Juan Menduiña

Resumen:
Seminario de Lenguajes (Android+Kotlin).

Nota de Clase 1: Introducción a Android.

* **Dominio del mercado:** Android es el sistema operativo para dispositivos móviles más utilizado en el mundo.
* **Fundamentos:** Es un sistema operativo de código abierto basado en Linux, respaldado por Google y desarrollado por la Open Handset Alliance.
* **Desarrollo de aplicaciones:**
  * Se pueden crear aplicaciones para una amplia gama de dispositivos: smartphones, tablets, smartwatches, TVs y automóviles.
  * Los lenguajes de programación utilizados son **Java** y **Kotlin**.
  * El entorno de desarrollo (IDE) oficial y recomendado es **Android Studio**, que incluye el **SDK de Android** con todas las herramientas necesarias para compilar, depurar y simular aplicaciones.
* **Fragmentación:** Existe una gran variedad de versiones de Android activas en el mercado simultáneamente, lo que se conoce como fragmentación. Esto es un factor importante a considerar durante el desarrollo.
* **Publicación de aplicaciones:**
  * Las aplicaciones se distribuyen principalmente a través de la **Google Play Store**.
  * Los desarrolladores utilizan la **Google Play Developer Console** para subir, gestionar y publicar sus aplicaciones.
  * El proceso de publicación incluye configurar la ficha de la aplicación (nombre, descripción, gráficos) y gestionar diferentes versiones (producción, pruebas internas, beta, etc.).
* **Formatos de publicación:**
  * **APK (.apk):** El formato de paquete tradicional que contiene toda la aplicación.
  * **AAB (.aab - Android App Bundle):** El formato moderno y recomendado. Google Play lo utiliza para generar y servir APKs optimizados para la configuración de cada dispositivo, lo que resulta en descargas más pequeñas y seguras.
* **Análisis:** La Play Console ofrece estadísticas detalladas sobre el rendimiento de la aplicación, segmentadas por país, versión de Android, tipo de dispositivo, etc.

Juan Menduiña

Nota de Clase 2: Primeros Pasos con Android.

* **Configuración del SDK:**
  * El **SDK Manager** en Android Studio se usa para instalar y administrar las diferentes versiones de la API de Android (SDK Platforms).
  * No es necesario instalar las APIs de todas las versiones antiguas para las que quieras dar soporte.
  * Se recomienda instalar la "plataforma" y las "fuentes" (Sources) del nivel de API con el que se compilará la aplicación.
  * Opcionalmente, se pueden instalar las **Google APIs** para usar servicios de Google (Maps, etc.) y las imágenes del sistema (System Images) para emular diferentes dispositivos y versiones de Android.

* **Creación de un nuevo proyecto:**
  * El proceso comienza seleccionando una plantilla, como **"Empty Views Activity"**.
  * Se debe configurar el nombre de la aplicación, el nombre del paquete (un identificador único), la ubicación, el lenguaje (se recomienda **Kotlin**) y el **Minimum SDK** (la versión de Android más antigua que soportará la app).
  * Android Studio ofrece una ayuda (`help me choose`) para decidir el SDK mínimo, mostrando el porcentaje de dispositivos que se alcanzarán.

* **Interfaz de Android Studio (IDE):**
  * Se presenta la interfaz principal, incluyendo la barra de herramientas, la barra de navegación, la ventana del editor y la barra de estado.

* **Ejecución de la aplicación:**
  * Se utiliza el **AVD (Android Virtual Device) Manager** para crear y gestionar emuladores.
  * Una vez configurado un emulador, se puede ejecutar la aplicación directamente desde Android Studio con el botón "Run" (o Mayus + F10).

* **El archivo `AndroidManifest.xml`:**
  * Es un archivo **esencial** en toda aplicación de Android.
  * Define la estructura y metadatos de la aplicación para el sistema operativo.
  * Contiene:
    * El **identificador único** de la aplicación (package name).
    * El **ícono**, **título** y **tema** de la aplicación.
    * La declaración de los **componentes** de la aplicación (como Activities).
    * La definición de la actividad principal que se lanzará al iniciar la app.
    * La solicitud de **permisos** que la aplicación necesita (ej. acceso a Internet).
  * Se menciona la evolución del modelo de permisos: antes de Android 6 se aceptaban todos al instalar, y a partir de Android 6 se solicitan en tiempo de ejecución.

Nota de Clase 3: Introducción a las Views.

* **Conceptos Fundamentales:**
  * **View:** Es el bloque de construcción básico para todos los componentes de la interfaz de usuario en Android. Cada `View` ocupa un área rectangular en la pantalla y es responsable de dibujar y manejar eventos. Ejemplos: `TextView`, `Button`, `ImageView`.
  * **ViewGroup:** Es una `View` especial que puede contener otras `View` (y `ViewGroup`), permitiendo organizar y anidar componentes. Actúa como un contenedor para definir la estructura del layout. Ejemplos: `LinearLayout`, `ConstraintLayout`.
  * **Jerarquía de Vistas:** Las `View` se organizan en una estructura de árbol. Un `ViewGroup` es un nodo que tiene hijos (otras `View` o `ViewGroup`), y una `View` es una hoja.

* **Definición de la Interfaz:**
  * La forma recomendada de definir la interfaz de usuario es a través de **archivos XML de layout**, ubicados en el directorio `res/layout/`.
  * Esto separa la presentación de la lógica de la aplicación, facilitando el mantenimiento.

* **Atributos Comunes de las Vistas (en XML):**
  * `android:id`: Asigna un identificador único a una `View` para poder referenciarla desde el código.
  * `android:layout_width` y `android:layout_height`: Definen el ancho y alto de la `View`. Valores comunes son `wrap_content` (se ajusta al contenido) y `match_parent` (ocupa todo el espacio del padre).
  * `android:text`: Establece el texto para `View` como `TextView` o `Button`.
  * `android:src`: Especifica la fuente de una imagen para un `ImageView`.
  * `android:padding`: Define un espacio interno entre el borde de la `View` y su contenido.
  * `android:margin`: Define un espacio externo alrededor de la `View`, separándola de otros elementos.
  * `android:background`: Establece un color o drawable de fondo.

* **Acceso a las Vistas desde el Código (Kotlin):**
  * En una `Activity`, después de inflar el layout con `setContentView()`, se puede obtener una referencia a una `View` específica usando el método `findViewById()`.
  * Una vez obtenida la referencia, se puede interactuar con la `View` para leer o modificar sus propiedades (ej. cambiar un texto, asignar un listener de clic).

* **Unidades de Medida:**
  * **dp (Density-independent Pixels):** Unidad abstracta basada en la densidad física de la pantalla. Es la unidad recomendada para especificar las dimensiones de las `View` para que se vean de manera consistente en diferentes densidades de pantalla.
  * **sp (Scale-independent Pixels):** Similar a `dp`, pero también se escala según la preferencia de tamaño de fuente del usuario. Es la unidad recomendada para el tamaño del texto.

Nota de Clase 4: Intents.

* **¿Qué es un Intent?:** Un `Intent` es un objeto de mensajería que se puede usar para solicitar una acción a otro componente de una aplicación. Es el mecanismo principal de comunicación entre los componentes de Android (Activities, Services, Broadcast Receivers).

* **Usos Principales de los Intents:**
  1. **Iniciar una Activity:** Es el uso más común. Permite navegar de una pantalla (Activity) a otra.
  2. **Iniciar un Service:** Para iniciar una operación en segundo plano sin interfaz de usuario.
  3. **Entregar un Broadcast:** Para enviar un mensaje a todo el sistema, al que cualquier aplicación con un `BroadcastReceiver` adecuado puede reaccionar.

* **Tipos de Intents:**
  * **Intents Explícitos:** Especifican el componente exacto que debe recibir el intent (por su nombre de clase). Se usan principalmente para la navegación y comunicación *dentro* de tu propia aplicación, ya que conoces las clases de tus Activities.
    * **Ejemplo:** `val intent = Intent(this, DetalleActivity::class.java)`

  * **Intents Implícitos:** No especifican un componente. En su lugar, declaran una acción general que se debe realizar. El sistema Android busca qué aplicación(es) pueden manejar esa acción y presenta al usuario las opciones si hay más de una.
    * **Ejemplo:** Para abrir una página web, se usa la acción `ACTION_VIEW` y se pasa la URL. El sistema abrirá el navegador web por defecto.
    * **Ejemplo:** Para compartir contenido, se usa la acción `ACTION_SEND`. El sistema mostrará todas las apps que pueden compartir (email, redes sociales, etc.).

* **Estructura de un Intent:**
  * **Acción (Action):** La acción a realizar (ej. `ACTION_VIEW`, `ACTION_EDIT`, `ACTION_DIAL`).
  * **Datos (Data):** La URI de los datos sobre los que se actuará (ej. una URL, un número de teléfono, un contacto).
  * **Extras:** Un `Bundle` (conjunto de pares clave-valor) para pasar información adicional entre componentes. Es la forma más común de pasar datos de una `Activity` a otra.
    * Para añadir datos: `intent.putExtra("clave", "valor")`
    * Para leer datos en la `Activity` receptora: `intent.getStringExtra("clave")`

* **Resolución de Intents y `Intent-Filter`:**
  * Para que un componente pueda responder a un `Intent` implícito, debe declarar un `<intent-filter>` en el archivo `AndroidManifest.xml`.
  * Este filtro especifica los tipos de `actions`, `data` y `categories` que el componente está preparado para recibir.

* **Iniciar una Activity para obtener un resultado:**
  * Se puede iniciar una `Activity` y esperar a que devuelva un resultado a la `Activity` original.
  * Esto es útil, por ejemplo, para que el usuario seleccione un contacto o tome una foto y la app principal reciba esa información. Se utilizan las APIs de `ActivityResultLauncher`.

Nota de Clase 5: Ciclo de Vida de una Activity.

* **Concepto:** Una `Activity` pasa por diferentes estados a lo largo de su existencia. El sistema operativo notifica estos cambios a través de una serie de métodos de *callback* que podemos sobrescribir para ejecutar código en momentos clave.

* **Estados Principales:**
  * **Created:** La actividad se está creando.
  * **Resumed (Running):** La actividad está en primer plano y tiene el foco del usuario. Es totalmente visible e interactiva.
  * **Paused:** La actividad está parcialmente visible, pero otra actividad (posiblemente semitransparente o una llamada entrante) tiene el foco. La actividad en pausa sigue "viva".
  * **Stopped:** La actividad está completamente oculta por otra actividad. Sigue "viva" en memoria, pero no está visible.
  * **Destroyed:** La actividad ha sido eliminada de la memoria, ya sea porque el usuario la cerró (con el botón "Atrás") o porque el sistema necesitó liberar recursos.

* **Callbacks del Ciclo de Vida:**
  * `onCreate()`: Se llama una sola vez, cuando la actividad se crea por primera vez. Aquí se realiza la inicialización fundamental: inflar la UI, inicializar variables, etc.
  * `onStart()`: Se llama cuando la actividad está a punto de volverse visible para el usuario.
  * `onResume()`: Se llama cuando la actividad está a punto de empezar a interactuar con el usuario. Es el lugar ideal para iniciar animaciones o acceder a recursos exclusivos como la cámara.
  * `onPause()`: Se llama cuando la actividad pierde el foco. Es el lugar para detener tareas que consumen CPU, guardar datos no persistentes y liberar recursos exclusivos. Debe ser muy rápido.
  * `onStop()`: Se llama cuando la actividad ya no es visible. Aquí se pueden detener tareas más pesadas.
  * `onRestart()`: Se llama cuando una actividad que estaba en estado `Stopped` está a punto de volver a iniciarse.
  * `onDestroy()`: Se llama justo antes de que la actividad sea destruida. Es la última oportunidad para limpiar todos los recursos.

* **Pérdida y Recuperación de Estado (Destrucción-Creación):**
  * **El Problema:** Cuando ocurre un cambio de configuración (como rotar la pantalla o cambiar el idioma del dispositivo), el sistema **destruye y vuelve a crear** la `Activity` para recargar los recursos adecuados. Esto provoca que se pierdan los datos de estado almacenados en variables de instancia (como un contador en un `TextView`).
  * **La Solución:** Para guardar el estado transitorio de la UI a través de este proceso, se utilizan dos callbacks adicionales:
    * `onSaveInstanceState(outState: Bundle)`: El sistema llama a este método antes de destruir la actividad, dándonos la oportunidad de guardar datos (como el valor de un contador) en un objeto `Bundle`.
    * `onRestoreInstanceState(savedInstanceState: Bundle)` o `onCreate(savedInstanceState: Bundle)`: Después de que la actividad se vuelve a crear, podemos recuperar los datos guardados del `Bundle` en cualquiera de estos dos métodos para restaurar el estado de la UI.
  * **Importante:** `onSaveInstanceState` **no** se llama cuando el usuario cierra la app explícitamente (ej. con el botón "Atrás"), ya que no se espera que el estado se guarde. Su propósito es manejar la recreación inesperada de la `Activity`. Para guardar datos de forma permanente (persistente), se deben usar otras técnicas como `SharedPreferences` o bases de datos, generalmente en el método `onPause()`.

Nota de Clase 6: Layouts (1).

* **Introducción a los Layouts:**
  * Un Layout (o `ViewGroup`) es un contenedor que define la estructura visual de la interfaz de usuario. Se encarga de organizar las `View` hijas (como `Button`, `TextView`, etc.) en la pantalla.
  * Los layouts se definen comúnmente en archivos XML en la carpeta `res/layout/`.

* **LinearLayout:**
  * Es uno de los layouts más simples. Organiza a sus hijos en una única dirección, ya sea horizontal o vertical.
  * **Atributo clave:** `android:orientation`
    * `"vertical"`: Apila los elementos uno debajo del otro.
    * `"horizontal"`: Coloca los elementos uno al lado del otro.
  * **Peso (Weight):** El atributo `android:layout_weight` permite distribuir el espacio sobrante entre las `View` hijas. Es especialmente útil para crear interfaces flexibles que se adaptan a diferentes tamaños de pantalla. Para que funcione correctamente, la dimensión correspondiente (`layout_width` para orientación horizontal o `layout_height` para vertical) suele establecerse en `0dp`.

* **RelativeLayout:**
  * Permite posicionar a los elementos hijos en relación con otros elementos hermanos o con el `RelativeLayout` padre.
  * Es más flexible que `LinearLayout` para crear interfaces complejas sin necesidad de anidar múltiples layouts.
  * **Atributos clave para posicionamiento relativo:**
    * **Respecto al padre:** `android:layout_alignParentTop`, `android:layout_alignParentBottom`, `android:layout_centerInParent`, `android:layout_centerHorizontal`, etc.
    * **Respecto a otros elementos (hermanos):** `android:layout_toRightOf="@id/otro_id"`, `android:layout_above="@id/otro_id"`, `android:layout_below="@id/otro_id"`, `android:layout_alignTop="@id/otro_id"`, etc.

* **FrameLayout:**
  * Es el layout más simple de todos. Está diseñado para contener un único elemento hijo.
  * Si se añaden múltiples hijos, los apila uno encima del otro (como una pila de cartas), dibujando el último que se añadió en la parte superior.
  * Es ideal para mostrar un solo elemento, como una imagen o un fragmento, o para superponer vistas (por ejemplo, un botón de "play" sobre una imagen).

* **TableLayout:**
  * Organiza a sus hijos en filas y columnas, de manera similar a una tabla HTML.
  * Las filas se definen con el elemento `<TableRow>`, que a su vez contiene las `View` para cada celda.
  * No muestra bordes para las filas, columnas o celdas por defecto.

* **ScrollView:**
  * Es un `ViewGroup` especial que permite hacer *scroll* (desplazamiento) de su contenido si este es más grande que el espacio disponible en pantalla.
  * **Importante:** Solo puede tener **un único hijo directo**. Por lo tanto, si se necesita que varios elementos sean desplazables, deben agruparse dentro de un único layout (como un `LinearLayout`) que será el hijo directo del `ScrollView`.

Juan Menduiña

Nota de Clase 7: Layouts (2).

* **ConstraintLayout:**
  * **Concepto:** Es el layout más potente y flexible de Android. Permite crear interfaces de usuario complejas y adaptables con una jerarquía de vistas plana, lo cual es muy bueno para el rendimiento, ya que evita anidar layouts (un layout dentro de otro).
  * **Funcionamiento:** En lugar de posicionar las vistas en un orden fijo, se definen "restricciones" (constraints) para cada vista, que determinan su posición y tamaño en relación con otros elementos o con el layout padre.
  * **Tipos de Restricciones:**
    * **Posición:** Se puede restringir cualquier lado de una vista (`top`, `bottom`, `start`, `end`) al lado correspondiente de otra vista o del padre. (Ej: `app:layout_constraintTop_toBottomOf="@id/otro_id"`).
    * **Centrado:** Se pueden centrar vistas horizontal o verticalmente.
    * **Bias:** Permite ajustar la posición de una vista a lo largo de un eje cuando está restringida por ambos lados (ej. `app:layout_constraintHorizontal_bias="0.7"`).
    * **Ratio de Dimensión:** Permite que el tamaño de una vista mantenga una proporción específica (ej. `app:layout_constraintDimensionRatio="16:9"` para un video).
  * **Herramientas Auxiliares:**
    * **Guidelines:** Líneas de guía invisibles a las que se pueden restringir otras vistas.
    * **Chains (Cadenas):** Permiten controlar cómo se distribuye un grupo de vistas (horizontal o verticalmente). Estilos de cadena: `spread`, `spread_inside`, `packed`.

* **RecyclerView:**
  * **Concepto:** Es un `ViewGroup` diseñado para mostrar listas largas o grillas de datos de manera muy eficiente. Es la evolución moderna y recomendada de `ListView` y `GridView`.
  * **Eficiencia (Reciclaje de Vistas):** Su principal ventaja es que **recicla** las vistas. En lugar de crear una vista nueva para cada elemento de la lista (lo cual consumiría mucha memoria), `RecyclerView` reutiliza las vistas que ya no son visibles en la pantalla para mostrar los nuevos elementos que aparecen.
  * **Componentes Clave:**
    1. **LayoutManager:** Se encarga de posicionar los elementos. Los más comunes son:
       * `LinearLayoutManager`: Para listas verticales u horizontales.
       * `GridLayoutManager`: Para grillas.
    2. **Adapter (`RecyclerView.Adapter`):** Es el puente entre los datos (ej. una lista de objetos) y el `RecyclerView`. Sus responsabilidades son:
       * Crear las vistas para cada elemento (`onCreateViewHolder`).
       * Vincular los datos de un elemento específico a su vista (`onBindViewHolder`).
       * Informar al `RecyclerView` del número total de elementos (`getItemCount`).
    3. **ViewHolder (`RecyclerView.ViewHolder`):** Es un objeto que representa y almacena las referencias a las vistas de un único elemento de la lista (ej. el `TextView` y el `ImageView` de un item). Esto evita las costosas llamadas a `findViewById()` cada vez que se recicla una vista.

Nota de Clase 8: Recursos.

* **Concepto Central:** En Android, es una práctica fundamental externalizar los recursos de la aplicación (como textos, imágenes, colores, estilos y layouts) separándolos del código fuente. Esto permite mantener y modificar la apariencia y el contenido de la app de forma independiente a su lógica.

* **Ubicación:** Todos los recursos se organizan en subdirectorios dentro de la carpeta `res/` del proyecto.

* **Ventaja Principal (Adaptabilidad):** La principal ventaja de este sistema es que permite proporcionar **recursos alternativos** para diferentes configuraciones de dispositivo. Android selecciona automáticamente el recurso más adecuado en tiempo de ejecución. Esto se logra mediante **calificadores (qualifiers)** en los nombres de los directorios.
  * **Ejemplos de calificadores:**
    * **Idioma:** `values-es/` para textos en español, `values-fr/` para francés.
    * **Tamaño de pantalla:** `layout-large/` para tablets.
    * **Orientación:** `layout-land/` para la vista en horizontal.
    * **Densidad de píxeles:** `drawable-hdpi/`, `drawable-xhdpi/` para imágenes de diferente resolución.
    * **Versión de API:** `values-v21/` para aplicar recursos solo en Android 5.0 y superior.

* **Tipos de Recursos Comunes:**
  * **Strings (`res/values/strings.xml`):** Para todos los textos de la UI. Esencial para la traducción (internacionalización).
  * **Colors (`res/values/colors.xml`):** Para definir valores de color hexadecimales y reutilizarlos en toda la app.
  * **Dimensions (`res/values/dimens.xml`):** Para definir medidas (márgenes, paddings, tamaños de texto) y mantener la consistencia.
  * **Drawables (`res/drawable/`):** Para gráficos e imágenes. Pueden ser imágenes de mapa de bits (PNG, JPG), gráficos vectoriales (VectorDrawable) o formas definidas en XML (Shape Drawable).
  * **Mipmap (`res/mipmap/`):** Directorio específico para los íconos de la aplicación en diferentes densidades.
  * **Layouts (`res/layout/`):** Archivos XML que definen la estructura de la interfaz de usuario.

* **Acceso a los Recursos:**
  * **Desde XML:** Se utiliza la sintaxis `@`. Por ejemplo: `@string/app_name`, `@color/colorPrimary`, `@drawable/my_icon`.
  * **Desde el código (Kotlin/Java):** Se utiliza la clase `R` generada automáticamente. Por ejemplo: `R.string.app_name`, `R.color.colorPrimary`, `R.drawable.my_icon`.

* **Estilos y Temas (`res/values/styles.xml` o `themes.xml`):**
  * **Estilo (Style):** Es una colección de atributos de apariencia (color, tamaño, fuente, etc.) que se pueden aplicar a una `View`. Funciona de manera similar a las hojas de estilo CSS.
  * **Tema (Theme):** Es un estilo que se aplica a toda una `Activity` o a la aplicación completa desde el `AndroidManifest.xml`. Define la apariencia por defecto de todos los componentes.

Nota de Clase 9: Recursos de Archivo. Menús.

* **Recursos de Archivo (Raw):**
  * **Concepto:** Permiten incluir archivos arbitrarios (como archivos de texto, audio, video, bases de datos) en su formato original y sin procesar dentro de la aplicación.
  * **Ubicación:** Estos archivos se colocan en el directorio `res/raw/`.
  * **Acceso:** Se accede a ellos desde el código a través de la clase `R`, utilizando su ID de recurso (ej. `R.raw.nombre_archivo`). Se pueden leer como un `InputStream`.

* **Menús en Android:**
  * **Concepto:** Son un componente de UI fundamental para presentar acciones al usuario de una manera organizada.
  * **Definición:** La forma recomendada de definir un menú es a través de un **recurso XML** en el directorio `res/menu/`. Dentro del archivo, se usan las etiquetas `<menu>` y `<item>` para definir la estructura y los elementos individuales.
  * **Atributos de `<item>`:**
    * `android:id`: Identificador único para manejar el clic en el código.
    * `android:title`: El texto que verá el usuario.
    * `android:icon`: El ícono que se mostrará junto al texto.
    * `app:showAsAction`: Define cómo y cuándo debe aparecer el ítem en la App Bar (si hay espacio, si siempre debe estar en el menú desplegable, etc.).

* **Tipos de Menús:**
  1. **Menú de Opciones (Options Menu):**
     * Es el menú principal de una `Activity`.
     * Generalmente se muestra en la **App Bar (o Toolbar)** en la parte superior de la pantalla.
     * **Implementación:**
       * Se sobrescribe `onCreateOptionsMenu()` en la `Activity` para inflar (cargar) el recurso XML del menú.
       * Se sobrescribe `onOptionsItemSelected()` para detectar qué ítem fue presionado (usando su ID) y ejecutar la acción correspondiente.

  2. **Menú Contextual (Context Menu):**
     * Aparece como un menú flotante cuando el usuario realiza una **pulsación larga** sobre una `View` específica.
     * Se usa para ofrecer acciones que afectan directamente al elemento presionado (ej. "eliminar" o "editar" un ítem de una lista).
     * **Implementación:**
       * Se registra la `View` para el menú contextual con `registerForContextMenu(view)`.
       * Se sobrescribe `onCreateContextMenu()` para inflar el menú.
       * Se sobrescribe `onContextItemSelected()` para manejar los clics.

  3. **Menú Emergente (Popup Menu):**
     * Es un menú modal que se ancla a una `View` específica y aparece cuando el usuario interactúa con dicha `View` (normalmente con un clic normal).
     * **Implementación:** Se crea y se muestra programáticamente, generalmente dentro del `onClickListener` de la `View` que lo dispara.

Nota de Clase 10: Diálogos y Notificaciones.

* **Diálogos (Dialogs):**
  * **Concepto:** Son pequeñas ventanas que aparecen sobre la `Activity` actual para solicitar al usuario que tome una decisión o introduzca información. No llenan la pantalla.
  * **`AlertDialog`:** Es el tipo de diálogo más común. Puede contener:
    * Un título y un mensaje.
    * Hasta tres botones de acción (positivo, negativo, neutro).
    * Una lista de elementos seleccionables o un layout personalizado.
  * **`DialogFragment`:** Es la forma **recomendada** de gestionar diálogos. Es un `Fragment` que muestra un diálogo. Su principal ventaja es que maneja correctamente los eventos del ciclo de vida (como rotaciones de pantalla), evitando que el diálogo se cierre inesperadamente.
  * **Diálogos Especializados:** Android provee diálogos ya listos para tareas comunes, como `DatePickerDialog` (para seleccionar una fecha) y `TimePickerDialog` (para seleccionar una hora).

* **Toast:**
  * **Concepto:** Es un mensaje emergente muy simple que aparece en la pantalla por un corto período y luego se desvanece. No requiere interacción del usuario.
  * **Uso:** Ideal para notificar al usuario sobre una operación que ha finalizado, como "Mensaje enviado" o "Archivo guardado".
  * **Implementación:** Se crea con `Toast.makeText(context, "Mensaje", Toast.LENGTH_SHORT).show()`.

* **Snackbar:**
  * **Concepto:** Similar a un `Toast`, pero más moderno y funcional. Aparece en la parte inferior de la pantalla.
  * **Ventajas sobre Toast:**
    * Puede incluir un **botón de acción** opcional, como "Deshacer" después de eliminar un elemento.
    * El usuario puede descartarlo deslizándolo fuera de la pantalla.
  * **Uso:** Forma parte de la librería de Material Design y es preferido sobre `Toast` para notificaciones que pueden requerir una acción inmediata.

* **Notificaciones (Notifications):**
  * **Concepto:** Son mensajes que se muestran **fuera de la interfaz de usuario** de tu aplicación (en la barra de estado y en el panel de notificaciones) para informar al usuario sobre eventos importantes, incluso cuando no está usando la app.
  * **Canales de Notificación (Notification Channels):** A partir de Android 8.0 (API 26), todas las notificaciones **deben** asignarse a un canal. Los canales permiten a los usuarios agrupar notificaciones y controlar su comportamiento (sonido, vibración, importancia) de forma granular. Los canales se deben crear antes de poder enviar una notificación.
  * **Creación de una Notificación:**
    1. Se utiliza `NotificationCompat.Builder` para construir la notificación de manera compatible con versiones anteriores de Android.
    2. Se establecen sus propiedades: ícono pequeño (`setSmallIcon`), título (`setContentTitle`), texto (`setContentText`), prioridad (`setPriority`), etc.
    3. Se crea un `PendingIntent` para definir qué `Intent` se ejecutará cuando el usuario toque la notificación (generalmente, abrir una `Activity` específica).
    4. Se utiliza `NotificationManagerCompat` para mostrar la notificación.

Nota de Clase 11: Cámara, GPS y Gestión de Permisos.

* **Gestión de Permisos en Android:**
  * **Concepto:** Para acceder a datos sensibles (como la ubicación, contactos, cámara) o a ciertas funcionalidades del sistema, una aplicación debe solicitar los permisos correspondientes al usuario.
  * **Proceso de 2 Pasos:**
    1. **Declaración en el Manifest:** Se debe declarar el permiso en el archivo `AndroidManifest.xml` usando la etiqueta `<uses-permission>`. (Ej: `<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />`).
    2. **Solicitud en Tiempo de Ejecución (Runtime):** A partir de Android 6.0 (API 23), no basta con declarar el permiso. Se debe solicitar explícitamente al usuario mientras la app está en uso. El usuario puede conceder o denegar el permiso.
  * **Flujo de Solicitud en el Código:**
    * `ContextCompat.checkSelfPermission()`: Se usa para verificar si el permiso ya ha sido concedido.
    * `ActivityCompat.requestPermissions()`: Si el permiso no ha sido concedido, se usa este método para mostrar el diálogo de solicitud al usuario.
    * `onRequestPermissionsResult()`: Se sobrescribe este método en la `Activity` para recibir la respuesta del usuario (concedido o denegado) y actuar en consecuencia.

* **Acceso a la Cámara:**
  * **Método 1: Intent Implícito (Recomendado para casos simples):**
    * Se puede delegar la tarea de tomar una foto a cualquier aplicación de cámara instalada en el dispositivo.
    * Se utiliza un `Intent` con la acción `MediaStore.ACTION_IMAGE_CAPTURE`. Esto **no requiere solicitar el permiso de cámara** explícitamente en el Manifest.
    * Para recibir la foto tomada, se debe lanzar el `Intent` usando `ActivityResultLauncher` y procesar el resultado (la imagen en formato Bitmap) en el callback.
  * **Filtrado en Google Play:** Se puede usar la etiqueta `<uses-feature>` en el Manifest para indicar si la cámara es un requisito (`android:required="true"`) o no para la instalación de la app.

* **Acceso a la Ubicación (GPS):**
  * **Permisos:** Es un dato sensible y siempre requiere permisos.
    * `ACCESS_COARSE_LOCATION`: Para obtener una ubicación aproximada (a nivel de ciudad).
    * `ACCESS_FINE_LOCATION`: Para obtener la ubicación precisa del GPS.
  * **Google Play Services:** La forma **recomendada** de obtener la ubicación es usando las APIs de localización de los Servicios de Google Play, ya que son más precisas, eficientes en el consumo de batería y fáciles de usar.
  * **Implementación con Google Play Services:**
    1. Añadir la dependencia `play-services-location` en el archivo `build.gradle`.
    2. Declarar los permisos de ubicación en el `AndroidManifest.xml`.
    3. Solicitar los permisos al usuario en tiempo de ejecución.
    4. Utilizar la clase `FusedLocationProviderClient` para obtener la última ubicación conocida (`lastLocation`) o para solicitar actualizaciones periódicas (`requestLocationUpdates`).
    5. `LocationRequest`: Se usa para configurar la frecuencia y precisión de las actualizaciones de ubicación.
    6. `LocationCallback`: Se crea un callback para recibir las nuevas ubicaciones cuando se solicitan actualizaciones en tiempo real.

Nota de Clase 12: Almacenamiento de Datos.

* **Opciones de Almacenamiento en Android:** Android ofrece varias opciones para almacenar datos, cada una adecuada para diferentes tipos de información y necesidades de persistencia.

* **1. Almacenamiento Interno (Internal Storage):**
  * **Concepto:** Almacena datos privados de la aplicación directamente en la memoria interna del dispositivo.
  * **Privacidad:** Los archivos guardados aquí son privados para la aplicación que los creó y no son accesibles por otras aplicaciones ni por el usuario directamente (a menos que el dispositivo esté rooteado).
  * **Persistencia:** Los datos persisten incluso si la aplicación se cierra, pero se eliminan automáticamente cuando la aplicación es desinstalada.
  * **Uso:** Ideal para datos sensibles o que no deben ser compartidos.
  * **Métodos:** `openFileOutput()` y `openFileInput()` para escribir y leer archivos.

* **2. Almacenamiento Externo (External Storage):**
  * **Concepto:** Almacena datos en un almacenamiento compartido, como una tarjeta SD o una partición de almacenamiento interno accesible públicamente.
  * **Accesibilidad:** Los datos son accesibles por otras aplicaciones y por el usuario.
  * **Persistencia:** Los datos persisten incluso si la aplicación se desinstala.
  * **Permisos:** Requiere permisos de lectura (`READ_EXTERNAL_STORAGE`) y/o escritura (`WRITE_EXTERNAL_STORAGE`) en el `AndroidManifest.xml` y en tiempo de ejecución.
  * **Uso:** Para archivos que deben ser compartidos o que son grandes (fotos, videos).
  * **Consideraciones:** Se debe verificar la disponibilidad del almacenamiento externo antes de usarlo.

* **3. SharedPreferences:**
  * **Concepto:** Almacena colecciones de pares clave-valor de datos primitivos (booleanos, flotantes, enteros, largos, cadenas).
  * **Uso:** Ideal para guardar pequeñas cantidades de datos, como preferencias de usuario, configuraciones de la aplicación o el estado de la UI.
  * **Privacidad:** Los datos son privados para la aplicación.
  * **Métodos:** Se obtiene una instancia de `SharedPreferences` (generalmente con `getSharedPreferences()` o `getDefaultSharedPreferences()`), se usa un `Editor` para escribir datos (`put...()`) y se aplica (`apply()`) o se confirma (`commit()`) el cambio. Para leer, se usan los métodos `get...()`.

* **4. Bases de Datos SQLite:**
  * **Concepto:** Android incluye soporte para bases de datos SQLite, una base de datos relacional ligera y embebida.
  * **Uso:** Ideal para almacenar datos estructurados y complejos que requieren consultas y relaciones.
  * **Persistencia:** Los datos persisten y son privados para la aplicación.
  * **`SQLiteOpenHelper`:** Es una clase de ayuda que facilita la creación y actualización de la base de datos. Se encarga de abrir la base de datos, crear las tablas si no existen y gestionar las actualizaciones de esquema.
  * **Operaciones CRUD:** Se utilizan métodos como `insert()`, `query()`, `update()`, `delete()` en un objeto `SQLiteDatabase` para interactuar con la base de datos.

* **5. Content Providers:**

* **Concepto:** Proporcionan una interfaz estandarizada para acceder a datos almacenados en una aplicación. Permiten que otras aplicaciones accedan a los datos de tu aplicación de forma segura y controlada.
* **Uso:** Principalmente para compartir datos entre aplicaciones o para acceder a datos del sistema (contactos, calendario, etc.).
* **URI:** Los datos se identifican mediante URIs (Uniform Resource Identifiers).
* **Métodos:** Implementan los métodos `query()`, `insert()`, `update()`, `delete()` para manejar las operaciones de datos.