Interacciones de usuario y recursos de Android (Parte II)

Data Binding

Competencias

- Activar Data Binding en un proyecto
- Generar el objeto binding
- Interactuar con elementos del layout usando Data Binding
- Manejar una fuente de datos asociada al layout

Introducción

En la Google I/O 2018 uno de los anuncios relevantes fue *Android JetPack*, que es una guía arquitectónica soportada por un conjunto de librerías y herramientas agrupadas en 4 áreas del desarrollo

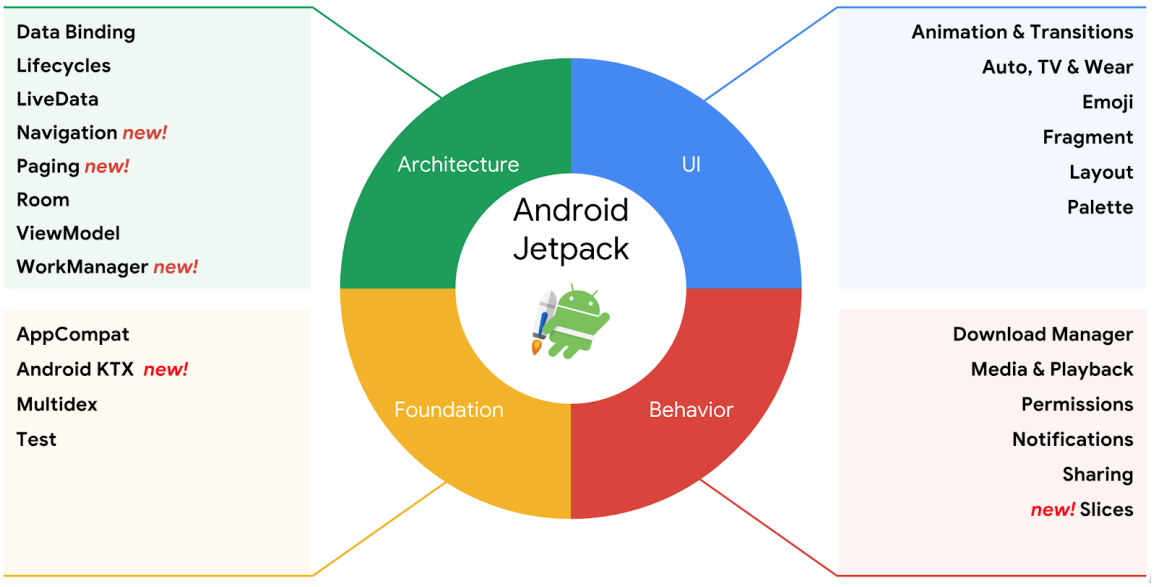

Imagen 1: Android Jetpack

Android Jetpack ayuda a acelerar el desarrollo de Android con componentes, herramientas y guías que eliminan las tareas repetitivas, permitiendo crear de forma más rápida y sencilla aplicaciones de alta calidad y testeables. [Android Developer Page]

Data Binding es una biblioteca de JetPack que permite enlazar los componentes de interfaz en los layouts con las fuentes de datos en nuestra app usando una forma **declarativa** más que **programática**.

Con Data Binding:

- Se elimina definitivamente la utilización de findViewById
- Las actividades y fragmentos son más limpios y con menos código
- Se puede utilizar expresiones directamente en el layout
- Se puede sincronizar los datos con las vistas, actualizándose automáticamente

Configurar Data Binding

Lo primero es habilitar Data Binding en el proyecto, activándolo en la configuración *android* en el archivo *build.gradle*

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

Generando el objeto Binding

Las binding classes son usadas para acceder a las variables y vistas del layout. Por cada archivo de layout se genera una *binding class*.

Por defecto, el nombre de la clase generada está basado en el nombre del layout luego de convertirlo en Pascal case y agregando el sufijo *Binding*. Por ejemplo, para el archivo **activity_main.xml** el correspondiente nombre generado es **ActivityMainBinding**.

Para que los layouts sean compatibles con Data Binding deben tener el elemento como raíz en el árbol jerárquico, de la siguiente forma

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools">

  <android.support.constraint.ConstraintLayout
     android:id="@+id/main"
     android:layout_width="match_parent"
     android:layout_height="match_parent"
     tools:context=".MainActivity">

     …

  </android.support.constraint.ConstraintLayout>
</layout>
```

La generación de las binding classes se realiza en tiempo de compilación y es necesario construir el proyecto luego de modificar los layout.

Creando el objeto Binding

El objeto binding debe ser creado inmediatamente después de que las vistas del layout fueron instanciadas (el layout fue “inflado”) para asegurar que el árbol jerárquico no ha sido modificado antes de poder enlazarlo.

La forma común utilizada es usar el método **setContentView** de la clase DataBindingUtil, reemplazando a setContentView de la clase Activity o Fragment como encargado de inflar la vista

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity_main);
}
```

Otra forma es utilizar los métodos estáticos de la binding class dentro de los métodos de creación de la Actividad o Fragment

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ActivityMainBinding binding = ActivityMainBinding.inflate(getLayoutInflater());
}
```

Vistas con ID

La binding class generada contiene un campo inmutable para cada vista que tiene un android:id definido en el layout. Por ejemplo, la biblioteca Data Binding crea los campos imageView y textView, de los tipos ImageView y TextView respectivamente, identificados en el siguiente layout.

```
<?xml version="1.0" encoding="utf-8"?>
<layout>
  <androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome!"
    />

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="128dp"
        android:layout_height="128dp"
        app:srcCompat="@drawable/androide" />


</androidx.constraintlayout.widget.ConstraintLayout></layout>
```

La biblioteca extrae las vistas incluyendo los IDs de la jerarquía de vistas en solo paso, logrando que este mecanismo sea más rápido que llamar a findViewById para cada una de las vistas en el layout.

Los IDs no son necesarios como lo son sin data binding, pero aún hay algunas instancias donde acceder a las vista por código sigue siendo necesario. Por ejemplo, al modificar el valor de una vista.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity_main);
    binding.textView.setText("Welcome again!");
}
```

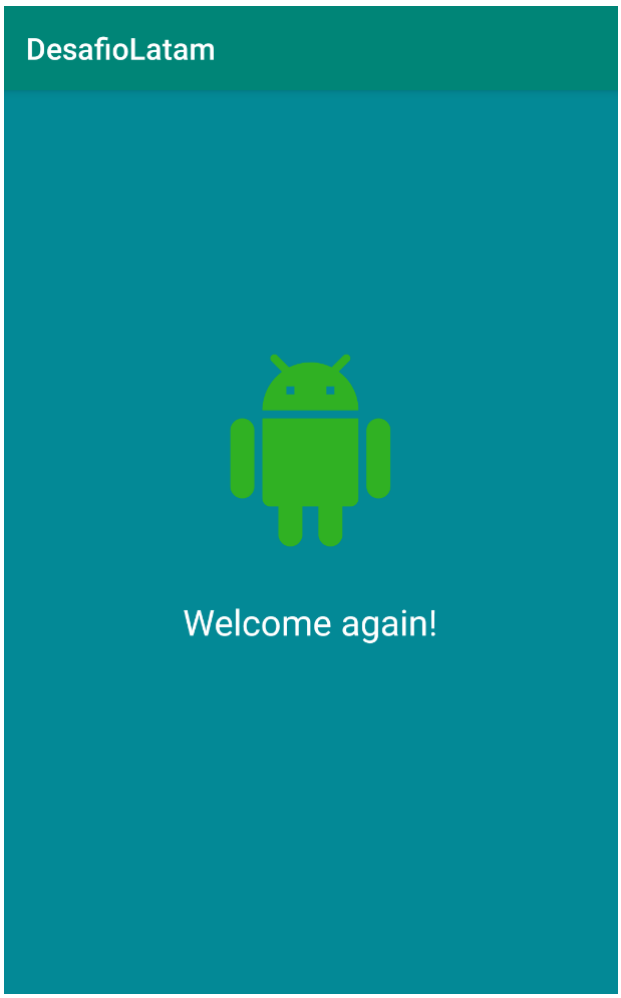

Imagen 2: Vista con ID

Variables

Dentro del tag `<data>` podemos definir variables que son utilizadas para llenar la información de las vistas.

Para cada una de estas variables, la biblioteca genera los métodos de acceso (*get* y *set*)

```
<data>
  <import type="android.graphics.drawable.Drawable"/>
  <variable name="user" type="cl.desafiolatam.User"/>
  <variable name="image" type="Drawable"/>
  <variable name="name" type="String"/>
</data>
```

Estas variables pueden ser utilizadas dentro del layout usando el format @{} dentro de los valores de los atributos. Por ejemplo, la variable name se puede utilizar como el texto en un TextView

```
<TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{name}" />
```

En el caso de ser una clase, se hacer referenciando directamente el atributo, debiendo existir el correspondiente método de acceso (*getter*) dentro de la clase utilizada

```
<TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{user.name}" />
```

Actualización automática

Cuando los valores de las variables cambian, el binding es agendado para cambiar en el futuro próximo, actualizando en forma automática la interfaz.Consideremos el siguiente layout

```
<?xml version="1.0" encoding="utf-8"?>
<layout>
  <data>
    <variable name="username" type="String" />
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="128dp"
        android:layout_height="128dp"
        app:srcCompat="@drawable/androide" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome"
    />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{username}!"
    />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

En el Activity que utiliza el layout anterior se aprecia que luego de obtener el objeto binding, se puede modificar el valor de la variable **username** declarada en el layout

```
package cl.desafiolatam;

import android.os.Bundle;

import androidx.appcompat.app.AppCompatActivity;
import androidx.databinding.DataBindingUtil;

import cl.desafiolatam.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    ActivityMainBinding binding = DataBindingUtil.setContentView(this, R.layout.activity_main);
    binding.setUsername("Tomás");
  }
}
```

Y la vista se actualiza sin tener que intervenir la misma directamente, desplegando el estado de la variable

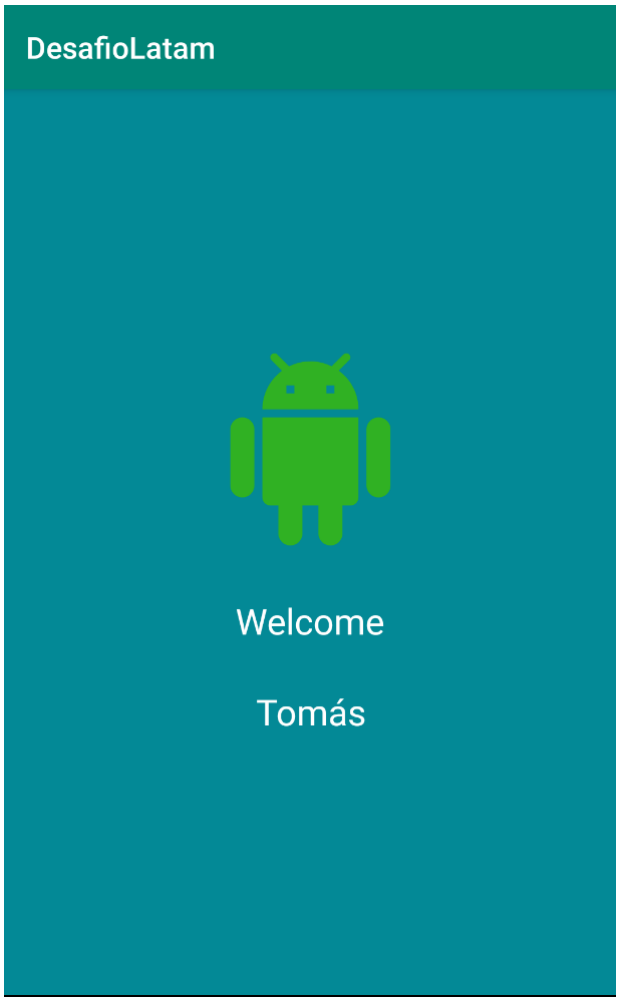

Imagen 3: Vista con estado de la variable

¿Qué sucede con el id del TextView para desplegar el nombre de usuario?

Usando data binding, el id de la vista deja de ser tan relevante como lo era para findViewById y ButterKnife

Lenguaje de expresiones

El lenguaje de expresiones se parece mucho a las expresiones que encontramos en nuestro código. Se pueden usar expresiones de tipo matemático, binario, lógicos, y además acceso a arreglos, clases y *cast*

Algunos ejemplos:

Para desplegar valores enteros se puede utilizar String.valueOf

```
android:text="@{String.valueOf(index + 1)}"
```

También se puede concatenar strings con variables usando comillas simples

```
android:src='@{"image_" + id}'
```

Para mostrar/ocultar una vista si una condición se cumple

```
android:visibility="@{age > 18 ? View.GONE : View.VISIBLE}"
```

Ejercicio

1. Crear un proyecto con una actividad vacía usando los templates y asignar como paquete `cl.desafiolatam.ejercicio1`

2. Crear clase Usuario que contenga dos atributos de tipo string: nombre, apellido y edad

```
package cl.desafiolatam.e1;

public class User {
  private String name;
  private String lastname;
  private int age;

  public User(String name, String lastname, int age) {
     this.name = name;
     this.lastname = lastname;
     this.age = age;
  }

  public String getName() {
     return name;
  }

  public String getLastname() {
     return lastname;
  }
  public int getAge() {
    return age;
  }
}
```

3. Agregar un usuario como fuente de datos al layout usando la clase Usuario definida

```
<data>
  <variable name="user" type="cl.desafiolatam.e1.User"/>
</data>
```

4. Agregar un textView para indicar el nombre del usuario. En la propiedad android:text se debe colocar el valor a utilizar obtenido desde la variable usuario (usuario.nombre)

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@{user.name}" />"
```

5. Enlazar el usuario a la vista

```
public class MainActivity extends AppCompatActivity {

    ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = DataBindingUtil.setContentView(this, R.layout.activity_main);
        binding.setUsuario(new User("Juan", "Ayala"));
    }
}
```

6. Verificar en la app que aparezca el nombre del usuario

Listener

Competencias

- Comprender el funcionamiento de los listeners
- Conocer los distintos gestores de eventos
- Utilizar una interfaz listener anidada para el evento onClick
- Utilizar una interfaz anónima para el evento onClick
- Implementar un listener en la activity para el evento onClick

Contexto

Listener (conocido también como “observer”) es un patrón de diseño común para crear llamadas asíncronas en el desarrollo. Los listener son utilizados en cualquier tipo de evento asíncrono para implementar el código que se va a ejecutar cuando ocurra el evento. Este es el patrón que se utiliza para todos los eventos de las vistas en la pantalla

La implementación de listener es un poderoso mecanismo para separar apropiadamente flujos dentro del código. Uno de los principios esenciales para escribir código mantenible es reducir el acoplamiento y la complejidad usando la encapsulación

Cuando ocurre un **evento** en la vista, por ejemplo, un click sobre un elemento, se verifica si existe algún listener registrado para ese evento particular. En el caso que exista se llama al **callback** correspondiente según lo definido en la **interfaz** del listener

De lo anterior, se rescatan 3 elementos:

- **Interfaz** es en orientación a objetos, una descripción de las acciones que un objeto puede realizar. La interfaz se declara no como clase (class) sino que como interfaz (interface).
- **Evento** es cualquier acción que realice el usuario interactuando con la app.
- **Callback** es un trozo de código ejecutable que se pasa como argumento a otro código para su ejecución cuando sea conveniente

Gestores de eventos

En particular para Android, **Event Listener** es una interfaz de la clase View que contiene una colección de interfaces anidadas con callbacks ya definidos.

Estas interfaces, llamadas gestores de eventos, permiten capturar la interacción del usuario con la interfaz.

Un receptor de eventos (*event listener*) es una interfaz en la clase View que contiene un solo método de callback.

En las interfaces de los gestores de eventos, se incluyen los siguientes métodos de callback:

- onClick()
- onLongClick()
- onFocusChange()
- onKey()
- onTouch()
- onCreateContextMenu()

Estos métodos son los únicos habitantes de su respectiva interfaz. Para definir alguno de estos métodos y manejar sus eventos, hay que implementar la interfaz anidada en su actividad o definirla como una clase anónima

Interfaz anidada

Consiste en implementar la interfaz creando una variable del tipo del listener e implementando el método requerido. Por ejemplo, para OnClickListener se debe implementar el método `onClick(View v)`. Una vez instanciado es posible asignarlo a una o varias vistas

```
public class MainActivity extends AppCompatActivity {

    ActivityMainBinding binding;
    ...

    private View.OnClickListener myListener = new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Toast.makeText(this, "OnClick event!!", Toast.LENGTH_SHORT).show();
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = DataBindingUtil.setContentView(this, R.layout.activity_main);

        binding.imageView.setOnClickListener(myListener);
    }
}
```

El listener se puede reutilizar si se quiere dar un comportamiento común a varios elementos.

```
public class MainActivity extends AppCompatActivity {

    ActivityMainBinding binding;

    private View.OnClickListener ocl = new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            if (v.getId() == binding.imageView.getId()) {
                showToast("onClick for imageView");
            }else if(v.getId() == binding.textView.getId()) {
                showToast("onClick for textView");
            }
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = DataBindingUtil.setContentView(this, R.layout.activity_main);

        binding.imageView.setOnClickListener(ocl);
        binding.textView.setOnClickListener(ocl);
    }

    private void showToast(final String message) {
        Toast.makeText(this, message, Toast.LENGTH_LONG).show();
    }
}
```

Al utilizar el mismo listener en ambas vistas se puede dar un comportamiento a varios elementos sin tener que crear listener para cada uno. Sin embargo, se puede utilizar el id de la vista para tener comportamientos distintos

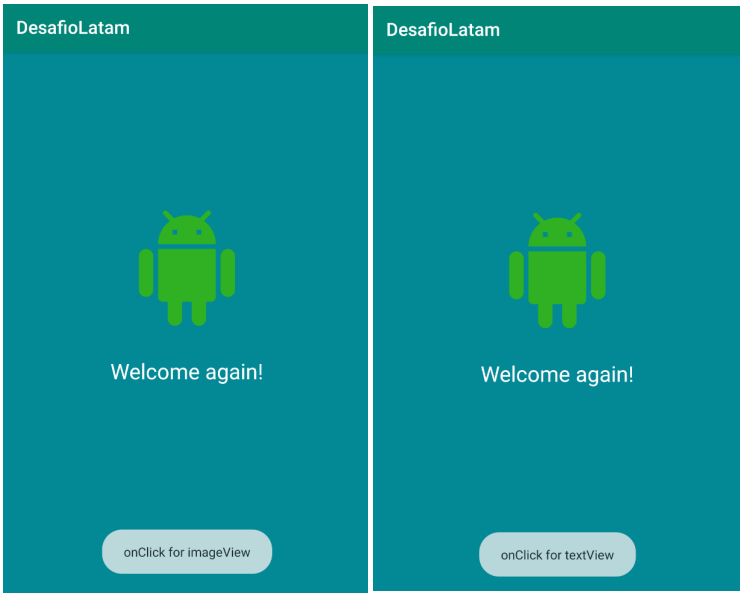

Imagen 4: id para comportamiento 1 y 2

Interfaz anónima

Es un listener anónimo porque no fue asignado a una variable y no puede ser reutilizado. Se crea un listener para cada uno de los componentes de forma independiente

```
public class MainActivity extends AppCompatActivity {

  ActivityMainBinding binding;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    binding = DataBindingUtil.setContentView(this, R.layout.activity_main);

    binding.imageView.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
        Toast.makeText(this, "OnClick event!!", Toast.LENGTH_SHORT).show();
      }
    });
  }
}
```

Es una forma rápida de asociar un listener, aunque si son muchos listener asociados a los eventos, el código tiende a ser más complejo de leer.

En ese caso, conviene crear implementaciones como clases separadas y utilizarlas en la actividad correspondiente

Usar la actividad para implementar el listener

Para ser considerada como un Activity, la clase debe extender una clase del tipo Activity, por ejemplo, AppCompatActivity y así sobreescribir el comportamiento de los callbacks utilizados en el ciclo de vida del Activity si lo estima conveniente

Es también una opción agregar al Activity la implementación de OnClickListener

Android Studio es amigable y ayuda a generar código para tareas rutinarias, en este caso basta escribir *implements View.OnClickListener* para que ofrezca agregar el método que debe ser implementado.

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {

    ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = DataBindingUtil.setContentView(this, R.layout.activity_main);

        binding.imageView.setOnClickListener(this);
    }

    @Override
    public void onClick(View v) {
        Toast.makeText(this, "OnClick event!", Toast.LENGTH_LONG).show();
    }
}
```

Cuando se asigna el listener al imageView (llamando a setOnClickListerner()) se hace referenciando al mismo Activity (*this*). El método onClick recibe la vista (View) donde se origina el evento

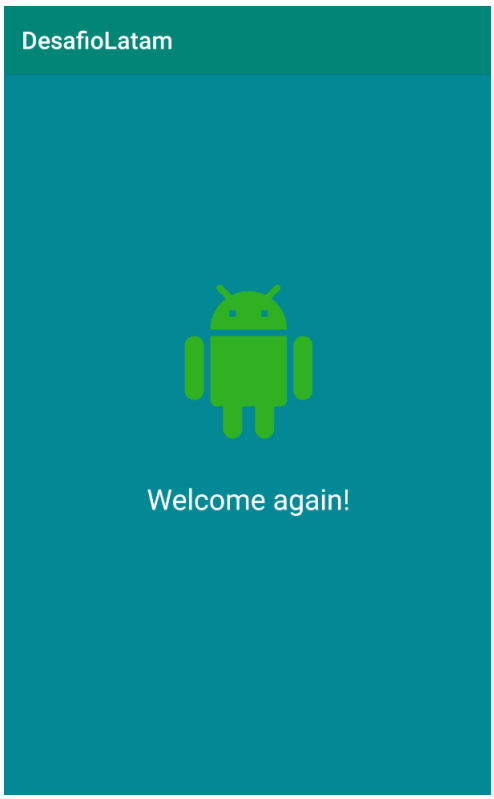

Imagen 5: Antes de ocurrir evento onClick

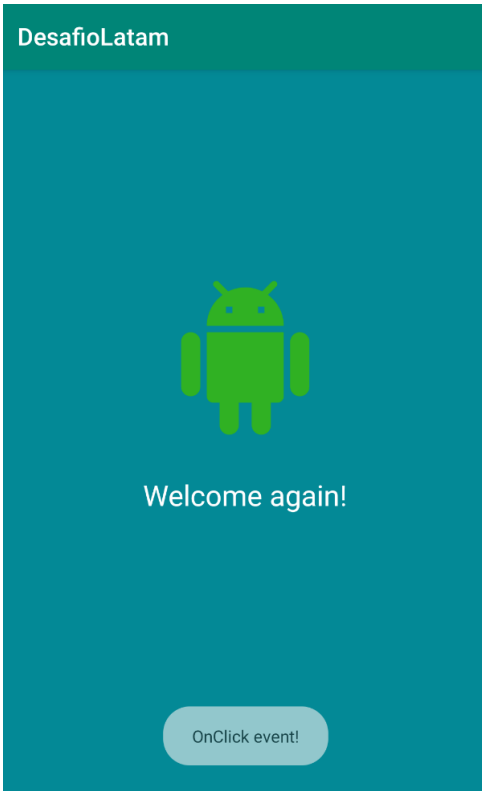

Imagen 6: Después de ocurrir evento onClick

Distintos tipos de tamaño de pantalla en los dispositivos

Competencias:

- Diferenciar las unidades de medida pixel, dp y sp
- Diferenciar la densidad y el tamaño de la pantalla
- Identificar diferentes densidades de pantalla

Introducción

A diferencia de Apple, Android corre en un conjunto diverso de dispositivos. Diversos en capacidades, tamaño y resolución, lo que obliga a diseñar e implementar las interfaces de las aplicaciones para que se adapten a estos distintos requerimientos

Gama	Modelo	Pantalla (")	Resolución (px)
Baja	Huawei Y5 2018	5,45	1440 × 720
	Samsung Galaxy J2	5	960 x 540
Media	Huawei Mate 10 Lite	5.9	1920 x 1080
	LG K11	5,3	1280 x 720
Alta	Samsung Galaxy S9	5.8	2960 x 1440
	Huawei P20	5.8	2244 x 1080

Imagen 7: Celulares más vendidos en 2018 según ohmygeek

Sin olvidar que Android también corre en tablet, wearables y tv con tamaños y resoluciones muy distintas.

Por esto es relevante entender y manejar algunos conceptos que permitan realizar un desarrollo capaz de mantener una interfaz adecuada dentro de la gran variedad de posibles configuraciones

Unidades de medida

- `Pixel`: Es la unidad mínima de una imagen digital. El pequeño cuadro que se aprecia a analizar con detalle una imagen
- `DP (density-independent pixels o DIP)`: Los píxeles de densidad independiente son unidades flexibles que se escalan a dimensiones uniformes en cualquier pantalla, por lo que las vistas serán escaladas según la densidad de la pantalla
- `SP (Scale-independent-Pixels)`: Los píxeles de escala independiente, tienen la misma función que los dp, la diferencia es que se aplican a textos, en el caso de Android para TextView o cualquier otro elemento que soporte una fuente.

El valor por defecto de un SP es el mismo que el valor por defecto de un dp . Por ejemplo:

```
<TextView
  android:id="@+id/textView"
  android:text="Tamaño de texto 25sp (Scale-independent-Pixels)"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:textSize="25sp"
/>
```

La diferencia entre DP y SP es que éste último preserva los ajustes del usuario respecto a la fuente. Los usuarios que tengan ajustes para agrandar el texto verán que el tamaño del texto concuerda con sus preferencias. Por ello, se recomienda el uso de esta unidad de medida para fuentes.

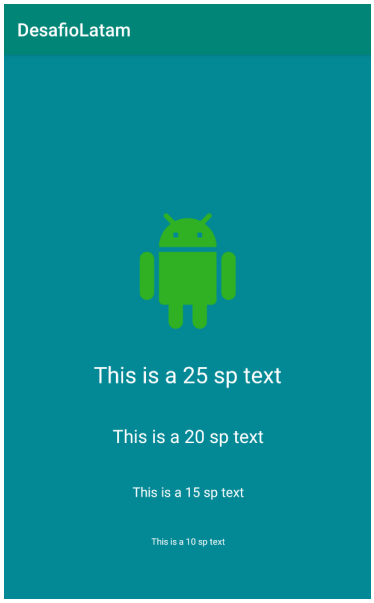

Imagen 8: Ejemplo unidades de medida

Densidad de píxeles

Es la cantidad de píxeles en un área física de la pantalla, y se conoce como ppp (puntos por pulgada).

Esto es diferente de la **resolución**, que es la cantidad total de píxeles en una pantalla.

Una mayor densidad de píxeles repercute en una mejor definición

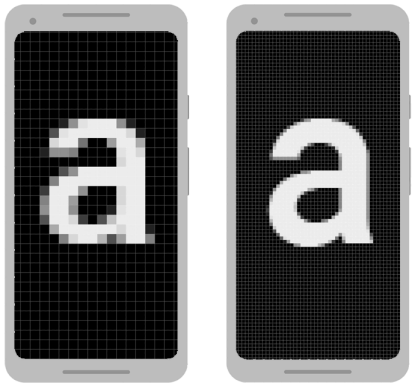

Imagen 9: Densidad de píxeles

Exageración de dos dispositivos del mismo tamaño y densidades de píxeles diferentes.

Compatibilidad con diferentes densidades

Dada la gran variedad de pantallas, Android agrupa las pantallas según su densidad de la siguiente forma

Table 1. Configuration qualifiers for different pixel densities.

Density qualifier	Description
`ldpi`	Resources for low-density (*ldpi*) screens (~120dpi).
`mdpi`	Resources for medium-density (*mdpi*) screens (~160dpi). (This is the baseline density.)
`hdpi`	Resources for high-density (*hdpi*) screens (~240dpi).
`xhdpi`	Resources for extra-high-density (*xhdpi*) screens (~320dpi).
`xxhdpi`	Resources for extra-extra-high-density (*xxhdpi*) screens (~480dpi).
`xxxhdpi`	Resources for extra-extra-extra-high-density (*xxxhdpi*) uses (~640dpi).
`nodpi`	Resources for all densities. These are density-independent resources. The system does not scale resources tagged with this qualifier, regardless of the current screen's density.
`tvdpi`	Resources for screens somewhere between mdpi and hdpi; approximately 213dpi. This is not considered a "primary" density group. It is mostly intended for televisions and most apps shouldn't need it—providing mdpi and hdpi resources is sufficient for most apps and the system will scale them as appropriate. If you find it necessary to provide tvdpi resources, you should size them at a factor of 1.33*mdpi. For example, a 100px x 100px image for mdpi screens should be 133px x 133px for tvdpi.

Imagen 10: Tabla de compatibilidad de diferentes densidades

Para dispositivos de media densidad tendremos recursos dedicados que permitan una mejor experiencia, y en los dispositivos de alta densidad se puede aprovechar la capacidad con una gráfica más detallada

Estas densidades se ven reflejadas en los íconos de lanzamiento (launcher) teniendo uno para cada densidad

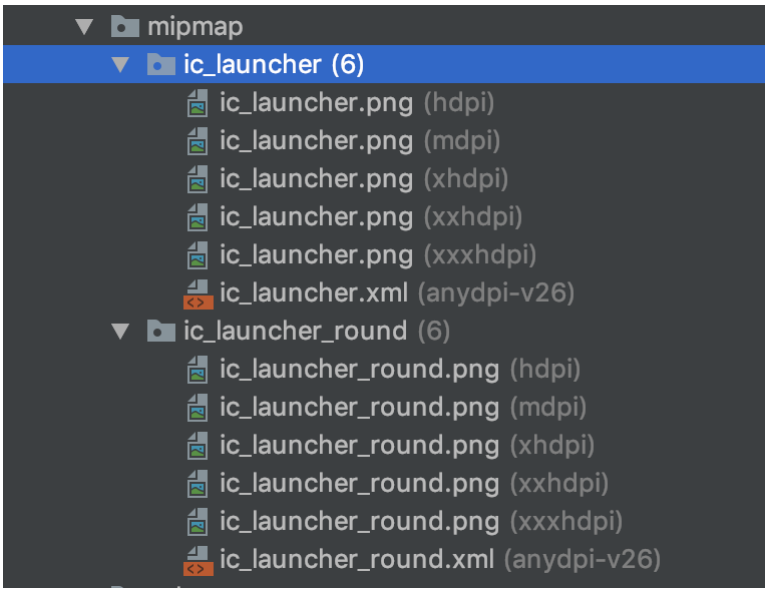

Imagen 11: Densidades reflejadas en los íconos de lanzamiento

Como vimos la herramienta Image Asset, cuándo creemos nuestros íconos personalizados para las apps podemos ver que se genera igualmente para las distintas densidades

Como curiosidad, en la actualidad (2019), Android ha “silenciado” a los teléfonos de gama baja. A pesar de existir en la documentación oficial y dar soporte a *ldpi*, actualmente Android Studio la ha abandonado. En la figura sobre los íconos lanzadores que son los valores por defecto, ldpi ya no tiene un recurso asignado

Compatibilidad de pantalla

El **tamaño de pantalla** es el espacio visible proporcionado por la interfaz de la app. El tamaño de la pantalla para la app, no es el tamaño real de la pantalla del dispositivo. Debemos considerar la orientación de la pantalla y las decoraciones del sistema, por ejemplo, de la barra de navegación y si la app fue redimensionada

Layouts

Competencias

- Manejar concepto View y ViewGroup
- Conocer cómo funciona Linear layout
- Conocer cómo funciona Relative layout
- Conocer cómo funciona ConstraintLayout
- Crear constraints usando ConstraintLayout

Introducción

El layout define la estructura de los componentes de la interfaz y su comportamiento para compatibilizar con los distintos tamaños de pantalla y densidades. Es la parte visible de la aplicación con la que interactúa el usuario, siendo el puente entre ambos. Debe ser llamativa en los visual pero sencilla en su uso para lograr que la app sea utilizada por el usuario

Está escrito en XML y compuesto por elementos que heredan de la clase View y ViewGroup, donde View dibuja un elemento con el que el usuario interactúa y ViewGroup es un contenedor invisible que define la estructura del diseño para los objetos View y ViewGroup

Un Viewgroup es una vista especial que puede contener otras vistas dentro, siendo éstas hijas de la vista contenedora.

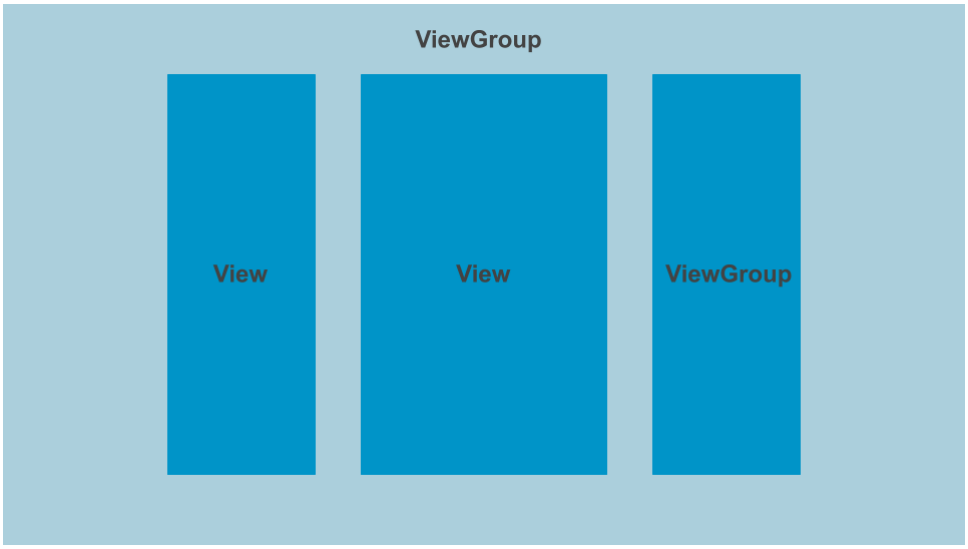

Imagen 12: ViewGroup

Cada subclase de ViewGroup tiene una manera única de desplegar las vistas, colocando los elementos según criterios distintos. En el SDK de Android se incluyen algunos tipos de layout de uso común

Linear Layout

Imagen 13: Linear Layout

Define líneas invisibles que ordenan los elementos según el orden de aparición dentro del XML. Es el layout más simple y de los primeros en utilizarse en Android y puede ser usado con orientación horizontal o vertical

Horizontal

Imagen 14: Layout horizontal

Vertical

Imagen 15: Layout vertical

Es una forma simple de ordenar layout que no son complejos al solo desplegar una línea o una columna, y para lograr una interfaz más compleja se necesita anidar layouts, lo que lo hace ineficiente a la hora de desplegarlos en tiempo de ejecución

Relative Layout

Imagen 16: Relative layout

Es un grupo de vistas que se agrupan en forma relativa, donde la posición de cada vista puede ser especificada en relación a otros elementos dentro del layout (arriba, abajo, a la izquierda o derecha de otro elemento) o en posiciones relativas al área padre, o sea, desde los márgenes

En el siguiente layout, el textView utiliza `android:layout_toEndOf="@+id/imageView"` para indicar que se va a ubicar al final del imageView, haciendo su posición relativa y `android:layout_width="match_parent"` para hacer que el texto se adapte al ancho disponible

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@drawable/background_grandient"
  tools:context=".MainActivity">

  <ImageView
     android:id="@+id/imageView"
     android:layout_width="93dp"
     android:layout_height="94dp"
     android:layout_alignParentStart="true"
     android:layout_alignParentTop="true"
     android:layout_marginTop="108dp"
     app:srcCompat="@drawable/ic_sunny" />

  <TextView
     android:id="@+id/textView"
     style="@style/HeaderTextView"
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:layout_below="@+id/imageView"
     android:layout_marginStart="23dp"
     android:layout_marginTop="-104dp"
     android:layout_marginEnd="10dp"
     android:layout_toEndOf="@+id/imageView"
     android:text="El sol está a una temperatura de 5,505 °C"
     app:layout_constraintTop_toBottomOf="@+id/imageView" />
</RelativeLayout>
```

Dependiendo del espacio disponible en la pantalla se adapta la ubicación de los elementos, por ejemplo, cuando se rota el dispositivo, el ancho de la pantalla aumenta y puede ser utilizado por los elementos

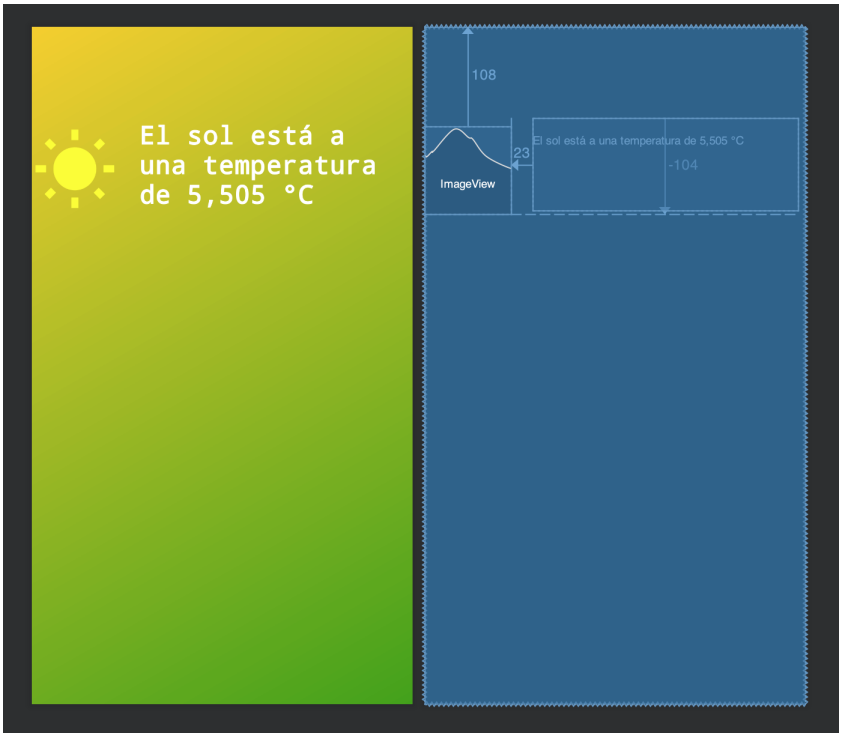

Imagen 17: Vista de diseño en portrait

Imagen 18: Vista de diseño en landscape

Si bien se mantienen los márgenes al parent y entre los elementos, el textView se adapta al tamaño de la pantalla

Los atributos para posicionar una vista son:

- layout_toRightOf
- layout_toLeftOf
- layout_toTopOf
- layout_toBottomOf

Constraint Layout

Con JetPack llega un nuevo layout que es parecido a RelativeLayout pero evolucionado, que usando restricciones basadas en las relaciones entre vistas dentro del layout permite reducir el número de vistas anidadas y así mejorar el rendimiento

A pesar de ser agregado recientemente tiene una compatibilidad desde la API version 9, por lo que en dispositivos antiguos funciona igualmente

Algunas de las ventajas sobre Relative layout son:

- El editor de diseño está desarrollado para y en función del Constraint layout, permitiendo hace el diseño de la interfaz visualmente
- Relative es un layout de dos pasos, debe medir dos veces el layout para mostrarlo. Constraint no, por lo que tiene mayor velocidad
- Constraint maneja el concepto de bias (tanto horizontal y vertical), que se define como el porcentaje de desplazamiento desde el centro.

Una **constraint** es una regla definida para una vista donde declara su posición horizontal y vertical dentro de la pantalla. Estas constraints son utilizadas para lograr distintos tipos de comportamiento dentro del layout

Tipos de constraints

Parent position

Restringe el lado de la vista al borde correspondiente del layout en forma horizontal y vertical

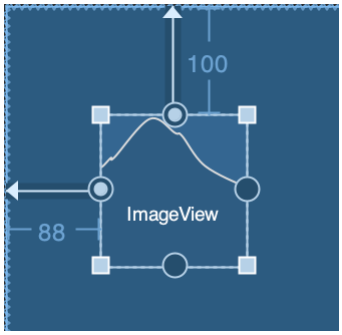

Imagen 19: Parent position

Order position

Define el orden de aparición de dos vistas, tanto vertical como horizontalmente. En este caso, Button B siempre estará a la derecha de la imagen por un margen de 56 dp que se logra usando `app:layout_constraintStart_toEndOf="@+id/imageView"` y `android:layout_marginStart="56dp"` como atributos de Button B

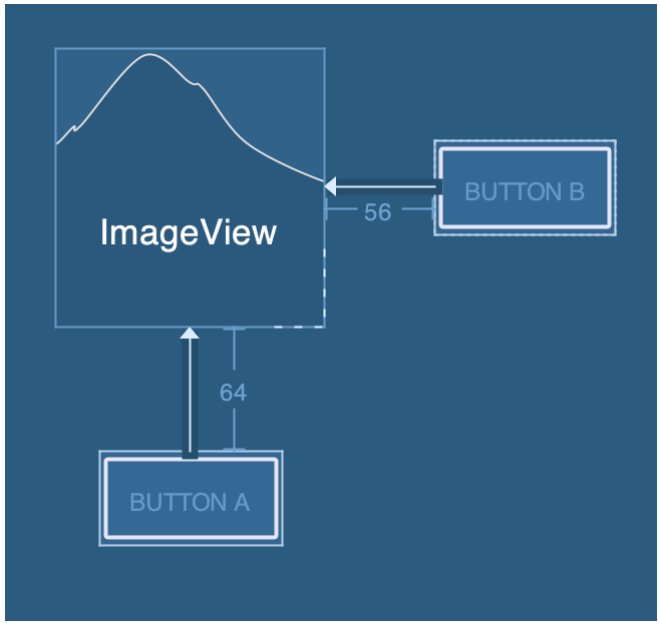

Imagen 20: Order position

Alignment

El alineamiento de un borde de una vista coincide con el mismo borde de otra vista

El borde izquierdo de ImageView es utilizado para alinearse por parte de Button A y B

También se pueden agregar márgenes a las vistas, por ejemplo, Button A tiene un offset de 24dp que se logra `android:layout_marginStart="24dp"` y `app:layout_constraintStart_toStartOf="@+id/ivWeather"`

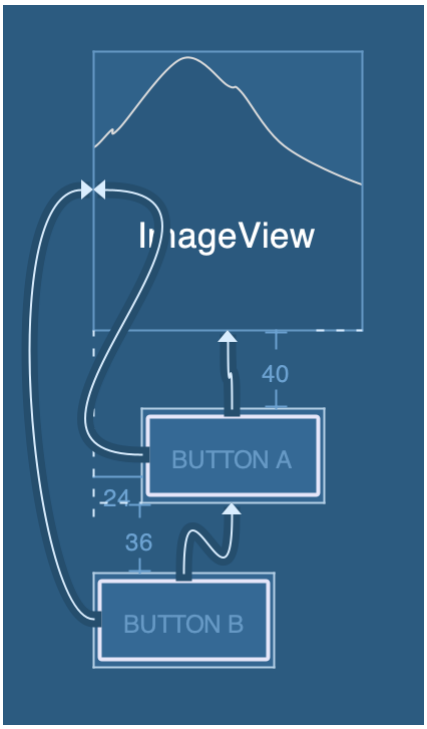

Imagen 21: Alignment

Baseline Alignment

Alinea la línea base del texto en una vista con la línea base del texto de otra vista

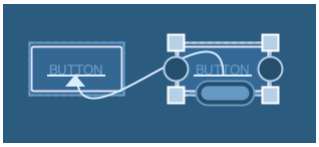

Imagen 22: Baseline alignment

Guideline

Se puede agregar una Guideline (línea guía) vertical u horizontal para restringir las vistas, usando dp o porcentaje relativo a los bordes del layout. La línea punteada que se agrega al diseño no es visible por el usuario cuando la app es ejecutada

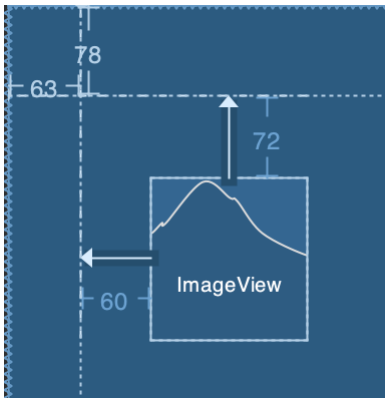

Imagen 23: Guideline

Bias

Usando los atributos *layout_constraintHorizontal_bias* y *layout_constraintVertical_bias* podemos descentrar la vista en un porcentaje relativo a los bordes de la pantalla

Bias ofrecen un posicionamiento simple y transparente de la vista a través de diferentes densidades y tamaños de pantalla.

En la vista de diseño, la paleta de atributos ubicada a la parte derecha tiene un layout que permite cambiar los bias.

En forma horizontal

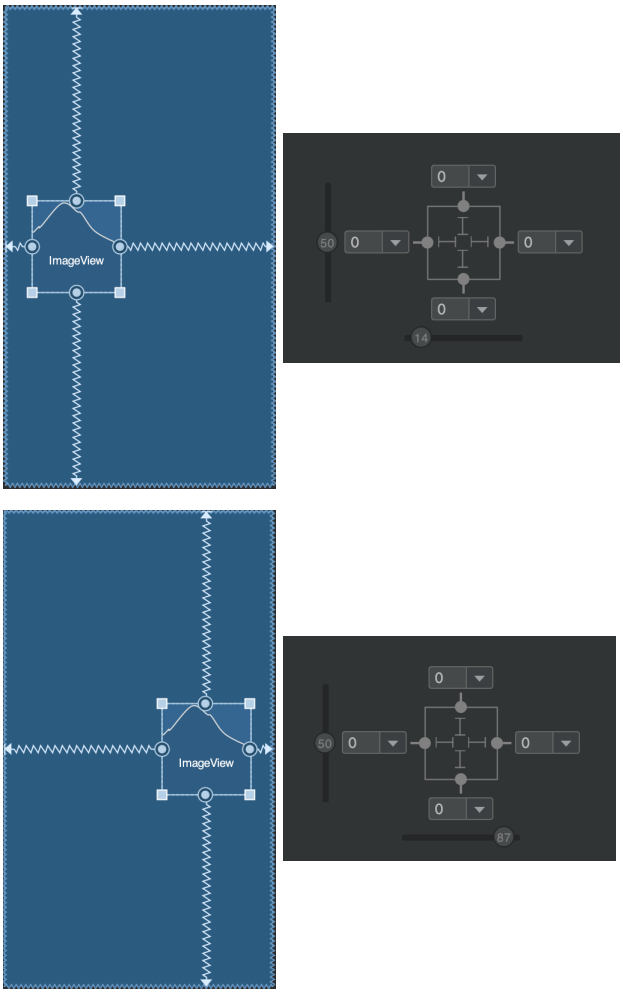

Imagen 24: Bias horizontal

En forma vertical

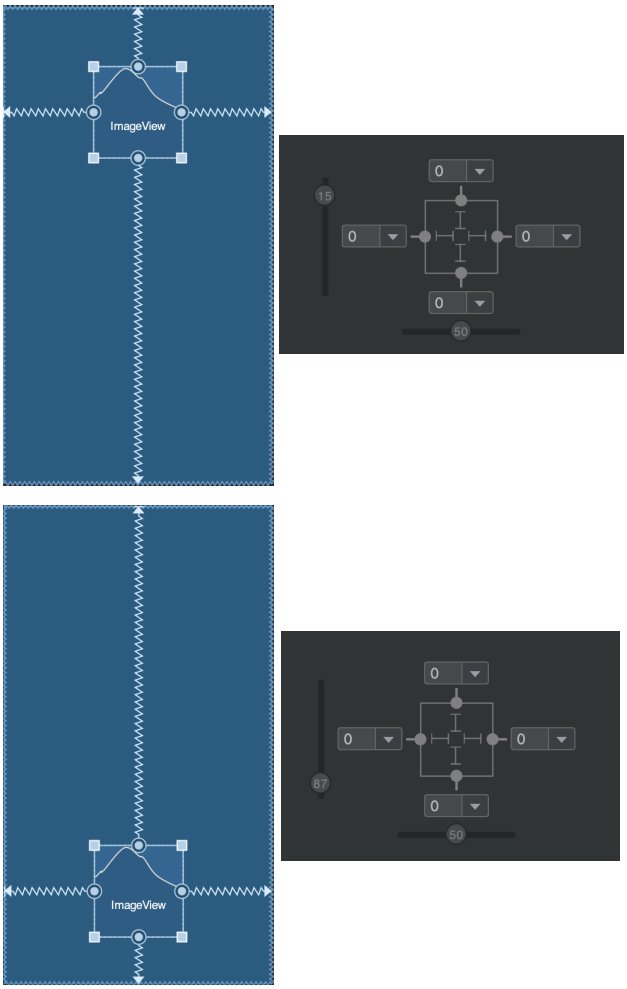

Imagen 25: Bias vertical

Finalmente, los beneficios de utilizar ConstraintLayout para manejar interfaces complejas o donde la actualización rápida sea importante son medibles

Los desarrolladores de Android el 2017 publicaron este interesante post sobre los beneficios en el rendimiento de ConstraintLayout vs RelativeLayout

Cómo usar las constraints

Las constraints se muestran para cada vista como 4 círculos en su borde que corresponden a *top, bottom, left, right*

Imagen 26: Vistas top, bottom, left, right

Cada círculo corresponde a una constraint y puede ser enlaza con los círculos de otras vistas uniendo los puntos en el editor

Imagen 27: Crer Top Constraint

Cuando se crea una constraint con otra vista son enlazadas visualmente por el editor

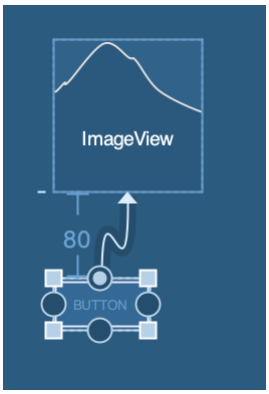

Imagen 28: Visualización con constraint creado

Una vez que la constraint ha sido creada puede ser modificada o eliminada. Si la vista se desplaza la constraint es inmediatamente recalculada

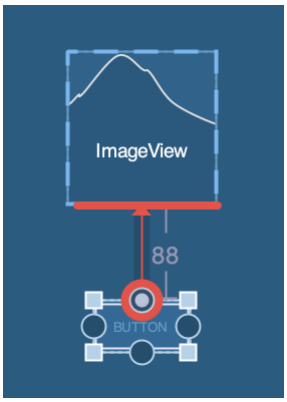

Imagen 29: Constraint recalculada