dispositivo. Si la intent coincide con un filtro de intents, el sistema inicia ese componente y le entrega el objeto Intent. Si varios filtros de intents son compatibles, el sistema muestra un cuadro de diálogo para que el usuario pueda elegir la aplicación que se debe usar.

Un filtro de intents es una expresión en el archivo de manifiesto de una aplicación que especifica el tipo de intent que el componente podría recibir. Por ejemplo, declarar un filtro de intents para una actividad permite que otras aplicaciones la inicien directamente con un tipo de intent específico. Asimismo, si no declaras ningún filtro de intent para una actividad, esta solo se puede iniciar con una intent explícita.

Un objeto **Intent** tiene información que el sistema Android usa para determinar qué componente debe iniciar (como el nombre exacto del componente o la categoría que debe recibir la intent), además de información que el componente receptor usa para realizar la acción correctamente (por ejemplo, la acción que debe efectuar y los datos en los que debe actuar).

La información principal que contiene una Intent es la siguiente:

- Nombre del componente

Esto es opcional, pero es la información clave que hace que una intent sea explícita, lo que significa que la intent debe enviarse solamente al componente de la aplicación definido en el nombre del componente. Sin un nombre de componente, la intent es implícita y el sistema decide qué componente debe recibir la intent conforme la información restante que esta contiene (como la acción, los datos y la categoría, que se describen a continuación). Por lo tanto, si necesitamos iniciar un componente específico en tu aplicación, debes especificar el nombre del componente.

Este campo de la Intent es un objeto `ComponentName` que puedes especificar con un nombre de clase completamente calificado del componente de destino, incluido el nombre del paquete de la aplicación, como `com.example.ExampleActivity`. Puedes establecer el nombre del componente con `setComponent()`, `setClass()`, `setClassName()` o el constructor `Intent`.

- Acción

Una string que especifica la acción genérica que se debe realizar (como ver o elegir).

En el caso de la intent de una emisión, es la acción que se produjo y que se está registrando. La acción determina cuál es la estructura del resto de la intent, especialmente la información que se incluye en los datos y extras.

Podemos especificar las acciones para que las usen las intents en la aplicación (o para que las usen otras aplicaciones a fin de invocar componentes en tu aplicación); pero, usualmente, debemos especificar acciones constantes definidas por la clase Intent u otras clases de marcos de trabajo. Estas son algunas acciones comunes para iniciar una actividad:

[ACTION_VIEW](#)

Usamos esta acción en una intent con startActivity() cuando tengamos información que la actividad pueda mostrar al usuario, como una foto para ver en una app de galería o una dirección para ver en una app de mapas.

[ACTION_SEND](#)

También se conoce como la intent de compartir y debemos usarla en una intent con startActivity() cuando tengamos información que el usuario pueda compartir mediante otra app, como una app de correo electrónico o intercambio social.

- Datos

El URI (un objeto Uri) que hace referencia a los datos en los que se debe realizar la acción o el tipo de MIME de esos datos. El tipo de datos suministrado está generalmente determinado por la acción de la intent. Por ejemplo, si la acción es ACTION_EDIT, los datos deben contener el URI del documento que se debe editar.

Cuando creamos una intent, es importante especificar el tipo de datos (su tipo de MIME) además de su URI. Por ejemplo, una actividad que puede mostrar imágenes probablemente no sea capaz de reproducir un archivo de audio aunque los formatos de URI sean similares. Especificar el tipo de MIME de los datos ayuda al sistema Android a encontrar el mejor componente para recibir la intent. Sin embargo, el tipo de MIME a veces se puede deducir del URI, especialmente cuando los datos son un URI content:. Un URI content: indica que los datos se encuentran en el dispositivo y son controlados por un ContentProvider, lo que hace que el sistema pueda ver el tipo de datos de MIME.

- Categoría

Un string que contiene información adicional sobre el tipo de componente que la intent debe manejar. En una intent, se puede incluir la cantidad deseada de descripciones de categorías, pero la mayoría de las intents no requieren una categoría. Estas son algunas categorías comunes:

[CATEGORY_BROWSABLE](#)

La actividad de destino permite que la inicie un navegador web para mostrar datos a los que hace referencia un vínculo, como una imagen o un mensaje de correo electrónico.

CATEGORY LAUNCHER

La actividad es la actividad inicial de una tarea y está enumerada en el selector de la aplicación del sistema.

Las propiedades nombradas anteriormente (nombre de componente, acción, datos y categoría) representan las características definitorias de una intent. Mediante la lectura de estas propiedades, el sistema Android puede resolver qué componente de la aplicación debe iniciar. Sin embargo, una intent puede tener información adicional que no afecte cómo se resuelve en un componente de la aplicación. Una intent también puede incluir Extras o Indicadores .

Ejemplo de intent explícita

Una intent explícita es una intent que se usa para iniciar un componente específico de la aplicación, como una actividad o un servicio particular en la aplicación. Para crear una intent explícita, definimos el nombre de componente del objeto Intent (todas las otras propiedades de la intent son opcionales).

Por ejemplo, si creamos un servicio en tu app denominado DownloadService, diseñado para descargar un archivo de la Web, podemos iniciarlo con el siguiente código:

```
// Executed in an Activity, so 'this' is the Context
// The fileUrl is a string URL, such as "http://www.example.com/
image.png"
Intent downloadIntent = new Intent(this, DownloadService.class);
downloadIntent.setData(Uri.parse(fileUrl));
startService(downloadIntent);
```

El constructor Intent(Context, Class) proporciona Context a la aplicación; y el componente, un objeto Class. De esta manera, la intent inicia explícitamente la clase DownloadService en la app.

Ejemplo de intent implícita

Una intent implícita especifica una acción que puede invocar cualquier aplicación en el dispositivo que puede realizar la acción. El uso de una intent implícita es útil cuando la aplicación no puede realizar la acción, pero otras aplicaciones probablemente sí, y queremos que el usuario elija qué aplicación usar.

Por ejemplo, si tenemos contenido que queremos que el usuario comparta con otras personas, creamos una intent con la acción ACTION_SEND y agregamos extras que especifiquen el contenido para compartir. Cuando llamamos a startActivity() con esa intent, el usuario puede elegir una aplicación mediante la cual compartir el contenido.

```
// Create the text message with a string
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType("text/plain");

// Verify that the intent will resolve to an activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(sendIntent);
}
```

Cuando se llama a `startActivity()`, el sistema examina todas las aplicaciones instaladas para determinar cuáles pueden manejar este tipo de intent (una intent con la acción `ACTION_SEND` y que tiene datos de texto/sin formato). Si solo hay una aplicación que puede manejarla, esta se abre inmediatamente y se le entrega la intent. Si varias actividades aceptan la intent, el sistema muestra un cuadro de diálogo, como el que se muestra en la figura 2, para que el usuario pueda elegir la aplicación que se debe usar.

Una funcionalidad útil al usar intent es el paso de parámetros que se adjuntan al intent generado y pueden ser capturados en la activity receptora. Para esto usamos el método `putExtra()`, el cual nos permite agregar parámetros al intent antes de ejecutar el método `startActivity()`, por ejemplo:

```
Intent intent = new Intent(getApplicationContext(), SignoutActivity.class);
intent.putExtra("EXTRA_SESSION_ID", sessionId);
startActivity(intent);
```

En el ejemplo anterior hemos incorporado un parámetro usando el método `putExtra()`, el parámetro `"EXTRA_SESSION_ID"` que corresponde al `sessionId`. Podemos aprovechar el método `putExtra` para agregar todos los parámetros que sean necesarios, solo debemos tener cuidado de mantener el nombre del parámetro (`Extra_Session_Id` en el ejemplo) único para evitar errores.

Luego en la activity receptora del Intent podemos recuperar el parámetro:

```
String sessionId = getIntent().getStringExtra("EXTRA_SESSION_ID");
```

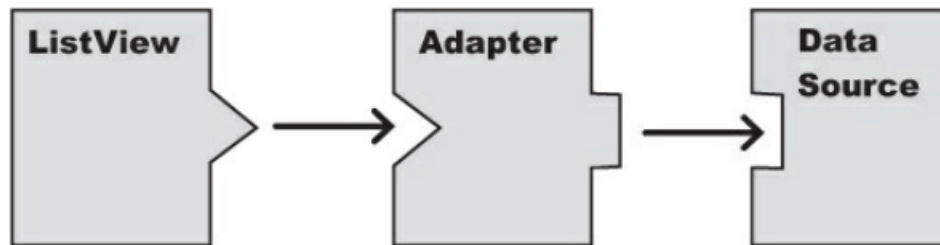
Adaptadores Android

Qué son los adaptadores, Adaptador para fragmentos, Historia de los adaptadores y porqué surge el reciclador, Adaptador reciclador para listas

Un adaptador **adapter** es un objeto que implementa la interface `Adapter`. Actúa como vínculo entre un conjunto de datos y una vista. Los datos pueden ser cualquier cosa que presente data de una forma estructurada, como, por ejemplo: Arreglos, Listas de Objetos y Cursores.

Un adaptador es responsable de recuperar los datos desde el conjunto de datos y generar el objeto vista basado en los datos. El objeto vista generado es usado para poblar cualquier *vista de adaptador* que esté ligada al adaptador.

Podemos crear nuestras propias clases adaptadores, pero comúnmente se extienden clases adaptadores incluidas en el SDK de Android, tales como ArrayAdapter o SimpleCursorAdapter.



Funcionamiento

Las vistas de adaptadores pueden desplegar grandes conjuntos de datos de forma eficiente. Por ejemplo, las vistas ListView y GridView pueden desplegar millones de ítems sin mayores problemas y manteniendo el uso de memoria relativamente bajo. Se aplican algunas estrategias como las siguientes:

- Se cargan solo las vistas de objetos que se encuentran efectivamente en la pantalla o que están a punto de pasar a esta. De esta forma el consumo de memoria se mantiene consistente e independiente del tamaño del conjunto de datos.
- Se permite a los desarrolladores minimizar costosos usos de operaciones de layouts y reciclar existentes vistas de objetos que se mueven fuera de pantalla, bajando el uso del procesador.

Para crear un adaptador necesitamos un conjunto de datos y un archivo de recursos conteniendo el layout de los objetos vista a usar. Consideremos el siguiente conjunto de datos (Array)

```
String[] cheeses = {  
    "Parmesan",  
    "Ricotta",  
    "Fontina",  
    "Mozzarella",  
    "Cheddar"  
};
```

Creamos un archivo de recursos XML cuya raíz es un elemento LinearLayout y lo llamamos item.xml, a continuación, se incluye un ejemplo:

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

```

        android:orientation="vertical" android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:padding="@dimen/activity_horizontal_margin">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textAppearance="?android:attr/textAppearanceLarge"
            android:text="Large Text"
            android:id="@+id/cheese_name" />
    </LinearLayout>

```

En nuestra activity creamos una nueva instancia de la clase ArrayAdapter usando su constructor. Como argumentos pasamos el nombre del archivo de recursos, el identificador del TextView y una referencia al arreglo definido:

```

ArrayAdapter<String> cheeseAdapter =
    new ArrayAdapter<String>(this,
        R.layout.item,
        R.id.cheese_name,
        cheeses
    );

```

Para desplegar los elementos en una vista que incluya los ítems, podemos usar la vista ListView. Es necesario agregar esta vista a la activity y pasamos este ListView al método setContentView() para que se presente en la pantalla

```

ListView cheeseList = new ListView(this);
setContentView(cheeseList);

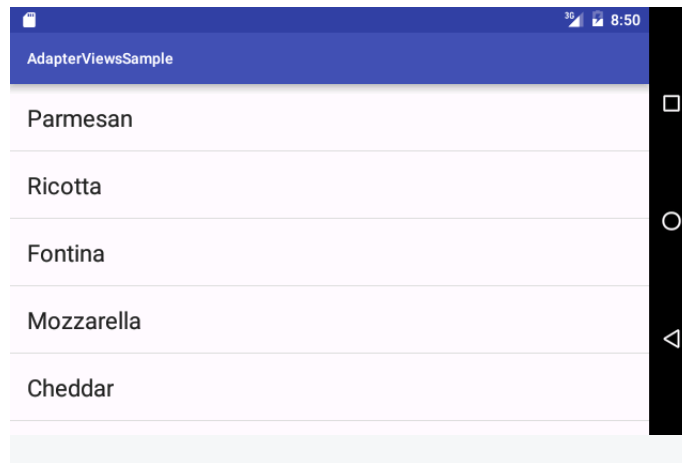
```

Para enlazar finalmente la vista ListView al adaptador creado anteriormente, debemos llamar al método setAdapter():

```

cheeseList.setAdapter(cheeseAdapter);

```



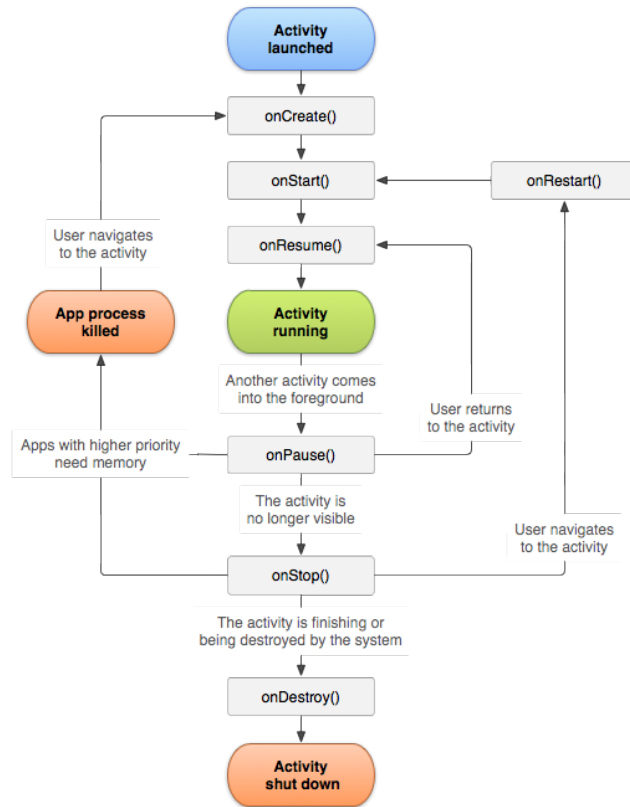
El ciclo de vida de Android

Uno de los temas más importantes para el desarrollo Android es comprender el *ciclo de vida*. Las activities pasan por un flujo establecido de estados que se ven reflejados en métodos que se van ejecutando, denominados *métodos de devolución*. Cada activity va reaccionando a la navegación del usuario, si este entra y sale de ella van ocurriendo diferentes partes del código de la activity. Esta situación también ocurre en otros elementos visuales como fragmentos (fragments).

Dentro de los métodos de devolución de llamada de ciclo de vida, podemos declarar el comportamiento que tendrá una actividad cuando el usuario la abandone y la reanude. Por ejemplo, si creamos un reproductor de video en streaming, podemos pausar el video y cancelar la conexión de red cuando el usuario cambia a otra app. Cuando el usuario vuelve, podemos volver a establecer la conexión con la red y permitir que el usuario reanude el video desde el mismo punto, controlando estas acciones. En otras palabras, cada devolución de llamada nos permite realizar un trabajo específico que es apropiado para un cambio de estado en particular. Hacer el trabajo preciso en el momento adecuado y administrar las transiciones correctamente hace que una app sea más sólida y eficiente. Por ejemplo, una buena implementación de las devoluciones de llamada de un ciclo de vida puede ayudar a garantizar que nuestra app:

- No falle si el usuario recibe una llamada telefónica o cambia a otra app.
- No consuma recursos valiosos del sistema cuando el usuario no la use de forma activa.
- No pierda el progreso del usuario si este abandona la app y regresa a ella posteriormente.
- No falle ni pierda el progreso del usuario cuando se gire la pantalla entre la orientación horizontal y la vertical.

Para navegar por las transiciones entre las etapas del ciclo de vida de una actividad, la clase Activity proporciona un conjunto básico de seis devoluciones de llamadas: onCreate(), onStart(), onResume(), onPause(), onStop() y onDestroy(). El sistema invoca cada una de estas devoluciones de llamada cuando una operación entra en un nuevo estado.



Cuando el usuario comienza a abandonar la actividad, el sistema llama a métodos para dismantlarla. En algunos casos, este dismantlamiento es solo parcial; la actividad todavía reside en la memoria (por ejemplo, cuando el usuario cambia a otra app) y aún puede volver al primer plano. Si el usuario regresa a esa actividad, se reanuda desde donde la dejó. Con algunas excepciones, se restringen las apps para que no inicien actividades cuando se ejecutan en segundo plano.

La probabilidad de que el sistema finalice un proceso determinado, junto con las actividades que contiene, depende del estado de la actividad en ese momento. En Estado de actividad y expulsión de memoria.

Según la complejidad de una actividad, es probable que no necesite implementar todos los métodos del ciclo de vida. Sin embargo, es importante que comprendamos cada uno de ellos y que implementemos aquellos que necesitemos de acuerdo al caso.

- **onCreate():**

Esta devolución de llamada se activa cuando el sistema crea la actividad por primera vez. Cuando se crea la actividad, esta entra en el estado Created. En el método onCreate(), ejecutamos la lógica de arranque básica de la aplicación que debe ocurrir una sola vez en

toda la vida de la actividad. Por ejemplo, la implementación de onCreate() podría vincular datos a listas, asociar la actividad con un ViewModel y crear instancias de algunas variables de alcance de clase. Este método recibe el parámetro savedInstanceState, que es un objeto Bundle que contiene el estado ya guardado de la actividad. Si la actividad nunca existió, el valor del objeto Bundle es nulo.

En el siguiente ejemplo del método onCreate(), se muestra la configuración básica de la actividad, como declarar la interfaz de usuario (definida en un archivo XML de diseño o layout), definir las variables de miembro y configurar parte de la IU

```
TextView textView;

// some transient state for the activity instance
String gameState;

@Override
public void onCreate(Bundle savedInstanceState) {
    // call the super class onCreate to complete the creation of
    activity like
    // the view hierarchy
    super.onCreate(savedInstanceState);

    // recovering the instance state
    if (savedInstanceState != null) {
        gameState = savedInstanceState.getString(GAME_STATE_KEY);
    }

    // set the user interface layout for this activity
    // the layout file is defined in the project res/layout/
    main_activity.xml file
    setContentView(R.layout.main_activity);

    // initialize member TextView so we can manipulate it later
    textView = (TextView) findViewById(R.id.text_view);
}

// This callback is called only when there is a saved instance that is
previously saved by using
// onSaveInstanceState(). We restore some state in onCreate(), while
we can optionally restore
// other state here, possibly usable after onStart() has completed.
// The savedInstanceState Bundle is same as the one used in
```

```

onCreate().
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
    textView.setText(savedInstanceState.getString(TEXT_VIEW_KEY));
}

// invoked when the activity may be temporarily destroyed, save the
instance state here
@Override
public void onSaveInstanceState(Bundle outState) {
    outState.putString(GAME_STATE_KEY, gameState);
    outState.putString(TEXT_VIEW_KEY, textView.getText());

    // call superclass to save any view hierarchy
    super.onSaveInstanceState(outState);
}

```

- onStart()

Cuando la actividad entra en el estado Started, el sistema invoca esta devolución de llamada. La llamada onStart() hace que el usuario pueda ver la actividad mientras la app se prepara para que esta entre en primer plano y se convierta en interactiva. Por ejemplo, este método es donde la app inicializa el código que mantiene la IU.

Cuando la actividad pase al estado Started, cualquier componente que priorice el ciclo de vida vinculado al de la actividad recibirá el evento ON_START.

El método onStart() se completa muy rápido y, al igual que con el estado Created, la actividad no permanece en el estado Started. Una vez finalizada esta devolución de llamada, la actividad entra en el estado Resumed, y el sistema invoca el método onResume().

- onResume():

Cuando la actividad entra en el estado Resumed, pasa al primer plano y, a continuación, el sistema invoca la devolución de llamada onResume(). Este es el estado en el que la app interactúa con el usuario. La app permanece en este estado hasta que ocurre algún evento que la quita de foco. Tal evento podría ser, por ejemplo, recibir una llamada telefónica, que el usuario navegue a otra actividad o que se apague la pantalla del dispositivo.

Cuando se reanude la actividad, cualquier componente que priorice el ciclo de vida vinculado al de la actividad recibirá el evento `ON_RESUME`. Aquí es donde los componentes del ciclo de vida pueden habilitar cualquier funcionalidad que necesite ejecutarse mientras el componente esté visible y en primer plano, como, por ejemplo, iniciar una vista previa de la cámara.

Cuando se produce un evento de interrupción, la actividad entra en el estado `Paused` y el sistema invoca la devolución de llamada `onPause()`.

Si la actividad regresa al estado `Resumed` desde `Paused`, el sistema volverá a llamar al método `onResume()`. Por esta razón, debemos implementar `onResume()` para inicializar los componentes que lancemos en `onPause()` y tenemos que realizar otras inicializaciones que deban ejecutarse cada vez que la actividad entre en el estado `Resumed`.

A continuación, se incluye un ejemplo de un componente que prioriza el ciclo de vida que accede a la cámara cuando el componente recibe el evento `ON_RESUME`:

```
public class CameraComponent implements LifecycleObserver {  
  
    ...  
  
    @OnLifecycleEvent(Lifecycle.Event.ON_RESUME)  
    public void initializeCamera() {  
        if (camera == null) {  
            getCamera();  
        }  
    }  
  
    ...  
}
```

- `onPause()`:

El sistema llama a este método a modo de primera indicación de que el usuario está abandonando la actividad (aunque no siempre significa que está finalizando la actividad); esto indica que la actividad ya no está en primer plano (aunque puede seguir siendo visible si el usuario está en el modo multiventana). Utilizamos el método `onPause()` para pausar o ajustar las operaciones que no deben continuar (o que deben continuar con moderación)

mientras Activity se encuentra en estado Paused y que esperamos reanudar en breve. Hay varias razones por las que una actividad puede entrar en este estado. Por ejemplo:

- Algunos eventos interrumpen la ejecución de la app, como se describe en la sección `onResume()`. Este es el caso más común.
- En Android 7.0 (API nivel 24) o versiones posteriores, varias apps se ejecutan en el modo multiventana. Debido a que solo una de las apps (ventanas) tiene foco en cualquier momento, el sistema pausa todas las demás.
- Se abre una nueva actividad semitransparente (como un diálogo). Mientras la actividad siga siendo parcialmente visible, pero no esté en foco, se mantendrá pausada.

Cuando la actividad pase al estado de pausa, cualquier componente que priorice el ciclo de vida vinculado al ciclo de vida de la actividad recibirá el evento `ON_PAUSE`. Aquí es donde los componentes del ciclo de vida pueden detener cualquier funcionalidad que no necesite ejecutarse mientras el componente no esté en primer plano, como detener una vista previa de la cámara.

■ `onStop()`:

Cuando el usuario ya no puede ver la actividad, significa que ha entrado en el estado `Stopped`, y el sistema invoca la devolución de llamada `onStop()`. Esto puede ocurrir, por ejemplo, cuando una actividad recién lanzada cubre toda la pantalla. El sistema también puede llamar a `onStop()` cuando haya terminado la actividad y esté a punto de finalizar.

Cuando la actividad pase al estado `Stopped`, cualquier componente que priorice el ciclo de vida vinculado al de la actividad recibirá el evento `ON_STOP`. Aquí es donde los componentes del ciclo de vida pueden detener cualquier funcionalidad que no necesite ejecutarse mientras el componente no sea visible en la pantalla.

En el método `onStop()`, la app debe liberar o ajustar los recursos que no son necesarios mientras no sea visible para el usuario. Por ejemplo, una app podría pausar animaciones o cambiar de actualizaciones de ubicación detalladas a más generales. Usar `onStop()` en lugar de `onPause()` garantiza que continúe el trabajo relacionado con la IU, incluso cuando el usuario esté viendo la actividad en el modo multiventana.

También debemos utilizar `onStop()` para realizar operaciones de finalización con un uso relativamente intensivo de la CPU. Por ejemplo, si no encontramos un momento más oportuno para guardar información en una base de datos, puedes hacerlo en `onStop()`. Por ejemplo, a continuación, se muestra una implementación de `onStop()` que guarda los contenidos del borrador de una nota en el almacenamiento persistente:


```

@Override
protected void onStop() {
    // call the superclass method first
    super.onStop();

    // save the note's current draft, because the activity is stopping
    // and we want to be sure the current note progress isn't lost.
    ContentValues values = new ContentValues();
    values.put(NotePad.Notes.COLUMN_NAME_NOTE, getCurrentNoteText());
    values.put(NotePad.Notes.COLUMN_NAME_TITLE,
getCurrentNoteTitle());

    // do this update in background on an AsyncQueryHandler or
    equivalent
    asyncQueryHandler.startUpdate (
        mToken, // int token to correlate calls
        null,   // cookie, not used here
        uri,    // The URI for the note to update.
        values, // The map of column names and new values to
apply to them.
        null,   // No SELECT criteria are used.
        null    // No WHERE columns are used.
    );
}

```

- onDestroy:

Se llama a onDestroy() antes de que finalice la actividad. El sistema invoca esta devolución de llamada por los siguientes motivos:

1. La actividad está terminando (debido a que el usuario la descarta por completo o a que se llama a finish()).
2. El sistema está finalizando temporalmente la actividad debido a un cambio de configuración (como la rotación del dispositivo o el modo multiventana).

Cuando la actividad pase al estado Destroyed, cualquier componente que priorice el ciclo de vida vinculado al de la actividad recibirá el evento ON_DESTROY. Aquí es donde los componentes del ciclo de vida pueden recuperar cualquier elemento que se necesite antes de que finalice el objeto Activity.

En lugar de poner lógica en ese objeto para determinar por qué está finalizando la actividad, deberíamos utilizar un objeto ViewModel a fin de contener los datos de vista relevantes para

Activity. Si se va a recrear el objeto Activity debido a un cambio de configuración, no es necesario que ViewModel realice ninguna acción, ya que se conservará y se entregará a la siguiente instancia del objeto Activity. Si no se va a recrear el objeto Activity, entonces ViewModel tendrá el método `onCleared()`, en el que podrá recuperar cualquier dato que necesite antes de que finalice la actividad.

Podemos diferenciar estos dos casos con el método `isFinishing()`.

Si la actividad está terminando, `onDestroy()` es la devolución de llamada del ciclo de vida final que recibe la actividad. Si se llama a `onDestroy()` como resultado de un cambio de configuración, el sistema crea inmediatamente una nueva instancia de actividad y luego llama a `onCreate()` en esa nueva instancia en la nueva configuración.

La devolución de llamada `onDestroy()` debe liberar todos los recursos que aún no han sido liberados por devoluciones de llamada anteriores, como `onStop()`.

Patrón MVP

Qué son los patrones de diseño y por qué se usan, Patrón MVP en Android, Patrones de diseño más comunes en Android, Los contratos del patrón MVP.

Patrones de diseño

Los patrones de diseño nacen como una idea de recolectar un conjunto de las mejores prácticas aplicadas en desarrollo de software. Un patrón de diseño documenta la mejor solución a un problema de desarrollo, comúnmente y en sus principios, software orientado a objetos, hoy en día aplicado a muchos paradigmas y situaciones. Gracias a los trabajos de destacados autores, es común encontrar los patrones de diseño en catálogos que agrupan patrones similares. Destacan generalmente los trabajos del Gang Of Four, autores del comúnmente conocido catálogo de patrones GOF.

Debemos recurrir a patrones de diseño para encontrar una solución correcta y estudiada a un problema de diseño. En desarrollo de sistemas, una idea común es la de buscar “reutilizar” el trabajo, generalmente aplicamos esta idea cuando usamos librerías o frameworks, un buen desarrollador debe ampliar esta idea a reutilizar también la forma de resolver problemas, lo que genera un ahorro de esfuerzo y tiempo.

El patrón MVP

El patrón MVP es un patrón de arquitectura usado en el desarrollo de aplicaciones Android. El MVP (Model View Presenter) es un patrón derivado del MVC (Model View Controller), que nos permite separar de forma muy clara nuestras vistas de la lógica de nuestras aplicaciones.

Para entender MVP debemos conocer un poco más sobre MVC. El modelo–vista–controlador (MVC) es un patrón de arquitectura de software, que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el

modelo, la vista y el controlador, es decir, por un lado, define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

Por otra parte, Modelo–Vista–Presentador (MVP) es una derivación del patrón arquitectónico modelo–vista–controlador (MVC), y es utilizado mayoritariamente para construir interfaces de usuario. En MVP el presentador asume la funcionalidad del "hombre-intermedio". En MVP, toda lógica de presentación es colocada al presentador.

Ambos patrones se parecen bastante, pero tienen las siguientes diferencias:

1. En el MVC, el modelo notifica a la vista cualquier cambio que sufra el estado del modelo. La información puede pasarse en la propia notificación, o después de la notificación, la vista puede consultar el modelo directamente para obtener los datos actualizados. Por el contrario, en el MVP, la vista no sabe nada sobre el modelo y la función del presentador es la de mediar entre ambos, enlazando los datos con la vista.
2. En el modelo MVC, la vista tiende a tener más lógica porque es responsable de manejar las notificaciones del modelo y de procesar los datos. En el modelo MVP, esa lógica se encuentra en el presentador, haciendo a la vista "estúpida". Su única función es representar la información que el presentador le ha proporcionado.
3. En MVC, el modelo tiene lógica extra para interactuar con la vista. En el MVP, esta lógica se encontraría en el presentador.

Los componentes del patrón MVP son:

- Modelo: Esta capa de acceso a datos, tales como bases de datos o API remotas.
- Vista: Se encarga de mostrar los datos. Aquí se encontrarían nuestros Fragmentos y Vistas.
- Presentador: es la capa que provee datos desde la vista hasta el modelo. También maneja las tareas en segundo plano.

En Android, MVP es una forma de separar las tareas de segundo plano de las actividades, vistas o fragmentos para hacerlos independientes de la mayoría de los eventos relacionados al ciclo de vida de Android. De esta forma la aplicación es mas simple y fácil de mantener.

Android no nos ofrece de forma nativa la posibilidad de desarrollar nuestras aplicaciones bajo el patrón MVP, de hecho, viola mucho de sus principios básicos. Aun así, podemos llevar a cabo alguna aproximación para este fin. Vamos a ver un posible ejemplo de implementación en el que el usuario dispone de un formulario dónde puede introducir contactos.

- El usuario introduce un contacto y pulsa el botón "añadir contacto".
- El Presentador crea el objeto Contacto con los datos introducidos por el usuario y se lo pasa al Modelo para que lo introduzca en la base de datos.
- El Modelo inserta el contacto en la base de datos.

- Si todo funciona correctamente, el Presentador limpia el formulario y refresca la lista de contactos para que aparezca el que acaba de añadir. En caso de que se haya producido algún error, muestra una alerta con un mensaje de error.

La comunicación entre las capas se va a llevar a cabo mediante el uso de interfaces. Para este ejemplo necesitaremos cuatro interfaces.

- `ProvidedPresenterOps`: Operaciones a las cuales tiene acceso la Vista y son implementadas por el Presentador. Permiten que la Vista se comunique con el Presentador.
- `ProvidedModelOps`: Operaciones a las cuales tiene acceso el Presentador y son implementadas por el Modelo. Permiten que el Presentador se comunique con el Modelo.
- `RequiredViewOps`: Operaciones a las cuales tiene acceso el Presentador y son implementadas por la Vista. Permiten que el Presentador se comunique con la Vista.
- `RequiredPresenterOps`: Operaciones a las cuales tiene acceso el Modelo y son implementadas por el Presentador. Permiten que el Modelo se comunique con el Presentador.

Una vez tenemos definidas las capas de nuestro MVP, necesitamos instanciarlas e insertar las referencias necesarias, además de un mecanismo responsable de mantener el estado del Presentador y del Modelo durante el ciclo de vida de nuestro Fragmento o Actividad.

Para más información sobre MVP en Android, por favor consultar:

http://konmik.com/post/introduction_to_model_view_presenter_on_android/

Callbacks

Un callback es una función "A" que se usa como argumento de otra función "B". Cuando se llama a "B", ésta ejecuta "A". Para conseguirlo, usualmente lo que se pasa a "B" es el puntero a "A".

Esto permite desarrollar capas de abstracción de código genérico a bajo nivel que pueden ser llamadas desde una función definida en una capa de mayor nivel. Usualmente, el código de alto-nivel inicia con el llamado de alguna función, definida a bajo-nivel, pasando a esta un puntero, o un puntero inteligente (conocido como handler), de alguna función. Mientras la función de bajo-nivel se ejecuta, esta puede ejecutar la función pasada como puntero para realizar alguna tarea. En otro escenario, las funciones de bajo nivel registran las funciones pasadas como un handler y luego pueden ser usadas de modo asincrónico.

Estas funciones de callbacks tienen una gran variedad de usos. Por ejemplo, podría ser una función que lee un archivo de configuración y que asocia valores con opciones.

Hay un aspecto importante del framework para UI de Android que es importante entender, el framework maneja un hilo a la vez (single-threaded). Existe solo un hilo quitando eventos desde la cola de eventos para hacer que los *callbacks* de controles puedan pintar la vista.

Una consecuencia de esto es que no es necesario usar bloques coordinados para manejar los estados entre vista y controlador. Otra ventaja es la garantía de que cada evento se procesa por completo y en el orden en que se encolaron, esto simplifica la manera de programar. Por ejemplo,

cuando un componente requiere múltiples cambios en su estado, cada estado causa una petición correspondiente para repintar la pantalla y se garantiza que la actualización de la pantalla no comenzará hasta que se haya completado su procesamiento, realizado todas las actualizaciones y retornado el control.

Una tercera razón importante para tener en consideración es que al tener solo una cola manejando eventos, si el código detiene su procesamiento, la UI se congelará. Si se trata de un evento simple es correcto y posible manejar el procesamiento en el hilo principal. Pero, por otro lado, si se trata de eventos más complejos que, por ejemplo, requieren de un procesamiento externo o ejecutar una query pesada, la UI completa se congelará, generando una mala experiencia para el usuario.

Algunos elementos de la UI de Android, tales como botones, hacen uso de handlers y callbacks de forma simple, sin embargo muchos otros elementos también definen *listeners*. La clase vista define varios eventos y listeners, otras clases definen otros tipos especializados de eventos y proveen handlers para esos eventos que son significativos solo para esas clases. Eso permite a sus clientes personalizar el comportamiento de un elemento visual sin tener que extenderlo para generar subclases. Este se conoce como el *patrón callback* y es una excelente forma de que el programa maneja sus propias acciones externas y asíncronas. Ya sea respondiendo a un cambio en el estado de un servidor remoto o una actualización de un servicio basado en localización, la aplicación puede definir sus propios eventos y listeners para permitir a sus clientes el reaccionar.

UNIDAD 7

Testing y Android

Android incluye algunas herramientas especiales para facilitar el testing. BuildVariants es una de estas herramientas, nos permite mantener diferentes versiones de la aplicación, generadas desde el mismo proyecto. Ejemplos típicos de estas variantes son “debug” y “release”, generalmente las diferencias entre estas versiones se encuentran a nivel técnico, pero no a nivel funcional.

Con las variantes (build variants) tenemos la capacidad de definir código de testing para versiones específicas de la app, pero también compartir código entre las diferentes versiones.

Esto funciona especificando diferentes propiedades en el archivo Gradle.build. Podemos crear diferentes carpetas de código fuente para estas propiedades. Esto significa que podemos mantener código fuente que pertenece a una propiedad específica definida.

Cada variante de compilación representa una versión diferente de la app que podemos compilar. Por ejemplo, es posible que necesitemos compilar una versión de la app que sea gratuita, con contenido limitado, y una versión paga que incluya más contenido. También podemos compilar diferentes versiones de la app para diferentes dispositivos, según el nivel de API y otras variantes de dispositivos. No obstante, si deseamos compilar diferentes versiones según la ABI del dispositivo o la densidad de pantalla, podemos compilar varios APK.

Podemos crear y configurar tipos de compilación en el archivo build.gradle del nivel del módulo dentro del bloque android. Cuando creamos un módulo nuevo, Android Studio genera automáticamente los tipos de compilación de depuración y lanzamiento. Si bien el tipo de compilación de depuración no aparece en el archivo de configuración de compilación, Android Studio lo configura con debuggable true. Esta configuración permite depurar la app en dispositivos Android seguros y configura la firma del APK con un almacén de claves de depuración genérico.

Podemos agregar el tipo de compilación de depuración a la configuración si deseamos agregar o cambiar determinadas opciones de configuración. En el siguiente ejemplo, se especifica un applicationIdSuffix para el tipo de compilación de depuración y se configura un tipo de compilación "staging" (etapa de pruebas) que se inicializa con la configuración del tipo de compilación de depuración.

```
android {
    defaultConfig {
        manifestPlaceholders = [hostName:"www.example.com"]
        ...
    }
    buildTypes {
        release {
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-
android.txt'), 'proguard-rules.pro'
        }

        debug {
            applicationIdSuffix ".debug"
            debuggable true
        }

        /**
         * The `initWith` property allows you to copy configurations
from other build types,
         * then configure only the settings you want to change. This
one copies the debug build
         * type, and then changes the manifest placeholder and
application ID.
         */
        staging {
            initWith debug
            manifestPlaceholders = [hostName:"internal.example.com"]
            applicationIdSuffix ".debugStaging"
        }
    }
}
```

Para más información, por favor consultar la documentación oficial:

<https://developer.android.com/studio/build/build-variants>

Test unitario usando Junit y Mockito

JUnit es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java.

Las pruebas unitarias consisten en aislar una parte del código y comprobar que funcionan según la especificación. Son pequeños tests que validan el comportamiento de una pieza de código.

Algunas ventajas de las pruebas unitarias son:

- Las pruebas unitarias demuestran que la lógica del código está en buen estado y que funcionará en todos los casos.
- Aumentan la legibilidad del código y ayudan a los desarrolladores a entender el código base, lo que facilita hacer cambios más rápidamente.
- Los test unitarios bien realizados sirven como documentación del proyecto.
- Se realizan en pocos milisegundos, por lo que podrás realizar cientos de ellas en muy poco tiempo.

JUnit es un framework que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

```
class SomeClassTest {  
  
    @Test  
    fun thisIsATest() {  
        assertEquals(4, 2 + 2)  
    }  
  
    @Test  
    fun thisIsAnotherTest() {  
        assertEquals(8, 4 + 4)  
    }  
  
}
```

Podemos evaluar la lógica de la app con pruebas unitarios (o de unidades) locales si necesitamos ejecutar pruebas con mayor rapidez y no precisas la fidelidad y la confianza asociadas con la ejecución de pruebas en un dispositivo real. Con este enfoque, cumplimos con las relaciones de dependencia utilizando Robolectric o un marco de trabajo ficticio, como Mockito. Por lo general, los tipos de dependencias asociados con las pruebas determinan qué herramienta usar:

- Si tenemos dependencias en el framework de Android, en especial, aquellas que crean interacciones complejas, se recomienda incluir dependencias del framework con Robolectric.
- Si las pruebas tienen dependencias mínimas del framework de Android, o si dependen solo de sus propios objetos, es correcto incluir dependencias ficticias con un framework ficticio, como Mockito.

Mockito es un framework para “mocking” (gestión de objetos de tipo “burla”, utilizados para la realización de pruebas en un sistema). Mockito nos permite escribir pruebas con una API simple y limpia.

Mocking es un proceso utilizado en testing unitario cuando la unidad o pieza a ser probada tiene dependencias externas. El propósito de realizar mocking es aislar y enfocar las pruebas en la funcionalidad que se desea verificar y no en el comportamiento de las dependencias externas.

Existe como alternativa el uso de *stubs*. Los stubs proporcionan respuestas predefinidas a las llamadas realizadas durante la prueba, por lo general no responden en absoluto a nada fuera de lo programado para la prueba. Los stubs también pueden grabar información sobre llamadas, como un código de acceso de correo electrónico que recuerda que se debe dar la salida de los mensajes que se enviaron con ese correo, o tal vez sólo cuántos mensajes fueron enviados.

Un stub por tanto es útil cuando el software bajo pruebas va a llamar a métodos externos, y se quiere evitar dicha llamada pero se necesita una simulación de la misma, e incluso se puede necesitar obtener una respuesta.

Un mock va más allá que un stub, un mock indica algo como “espero que llames a este método y que si lo haces con un específico argumento de entrada, obtendrás una respuesta concreta”.

Test de integración usando espresso

Espresso

Podemos usar Espresso para escribir pruebas de la IU de Android concisas, eficaces y confiables. En el siguiente fragmento de código, se muestra un ejemplo de una prueba de Espresso:

```
@Test
public void greeterSaysHello() {
    onView(withId(R.id.name_field)).perform(typeText("Steve"));
    onView(withId(R.id.greet_button)).perform(click());
    onView(withText("Hello
```



```
Steve!")) .check(matches(isDisplayed()));  
}
```

La API principal es pequeña, predecible y fácil de aprender, pero podemos personalizarla. Las pruebas de Espresso exponen claramente las expectativas, las interacciones y las afirmaciones sin la distracción del contenido estándar, la infraestructura personalizada o los complicados detalles de implementación que se interponen en el camino.

Las pruebas de Espresso se ejecutan con una rapidez óptima. Permiten dejar atrás esperas, sincronizaciones, tiempos inactivos y sondeos, mientras manipulan y afirman en la IU cuándo están en reposo.

Cada vez que la prueba invoca a `onView()`, Espresso espera para realizar la acción o aserción de la IU correspondiente hasta que se cumplan las siguientes condiciones de sincronización:

- La cola de mensajes está vacía.
- No hay instancias de `AsyncTask` ejecutando una tarea.
- Todos los recursos inactivos definidos por el desarrollador están inactivos.

Al realizar estas comprobaciones, Espresso aumenta sustancialmente la probabilidad de que solo una acción o afirmación de la IU pueda ocurrir en un momento específico. Esta capacidad brinda resultados de prueba más confiables y seguros.

Configuración

Para evitar la fragilidad, se recomienda desactivar las animaciones del sistema en los dispositivos virtuales o físicos utilizados para las pruebas. En el dispositivo, en Configuración > Opciones para desarrolladores, desactiva las siguientes 3 configuraciones:

- Escala de animación de ventana
- Escala de animación de transición
- Escala de duración de animador

Para agregar dependencias de Espresso a tu proyecto, completa los siguientes pasos:

- Abre el archivo `build.gradle` de tu app. Por lo general, este no es el archivo `build.gradle` de nivel superior, sino `app/build.gradle`.
- Agrega las siguientes líneas dentro de las dependencias:

```
androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.0'  
androidTestImplementation 'androidx.test:runner:1.1.0'  
androidTestImplementation 'androidx.test:rules:1.1.0'
```

Para agregar el ejecutor de instrumentación, Agrega la siguiente línea en android.defaultConfig al mismo archivo build.gradle:

```
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
```

A continuación, un ejemplo de archivo Gradle:

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 28

    defaultConfig {
        applicationId "com.my.awesome.app"
        minSdkVersion 15
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner
            "androidx.test.runner.AndroidJUnitRunner"
    }
}

dependencies {
    androidTestImplementation 'androidx.test:runner:1.1.0'
    androidTestImplementation 'androidx.test.espresso:espresso-
core:3.1.0'
}
```

La API de Espresso alienta a los autores de pruebas a pensar en términos de lo que un usuario podría hacer mientras interactúa con la app: ubicar elementos de la IU e interactuar con ellos. Al mismo tiempo, el framework impide el acceso directo a las actividades y las vistas de la aplicación, ya que conservar esos objetos y realizar operaciones con ellos por fuera del subproceso de la IU suele causar gran inestabilidad en las pruebas. Por lo tanto, no veremos métodos como `getView()` y `getCurrentActivity()` en la API de Espresso. Podemos realizar operaciones en las vistas de forma segura mediante la implementación de tus propias subclases de `ViewAction` y `ViewAssertion`.

Estos son algunos de los componentes principales de Espresso:

- Espresso: punto de entrada a las interacciones con las vistas (a través de `onView()` y `onData()`). También expone las API que no están necesariamente vinculadas a ninguna vista, como `pressBack()`.

- ViewMatchers: colección de objetos que implementan la interfaz `Matcher<? super View>`. Puedes pasar uno o más de estos al método `onView()` para ubicar una vista dentro de la jerarquía de vistas actual.
- ViewActions: colección de objetos `ViewAction` que se pueden pasar al método `ViewInteraction.perform()`, como `click()`.
- ViewAssertions: colección de objetos `ViewAssertion` a los que se puede pasar el método `ViewInteraction.check()`. La mayoría de las veces, usarás la aserción de coincidencias, que utiliza un comparador de vistas para confirmar el estado de la vista seleccionada.

A continuación, un ejemplo:

```
// withId(R.id.my_view) is a ViewMatcher
// click() is a ViewAction
// matches(isDisplayed()) is a ViewAssertion
onView(withId(R.id.my_view))
    .perform(click())
    .check(matches(isDisplayed()));
```

En la gran mayoría de los casos, el método `onView()` toma un comparador que se espera que coincida con una sola vista dentro de la jerarquía de vistas actual. Los comparadores son eficaces y les resultarán familiares a quienes los hayan usado con Mockito o Junit.

A menudo, la vista deseada tiene un `R.id` único, y un comparador `withId` simple limitará la búsqueda de vistas. Sin embargo, hay muchos casos legítimos en los que no se puede determinar `R.id` al momento del desarrollo de la prueba. Por ejemplo, es posible que la vista específica no tenga un `R.id` o que el `R.id` no sea único. Esto puede hacer que las pruebas de instrumentación normales sean inestables y complejas en cuanto a su escritura, porque la forma normal de acceder a la vista (con `findViewById()`) no funciona. Por lo tanto, es posible que debamos acceder a los miembros privados de la actividad o el fragmento que conservan la vista, o bien buscar un contenedor con un `R.id` conocido y navegar a su contenido para llegar a la vista particular.

Espresso da una respuesta eficaz a este problema, ya que te permite acotar la vista empleando objetos `ViewMatcher` existentes o tus propios objetos personalizados.

Encontrar una vista por su `R.id` es tan simple como llamar a `onView()`:

```
onView(withId(R.id.my_view));
```

A veces, los valores de `R.id` se comparten entre varias vistas. Cuando eso sucede, si intentamos usar un `R.id` en particular, el sistema arroja una excepción, como `AmbiguousViewMatcherException`. El mensaje de excepción proporciona una representación de texto de la jerarquía de vistas actual, que podemos usar para buscar las vistas que coincidan con los `R.id` no únicos.

```
java.lang.RuntimeException:
    androidx.test.espresso.AmbiguousViewMatcherException
```

```
This matcher matches multiple views in the hierarchy: (withId: is
<123456789>)
```

```
...

+----->SomeView{id=123456789, res-
name=plus_one_standard_ann_button,
visibility=VISIBLE, width=523, height=48, has-focus=false, has-
focusable=true,
window-focus=true, is-focused=false, is-focusable=false,
enabled=true,
selected=false, is-layout-requested=false, text=,
root-is-layout-requested=false, x=0.0, y=625.0, child-count=1}
****MATCHES****
|
+----->OtherView{id=123456789, res-
name=plus_one_standard_ann_button,
visibility=VISIBLE, width=523, height=48, has-focus=false, has-
focusable=true,
window-focus=true, is-focused=false, is-focusable=true,
enabled=true,
selected=false, is-layout-requested=false, text=Hello!,
root-is-layout-requested=false, x=0.0, y=0.0, child-count=1}
****MATCHES****
```

Si observamos los distintos atributos de las vistas, es posible encontrar propiedades que se puedan identificar de forma única. En el ejemplo anterior, una de las vistas tiene el texto "Hello!". Podemos usar esto para acotar la búsqueda usando comparadores con combinaciones:

```
onView(allOf(withId(R.id.my_view), withText("Hello!")));
```

También podemos optar por no revertir ninguno de los comparadores:

```
onView(allOf(withId(R.id.my_view), not(withText("Unwanted"))));
```

Algunas consideraciones:

1. En una app que se comporta correctamente, todas las vistas con las que un usuario puede interactuar deben incluir texto descriptivo o tener una descripción del contenido. Si no podemos acotar una búsqueda con `withText()` o `withContentDescription()`, procuraremos tratarla como un error de accesibilidad.
2. Usa el comparador menos descriptivo que encuentre la vista que estás buscando. No incluyas demasiadas especificaciones, ya que esto obligará al framework a realizar más

trabajo del necesario. Por ejemplo, si una vista puede identificarse de forma única por su texto, no es necesario especificar que también se puede asignar desde `TextView`. Para muchas vistas, el `R.id` de la vista debería ser suficiente.

3. Si la vista de destino está dentro de una `AdapterView`, como `ListView`, `GridView` o `Spinner`, es posible que el método `onView()` no funcione. En estos casos, debes usar `onData()`.

Para más detalles sobre la utilización de Espresso, por favor consultar:

<https://developer.android.com/training/testing/espresso>

Otras alternativas para el testing

Existen otras ideas y alternativas para realizar testing en Android Studio. A continuación presentamos algunos de estos frameworks para el testing en Android:

1. Appium: es un framework para aplicaciones nativas e híbridas, usa el protocolo JSONWireProtocol internamente y Selenium Webdriver. Permite crear *scripts* en casi cualquier lenguaje, permitiendo no depende de alguna tecnología particular y beneficiando la compatibilidad. Es muy similar a Selenium.
2. Detox: es un framework JavaScript que se construye en al aplicación y permite testearla al ejecutarla. Permite trabajar con apps híbridas y es más rápido que Appium en su ejecución.
3. Calabash: es un framework de múltiple plataforma para automatizar la ejecución de pruebas en apps nativas e híbridas. Requiere el uso de una sintaxis muy simple que permite a usuarios no técnicos su utilización.
4. Robotium: en algún momento fue el framework más popular para el testing en Android. Es un framework de código libre que extiende a Junit y agrega muchos métodos útiles para el testing. Permite automatizar pruebas de caja negra para aplicaciones Android.

UNIDAD 8

Unidad 1

- **Zaman Khawar y Umrysh Cary** Developing Enterprise Java Applications with J2EE and UML [Libro].
- Wikipedia [En línea]. - https://en.wikipedia.org/wiki/Unified_Modeling_Language Unidad 2

Unidad 2

- **Haseman Chris** Android Essentials [Libro]
- Guía para Desarrolladores [En línea]. - 02 de 2021. - <https://developer.android.com/guide>

Unidad 3

- **Griffiths Dawn Griffiths & David** Head First Android Development [Libro]
- **Ableson Frank, Collins Charlie y Sen Robi** Unlocking Android [Libro]. – 2009
- Guía para Desarrolladores [En línea]. - 02 de 2021. - <https://developer.android.com/guide>

Unidad 4

- **Griffiths Dawn Griffiths & David** Head First Android Development [Libro]
- Guía para Desarrolladores [En línea]. - 02 de 2021. - <https://developer.android.com/guide>

Unidad 5

- **GitHub.com** GIT Handbook [En línea]. - <https://guides.github.com/introduction/git-handbook/>
- **Atlassian** GIT Guide [En línea]. - <https://www.atlassian.com/git>
- **Guía para Desarrolladores** [En línea]. - 02 de 2021. - <https://developer.android.com/guide>

Unidad 6

- **Griffiths Dawn Griffiths & David Head** First Android Development [Libro]
- **Haseman Chris** Android Essentials [Libro]
- **Guía para Desarrolladores** [En línea]. - 02 de 2021. - <https://developer.android.com/guide>

Unidad 7

- **Rogers Rick [y otros]** Android Application Development [Libro]
- **Griffiths Dawn Griffiths & David Head** First Android Development [Libro]
- **Guía para Desarrolladores** [En línea]. - 02 de 2021. - <https://developer.android.com/guide>