# Elementos de la interfaz, navegación e interacción

## Elementos de interacción (Parte II)

{desafío} latam_

***Utilizar elementos de navegación e interacción de usuario disponibles en el entorno Android Studio para dar solución a un requerimiento.***

- Unidad 1: Ambiente de desarrollo y sus elementos de configuración.
- Unidad 2: Elementos de la interfaz, navegación e interacción.
- Unidad 3: Fundamentos de GIT y GitHub.

***Te encuentras aquí***

{desafío} latam_

# ¿Qué aprenderás en esta sesión?

- *Utiliza Event Listeners en las vistas para el manejo de la interacción del usuario y del comportamiento a la pantalla*

# ¿Cuál es el objetivo de la binding class?

# /* Obtener el input del usuario */

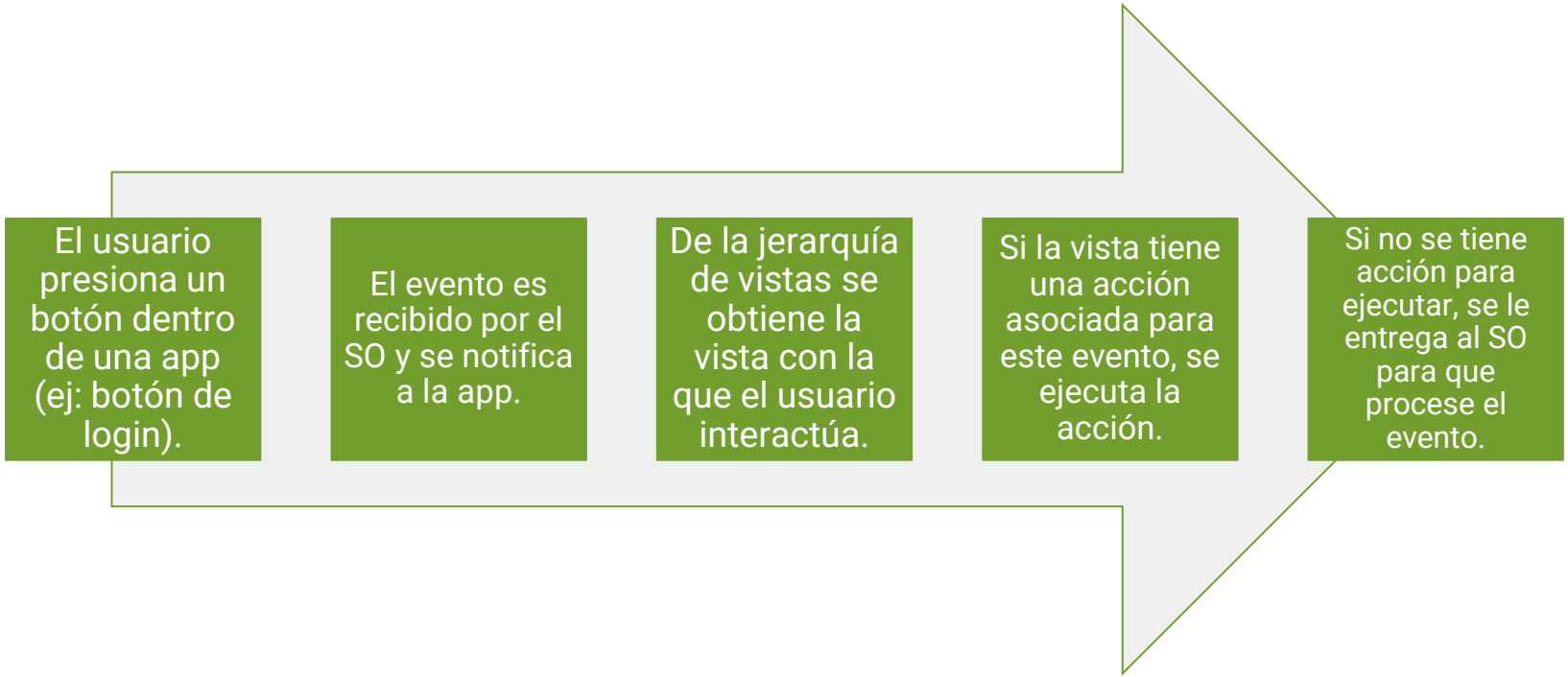
# Eventos de entrada

*¿Qué son?*

- Es un **evento de interacción entre usuario y app.**
- Cada objeto View maneja los eventos con los que el usuario interactúa.
- La clase View tiene varios callbacks que sirven para los eventos de UI.
- El framework de Android utiliza callbacks cuando ocurre la acción respectiva.

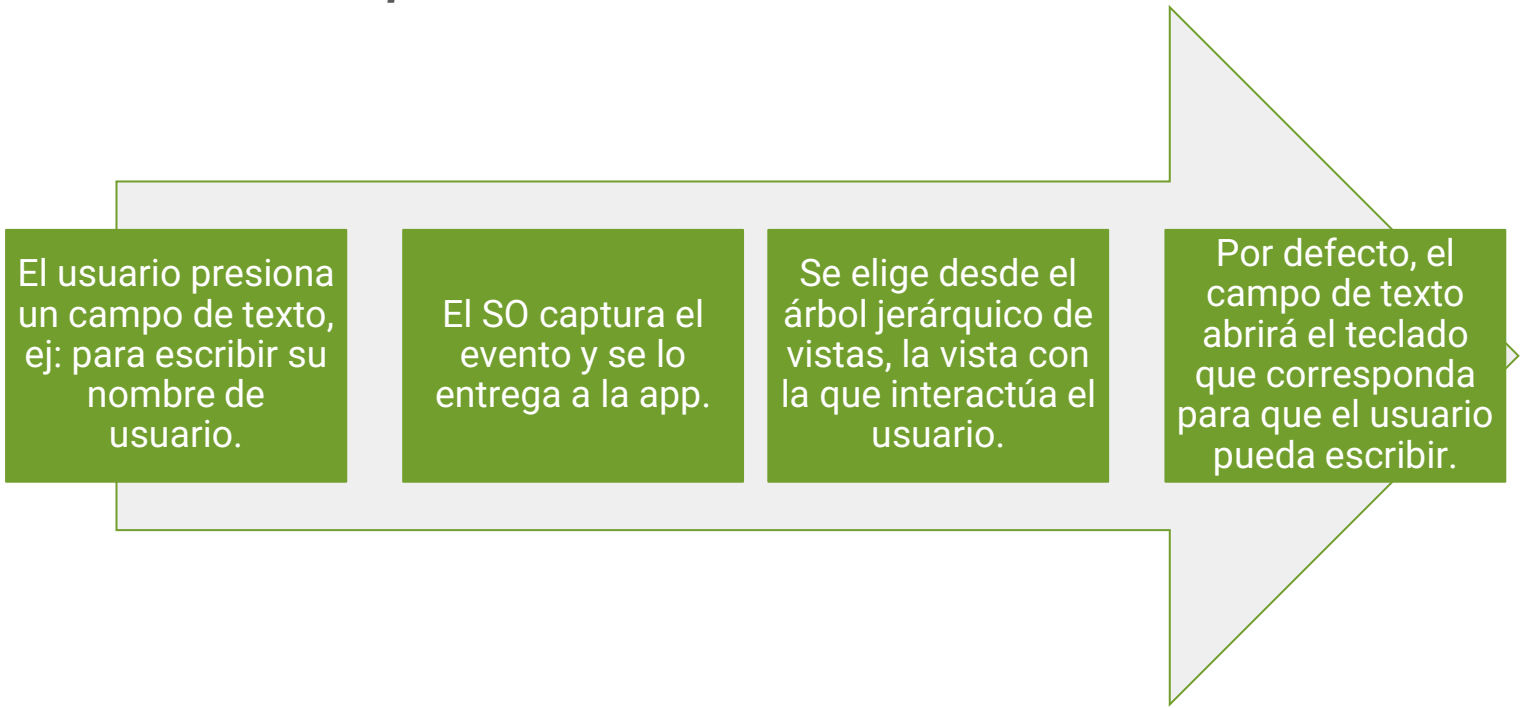
# Eventos de entrada

## *Acción sobre un botón*



# Eventos de entrada

***Acción sobre un campo de texto***



# Callback

## *¿Qué es?*

También llamado *call-after function*, es un código que se entrega a otra función para que sea ejecutado en un momento futuro, ya sea de forma síncrona o asíncrona.

La implementación depende del lenguaje de programación, y puede ser, por ejemplo, usando:

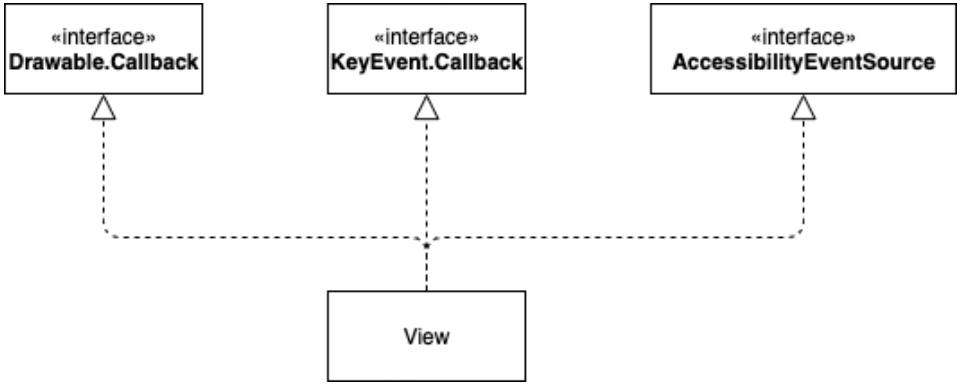
- Métodos
- Lambdas
- Bloques

# Clase View

Recordemos que la clase View es la clase base para todos los componentes visuales como botones, textos o imágenes.

La clase View implementa 3 interfaces que modelan la interacción con el usuario:

- Drawable.Callback
- KeyEvent
- AccessibilityEventSource




Fuente: ADL.

# Clase View

Las interacciones entre cada vista y el usuario se manejan con los eventos básicos en la categoría de *Event processing*.

Cada subclase de vista, por ejemplo, un botón (Button) o un texto (TextView), tienen su propia implementación de estos eventos.

| | | |
|---|---|---|
| Event processing | `onKeyDown(int, android.view.KeyEvent)` | Called when a new hardware key event occurs. |
| | `onKeyUp(int, android.view.KeyEvent)` | Called when a hardware key up event occurs. |
| | `onTrackballEvent(android.view.MotionEvent)` | Called when a trackball motion event occurs. |
| | `onTouchEvent(android.view.MotionEvent)` | Called when a touch screen motion event occurs. |

Fuente:
developer.android.com/view

# Vistas personalizadas

## *Actividad*

Vamos a implementar nuestro propio TextView que cambie de color cuando se presione.

Para eso necesitamos:

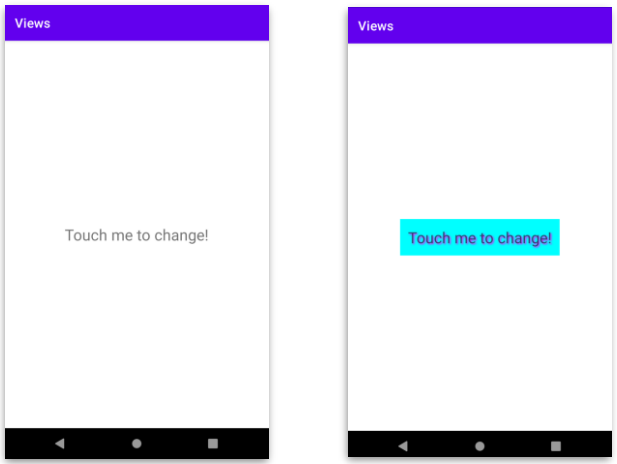
1. Heredar de TextView.
2. Sobreescribir el método `onTouchEvent`.
3. Modificar el aspecto del texto.

```
public class MyCustomView extends AppCompatTextView {
    private static final String TAG = "MyCustomView";
    public MyCustomView(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        Log.d(TAG, "onTouchEvent "  + event.toString());
        decorateText();
        return super.onTouchEvent(event);
    }
    private void decorateText() {
        if (!getText().toString().isEmpty()) {
            setShadowLayer(6, 4, 4, Color.rgb(250, 00, 250));
            setBackgroundColor(Color.CYAN);
        } else {
            setBackgroundColor(Color.RED);
        }
    }
}
```

# Vistas personalizadas

## *Actividad*

Nuestra nueva vista CustomView se utiliza directamente en el layout.




Fuente: ADL.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout

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"
    tools:context=".MainActivity">


    <cl.dal.viewexample.MyCustomView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="12dp"
        android:text="@string/touch_me_to_change"
        android:textSize="24sp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

# Event handlers

## *Uso*

Al construir un componente personalizado, existen varios métodos utilizados como *callback* para manejar eventos:

| Event Handler | Descripción |
|---|---|
| onKeyDown(int, KeyEvent) | Llamado cuando se presiona una tecla |
| onKeyUp(int, KeyEvent) | Llamado cuando se libera un tecla |
| onTrackballEvent(MotionEvent) | Cuando ocurre un movimiento de trackball |
| onTouchEvent(MotionEvent) | Cuando el usuario toca la pantalla |
| onFocusChanged(boolean, int, Rect) | Cuando la vista gana o pierde el foco |

# /* Vistas y listeners */

# Listener

## *¿Qué es?*

- Recordemos que un evento de entrada es un **evento de interacción entre usuario y app.**
- Los listeners se encargan de controlar estos eventos y realizar una o unas serie/s de acciones especialmente diseñadas.
- A nivel de código, el listener es una interfaz que define el comportamiento, por lo que debemos implementar todos sus métodos definidos.

# Algunos tipos de listener

- El más utilizado es OnClickListener, que reacciona cuando el usuario hace tap (clic) sobre una vista.
- Hay otros listeners que permiten tener comportamientos más específicos, como dejar presionado un botón durante unos segundos, o escribir una letra usando el teclado.

| Interfaz | Event listener |
|---|---|
| View.OnClickListener | onClick(View v) |
| View.OnLongClickListener | onLongClick(View v) |
| View.OnKeyListener | onKey(View v, int keyCode, KeyEvent event) |
| View.OnTouchListener | onTouch(View v, MotionEvent event) |
| View.OnFocusChangeListener | onFocusChange(View v, boolean hasFocus) |

# /* Alternativas para implementar event listeners */

# a. Implementación anónima

## *Usando un atributo*

- Se define una variable que implemente el listener (OnClickListener) sin crear una clase.
- Para asignar el listener, se utiliza el método `setOnClickListener` de la vista.

```
public class MainFragment extends Fragment {
    private static final String TAG = "FirstFragment";
    private FragmentMainBinding binding;
    private View.OnClickListener onClickListener = new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Log.d(TAG, "onClick called with v = " + view.getId());
        }
    };
    public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        binding.changeColorButton.setOnClickListener(onClickListener);
    }
```

# b. Implementación anónima

***En la vista***

Cada vista tiene asociado su propio listener, el que se entrega en forma anónima, es decir, no existe una clase explícitamente creada.

```
public class MainFragment extends Fragment {
   private static final String TAG = "FirstFragment";
   public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
     super.onViewCreated(view, savedInstanceState);
     binding.changeColorButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Log.d(TAG, "onClick called with v = " + view.getId());
        }
      });
   }
}
```

# c. Lambda

- Se implementa usando **predicado**.
- Esta es la opción preferente en la actualidad.
- Cada vista tiene asociado su propio listener.

```
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
   super.onViewCreated(view, savedInstanceState);
   binding.changeColorButton.setOnClickListener(v1 -> Log.d(TAG, "onClick called with v = " + v1.getId()));
}
```

# Expresiones Lambda

## *Predicado*

Expresión compuesta por argumentos y un cuerpo separados por el operador flecha ->

```
( argument ) -> { expression body }
```

| Argumento | Body |
| --- | --- |
| Son los parámetros que se le entregan a la función (cero o más). | Función a ejecutar. |
| **Ejemplos** | |

```
( ) -> Log.d(TAG, “Lambda sin parámetros”);
```

```
event -> Log.d(TAG, “Lambda con 1 parámetros ” + event);
```

```
(event1, event2) -> Log.d(TAG, “Lambda con 2 parámetros ”);
```

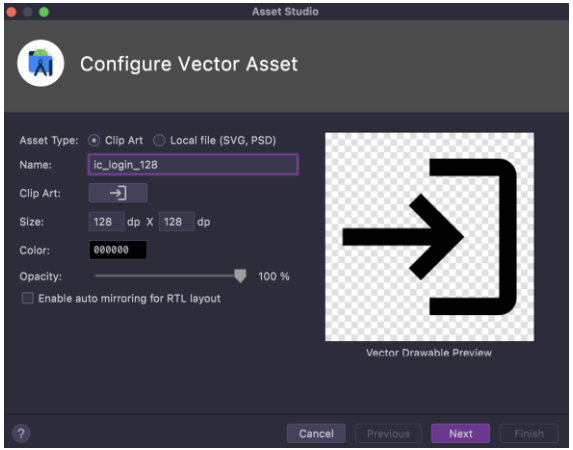
# /* Ejemplo - Pantalla de login */

# Ejemplo

## *Pantalla de login personalizada*

Vamos a construir una pantalla de login que nos permita probar el evento onClick sobre el botón Ingresar.

1. Crearemos un nuevo proyecto con al menos un fragmento.
2. Vamos a agregar un nuevo vector que usaremos en la pantalla de login.



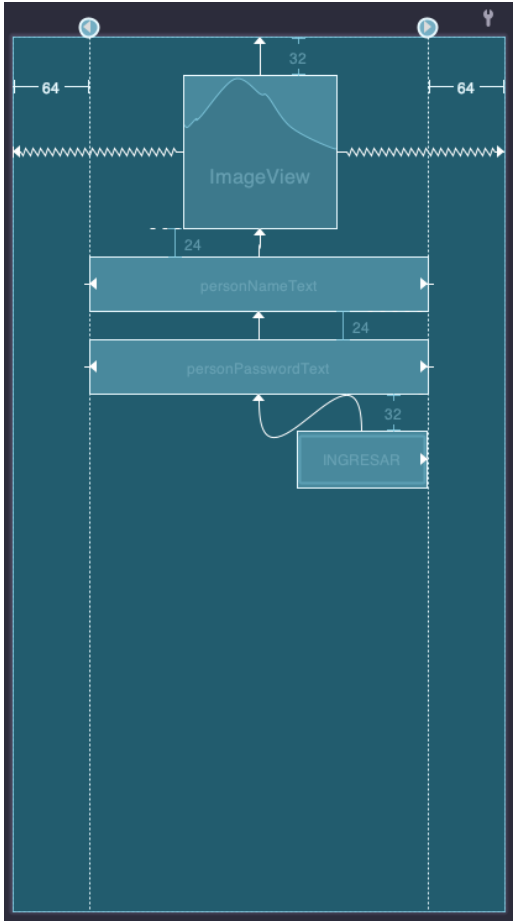

Fuente: ADL.

# Ejemplo

## *Pantalla de login personalizada*

3. Modificar el layout para que cumpla con el siguiente diseño:

- 2 guidelines verticales.
- 1 ImageView como cabecera.
- 2 EditText para ingresar nombre y correo.
- 1 botón para crear la cuenta.




Fuente: ADL.

# Ejemplo

## *Pantalla de login personalizada*

Usando `setOnClickListener()` se asocia el evento onClick con el método `printInfo()`.

En el método `printInfo()` se recupera la información del nombre y la contraseña, para luego imprimir por consola.

```
public class MainActivity extends AppCompatActivity {
    private ActivityMainBinding binding;
    private static final String TAG = "MainActivity";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());
        binding.loginButton.setOnClickListener(view -> printInfo());
    }
    private void printInfo() {
        String inputName = binding.personNameText.getText().toString();
        String inputPass = binding.personPasswordText.getText().toString();
        Log.d(TAG, "Información ingresada " + inputName + " - " + inputPass);
    }
}
```

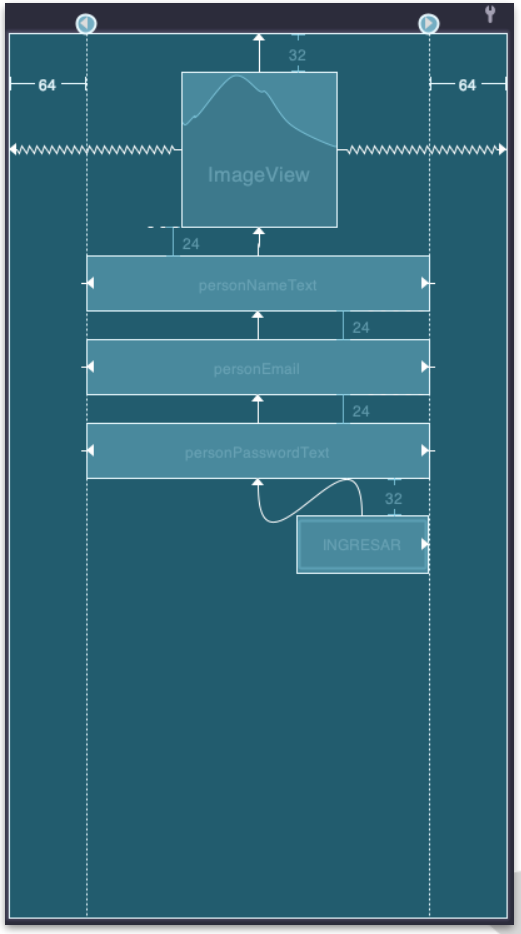
# Ejercicio práctico

# Ejercicio práctico

El botón “CREATE ACCOUNT” debe imprimir por consola los datos ingresados.

Acciones a realizar:

- Definir el layout.
- Implementar el listener del botón.
- Obtener los datos.
- Imprimir por consola los datos recuperados.




Fuente: ADL.

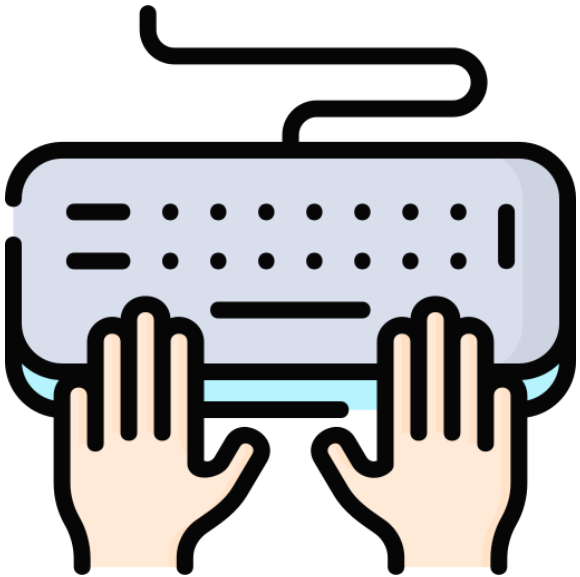
# /* Change listeners */

# Change listeners

En general ocupamos los listeners asociados a botones, imágenes, incluso CardView.

Pero hay un tipo de listeners que los usamos para reaccionar a eventos del usuario cuando por ejemplo, se está escribiendo un texto al ingresar una letra o al seleccionar un elemento de un spinner.

Este tipo de listeners reacciona a un cambio de estado del componente que es provocado por el usuario generalmente.



# Cambios en el texto

## *TextWatcher*

Se crea un implementación anónima de TextWatcher y se asocia al EditText para que cuando se escriba cualquier carácter se ejecute el método *onTextChanged*.

```
binding.editTextPersonName.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence charSequence, int i, int i1, int i2) {
    }
    @Override
    public void onTextChanged(CharSequence charSequence, int i, int i1, int i2) {
        Log.d(TAG, "texto escrito " + charSequence);
    }
    @Override
    public void afterTextChanged(Editable editable) {
    }
});
```

# Cambios en un spinner

## *OnItemSelectedListener*

Se crea un implementación anónima de OnItemSelectedListener, y se asocia al spinner para que al seleccionar un elemento se ejecute el método `onItemSelected`.

```
binding.spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {

    @Override
    public void onItemSelected(AdapterView<?> adapterView, View view, int i, long l) {
        Log.d(TAG, "elemento seleccionado " + adapterView.getItemAtPosition(i));
    }

    @Override
    public void onNothingSelected(AdapterView<?> adapterView) {

    }
});
```

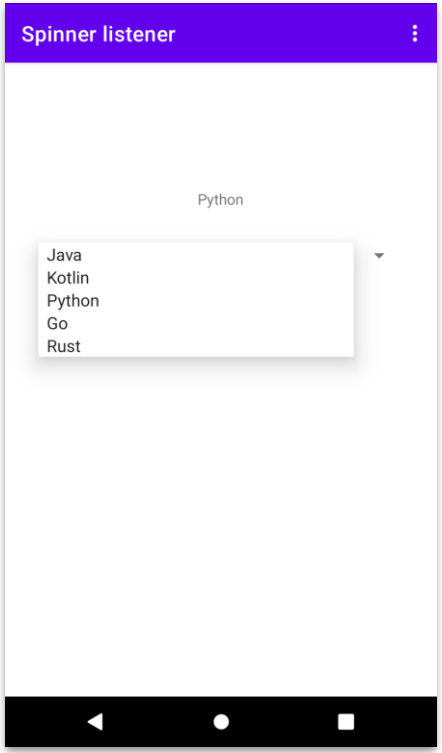
# Ejercicio práctico

Crear una pantalla que tenga un spinner para seleccionar un lenguaje de programación de la lista y que la selección se muestre en un TextView.

Tip: Recuerda usar un adaptador para contener la información que mostrará el Spinner.

```
String[] planets = {"Mercurio", "Venus", "Tierra", "Marte",
"Jupiter", "Saturno", "Urano", "Neptuno"};

ArrayAdapter<String> adapter = new
ArrayAdapter(getApplicationContext(),
             android.R.layout.simple_spinner_item,
             planets);
```




Fuente: ADL.

# ¿Cómo podemos hacer que una vista reaccione a una interacción del usuario?

# Próxima sesión...

- *Revisar el material de estudio sincrónico que consiste en una guía de estudio para practicar los conceptos aprendidos en las sesiones anteriores.*

{desafío} latam_

{desafío}
latam_

Academia de
talentos digitales