ING. SISTEMAS E INFORMÁTICA

APLICACIONES MÓVILES

Explora el mundo del desarrollo móvil y abre puertas a nuevas oportunidades

UNS
UNIVERSIDAD NACIONAL DEL SANTA

MS. MIRKO MANRIQUE RONCEROS

INTRODUCCIÓN

En el desarrollo de aplicaciones móviles Android, el diseño de interfaces de usuario intuitivas, funcionales y visualmente atractivas es un factor clave para garantizar una buena experiencia del usuario. Para ello, el sistema operativo Android proporciona una serie de componentes y herramientas especializadas que permiten construir dichas interfaces de manera flexible y eficiente. Entre estos elementos destacan las Views, las Views personalizadas (Customize Views), el RecyclerView y el CardView, cada uno con funciones específicas dentro del ecosistema visual de la aplicación.

Las **Views** constituyen la base de todos los elementos visuales en Android. Botones, textos, cuadros de entrada, imágenes, barras de progreso, entre otros, son todos ejemplos de Views. Cuando las necesidades del diseño exceden las capacidades de los componentes estándar, los desarrolladores pueden crear **Views personalizadas**, adaptando el comportamiento y la apariencia mediante la herencia de clases y la redefinición de métodos clave como onDraw() o onMeasure(). Esto permite, por ejemplo, construir un botón con animaciones especiales o un gráfico interactivo acorde a los requerimientos de una aplicación profesional.

Por otro lado, cuando se trata de mostrar listas extensas de datos o elementos repetitivos, el componente ideal es el **RecyclerView**. Esta clase moderna y altamente eficiente reemplaza al tradicional ListView, ofreciendo una arquitectura basada en el patrón ViewHolder que minimiza el uso de memoria y mejora el rendimiento. RecyclerView permite configurar el tipo de distribución (lineal, en cuadrícula o escalonada) y soporta la incorporación de animaciones, múltiples tipos de vistas y manejo de eventos en cada elemento. Su diseño modular facilita

además la separación de responsabilidades, fomentando la reutilización de código y la escalabilidad de la aplicación.

En complemento, el **CardView** es un contenedor visual que proporciona un marco atractivo con esquinas redondeadas y sombra, útil para presentar información agrupada en forma de tarjeta. Es comúnmente utilizado junto con RecyclerView para mejorar la estética y la jerarquía visual de los contenidos, como ocurre en aplicaciones de noticias, redes sociales o catálogos de productos. Su implementación sencilla y su compatibilidad con otros layouts lo convierten en una herramienta poderosa para enriquecer la experiencia visual del usuario.

En conjunto, estos tres elementos —Views, RecyclerView y CardView— permiten a los desarrolladores diseñar interfaces más robustas, modernas y centradas en el usuario, alineadas con los estándares actuales del desarrollo móvil y con proyección al despliegue en la nube.

I. VIEW Y CUSTOMIZE VIEWS

1) Definición de View en Android y su papel en la interfaz de usuario

- **Concepto**: En Android, una View es el bloque de construcción fundamental para cualquier elemento de la interfaz de usuario (UI). Representa un área rectangular en la pantalla y es responsable de dibujar contenido y manejar la interacción del usuario. Casi todo lo que el usuario ve e interactúa en una aplicación Android (textos, botones, imágenes, campos de entrada, etc.) es una instancia de la clase android.view.View o de una de sus subclases.

- **Papel en la Interfaz de Usuario**:
  - **Visualización**: Las Views son responsables de dibujar contenido en la pantalla. El sistema Android las organiza en una jerarquía de vistas (View Hierarchy), donde cada View se dibuja dentro de los límites de su padre (ViewGroup).
  - **Interacción**: Las Views son interactivas y pueden responder a eventos de entrada del usuario, como toques, clics, deslizamientos, y eventos del sistema. Proporcionan mecanismos para registrar listeners que se activan cuando ocurren estos eventos.
  - **Presentación de Datos**: Muchas Views se utilizan para mostrar datos al usuario (ej., TextView para texto, ImageView para imágenes).
  - **Entrada de Datos**: Otras Views permiten al usuario ingresar información (ej., EditText).
  - **Control de la Aplicación**: Views como Button y Switch permiten al usuario interactuar con la lógica de la aplicación.

  - **Contenedores**: Aunque técnicamente los ViewGroups (como LinearLayout, ConstraintLayout) son subclases de View, su papel principal es contener y organizar otras Views, definiendo la estructura del layout.
- **Atributos de las Views**: Todas las Views comparten un conjunto común de atributos (definidos en la clase android.view.View) que controlan su apariencia, tamaño, posición, visibilidad, comportamiento, etc. Estos atributos se pueden configurar en el archivo de layout XML o dinámicamente en el código.

2) Creación de Views Personalizadas Mediante la Extensión de Clases Base:

Android proporciona una amplia gama de Views predefinidas, pero en muchos casos, se necesita crear componentes de UI que tengan un comportamiento o una apariencia específicos. Esto se logra creando Views personalizadas extendiendo las clases base de View o sus subclases.

- **Clases Base para Extender**:
  - **android.view.View**: La clase base para todos los componentes visuales. Se extiende directamente cuando se necesita un control total sobre el dibujo y el comportamiento.
  - **Subclases de View**: Se pueden extender subclases existentes como TextView, ImageView, Button, etc., para modificar su comportamiento o apariencia predeterminados. Esto es útil cuando se necesita una ligera variación de un componente existente.
  - **android.view.ViewGroup**: Si la View personalizada necesita contener y organizar otras Views, se debe extender una subclase de ViewGroup (como LinearLayout, FrameLayout, o crear una subclase personalizada de ViewGroup).

- **Pasos para Crear una View Personalizada**:
  - **Crear una Nueva Clase**: Crear una nueva clase en Java o Kotlin que extienda la clase base apropiada (View o una de sus subclases).
  - **Constructores**: Implementar los constructores necesarios. Es importante proporcionar al menos los siguientes constructores para que la View personalizada pueda ser instanciada desde XML y desde código:

    Kotlin

    constructor(context: Context) : super(context)

    constructor(context: Context, attrs: AttributeSet?) : super(context, attrs)

    constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int) : super(context, attrs, defStyleAttr)

    constructor(context: Context, attrs: AttributeSet?, defStyleAttr: Int, defStyleRes: Int) : super(context, attrs, defStyleAttr, defStyleRes)
    - **context**: El contexto actual.
    - **attrs**: Un conjunto de atributos XML que se especificaron al instanciar la View desde un layout.
    - **defStyleAttr**: Una referencia a un estilo en el tema actual que contiene atributos predeterminados para la View.
    - **defStyleRes**: Una referencia a un estilo predeterminado dentro del tema de la aplicación (solo se usa si no se encuentra defStyleAttr).
  - **Sobrescribir onDraw(Canvas canvas):** Este método es donde se realiza el dibujo real de la View. Se utiliza el objeto Canvas proporcionado para dibujar formas, texto, imágenes, etc., utilizando objetos Paint para definir el color, estilo, fuente, etc.

- **Manejar Atributos Personalizados (Opcional):** Si se desea que la View personalizada tenga atributos específicos que se puedan configurar desde XML, se deben:
  - Declarar los atributos en un archivo attrs.xml dentro del directorio res/values/.
  - Obtener los valores de estos atributos en el constructor utilizando context.obtainStyledAttributes(attrs, R.styleable.MiVistaPersonalizada).
  - Utilizar los valores obtenidos para configurar la apariencia o el comportamiento de la View.
  - Reciclar el TypedArray utilizando typedArray.recycle() después de obtener los valores.
- **Manejar Eventos de Entrada (Opcional):** Sobrescribir métodos como onTouchEvent(MotionEvent event), onKeyDown(int keyCode, KeyEvent event), etc., para detectar y responder a las interacciones del usuario.
- **Medir la View (Opcional):** Sobrescribir onMeasure(int widthMeasureSpec, int heightMeasureSpec) para especificar el tamaño deseado de la View. Es importante tener en cuenta los "MeasureSpec" proporcionados por el padre, que indican las restricciones de tamaño.

3) Manejo de Eventos y Atributos Personalizados en Views

- **Manejo de Eventos:**
  - **onTouchEvent(MotionEvent event):** Proporciona información detallada sobre los eventos táctiles (presionar, mover, levantar). Se puede sobrescribir para implementar gestos personalizados o interacciones táctiles específicas.

- **onClickListener:** Un listener estándar para eventos de clic. Se puede establecer utilizando setOnClickListener().
- **Otros Listeners:** Dependiendo de la necesidad, se pueden usar otros listeners para eventos como toques largos (OnLongClickListener), eventos de foco (OnFocusChangeListener), eventos de teclado (OnKeyListener), etc.

- **Manejo de Atributos Personalizados:**
  - **Declaración en attrs.xml:**

    XML

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <declare-styleable name="MiVistaPersonalizada">
    <attr name="colorDeFondo" format="color" />
    <attr name="texto" format="string" />
    <attr name="tamanoTexto" format="dimension" />
  </declare-styleable>
</resources>
```

  - **Obtención de Atributos en el Constructor:**

    Kotlin

```
class MiVistaPersonalizada(context: Context, attrs: AttributeSet?) :
View(context, attrs) {
  private var colorFondo: Int = Color.RED
  private var textoAMostrar: String = ""
  private var tamanoTexto: Float = 12f

  private val paint = Paint(Paint.ANTI_ALIAS_FLAG)
```

```
    init {
        attrs?.let {
            val typedArray = context.obtainStyledAttributes(it,
R.styleable.MiVistaPersonalizada)
            colorFondo =
typedArray.getColor(R.styleable.MiVistaPersonalizada_colorDeFond
o, Color.RED)
            textoAMostrar =
typedArray.getString(R.styleable.MiVistaPersonalizada_texto) ?: ""
            tamanoTexto =
typedArray.getDimension(R.styleable.MiVistaPersonalizada_taman
oTexto, 12f)
            typedArray.recycle()
        }
        paint.color = colorFondo
        paint.textSize = tamanoTexto
    }
    override fun onDraw(canvas: Canvas) {
        super.onDraw(canvas)
        canvas.drawRect(0f, 0f, width.toFloat(), height.toFloat(), paint)
        paint.color = Color.WHITE
        canvas.drawText(textoAMostrar, 10f, height / 2f + tamanoTexto /
2f, paint)
    }
}
```

- **Uso en el Layout XML**:

```
<com.example.app.MiVistaPersonalizada
    android:layout_width="200dp"
```

```
android:layout_height="100dp"
app:colorDeFondo="#00FF00"
app:texto="¡Hola!"
app:tamanoTexto="20sp" />
```

- (Recuerda añadir el namespace xmlns:app="http://schemas.android.com/apk/res-auto" en el layout raíz).

4) **Ejemplos de Implementación de Views Personalizadas para Componentes Específicos**

a) **Indicador de Progreso Circular Personalizado**: Se puede crear una View que dibuje un círculo de progreso animado, con control sobre el color, grosor y animación:

Kotlin

```
class CircularProgressBar(context: Context, attrs: AttributeSet?) : View(context, attrs) {
  // ... (implementación para dibujar el círculo, el arco de progreso y la animación)
  override fun onDraw(canvas: Canvas) {
    // ... (dibujo utilizando Paint y Canvas)
  }
}
```

b) **Botón con Icono y Texto:** Se puede extender View o LinearLayout para crear un botón que combine un icono (dibujado o desde un recurso) y texto, con control sobre la posición del icono.

Kotlin

```
class IconButton(context: Context, attrs: AttributeSet?) : LinearLayout(context, attrs) {
  private val imageView: ImageView
  private val textView: TextView

  init {
    orientation = HORIZONTAL
    // ... (inflar layout o crear ImageView y TextView programáticamente)
    // ... (obtener y aplicar atributos personalizados para el icono y el texto)
  }
}
```

c) **Gráfico de Líneas Personalizado:** Se puede crear una View que dibuje un gráfico de líneas basado en un conjunto de datos, con control sobre los colores de las líneas, etiquetas de los ejes, etc.

Kotlin

```
class LineChartView(context: Context, attrs: AttributeSet?) : View(context, attrs) {
  private var dataPoints: List<Float> = emptyList()
  private val linePaint = Paint(Paint.ANTI_ALIAS_FLAG)
  private val textPaint = Paint(Paint.ANTI_ALIAS_FLAG)
```

```
    fun setData(data: List<Float>) {
        dataPoints = data
        invalidate() // Forzar un redibujo
    }

    override fun onDraw(canvas: Canvas) {
        // ... (lógica para dibujar los ejes, las líneas y las etiquetas)
    }
}
```

d) **Selector de Color Personalizado: Una View que permite al usuario seleccionar un color a través de un círculo de colores o un selector de espectro.**

Kotlin

```
class ColorPickerView(context: Context, attrs: AttributeSet?) : View(context, attrs) {
    // ... (lógica para dibujar el selector de color y manejar los eventos táctiles para la selección)
    override fun onTouchEvent(event: MotionEvent): Boolean {
        // ... (detectar la selección del color)
        return true
    }

    override fun onDraw(canvas: Canvas) {
        // ... (dibujar el selector de color)
    }
}
```

II. RECYCLERVIEW

1) Introducción a RecyclerView y su Ventaja sobre ListView

a) **Concepto**: muestra un conjunto de elementos de datos presentando un número limitado de Views a la vez y reutilizando aquellas que ya no son visibles para mostrar nuevos datos. Esto se conoce como el patrón ViewHolder.

b) **Ventajas sobre ListView:**

- **Rendimiento:** RecyclerView es significativamente más eficiente que ListView para mostrar grandes conjuntos de datos. La reutilización de Views evita la creación y destrucción constante de objetos View, lo que reduce el consumo de memoria y mejora la fluidez del desplazamiento.
- **Flexibilidad:** RecyclerView es más flexible que ListView en términos de la disposición de los elementos. Permite diferentes tipos de layouts (listas verticales u horizontales, cuadrículas, layouts personalizados) a través de los LayoutManagers**.**
- **Animaciones:** RecyclerView facilita la implementación de animaciones para insertar, eliminar o mover elementos en la lista**.**
- **Separación de Responsabilidades:** RecyclerView promueve una mejor separación de responsabilidades entre la presentación de los datos (el Adapter) y la disposición de los elementos (el LayoutManager).
- **Extensibilidad:** RecyclerView está diseñado para ser extensible y personalizable. Permite crear LayoutManagers y ItemAnimators personalizados para lograr comportamientos específicos.

2) Estructura de RecyclerView: Adapter, ViewHolder y LayoutManager

RecyclerView funciona en conjunto con tres componentes clave:

- **Adapter**: El Adapter es responsable de crear las Views para cada elemento de datos y de vincular los datos a esas Views. Actúa como un puente entre la fuente de datos (una lista, una base de datos, etc.) y el RecyclerView. El Adapter crea instancias de ViewHolders.
- **ViewHolder**: El ViewHolder es una clase que contiene las referencias a las Views dentro de un elemento de la lista (por ejemplo, un TextView para el título y un ImageView para la imagen). El patrón ViewHolder evita la búsqueda repetida de las Views dentro del layout de cada elemento, lo que mejora el rendimiento.
- **LayoutManager**: El LayoutManager es responsable de la disposición de los elementos dentro del RecyclerView. Define si los elementos se muestran en una lista vertical, una lista horizontal, una cuadrícula, o un layout personalizado. Android proporciona LinearLayoutManager (para listas), GridLayoutManager (para cuadrículas) y StaggeredGridLayoutManager.

3) Implementación de Listas Dinámicas con RecyclerView:

Para implementar una lista dinámica con RecyclerView, se deben seguir estos pasos:

a) Añadir RecyclerView al Layout: Añadir un RecyclerView al layout XML de la Activity o Fragment.

XML

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/mi_recycler_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

b) Crear un Adapter: Crear una clase que extienda RecyclerView.Adapter. Esta clase debe implementar los métodos onCreateViewHolder(), onBindViewHolder() y getItemCount().

- onCreateViewHolder(): Crea una nueva instancia de ViewHolder para cada elemento. Infla el layout del elemento y lo pasa al constructor del ViewHolder.
- onBindViewHolder(): Vincula los datos del elemento en la posición dada a las Views dentro del ViewHolder.
- getItemCount(): Devuelve el número total de elementos en el conjunto de datos.

Kotlin

```
class MiAdaptador(private val datos: List<String>) : RecyclerView.Adapter<MiAdaptador.MiViewHolder>() {

    class MiViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val textView: TextView = itemView.findViewById(R.id.texto_elemento)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MiViewHolder {
        val itemView = LayoutInflater.from(parent.context).inflate(R.layout.elemento_lista, parent, false)
        return MiViewHolder(itemView)
    }
```

```
    override fun onBindViewHolder(holder: MiViewHolder, position: Int)
{
    holder.textView.text = datos[position]
  }

  override fun getItemCount(): Int {
    return datos.size
  }
}
```

c) **Crear un ViewHolder**: Crear una clase que extienda RecyclerView.ViewHolder. Esta clase debe contener las referencias a las Views dentro del layout de cada elemento.

```
class MiViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
  val textView: TextView = itemView.findViewById(R.id.texto_elemento)
}
```

d) **Crear un LayoutManager:** Crear una instancia del LayoutManager deseado (por ejemplo, LinearLayoutManager para una lista vertical).

```
val layoutManager = LinearLayoutManager(this) // 'this' es el Contexto (Activity o Fragment)
```

e) **Configurar el RecyclerView:** Obtener una referencia al RecyclerView en el código, establecer el LayoutManager y el Adapter.

```
val recyclerView = findViewById<RecyclerView>(R.id.mi_recycler_view)
recyclerView.layoutManager = layoutManager
recyclerView.adapter = miAdaptador
```

4) Manejo de Eventos de Clic y Actualización de Datos en RecyclerView

- Manejo de Eventos de Clic: Los eventos de clic se manejan generalmente dentro del ViewHolder. Se puede establecer un OnClickListener en la View raíz del elemento o en Views específicas dentro del elemento:

```
class MiViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    val textView: TextView = itemView.findViewById(R.id.texto_elemento)

    init {
        itemView.setOnClickListener {
            val position = adapterPosition // Obtener la posición del elemento
            if (position != RecyclerView.NO_POSITION) {
                // Hacer algo con la posición o los datos del elemento
                Log.d("MiAdaptador", "Elemento clicado en la posición: $position")
            }
        }
    }
}
```

- **Actualización de Datos:** Para actualizar los datos en el RecyclerView, se debe modificar la fuente de datos (la lista, etc.) y luego notificar al Adapter sobre los cambios. El Adapter proporciona métodos como notifyDataSetChanged(), notifyItemInserted(), notifyItemRemoved(), notifyItemChanged() y otros para notificar los cambios de manera eficiente.

```
// Ejemplo de actualización de datos:
datos.add("Nuevo Elemento") // Añadir un nuevo elemento a la lista
adaptador.notifyItemInserted(datos.size - 1) // Notificar al adaptador que se ha insertado un nuevo elemento

datos.removeAt(0) // Eliminar el primer elemento
adaptador.notifyItemRemoved(0) // Notificar al adaptador que se ha eliminado un elemento
```

III. CARDVIEW

1) Concepto y Beneficios de Utilizar CardView en Interfaces de Usuario:

- **Concepto**: CardView es un FrameLayout con funcionalidades adicionales para mostrar contenido de manera visualmente destacada. Aplica un estilo consistente a sus hijos, típicamente con esquinas redondeadas y una sombra sutil, simulando la apariencia de una tarjeta física.
- **Beneficios de** Utilizar CardView:
  - Atractivo Visual: Las tarjetas proporcionan una forma organizada y visualmente agradable de presentar información, mejorando la estética general de la interfaz de usuario.
  - **Organización del Contenido**: Ayudan a agrupar información relacionada de manera lógica, facilitando la comprensión y el escaneo por parte del usuario.
  - **Énfasis Visual**: La sombra y las esquinas redondeadas ayudan a destacar el contenido de la tarjeta del resto de la interfaz.
  - **Consistencia**: Proporciona un estilo consistente que se puede aplicar fácilmente a diferentes partes de la aplicación.

- **Facilita la Interacción**: Las tarjetas pueden servir como contenedores para elementos interactivos, como botones o menús, proporcionando un área táctil clara.
- **Integración con Material Design**: CardView es un componente que se alinea con los principios de Material Design, contribuyendo a una experiencia de usuario moderna y coherente.

2) Diseño de Tarjetas Informativas Utilizando CardView:

Para diseñar tarjetas informativas con CardView, simplemente se añade el componente al layout XML y se colocan dentro de él los Views que contendrán la información.

- **Añadir la Dependencia:** Primero, asegúrate de tener la dependencia de la biblioteca androidx.cardview:cardview en tu archivo build.gradle (app):

  Gradle

  dependencies {

  implementation("androidx.cardview:cardview:1.0.0") // O la versión más reciente

  // ... otras dependencias ...

  }

  Luego, sincroniza tu proyecto con Gradle.

- **Usar CardView en el Layout XML:**

  XML

  <androidx.cardview.widget.CardView

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

  xmlns:app="http://schemas.android.com/apk/res-auto"

  android:layout_width="match_parent"

```
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    app:cardCornerRadius="4dp"
    app:cardElevation="4dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="16dp">

        <TextView
            android:id="@+id/titulo_tarjeta"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Título de la Tarjeta"
            android:textAppearance="?android:textAppearanceMedium"
            android:textStyle="bold" />

        <TextView
            android:id="@+id/descripcion_tarjeta"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="8dp"
            android:text="Descripción detallada de la información que
contiene la tarjeta." />

        <Button
```

```
        android:id="@+id/boton_accion"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="16dp"
        android:text="Acción" />

    </LinearLayout>

</androidx.cardview.widget.CardView>
```

- **En este ejemplo:**
  - **androidx.cardview.widget.CardView** es el contenedor principal**.**
  - **android:layout_width y android:layout_height** definen las dimensiones de la tarjeta.
  - **android:layout_margin añade espacio alrededor de la tarjeta.**
  - **app:cardCornerRadius** define el radio de las esquinas redondeadas.
  - **app:cardElevation** controla la elevación de la tarjeta, lo que afecta la sombra.
  - **Dentro del CardView,** se utiliza un LinearLayout para organizar el título, la descripción y un botón verticalmente**.**
  - **android:padding** dentro del LinearLayout añade espacio alrededor del contenido dentro de la tarjeta**.**

3) **Integración de CardView dentro de RecyclerView para Listas más Atractivas:**

CardView es una excelente opción para estilizar los elementos individuales de una lista mostrada con RecyclerView. En lugar de usar un layout simple para cada elemento, envolver el contenido de cada elemento en un CardView puede mejorar significativamente la apariencia de la lista.

- **Layout del Elemento del RecyclerView con CardView:**: Crea un layout XML para cada elemento del RecyclerView que utilice CardView como el contenedor raíz:.

XML

```
<androidx.cardview.widget.CardView
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_margin="4dp"
  app:cardCornerRadius="4dp"
  app:cardElevation="2dp">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="8dp">

    <TextView
      android:id="@+id/item_titulo"
      android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="Título del Item"
        android:textAppearance="?android:textAppearanceMedium"
        android:textStyle="bold" />

    <TextView
        android:id="@+id/item_descripcion"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="4dp"
        android:text="Descripción breve del item." />

  </LinearLayout>

</androidx.cardview.widget.CardView>
```

- **Usar este Layout en el Adapter del RecyclerView:**
  En el método onCreateViewHolder() de tu RecyclerView.Adapter, infla este layout (R.layout.item_layout_con_cardview en este ejemplo) para crear la ViewHolder de cada elemento:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MiViewHolder {
  val itemView = LayoutInflater.from(parent.context)
    .inflate(R.layout.item_layout_con_cardview, parent, false)
  return MiViewHolder(itemView)
}
```

El resto de la implementación del Adapter (como onBindViewHolder() y getItemCount()) seguiría la lógica normal para vincular los datos a las TextViews dentro de cada CardView.

4) Personalización de CardView: esquinas redondeadas, sombras y elevación:

CardView ofrece varios atributos XML para personalizar su apariencia

- **Esquinas Redondeadas (app:cardCornerRadius):**
  - Define el radio de las esquinas de la tarjeta.
  - Se especifica en unidades de densidad independiente (dp).
  - Ejemplo: app:cardCornerRadius="8dp"
- **Sombras (app:cardElevation y app:cardMaxElevation):**
  - app:cardElevation: Controla la elevación base de la tarjeta, lo que determina el tamaño de la sombra. Valores más altos producen sombras más grandes. Se especifica en dp.
  - app:cardMaxElevation: Define la elevación máxima que la tarjeta puede alcanzar dinámicamente (por ejemplo, al hacer clic). Se especifica en dp.
  - Para deshabilitar la sombra, se puede establecer app:cardElevation="0dp".
- **Color de Fondo de la Tarjeta (android:cardBackgroundColor):**
  - Permite cambiar el color de fondo de la tarjeta.
  - Se especifica con un valor de color (hexadecimal, recurso de color, etc.).
  - Ejemplo: android:cardBackgroundColor="@color/white"
- **Márgenes Internos del Contenido** (app:contentPadding, app:contentPaddingLeft, app:contentPaddingRight, app:contentPaddingTop, app:contentPaddingBottom):

  - Controlan el espacio entre los bordes de la CardView y su contenido hijo.
  - Se especifican en dp.

- **Usar Compatibilidad para Sombras en Versiones Antiguas (app:cardUseCompatPadding y app:cardPreventCornerOverlap):**
  - **app:cardUseCompatPadding="true":** Añade relleno (padding) alrededor de la tarjeta para compensar la falta de soporte nativo para sombras con elevación en versiones de Android anteriores a Lollipop (API 21).
- **app:cardPreventCornerOverlap="true":** Evita que el contenido en versiones anteriores a Lollipop se superponga con las esquinas redondeadas. Generalmente se debe establecer en true cuando se usa cardUseCompatPadding.

- **Ejemplo de Personalización**

  XML

```
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    app:cardBackgroundColor="#F0F0F0"
    app:cardCornerRadius="8dp"
    app:cardElevation="6dp"
    app:cardMaxElevation="12dp"
    app:cardUseCompatPadding="true"
    app:contentPadding="16dp">
```

```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Tarjeta Personalizada"
        android:textAppearance="?android:textAppearanceLarge" />

</androidx.cardview.widget.CardView>
```