Capitulo 5

UD05. MATERIAL VIEWS

Resultados de avaliación

RA1. Aplica tecnoloxías de desenvolvemento para dispositivos móbiles, e avalía as súas características e as súas capacidades.

RA2. Desenvolve aplicacións para dispositivos móbiles, para o que analiza e emprega as tecnoloxías e as librarías específicas.

Criterios de avaliación

- CA1.8 Realizáronse modificacións sobre aplicacións existentes.
- CA1.9 Utilizáronse emuladores para comprobar o funcionamento das aplicacións.
- CA2.1 Xerouse a estrutura de clases necesaria para a aplicación.
- CA2.4 Utilizáronse as clases necesarias para a conexión e a comunicación con dispositivos sen fíos.
- CA2.5 Utilizáronse as clases necesarias para o intercambio de mensaxes de texto e multimedia.
- CA2.8 Realizáronse probas de interacción entre o usuario e a aplicación para mellorar as aplicacións desenvolvidas a partir de emuladores.
- CA2.9 Empaquetáronse e despregáronse as aplicacións desenvolvidas en dispositivos móbiles reais.
- CA2.10 Documentáronse os procesos necesarios para o desenvolvemento das aplicacións

BC1. Análise de tecnoloxías para desenvolvemento de aplicacións en dispositivos móbiles.

- Contidos
- Contornos integrados de traballo.
- Módulos para o desenvolvemento de aplicacións móbiles.
- Emuladores.
- Estrutura dunha aplicación para dispositivo móbil.
- Modificación de aplicacións existentes.
- Uso do contorno de execución do administrador de aplicacións.
- Ferramentas e fases de construción.
- Técnicas de animación e son.
- Comunicacións: clases asociadas. Tipos de conexións.
- Xestión da comunicación sen fíos.
- Envío e recepción de mensaxes de texto: seguridade e permisos.
- Envío e recepción de mensaxaría multimedia: sincronización de contido; seguridade e permisos.
- Manexo de conexións HTTP e HTTPS.
- Eventos da interface.
- Probas de interacción.

- Empaquetaxe e distribución.
- Documentación do desenvolvemento das aplicacións.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SUBSECCIONES DE UD05. MATERIAL VIEWS

CREAR PROYECTO

Para ver cómo funcionan los ciclos de vida en Android vamos a generar un proyecto sobre el que veremos los distintos estados de forma práctica. Así, lo primero que haremos será generar nuestro protecyo en Android Studio siguiendo los siguientes pasos:

1. **Abrir Android Studio**: Abre Android Studio en tu ordenador.
2. **Crear un Nuevo Proyecto**:
    - Selecciona “Start a new Android Studio project” en la pantalla de inicio.
    - Elige “Phone and Tablet” como tipo de dispositivo.
    - Selecciona “Empty Activity” como plantilla para comenzar con una actividad vacía.
3. **Configuración del Proyecto**:
    - En la siguiente pantalla, completa la información básica sobre el proyecto:
        - **Name**: Ingresa “UF1_UD05_1_Pizz”.
        - **Package name**: Deja el nombre de paquete predeterminado o personalízalo según tus necesidades.
        - **Save location**: Elige la ubicación donde deseas guardar el proyecto en tu sistema.
        - **Language**: Selecciona “Kotlin” como lenguaje de programación.
        - **Minimum API level**: Selecciona “API 35: Android 7.0 (Nougat)” como SDK mínimo.
4. **Finalizar Configuración**:
    - Revisa la configuración y ajusta cualquier otra opción según tus preferencias.
    - Haz clic en “Finish” para crear el proyecto.

Android Studio generará automáticamente la estructura básica del proyecto, incluyendo los archivos necesarios para la actividad principal que has creado. Puedes comenzar a desarrollar tu aplicación agregando código a la actividad `MainActivity.kt` y diseñando la interfaz de usuario en el archivo de diseño correspondiente.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

MATERIAL

En la biblioteca estándar de Android y en **Material**, contamos con diversos componentes para mejorar nuestra interfaz de usuario. A continuación, exploraremos cómo utilizar algunos de ellos:

- **Toolbars** dinámicas que se ajustan al desplazamiento de la pantalla o se colapsan.
- **Radio buttons**, **checkboxes** y **chips** que facilitan la selección de opciones.
- **Botones de acción flotantes** (Floating Action Buttons o FAB's).
- **Toasts** y **Snackbars** para mostrar mensajes emergentes.

Dependencia

Antes que nada, es crucial verificar si nuestro proyecto cuenta con la dependencia de Material agregada. En el archivo Gradle de la aplicación, asegurémonos de que la dependencia esté incluida de la siguiente manera:

```
implementation("com.google.android.material:material:1.10.0")
```

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

FRAGMENTO

Hemos creado el fragmento OrderFragment y simplificado el archivo Kotlin para que solo contenga el método `onCreateView`:

```
class OrderFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflar el diseño para este fragmento
        return inflater.inflate(R.layout.fragment_order, container, false)
    }

}
```

Además, hemos ajustado el diseño para que muestre únicamente un texto con la pregunta "¿Qué tipo de pizza deseas?". Para lograrlo, hemos agregado una cadena:

```
<string name="order_question">¿Qué tipo de pizza deseas?</string>
```

Y hemos modificado el diseño:

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

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="@string/order_question" />

</LinearLayout>
```

Pasos

1. **Añadiendo el Fragmento a la Actividad Principal:** Para integrar el fragmento en la actividad principal, creamos un LinearLayout que utiliza el componente FragmentContainerView.

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

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/fragment_container_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:name="com.example.uf1_ud05_1_pizza.OrderFragment"/>

</LinearLayout>
```

Ejecutamos la aplicación hasta este punto para confirmar que el fragmento se carga correctamente en el contenedor.

2. **Cambiando la Barra por Defecto:**

- Verificamos que estamos utilizando un tema sin la barra por defecto en ambos archivos “themes.xml”:

```
<style name="Base.Theme.UF1_UD05_1_Pizza" parent="Theme.Material3.DayNight.NoActionBar">
```

- Creamos nuestra propia barra en el archivo del fragmento, asociándola específicamente al fragmento:

```
<com.google.android.material.appbar.MaterialToolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    style="@style/Widget.MaterialComponents.Toolbar.Primary" />
```

Ejecutamos la aplicación para notar que hay una barra presente, aunque aún no está definida como la barra por defecto.

3. **Toolbar como Barra Principal:** En el código del fragmento, recuperamos la toolbar y la establecemos como la barra principal:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    // Inflar el diseño para este fragmento
    val view = inflater.inflate(R.layout.fragment_order, container, false)
```

```
    val toolbar = view.findViewById<MaterialToolbar>(R.id.toolbar)
    (activity as AppCompatActivity).setSupportActionBar(toolbar)
    return view
}
```

Con este código, aunque en un fragmento no tenemos acceso directo al método `setSupportActionBar`, podemos acceder a la actividad que contiene el fragmento para establecer la toolbar como la barra principal.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

SCROLL

En esta fase, deseamos implementar el **desplazamiento** en el fragmento, de modo que al hacer scroll, la toolbar también se desplace hacia abajo.

Para lograr que la toolbar se mueva al hacer scroll, necesitamos emplear un `CoordinatorLayout`, encargado de coordinar las animaciones entre las diversas vistas. Se incluirá un `AppBarLayout`, que posibilitará la animación de la toolbar, y un `NestedScrollView`, que permitirá que el contenido del diseño sea desplazable.

```
<CoordinatorLayout>
    <AppBarLayout>
        <CollapsingToolbarLayout>
            <MaterialToolbar/>
        </CollapsingToolbarLayout>
    </AppBarLayout>
    <NestedScrollView>
        <LinearLayout> <!-- Solo puede haber un view -->
            . . .
        </LinearLayout>
    </NestedScrollView>
</CoordinatorLayout>
```

Para implementar una barra colapsable, introduciremos un nuevo tipo de diseño dentro del `AppBarLayout`: el `CollapsingToolbarLayout`. Este, a su vez, contendrá la `MaterialToolbar`.

```
<CoordinatorLayout>
    <AppBarLayout>
        <CollapsingToolbarLayout>
            <MaterialToolbar/>
        </CollapsingToolbarLayout>
    </AppBarLayout>
    <NestedScrollView>
        <LinearLayout> <!-- Solo puede haber un view -->
            . . .
        </LinearLayout>
    </NestedScrollView>
</CoordinatorLayout>
```

1. **Cambiar a CoordinatorLayout:** Primero, actualizamos nuestro diseño reemplazando el LinearLayout por el CoordinatorLayout. Todo permanece igual, excepto la orientación, que es específica del LinearLayout.

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".OrderFragment">
```

2. **Añadir AppBarLayout:** Agregamos el diseño específico `appBarLayout`:

```
<com.google.android.material.appbar.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/ThemeOverlay.MaterialComponents.Dark">
```

3. **Añadir Toolbar dentro del AppBarLayout:** Dentro de este diseño específico para la barra de aplicación, añadimos la toolbar para que responda al scroll:

```
<com.google.android.material.appbar.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/ThemeOverlay.MaterialComponents.Dark">

    <com.google.android.material.appbar.MaterialToolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize" />
</com.google.android.material.appbar.AppBarLayout>
```

Ya no necesitamos el estilo de la barra porque aplicaremos el tema al diseño `AppBarLayout`.

4. **Definir Propiedad de Scroll:** Definimos una propiedad para especificar cómo reaccionará ante los scrolls:

```
app:layout_scrollFlags="scroll|enterAlways"
```

“scroll” para permitir el desplazamiento y “enterAlways” para ocultar la barra superior al hacer scroll.

5. **Scroll en el Resto del Fragmento:**

Además de la toolbar, queremos que el resto del contenido del fragmento haga scroll. Por lo tanto, añadimos los demás componentes dentro del elemento NestedScrollView. También le añadimos la propiedad `app:layout_behavior="@string/appbar_scrolling_view_behavior"` para ajustar su posición en función de si la toolbar es visible o no.

```
<androidx.core.widget.NestedScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

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

```
</androidx.core.widget.NestedScrollView>
```

Al ejecutar la aplicación, podemos observar que al pulsar sobre el texto, hacemos scroll, ocultando la toolbar.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

COLLAPSING BAR

Vamos a ajustar el comportamiento de desplazamiento de la barra de aplicación para que, en lugar de hacer scroll, se colapse. Para lograr esto, agregamos otro diseño del tipo CollapsingToolbarLayout dentro del diseño de AppBarLayout. A través de la propiedad `layout_behavior`, indicamos que este diseño debe colapsar:

```
<com.google.android.material.appbar.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/ThemeOverlay.MaterialComponents.Dark">

    <com.google.android.material.appbar.CollapsingToolbarLayout
        android:layout_width="match_parent"
        android:layout_height="300dp"
        app:layout_scrollFlags="scroll|exitUntilCollapsed">

        <com.google.android.material.appbar.MaterialToolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            app:layout_collapseMode="pin" />

    </com.google.android.material.appbar.CollapsingToolbarLayout>

</com.google.android.material.appbar.AppBarLayout>
```

Ahora es el CollapsingToolbarLayout el que responde al comportamiento de desplazamiento, por lo que la propiedad `layout_scrollFlags` ya no es necesaria en la MaterialToolbar. No obstante, vamos a indicar con otra propiedad cómo debe colapsar. En este caso, si tenemos un título, botones, texto, etc., colapsará pero dejará visible esos elementos:

```
<com.google.android.material.appbar.MaterialToolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    app:layout_collapseMode="pin" />
```

Al ejecutar la aplicación, al pulsar sobre el texto, observaremos que hacemos scroll y la toolbar se colapsa, pero siempre deja visible el texto.

ud4-scroll2

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

IMAGEN

Vamos a incorporar una imagen a la toolbar. Inicialmente, la añadimos a la carpeta “drawable” y luego insertamos un componente ImageView dentro de nuestro CollapsingBarToolBar.

Al utilizar el componente parallax, podemos tener varios elementos dentro de nuestra toolbar y, al colapsar, experimentarán una animación específica, creando un estilo distintivo.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

COLOR

Adicionalmente, podemos incluir una propiedad para que, al colapsar, se aplique un color específico a la barra de título:

```
app:contentScrim="?attr/colorPrimary"
```

Al ejecutar la aplicación, observamos el siguiente resultado:

ud4-scroll3

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

Capitulo 8

RADIO BUTTONS

Los botones de radio, también conocidos como radio buttons, posibilitan la elección de una opción de entre varias alternativas mutuamente excluyentes.

- Para implementar un botón de radio, se utilizará la vista `RadioButton`.
- El método `checkRadioButtonId` proporciona acceso al ID del RadioButton seleccionado (devuelve -1 si no se ha seleccionado ninguno).
- Dado que representan opciones excluyentes, es necesario agruparlos dentro de un RadioGroup. Esto garantiza que la selección de uno desactive automáticamente los demás.

Incorporar al proyecto

Para incorporar nuevos componentes a nuestro proyecto, es necesario realizarlo dentro del elemento `NestedScrollView`. No obstante, dado que solo se permite una vista dentro de este elemento, y ya hemos incluido un `TextView`, ahora deseamos agregar más componentes. Para lograr esto, es preciso insertar un diseño (layout) dentro de él, que sirva como raíz para los demás elementos que planeamos añadir. El código modificado es el siguiente:

```
<androidx.core.widget.NestedScrollView
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior">

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

        <TextView
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:text="@string/order_question" />

        <!-- Aquí puedes añadir más componentes según sea necesario -->
        <!-- Por ejemplo: -->
        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Mi Botón" />

        <EditText
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:hint="Mi Campo de Texto" />

        <!-- Agrega otros componentes aquí según tus requisitos -->

    </LinearLayout>
</androidx.core.widget.NestedScrollView>
```

Es importante destacar que este ajuste no debería afectar la apariencia de nuestra aplicación.

Radio Group

Dado que los botones de radio son mutuamente excluyentes, es necesario crear un grupo que los agrupe:

```
<RadioGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/group_pizza_type">

    <!-- Aquí se agregarán los botones de radio según sea necesario -->

</RadioGroup>
```

Radio Button

Dentro de este conjunto, incorporamos los distintos botones de radio:

```
<RadioGroup
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/group_pizza_type">
```

```
    <RadioButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/radio_margarita"
        android:text="Margarita" />

    <RadioButton
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/radio_calzone"
        android:text="Calzone" />
</RadioGroup>
```

Cuando ninguno de los botones está seleccionado, el ID devolverá -1. Si seleccionamos alguno de ellos, obtendremos esos IDs.

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

CHIPS

Los Chips son elementos interactivos que posibilitan tanto selecciones como acciones:

- Pueden contener hasta dos etiquetas (principal y secundaria) y un icono, y también admiten una imagen de fondo en el caso de Chips de imagen.
- Se implementan mediante la vista `Chip` y, aunque operan de manera independiente, es posible agruparlos visualmente utilizando `ChipGroup`.
- Su estado de selección puede ser verificado a través del método `isChecked`.

Incorporar al proyecto

Vamos a incorporar dos chips a nuestro proyecto, ofreciendo al usuario la posibilidad de seleccionar entre dos opciones no excluyentes. Dentro del `LinearLayout`, añadimos ambos componentes:

```
<com.google.android.material.chip.Chip
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Parmesano"
    android:id="@+id/chip_parmesano"
    style="@style/Widget.MaterialComponents.Chip.Choice" />

<com.google.android.material.chip.Chip
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/chip_tomate_cherry"
    android:text="Tomate Cherry"
    style="@style/Widget.MaterialComponents.Chip.Choice" />
```

Verificamos que se posicionan uno debajo del otro debido al diseño del `LinearLayout`. Para aplicar estilos específicos, podemos agruparlos mediante un `ChipGroup`:

```
<com.google.android.material.chip.ChipGroup
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content">

    <com.google.android.material.chip.Chip ... />
    <com.google.android.material.chip.Chip ... />
</com.google.android.material.chip.ChipGroup>
```

Una vez incluidos los dos chips dentro del `ChipGroup`, observamos que ahora se presentan horizontalmente.

Estos chips representan una evolución de los checkboxes, permitiendo comportamientos adicionales. Si modificamos la propiedad de selección:

```
style="@style/Widget.MaterialComponents.Chip.Action"
```

Funcionarán como botones, desencadenando eventos que podemos controlar desde nuestro código.

En el caso de aplicar otro estilo, como el de filtro:

```
style="@style/Widget.MaterialComponents.Chip.Filter"
```

Al ejecutar nuestra aplicación, notamos que la apariencia ha cambiado.

Autor/a: Sabela Sobrino Última actualización: 20.01.2025

FABS

Los botones de acción flotantes son elementos circulares cuya función principal es activar la acción primaria en la interfaz de usuario de tu aplicación.

- Los íconos deben ser claros, ya que no cuentan con etiquetas explícitas.
- Mantienen su presencia en la pantalla incluso cuando se realiza un desplazamiento.
- A lo largo de las distintas versiones del sistema Material Design, su apariencia ha experimentado modificaciones.

Para incorporar estos elementos a nuestro proyecto, los añadiremos fuera del `NestedScrollView`, ya que no forman parte de los elementos básicos de nuestro sistema. Por lo tanto, nos posicionamos en el elemento raíz `CoordinatorLayout` y añadimos el siguiente componente:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/fab"
    android:layout_margin="16dp"
    android:src="@android:drawable/ic_menu_send"
    android:layout_gravity="bottom|end" />
```

Integración en la Barra de Herramientas

Vamos a realizar otra integración, esta vez dentro de la barra de herramientas.

1. Para incorporar este botón en la barra de herramientas, lo primero que debemos hacer es asignarle un identificador a esa barra:

```
<com.google.android.material.appbar.CollapsingToolbarLayout
    android:layout_width="match_parent"
    android:layout_height="300dp"
    app:layout_scrollFlags="scroll|exitUntilCollapsed"
    app:contentScrim="?attr/colorPrimary"
    android:id="@+id/collapsing_toolbar">
```

7. Copiamos el código del botón flotante anterior (`fab`) y le añadimos dos propiedades adicionales: `layout_anchor`, donde indicamos dónde vamos a anclar este botón flotante (en este caso, a la barra de herramientas), y `layout_anchorGravity`, donde especificamos en qué parte de la barra de herramientas queremos que se posicione:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/fab2"
    android:layout_margin="16dp"
    android:src="@android:drawable/ic_menu_help"
    android:layout_gravity="bottom|end"
    app:layout_anchor="@id/collapsing_toolbar"
    app:layout_anchorGravity="bottom|end" />
```

Autor/a: Sabela Sobrino Última actualización: 08.02.2024

TOAST Y SNACKBAR

Los Toasts y Snackbars son herramientas eficaces para proporcionar al usuario información o avisos de manera concisa en una ventana emergente que desaparece automáticamente después de un tiempo.

Toasts

Obtendremos una referencia a nuestro botón:

```kotlin
val fab = view.findViewById<FloatingActionButton>(R.id.fab)
fab.setOnClickListener {
    val pizzaGroup = view.findViewById<RadioGroup>(R.id.group_pizza_type)
    val pizzaType = pizzaGroup.checkedRadioButtonId

    if (pizzaType == -1) {
        val msg = "Debes seleccionar un tipo de pizza"
        Toast.makeText(activity, msg, Toast.LENGTH_LONG).show()
    } else {
        // Lógica adicional, si es necesario
    }
}
```

Si no se ha seleccionado ningún tipo de pizza, generamos un Toast con un mensaje.

```
![chips](ud4-toast.gif)
```

Snackbar

Añadir un mensaje de Snackbar es similar a Toast:

```kotlin
Snackbar.make(fab, msg, Snackbar.LENGTH_SHORT).show()
```

Completamos la lógica escribiendo un mensaje con el tipo de pizza seleccionado y los elementos adicionales que queremos:

```kotlin
var msg = ""
if (pizzaType == -1) {
    msg = "Debes seleccionar un tipo de pizza"
    // Toast.makeText(activity, msg, Toast.LENGTH_LONG).show()
} else {
    msg = "Has seleccionado una pizza "
    // Tipo de pizza
    msg += when (pizzaType) {
        R.id.radio_margarita -> "Margarita"
        else -> "Calzone"
    }
    // Extras
    var parmesano = view.findViewById<Chip>(R.id.chip_parmesano)
    if (parmesano.isChecked) msg += " Parmesano"
    var tomate = view.findViewById<Chip>(R.id.chip_tomate_cherry)
    if (tomate.isChecked) msg += " Tomate Cherry"
}
Snackbar.make(fab, msg, Snackbar.LENGTH_SHORT).show()
```

![chips](ud4-total.gif)

Estas herramientas proporcionan una manera efectiva de comunicarse con el usuario de forma breve y no intrusiva. Otra opción sería recuperar el elemento directamente de la vista y obtener el texto mostrado: ```kotlin

var msn = “Debes seleccionar un tipo de pizza”.plus(view.findViewById(pizzaType).text) /* msn += when(burguerType){ R.id.radio_american -> getString(R.string.american) R.id.radio_vegan -> getString(R.string.vegan) R.id.radio_chicken -> getString(R.string.chicken) else -> “Error” }*/ val snackbar = Snackbar.make(fabSend, msn, Snackbar.LENGTH_SHORT) .setAction(“Undo”){

        }.show()
Acción SnackBar

Estos Snackbars también nos brindan la posibilidad de agregar acciones. Esto se logra antes de mostrar el SnackBar. A continuación, se describen los pasos:

1. Primero, obtenemos una referencia al SnackBar:

```kotlin
val snackbar = Snackbar.make(fab, msg, Snackbar.LENGTH_SHORT)
```

2. Luego, añadimos una acción “Undo”:

```
val snackbar = Snackbar.make(fab, msg, Snackbar.LENGTH_SHORT)
snackbar.setAction("Undo") {
    // Acciones a realizar cuando se selecciona "Undo"
}
snackbar.show()
```

Estos pasos nos permiten personalizar el SnackBar con una acción específica que el usuario puede ejecutar, en este caso, deshacer la acción previa.