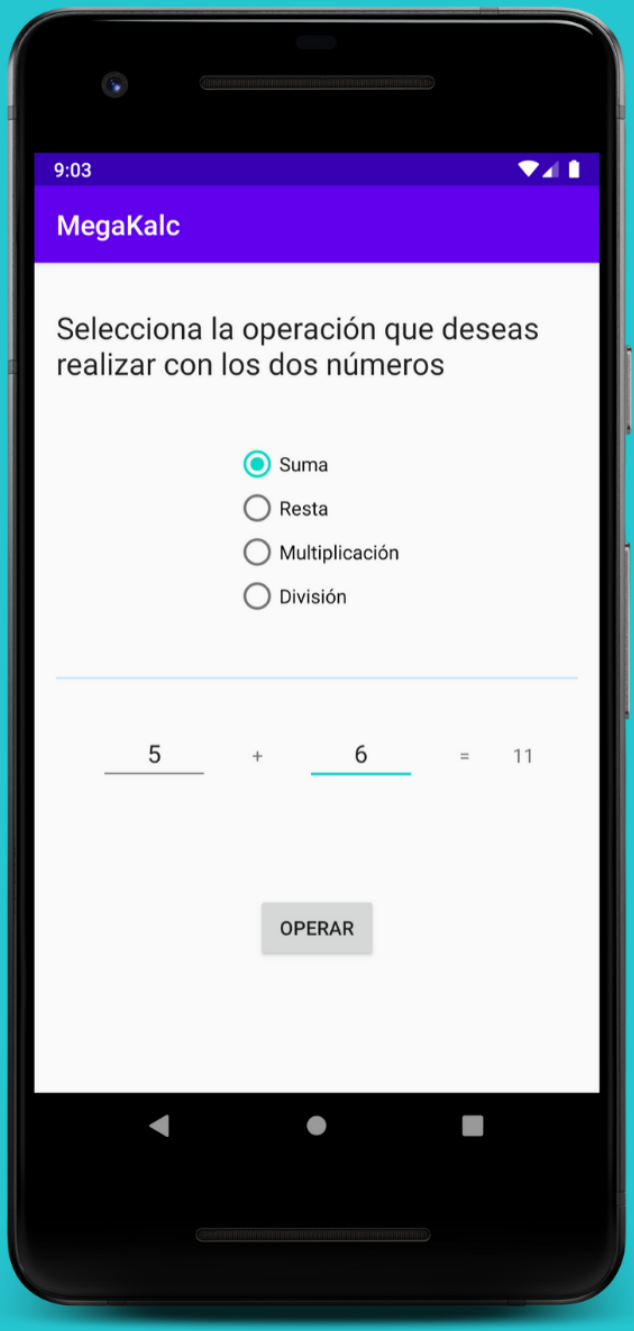
Crea Tu Primera Aplicación Android Desde Cero



Hermosa Programación

¿Acabas de comenzar a desarrollar apps Android y ya quieres rendirte?…

¡Tranquilo!

Esa es la sensación más común entre todos los seres humanos cuando se aventuran en algo que aún no conocen.

Así que es normal que te angusties y sientas ganas de abandonar. Ya te pasará con el tiempo si aplicas lo que te digo en esta guía.

Te Mostraré Como Crear Tu Primer App

Estás emocionado porque instalaste *Android Studio*...

Al fin se han actualizado los componentes del *SDK de Android* y también se configuró el *JDK de Java*.

Con emoción creas un nuevo proyecto de prueba, luego lo ejecutas y puedes ver como tu *AVD* reproduce un increíble `"Hello World!"`.

Sientes esa gran satisfacción de logro y sonríes a la vida, ya que diste tus primeros pasos en el desarrollo Android.

Lee también Instalar Android Studio

Pero… ¿qué pasa luego?...

Intentas realizar más acciones con tu miniaplicación que Android Studio ha creado automáticamente, pero las carpetas y archivos que se encuentran en la jerarquía de tu proyecto te abruman con extensiones y nombres raros.

Arrastras botones y campos de texto en la pestaña de diseño intentando llevar a cabo alguna acción.

Navegas por todas las opciones del menú superior, cambias variables, pruebas métodos y nada funciona.

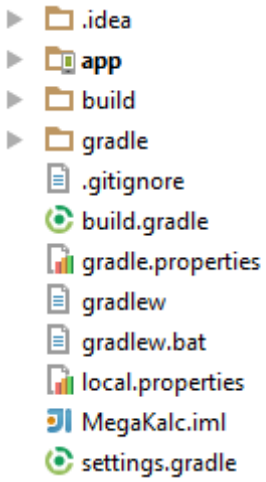
Ni siquiera se te pasa por la cabeza leer la documentación del sitio oficial o descargar un libro para entender mejor de que va esa condenada estructura de un proyecto Android.

Pero no te preocupes, voy darte las luces necesarias para comprender la estructura de un proyecto en **Android Studio** y la forma en que se compone un **Módulo de Aplicación**.

Los directorios principales en Android Studio

Al igual que otros *IDEs*, existe un panel donde se muestra la estructura del proyecto que se ha generado para comenzar a desarrollar la aplicación. Esta estructura representa la jerarquía o esqueleto de tu espacio de trabajo y como desarrollador es primordial aprender a moverte sobre él.

¿Qué archivos y carpetas contiene el proyecto?



Por el momento son los que ves anteriormente.

¿Quiere saber de qué se trata?

Bien:

`.idea`

Sistema de archivos de intelliJ *IDEA* que contiene los archivos de configuración del proyecto que se está desarrollando.

`app`

Representa el módulo funcional de la aplicación que se está desarrollando. Dentro de ella se encuentra todos los directorios y archivos que componen el aplicativo.

`build`

Guarda un historial de las construcciones generadas del módulo.

`gradle`

Contiene el archivo **gradle-wrapper** para la construcción automática de la construcción.

`.gitignore`

Archivo de **Git** para ignorar el seguimiento de la versión de otros archivos.

`build.gradle`

Permite configurar las características del sistema de construcción **Gradle**, como la inclusión de librerías de soporte, la certificación de aplicaciones, versiones de compilación, etc.

`gradle.properties`

En este archivo se pueden añadir líneas de comando para modificar las opciones de construcción de gradle.

`gradlew`

Script para iniciar gradle en el sistema operativo *Unix*.

`gradlew.bat`

Script para iniciar gradle en el sistema operativo *Windows*.

`local.properties`

Archivo del sistema de construcción que referencia la ubicación del **SDK** en tu computadora.

`<NombreDelProyecto>.iml`

Este archivo es parte del sistema de módulos de IntelliJ. Su propósito es guardar toda la metainformación del módulo que será construido en el proyecto.

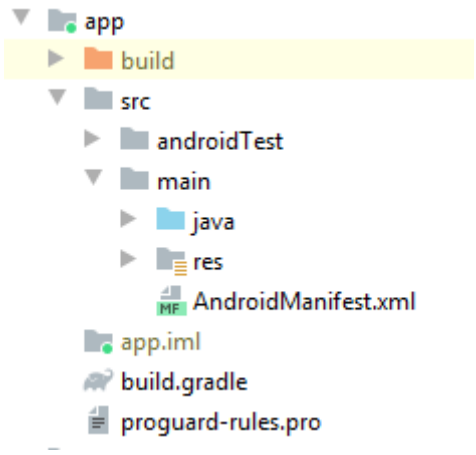
`settings.gradle`

Contiene la definición de los subproyectos que se construirán. Normalmente solo encontrarás el modulo *app*.

Este resumen demuestra que no es complicado el propósito de cada archivo y directorio, es más, por el momento la mayoría no se interesa para comprender las bases del desarrollo Android. Como principiante debes enfocarte en la lógica de la programación.

Estructura de un Módulo Android

Como acabas de ver, la carpeta `app` es la representación del módulo de ejecución de nuestra aplicación y es allí donde debemos enfocarnos a la hora de implementar la aplicación y obtener un archivo `.apk`.



Si despliegas su contenido se verán varios archivos y carpetas que componen la estructura de la aplicación. Veamos una breve definición para cada uno:

`build`

En esta carpeta se ubican los archivos de las construcciones que se generan del módulo. SU gestión es automática por lo que no debemos preocuparnos por modificar o crear dependencias aquí.

`src`

Aquí se encuentran todos los archivos fuente Java de tu proyecto y los recursos necesarios para integrar el funcionamiento de la aplicación.

`src/androidTest`

Contiene los casos de prueba de la aplicación. El framework de Android brinda facilidad para generar casos de prueba con *JUnit* en tus aplicaciones.

`src/main/java/<paquete>`

Esta carpeta contiene el código fuente java que construye la lógica y operatividad a la aplicación.

`src/main/assets`

No encontrarás creado este directorio, pero puedes generarlo para guardar recursos que tengan formato especial o sean tengan fines específicos, como archivos binarios, texturas o texto plano.

`src/main/res`

Contiene subcarpetas clasificadas por el tipo de recursos que usará tu aplicación. Recursos como cadenas, enteros, imágenes, layouts, etc.

`src/AndroidManifest.xml`

Archivo estandarizado en las aplicaciones Android que define el alcance y limitaciones de una aplicación Android, como la cantidad de actividades que tiene, las versiones de Android

donde la aplicación se ejecutará, los permisos que requiere la aplicación para acceder a datos externos, etc.

`app.iml`

Representa la composición del módulo app para intelliJ.

`build.gradle`

Archivo de configuración del sistema de construcción Gradle que permite personalizar las características de dependencia del módulo.

`proguard-rules.pro`

Archivo de configuración que optimiza el código del módulo para generar apks más livianas.

Por ahora ignoraremos los archivos del **Sistema de Construcción Gradle**, las **Pruebas Unitarias**, el **Control de Versiones en Git** y las **Reglas ProGuard**. La explicación de cada uno de estos temas es especializada, por lo que serán temas de otros ebooks o artículos. Quiero que te concentres en los archivos fuente y los recursos, ya que son la base de las aplicaciones.

Si te abrumas intentando comprender el significado de todas las instrucciones de esos archivos "*raros*" te desenfocarás del objetivo que es implementar el código de tu primera aplicación Android.

Aquí está tu mapa de navegación

Con frecuencia me encuentro personas que tienen un gran deseo por crear apps Android pero no saben cómo abordar dicho objetivo.

Les abruma la cantidad de información que ronda en la web sin saber por dónde iniciar.

Si esa es tu situación, tranquilo… los siguientes son pasos que según mi experiencia, debes seguir:

Paso 1. Pregúntate:

¿Soy graduado o estudiante de una profesión asociada a la programación?

Si la respuesta es sí, avanza al paso dos, de lo contrario pon atención a lo siguiente:

> Ser programador requiere como mínimo estudios de un campo llamado "Algoritmia". Además se requiere un razonable nivel de pensamiento lógico y cuantitativo.
>
> ¿Esto tiene solución?
>
> Claramente. Busca un buen curso online, curso técnico, blog, canal de youtube, etc. y práctica lo más que puedas. La idea es potenciar tu cerebro para la resolución de problemas.

De lo contrario en el mejor de los casos estarás perdiendo tiempo aprendiendo desarrollo Android sabiendo que existen prerrequisitos mínimos.

Anímate, con dedicación y un ritmo constante es más que posible.

Lectura recomendada: Fundamentos de Programación de Joyanes

Paso 2. Perfecto, ya tienes clara la movida. Por el momento Google usa **Java** y **Kotlin** para desarrollar apps, así que búscate un buen libro, videotutoriales y blogs para aprender con el que mejor estés asociado (si vas desde cero, entonces ve por Kotlin sin dudarlo).

Lecturas recomendadas:

- 5 Libros en español para aprender a programar en Java
- Introducción A Java Para Desarrollo Android
- Desarrolla apps para Android con Kotlin

Paso 3. Instala el entorno de desarrollo para Android. Por el momento este se compone del SDK de Android, el JDK de Java y el IDE Android Studio. Es todo… puedes ver como configurarlos en conjunto aquí:

Click aquí para ver cómo preparar el entorno de desarrollo Android

Paso 4. Si ya estás listo, entonces comienza a devorarte el resto de este ebook.

Te mostraré como crear una pequeña app de ejemplo que te mostrará a grandes rasgos la actividad de un Android Developer.

Inicia la lectura y da tu primer paso para convertirte en beginner.

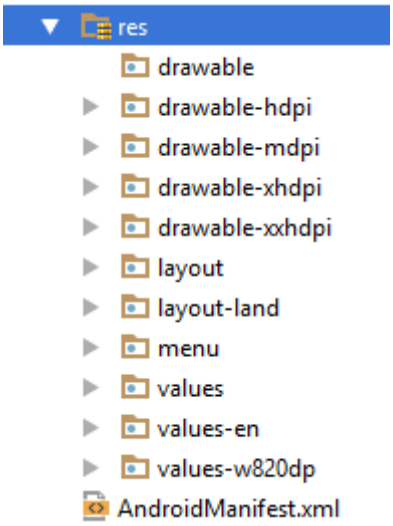
Recursos De Una Aplicación Android

Los recursos son archivos externos destinados a soportar el comportamiento integral de una aplicación.

Estos pueden ser **imágenes**, **cadenas**, **audios**, **archivos de texto**, **animaciones**, etc. El hecho es que al ser independientes permiten a tu aplicación adaptarse a cualquier configuración en distintos dispositivos móviles.

Configuraciones como el cambio de **Orientación de la Pantalla**, la variación del **Idioma**, selección de estándares de **Teclado**, las **Dimensiones de la Pantalla**, y mucho más.

Recuerda que en el apartado anterior vimos que por estándar se usa la carpeta ***res*** para guardar todos estos recursos. En su interior contiene subdivisiones respecto al tipo de recurso que se alberga. Cada uno de estos subdirectorios tiene un nombre especificado para la lectura de la aplicación Android.



> **Intentar crear carpetas de recursos distintas**: Android trae automatizado y preconfigurado los tipos y nombres de recursos que pueden usarse, por lo que no se reconocerían carpetas con formatos distintos creadas por el programador.

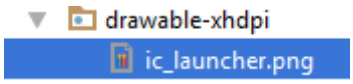
Tipos de Recursos

La siguiente tabla ilustra los tipos de recursos más usados en un proyecto Android:

Nombre del Directorio	Contenido
`animator/`	Archivos XML que contienen definiciones sobre animaciones.
`anim/`	Archivos XML que contienen definiciones de otro tipo de animaciones.
`color/`	Archivos XML que contienen definiciones de listas de estados de colores.
`drawable/`	Contiene mapas de bits (`.png`, `.9.png`, `.jpg`, `.gif`) o archivo XML que contienen definiciones de elementos especializados en el API de dibujo de Android.

`layout/`	Archivos XML que contienen el diseño de interfaz de usuario.
`menu/`	Archivos XML que contienen la definición de la estructura de un menú.
`raw/`	Archivos con otro tipo de extensión.
`values/`	Archivos XML que contienen definiciones sobre valores simples como colores, cadenas y números enteros. A continuación algunas de las convenciones más comunes: • `arrays.xml` para arreglos • `colors.xml` para colores • `dimens.xml` para medidas • `strings.xml` para cadenas • `styles.xml` para estilos de la aplicación
`xml/`	Archivos XML que tienen como propósito ser leídos en tiempo real para alguna tarea de parsing o estructuración.

Un ejemplo de un recurso *drawable* es la imagen *png* que representa el icono de la aplicación. Si exploras su contenido notarás que se le llamada `ic_launcher.png`.



Calificadores

Un calificador es una notación gramatical que condiciona el uso de los recursos en una aplicación. Los calificadores están directamente relacionados con las características del dispositivo donde se ejecuta la aplicación Android.

Como has visto cuando creas un nuevo proyecto en Android Studio, vienen una serie de carpetas *drawable* con diferentes nombres.

Cada especificación añadida más un guion es un calificador que condiciona los recursos por el tipo de densidad de pantalla establecida.

Por ejemplo:

`drawable-hdpi` contendrá los recursos drawables que la aplicación usará si es ejecutada en un dispositivo con alta densidad (más de *240dpi* para ser exactos).

En ese caso, el framework de Android ayudará a la aplicación a elegir los drawables en el directorio correspondiente.

La sintaxis para crear un calificador en un tipo de recurso es la siguiente:

```
<nombre_del_recurso>-<calificador>
```

Donde `nombre_del_recurso` es claramente el nombre del directorio, por ejemplo, `values`. Y `calificador` es una notación estandarizada que identifica el tipo de configuración.

Si deseas usar varios calificadores debes separar por guiones cada uno, por ejemplo:

```
drawable-es-hdpi
```

El anterior ejemplo adapta un conjunto de drawables a un dispositivo cuya configuración se basa en el idioma Español más una densidad de pantalla alta.

Al igual que la densidad del dispositivo, también podemos encontrar un sin número de características de configuración adicionales. A continuación veremos una tabla que mostrará las categorías de calificadores más populares:

Configuración	Sintaxis del Calificador	Definición

Idioma y Región	`en`
	`fr`
	`en-rUS`
	`fr-rFR`
	`fr-rCA`
		La sintaxis del lenguaje se establece en la norma ISO 639-1 , la cual explica que se usan dos letras descriptivas. Si deseas añadir la región específica, entonces usa la descripción estipulada en la normar ISO 3166-1-alpha-2 .

Dirección de la escritura	`ldrtl` `ldltr`	Usa el valor *ldrtl* para definir "layout-direction-right-to-left", es decir, escritura de derecha a izquierda. Y ldltr para expresar "layout-direction-left-to-right" o escritura de izquierda a derecha (que es el valor por defecto).

Densidad mínima	`sw<N>dp`
	
	Ejemplos:
	`sw320dp`
	`sw600dp`
	`sw720dp`
		Define la densidad mínima para filtrar los recursos de la aplicación (*Smallest Witdh*).

Ancho mínimo disponible	`w<N>dp`
	
	Ejemplos:
	`w720dp`
	`w1024dp`
	etc.
		Filtra por el ancho de pantalla actual de la pantalla en dps. Usa el identificador `<N>` para establecer el número respectivo.

Alto mínimo disponible	`h<N>dp`
	
	Ejemplos:
	`h720dp`
	`h1024dp`
	etc.
		Filtra por el alto mínimo que debe estar disponible en la pantalla del dispositivo. Usa el identificador `<N>`para establecer el número respectivo en unidades *dp*.

Tamaño de la pantalla	`small`
	`normal`
	`large`
	`xlarge`
		Permite configurar los recursos dependiendo del tamaño de la pantalla. Por tamaño entiéndase las siguientes resoluciones aproximadas:
		
		small: 320x426 dp
		
		normal: 320x470 dp
		
		large: 480x640 dp
		
		xlarge: 720x960 dp

Proporción de la pantalla	`long` `notlong`	Se usa la proporción o ratio de la pantalla del dispositivo para incluir los recursos. **long**: Pantallas largas como los formatos WQVGA, WVGA, FWVGA. **notlong**: Pantallas cortas como QVGA, HVGA, and VGA.

Orientación de la pantalla	`port` `land`	**port**: El dispositivo se despliega como portarretrato o verticalmente **land**: El dispositivo se despliega horizontalmente.

Densidad de la pantalla (dpi)	`Ldpi`
	`mdpi`
	`hdpi`
	`xhdpi`
	`xxhdpi`
	`xxxhdpi`
	`nodpi`
		ldpi: Low-density, densidades de máximo 120dpi.
		mdpi: Medium-density, máximo 160dpi.
		hdpi: High-density, máximo 240dpi.
		xhdpi: Extra-high-density, máximo 320dpi
		xxhdpi: *Extra-Extra-High-Density*, máximo 480dpi.
		xxxhdpi: Extra-extra-extra-high-density máximo 640dpi.
		nodpi: aquí puedes situar objetos que no quieres redimensionar sea cual sea la densidad.

Versión de Android o nivel del API	Ejemplos:
	`v3`
	`v4`
	`v7`
	etc.
		Nivel del API soportado por el dispositivo. Por ejemplo v16 para referirse a los dispositivos con Android *JellyBeans 4.1*.

Los recursos string

Para aquellos recursos que contienen información de texto se debe usar el archivo *strings.xml*. Este contiene una serie de ítems con un identificador llamado `name` y su contenido respectivo.

Si abres este archivo ubicado en la carpeta ***values***, verás una definición parecida a la siguiente:

```
<resources>
    <string name="app_name">MegaKalc</string>
</resources>
```

Este archivo de recursos debe contener como nodo principal la etiqueta `<resources>` para hacer énfasis en la creación de recursos. Cada cadena de texto debe ser declarada dentro de elementos `<string>`, donde su clave se establece con el atributo `name` y el contenido va en línea.

¿Cómo acceder a los recursos?

Existen dos formas para acceder a un recurso: En el código a través de la clase `R` o en definiciones XML.

Acceder a los recursos en el código

Para acceder en el código debes referirte a la clase `R.java`, un archivo especial que es generado automáticamente cuando tu aplicación Android es compilada.

Esta clase contiene la referencia a todos los recursos que usarás en tu proyecto.

Cada referencia está declarada como un número hexadecimal ligado al recurso lógico.

Las tipos de recursos son representados por clases internas y sus atributos son los recursos particulares como tal.

Veamos un ejemplo de cómo luce el archivo:

```
public final class R {
    public static final class anim {
        public static final int abc_fade_in=0x7f040000;
        public static final int abc_fade_out=0x7f040001;
        public static final int abc_grow_fade_in_from_bottom=0x7f040002;
        public static final int abc_popup_enter=0x7f040003;
        public static final int abc_popup_exit=0x7f040004;
        public static final int abc_shrink_fade_out_from_bottom=0x7f040005;
        public static final int abc_slide_in_bottom=0x7f040006;
        public static final int abc_slide_in_top=0x7f040007;
        public static final int abc_slide_out_bottom=0x7f040008;
        public static final int abc_slide_out_top=0x7f040009;
    }
    public static final class attr {
    ...
```

Eso significa que para llamar un recurso mediante a la clase `R`, sigues la sintaxis:

```
[<paquete>].R.<tipo_de_recurso>.<nombre_del_recurso>
```

Donde,

- `<paquete>`: Es el nombre del paquete en que se encuentra el recurso. Si el recurso es de tu propio proyecto no requiere este parámetro.

- `<tipo_de_recurso>`: Representa que tipo de recurso buscarás. Por ejemplo si buscas un String entonces usas la subclase `string`.

- `<nombre_del_recurso>`: Si buscas un archivo, entonces es el nombre de este. Si lo que buscas es un valor simple definido en XML, entonces usas el atributo `android:name` de dicho recurso.

Un ejemplo de acceso sería el siguiente:

```
TextView titulo = (TextView) findViewById(R.id.textoTitulo);
titulo.setText(R.string.titulo);
```

La primer línea obtiene un `TextView` y en la segunda se le asigna una cadena guardada en un string llamado título, que ha sido definido en el archivo `strings.xml`.

Recuerda que el API de Android nos provee todos los recursos que usa el sistema operativo en sus aplicaciones y comportamiento corriente.

Para accederlos debes hacer referencia al paquete Android, donde encontrarás una clase `R` que proporciona el acceso correspondiente.

Acceder a los recursos en XML

Similar que en el código Java, los recursos pueden ser accedidos a través de referencias XML con la siguiente sintaxis:

```
@[<paquete>:]<tipo_del_recurso>/<nombre_del_recurso>
```

En este caso se usa el símbolo `'@'` para navegar en el *DOM* de cada recurso XML definido con anterioridad. Si el recurso se encuentra en el paquete de tu aplicación no debes usar el nombre. Pero si está por ejemplo en el paquete de Android, entonces usa `android`.

Ejemplo:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceMedium"
    android:text="@string/titulo"
    android:id="@+id/textoTitulo"/>
```

Como ves, se le asigna al atributo `android:text` el string “*titulo*”.

Si quisieras asignar un color del paquete Android al fondo de un view puedes acceder de la siguiente forma:

```
android:background="@android:color/white"
```

Donde `white` es un recurso del framework para el color blanco.

Lee también *Diseñar Temas y Estilos para tus Aplicaciones Android*

Todos Hablan De “Activities”, Pero… ¿Qué Son?

Una **actividad** es un "pantallazo" de tu aplicación. Actúa como el contenedor principal de todos los componentes de interfaz siendo un equivalente a las ventanas que abren los sistemas operativos de escritorio.

Por ejemplo, el home de *Google Play* es una actividad:

Y cada una de las secciones de pantalla completa por donde vayas navegando es una actividad. En ellas se despliega todo el poder del diseño artístico y programático para obtener el mejor rendimiento y experiencia de usuario.

Ciclo de vida de una Actividad

Las actividades actúan como entidades vivas ante las acciones del usuario o del sistema.

Estas se autogestionan.

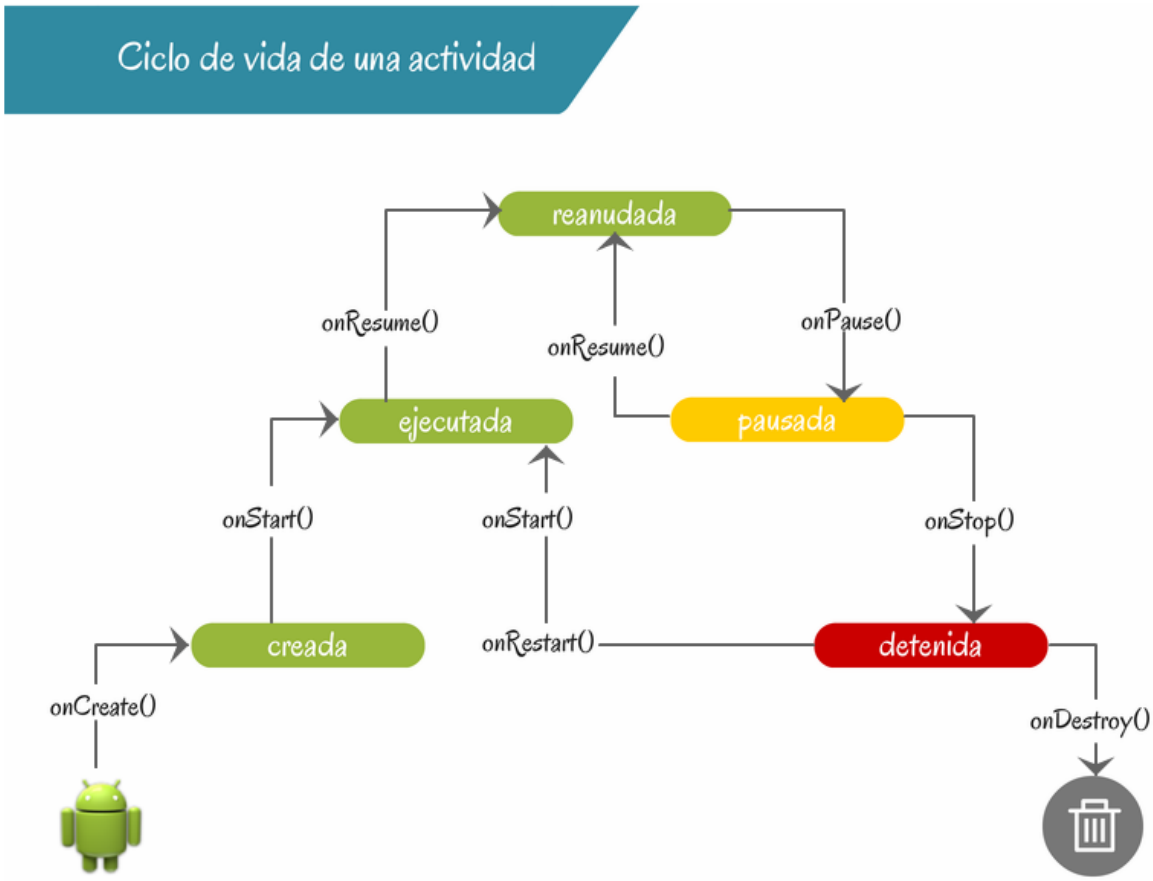
Tu solo automatiza su comportamiento. No tienes que iniciarlas, destruirlas, reanudarlas, dibujarlas, pausarlas, etc.

Este comportamiento es posible gracias al ciclo de vida de la actividad, el cual se creó para automatizar las acciones de una actividad en el sistema. Si has estudiado autómatas podrás hacerte una idea del flujo de trabajo que se empleará.

Cuando Android inicia la **Actividad Principal** de una aplicación, esta comienza a atravesar por una serie de estados que le permiten definir su comportamiento. No importa lo que pase en el sistema, la actividad siempre tendrá un curso definido de operación.

El rol del programador es solo definir qué acciones ocurrirán en la transición de estado a estado para que se cumpla el objetivo de la aplicación.

La siguiente ilustración muestra el ciclo de vida de una actividad:



Cada transición entre estados es representada por métodos callback en la API de Android. Recuerda que un método callback es un método que se llama automáticamente por el sistema en el momento que este lo requiera. El programador no tiene control sobre esta ejecución, pero si puede sobrescribir los métodos para que se lleven a cabo tareas personalizadas.

A continuación se explica cada estado de la actividad y sus respectivas transiciones:

- **Creada**: Una actividad se ha creado cuando su estructura se encuentra en memoria, pero esta no es visible aun. Cuando el usuario presiona sobre el icono de la aplicación en su dispositivo, el método `onCreate()` es ejecutado inmediatamente para cargar el layout de la actividad principal en memoria.

- **Ejecutada**: Después de haber sido cargada la actividad se ejecutan en secuencia a el método `onStart()` y `onResume()`. Aunque `onStart()` hace visible la actividad, es `onResume()` quien le transfiere el foco para que interactúe con el usuario.

- **Pausada**: Una actividad está en pausa cuando se encuentra en la pantalla parcialmente visible. Un ejemplo sería cuando se abren diálogos que toman el foco

superponiéndose a la actividad. El método llamado para la transición hacia la pausa es `onPause()`.

- **Detenida**: Una actividad está detenida cuando no es visible en la pantalla, pero aún se encuentra en memoria y en cualquier momento puede ser reanudada. Cuando una aplicación es enviada a segundo plano se ejecuta el método `onStop()`.
  Al reanudar la actividad, se pasa por el método `onRestart()` hasta llegar a el estado de ejecución y luego al de reanudación.

- **Destruida**: Cuando la actividad ya no existe en memoria se encuentra en estado de destrucción. Antes de pasar a destruir la aplicación se ejecuta el método `onDestroy()`. Es común que la mayoría de actividades no implementen este método, a menos que deban destruir procesos (como servicios) en segundo plano.

Este ebook se enfocará en el método `onCreate()`, debido a que es la base de la construcción de la actividad porque se incorporan las declaraciones de interfaz y controles. Los otros métodos se sobrescriben en situaciones más avanzadas, como por ejemplo, conservar datos cuando la aplicación se minimiza, cuando se cierra la aplicación, etc.

Otro aspecto a tener en cuenta es la existencia de una actividad principal en toda aplicación Android. Esta se caracteriza por ser aquella que se proyecta primero cuando el usuario hace **Tap** en el icono de la aplicación en su escritorio.

La clase Activity

Las actividades en el API de Android se representan por la clase `Activity`. Si ves tu proyecto de Android Studio, veras que ha creado una nueva clase que hereda de `AppCompatActivity`.

Esta es una subclase que soporta varios elementos nuevos en versiones anteriores de Android.

Normalmente su contenido es la sobrescritura del método `onCreate()`.

Veamos un ejemplo:

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

El método `onCreate()` del código anterior tiene dos instrucciones necesarias en cualquier actividad, la llamada a la referencia `super` obligatoria de la superclase y la creación del contenido a través de un recurso layout con `setContentView()`.

Si puedes notar, se accede a un recurso layout con `R.layout.activity_main` como parámetro de `setContentView()`. Esto se debe a que internamente se lleva a cabo el proceso de "inflado" de la interfaz.

Cuando leas o escuches este término debes saber qué hace referencia al proceso de convertir definiciones de elementos XML en objetos Java, es decir, un proceso de parsing.

El Archivo AndroidManifest.xml

Este archivo es como el pegamento entre los componentes de las aplicaciones para que funcionen como un todo dentro de Android. Cada aplicación debe contenerlo obligatoriamente debido a que la compilación y parsing se orienta en su contenido.

El archivo `AndroidManifest.xml` actúa como un indicador de las características de tu aplicación, estableciendo todos los componentes que tiene, sus limitaciones, los permisos que necesita, las versiones de Android donde correrá tu aplicación, etc.

Lee también Tutorial básico del lenguaje XML

Su definición XML se inicia con el elemento raíz `<manifest>`, el cual debe contener un nodo `<application>` que represente la existencia de una aplicación.

Veamos un ejemplo típico para un proyecto en blanco de Android Studio:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.hermosaprogramacion.megakalc" >

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="com.hermosaprogramacion.megakalc.MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Un archivo de manifesto tiene las siguientes características:

- El nodo raíz `<manifest>` hace referencia al namespace `android` definido en el API de Android. También usa el atributo `package` para referenciar el paquete donde se alberga el código fuente de tu aplicación.

- Es obligatorio que contenga un elemento `<application>` para declarar la presencia de una aplicación Android. Este elemento puede contener cualquier tipo de componentes. Generalmente siempre veremos un elemento `<activity>` para añadir una actividad dentro de la aplicación. Cada bloque de construcción que exista dentro de la aplicación se refiere a una clase Java que debe haber sido declarada con anterioridad.

- Para relacionar ambos entornos se usa el atributo `android:name` de los bloques con el fin de establecer el nombre del archivo java que contiene su referencia programática. En el ejemplo anterior el nodo `<activity>` se liga con `MainActivity`.

- El atributo `android:label` se refiere al título que desplegará la actividad en los lugares que se referencie (como la **Toolbar**). Normalmente Android Studio crea por defecto el string `app_name`, el cual contiene el nombre que asignaste en el asistente de creación del proyecto.

En el apartado la sección 3 se declaró que las aplicaciones Android siempre tendrán una actividad principal.

Precisamente los elementos `<intent-filter>` y `<action>` configuran esta característica.

La propiedad `android.intent.action.MAIN` especifica que la actividad en cuestión es la principal. Además será la primera que se iniciará ante el usuario (`android.intent.category.LAUNCHER`).

Este tema es más avanzado, por lo que la explicación se escapa de este ebook.

> Recuerda añadir cada actividad nueva declarada en tu código como elemento `<activity>` en el `AndroidManifest.xml`. Regularmente no es necesario porque Android Studio lo hará por ti, pero muchas personas redactan sus actividades desde cero y olvidan declararlas.

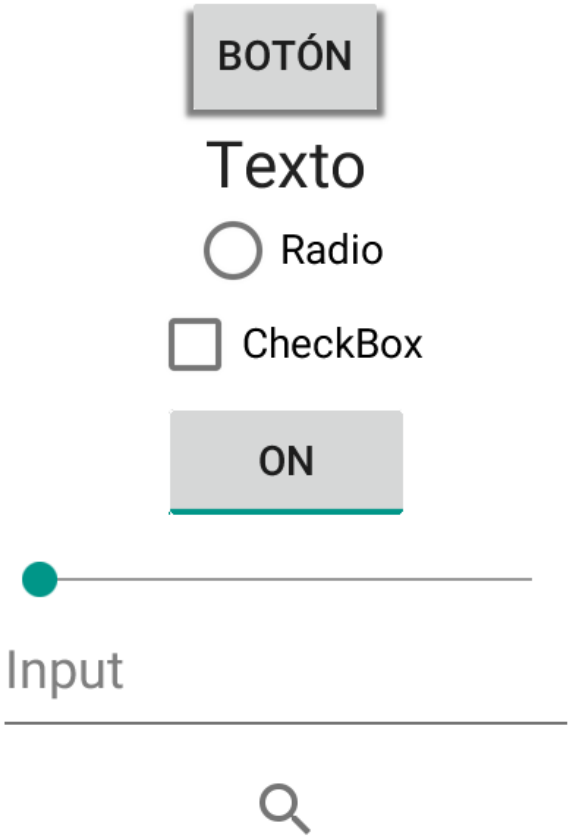
> **“¿Qué hago? La compilación genera 2 Apks”**: esto sucede porque declaraste dos actividades con la propiedad LAUNCHER. Elige solo la que cumpla el rol de principal si no tienes pensado realizar un proyecto con afinidades.

Diseñar la interfaz de una aplicación Android con Layouts

La definición de ***Layout*** se desprende de la definición de ***View***.

Un View es un componente que tiene representación gráfica. Se despliega en pantalla y además puede tener interacción con el usuario.

Ejemplos de estos son **botones**, **campos de texto**, **etiquetas**, **listas**, **radios**, etc.



Ahora, un Layout es un View que contiene más views. Su fin es proporcionar al programador un mecanismo para crear un diseño óptimo de los views que estarán disponibles para el usuario y su interacción.

Los layouts programáticamente son subclases de la superclase `ViewGroup`.

Aunque se pueden crear layouts personalizados, la API trae consigo implementaciones prefabricadas para no complicarse la vida.

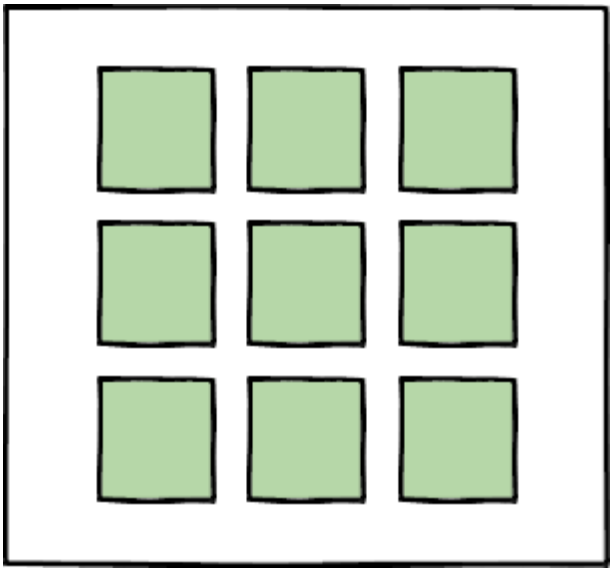
Algunas de ellas son: *ConstraintLayout*, *LinearLayout*, *RelativeLayout*, *GridView* y *WebView*.

Tipos de Layouts en Android

A continuación se verá una breve definición de las clases que representan los layouts más populares:

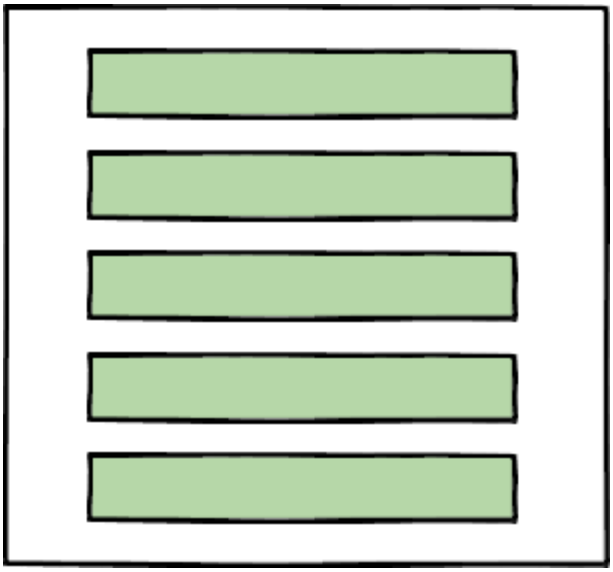
`FrameLayout`

Este layout bloquea secciones de la pantalla para mostrar un elemento por cada sección.



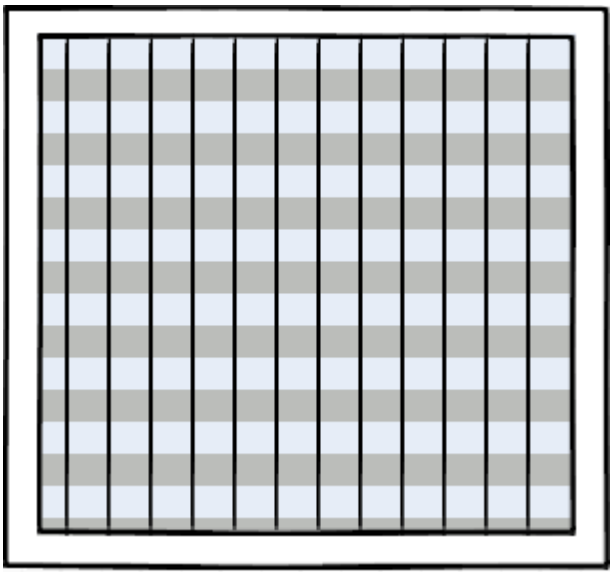
`LinearLayout`

Organiza los elementos en forma de filas o columnas.



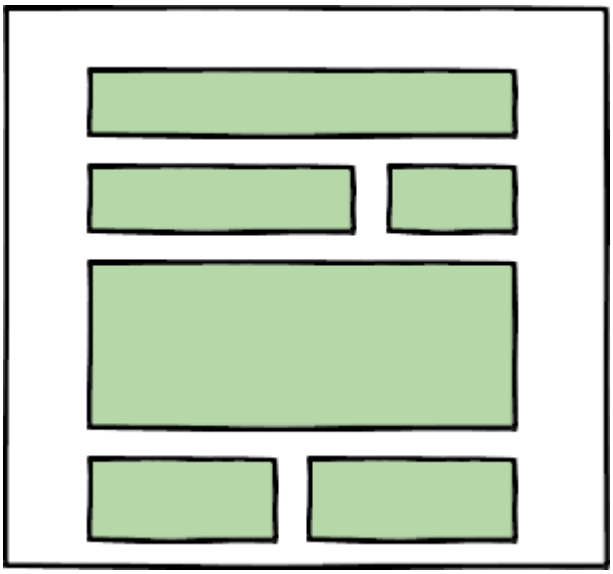
`GridLayout`

Este tipo de Layout tiene la capacidad de ubicar a sus hijos en forma de una grilla rectangular. Se basa en el recorrido de los índices que sectorizan sus celdas para añadir cada view.



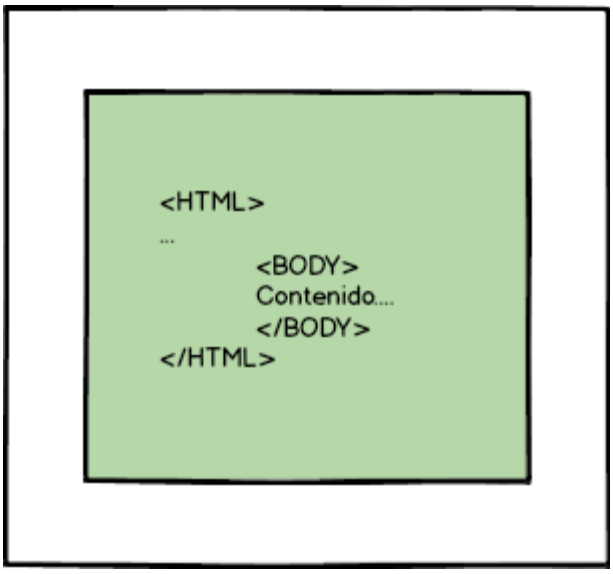
`RelativeLayout`

Este es uno de los views más utilizados, debido a que acomoda sus hijos con posiciones relativas. Cuando digo relativas, me refiero a que dependiendo de la ubicación de un view se determinará la posición de otro, lo que permite un ajuste instantáneo por si en algún momento las pociones cambian.

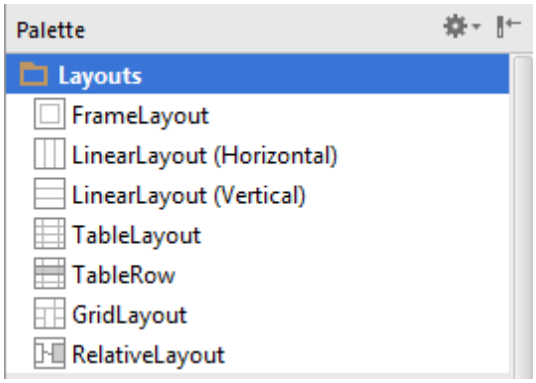


`WebView`

Muestra contenido web parseando código HTML.



La primera sección de la paleta de componentes en Android Studio precisamente se llama *Layouts*, donde encontrarás varios de estos elementos:



Entendiendo el RelativeLayout

El `RelativeLayout` es uno de los layouts más populares y usados para crear diseños flexibles y suavizados.

En él se puedes crear infinidad de distribuciones de views a través de referencias nominativas.

A diferencia de otros layouts que requieren contener a otros layouts para estructurar el diseño, situación que reduce el rendimiento de la aplicación.

Muchos desarrolladores se confunden a la hora de emplearlo, ya que las referencias pueden llegar a ser confusas si el conocimiento de inglés no es intuitivo.

Cuando digo referencias me refiero a que no expresaremos la ubicación de los componentes de esta forma:

> "*El botón OK* estará *ubicado en el punto (200, 120) del layout y sus dimensiones son 200x30 dp*"

A esa definición de atributos se le llama **definición absoluta**, y describe con pelos y señales cuales son las medidas numéricas para el **View**.

A diferencia de esa declaración, dentro de un `RelativeLayout` usaremos expresiones como la siguiente:

> "*El botón OK estará ubicado a la izquierda del extremo derecho del TextView 2 y sus dimensiones serán ajustadas al padre*"

¿Notas la diferencia?

No importa de qué tamaño sea la pantalla o que densidad maneje, el botón se ajustará relativamente a las condiciones que se le han impuesto.

Lo que significa: mejor experiencia para distintos usuarios sin importar las características de su dispositivo.

Ahora, para llevar estas descripciones a un view en su declaración XML se han de usar atributos especializados. Atributos que permiten acomodar el view en relación a su padre u otro view. Veamos algunos:

`android:layout_above`

Ubica el borde inferior del view en cuestión sobre el view que se declaró en su valor.

`android:layout_below`

El borde superior del view es ubicado por debajo del view asignado.

`android:layout_toRightOf`

El borde izquierdo del view es ubicado al lado derecho de otro.

`android:layout_toLeftOf`

El borde derecho del view es ubicado al lado izquierdo de otro.

`android:layout_alignTop`

El borde superior del view es alineado con el borde superior del relative layout.

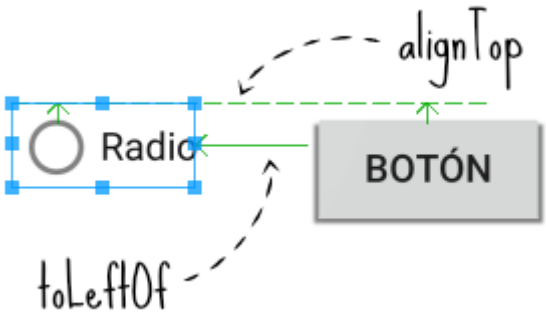
Existen muchos más atributos que se refieren a los lados restantes que podemos utilizar. Todos son intuitivos si conoces el significado en español de los adverbios empleados.

> Puedes conocer todos los atributos disponibles en la clase RelativeLayout.LayoutParams.

Por ejemplo…

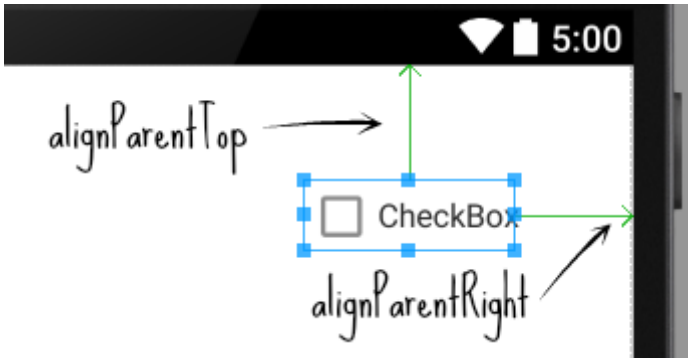
La siguiente ilustración muestra dos views alineados en un *RelativeLayout*.

El `RadioButton` se encuentra alineado a la izquierda del botón y además su borde superior coincide también:



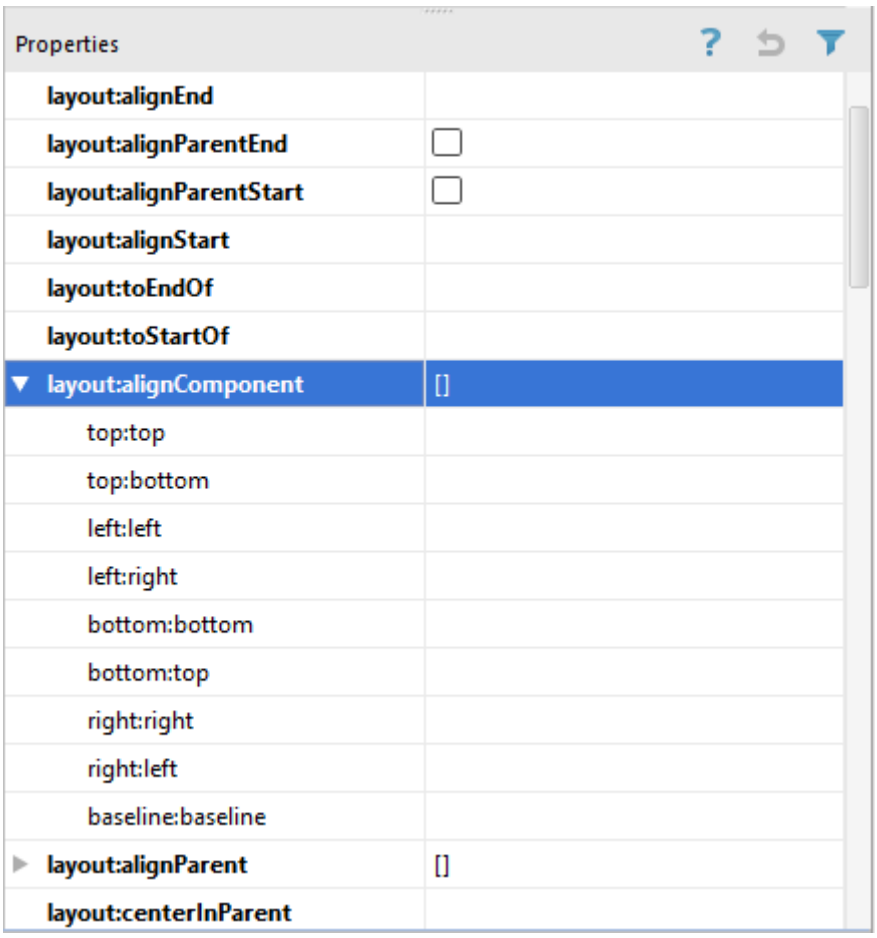
> **Android Studio** visualiza las posibles referencias entre views en su sección de diseño. No obstante se vuelve complejo a medida que se añaden más views, ya que las relaciones no se entienden o no son las más acordes. En este caso es necesario asignar las relaciones manualmente en el *Panel de Propiedades*.

Veamos otro ejemplo donde un `CheckBox` se alinea a los bordes superior y derecho del `RelativeLayout`:



El RelativeLayout en Android Studio

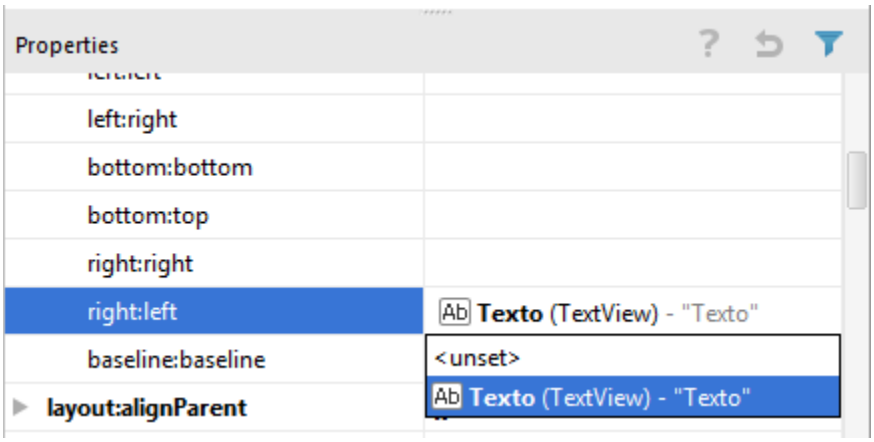
Android Studio provee un sistema muy interesante en la sección de diseño. Cuando se están distribuyendo los elementos este resume los atributos de referencia dentro del panel de propiedades de cada view.



La anterior ilustración visualiza todos los atributos que un view dentro de un `RelativeLayout` puede adquirir. Sin embargo para los atributos de relación entre views aparece una propiedad llamada `layout:alignComponent`, la cual contiene a todos los atributos de ubicación.

El sistema funciona de la siguiente manera: Existen una serie de pares de alineación, donde el primer término es el borde del view actualmente seleccionado y el segundo término es el borde del view con el que se quiere relacionar.

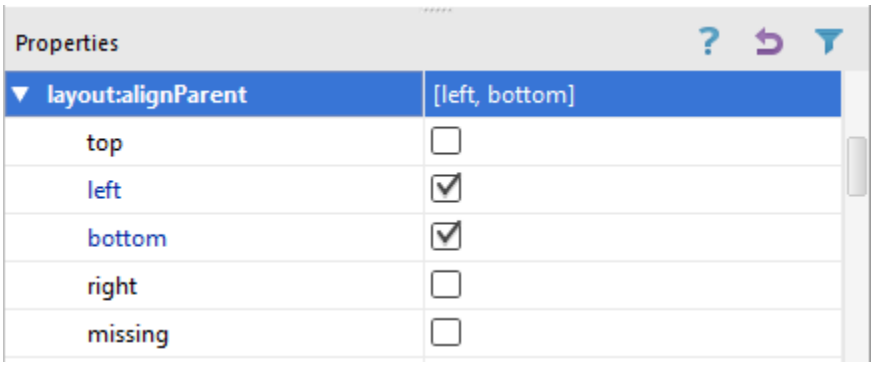
Ejemplo: Si quieres que el margen derecho de tu componente colinde con el borde izquierdo de otro view, entonces usas el par *right:left*. Solo se elige entre la lista que se despliega para seleccionar el view necesario.



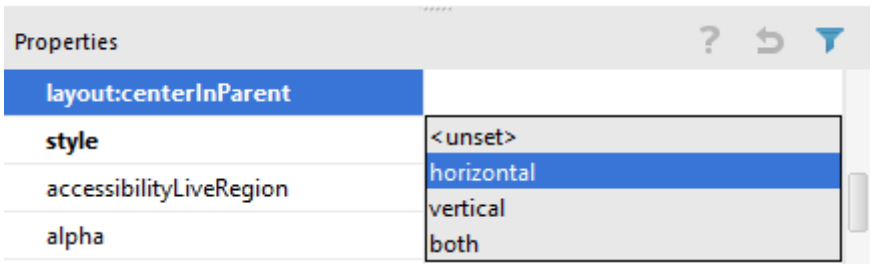
Esta sintaxis actualizará directamente la definición de tu view, modificando el valor de su atributo `android:layout_toLeftOf`:

```
android:layout_toLeftOf="@+id/Texto"
```

Adicionalmente existen las propiedades `layout:alignParent` y `layout:centerInParent`. La primera contiene resumidamente las características de relación entre un view y el padre. Puedes usar **check boxes** para seleccionar la que desees.



En la segunda propiedad puede elegir tres opciones de centrado con respecto al padre:*centrado horizontal*, *centrado vertical*, *ambos centrados*. Cada una de estas elecciones hace referencia a los atributos de referencia y se irán actualizando en la definición XML cuando los elijas.



Al probar todas estas propiedades sentirás la flexibilidad del `RelativeLayout`. Te recomiendo probar cada uno de los comportamientos de este sistema de diseño, ya que más adelante necesitarás estos conocimientos.

Manejar Eventos Cuando Un View Es Presionado

Supongo que ahora que sabes algo de views, ya quieres interactuar con ellos, ¿no es así?

Bien.

¿Recuerdas el patrón Observer de Java?

Púes el manejo de eventos en Android lo usa a través de escuchas (**listeners**).

Recuerda que una escucha es un objeto relacionado a otro objeto, cuyo fin es estar pendiente a las acciones que lo afectan en el exterior.

Su trabajo es registrar estos sucesos y notificarlo al entorno principal para que se tomen acciones preprogramadas y dirigir correctamente el flujo.

Los eventos con mayor frecuencia de ocurrencia son los eventos de entrada a través de los dispositivos de interfaz humana, como lo son los mouses, teclados, joysticks, o en este caso el tap de una pantalla móvil.

Ejemplo de implementación de OnClickListener

OnClickListener es una escucha que permite implementar acciones cuando un view es presionado en la interfaz por el usuario.

Para implementarla lo primero que debes realizar es relacionar la interfaz `OnClickListener` ya sea a través del uso de interfaz en la actividad donde se encuentra el botón o creando una nueva clase anónima y setearla en el atributo interno del botón:

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

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button boton = findViewById(R.id.boton);
        // this = MainActivity
        boton.setOnClickListener(this);
```

```
    }
}
```

El método `setOnClickListener()` asigna al botón declarado la instancia de la clase que implementa la interfaz `OnClickListener`, que en este caso es la misma clase contenedora.

Luego se debe diligenciar el contenido del método callback `onClick()`. Sin él, la escucha no tendría sentido, ya que debe existir para dirigir los eventos registrados hacia su contenido.

```
@Override
public void onClick(View v) {
    // Acciones
}
```

Dentro de este método debes ubicar todas las acciones que desees que ocurran al presionarse el botón. Recibe un parámetro de tipo `View`, el cual hace alusión a la instancia del botón seleccionado.

Manejo anónimo

Otra forma de usar esta interfaz es creando una clase anónima en línea:

```
boton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {

    }
});
```

Atributo android:onclick

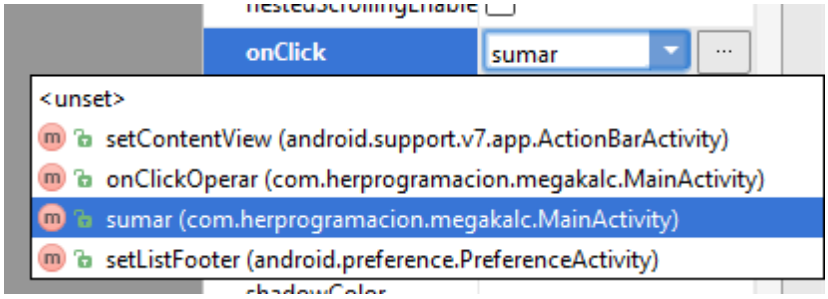
Usa la propiedad llamada `android:onclick`. En ella puedes asignar un método que tenga la forma de `onClick()`. Lógicamente el método debe estar previamente definido, retornar en `void` y recibir como parámetro un objeto de tipo `View`.

Por ejemplo…

Voy a declarar un método llamado `sumar()`, el cual actuará como *onClick*.

```
public void sumar (View v){
    suma = a+b;
}
```

Ahora supón que este método debe ejecutarse cuando se presione el botón. Así que voy al panel de propiedades de Android Studio y se lo asigno directamente:



O a través de la definición XML:

```
android:onClick="sumar"
```

¿Cuál crees que es la diferencia entre estos tres caminos?

Que usar clases anónimas o asignar métodos directos en el layout es ideal para actividades que monitorean a lo máximo 2 views.

No obstante existirán casos especiales donde múltiples views estén registrados para recibir eventos de toque.

Por lo que la primera opción funciona de maravilla, ya que puedes estar alerta de todos los elementos que están dentro de la actividad a través de una estructura de selección múltiple:

```
@Override
public void onClick(View v) {
    int idView = v.getId();

    switch (idView) {
        case R.id.primerView:
            // acciones
            break;
        case R.id.segundoView:
            // acciones
            break;
        …
        case R.id.enesimoView:
            // acciones
            break;

    }
}
```

El ejemplo anterior ilustra que dependiendo del `id` obtenido y la confrontación con los **ids** registrados en la clase `R` se implementarán las acciones necesarias.

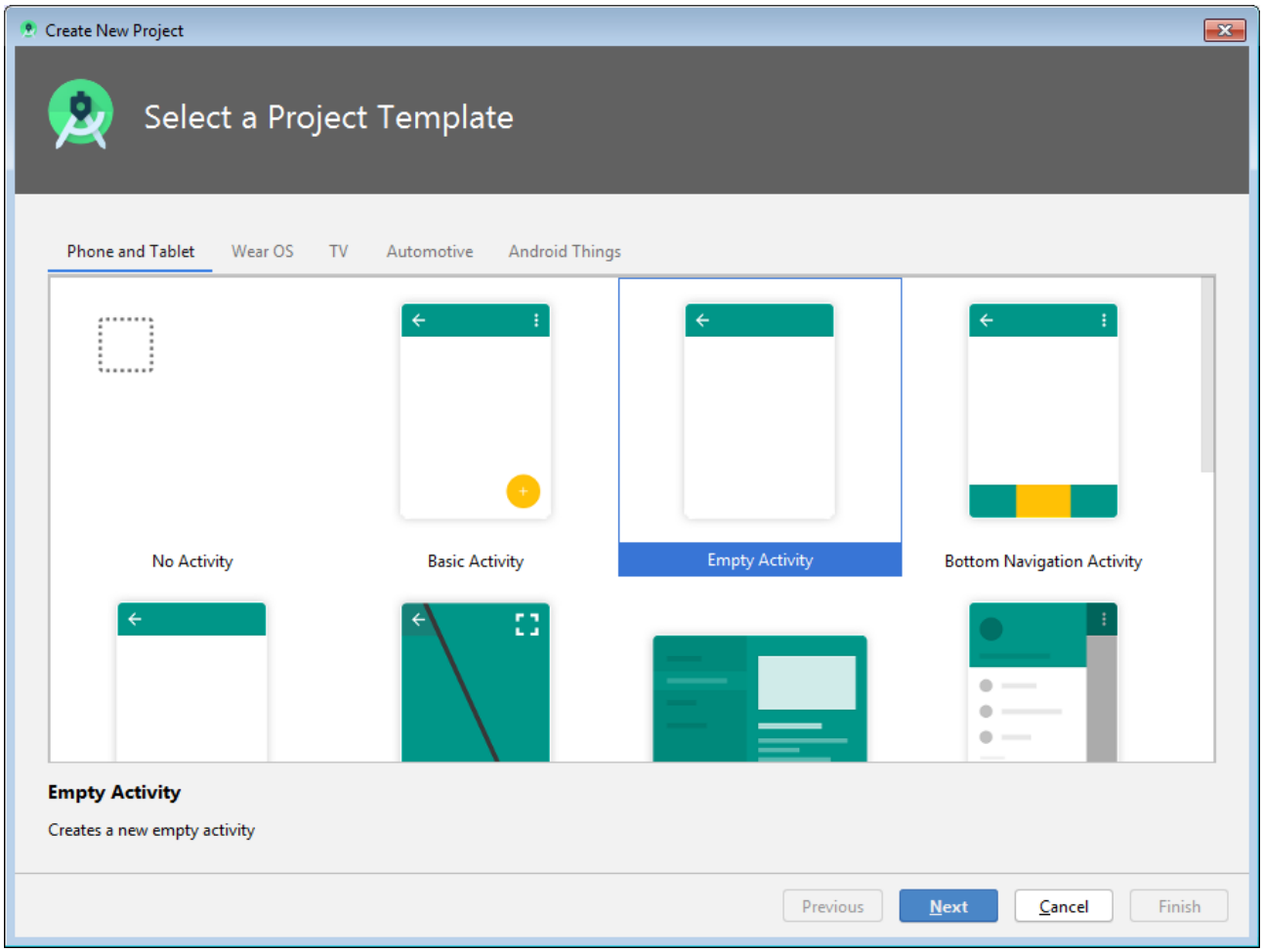
MegaKalc: El Ejemplo Que Tanto Esperabas

El siguiente ejemplo es básico, por lo que te permitirá familiarizarte con el entorno de Android Studio y la dinámica del diseño e implementación de interfaces.

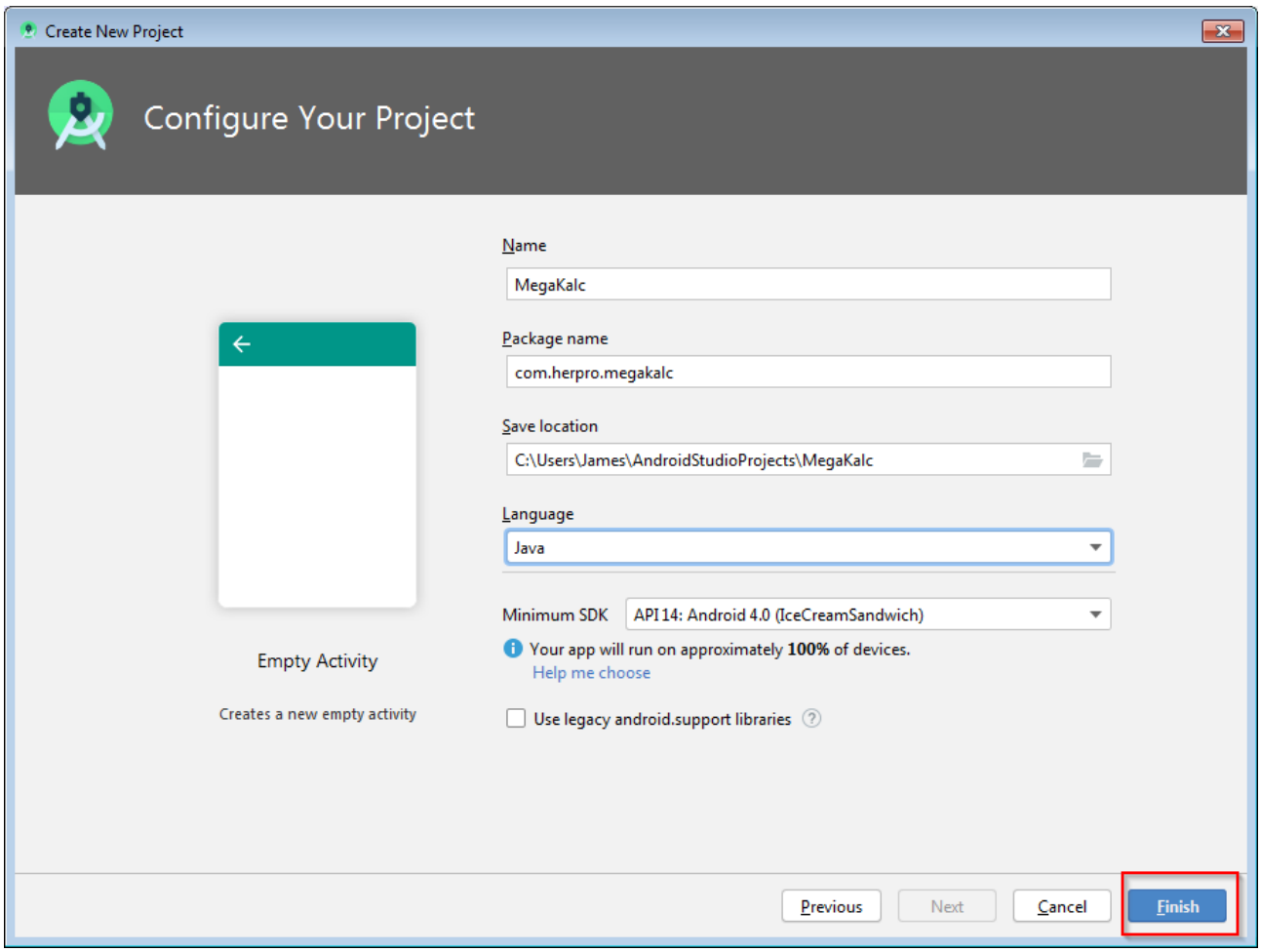
El objetivo de la aplicación *MegaKalc* es sencillo, operará dos números que el usuario ingrese en los campos. Además de eso se añadirá una radio para que seleccione una de las cuatro operaciones básicas de la aritmética: suma, resta, multiplicación y división.

Paso 1: Crear un nuevo proyecto en Android Studio

Primero lo primero. Inicia Android Studio y crea un proyecto que se llame “MegaKalc” con una actividad en blanco (*Empty Activity*).



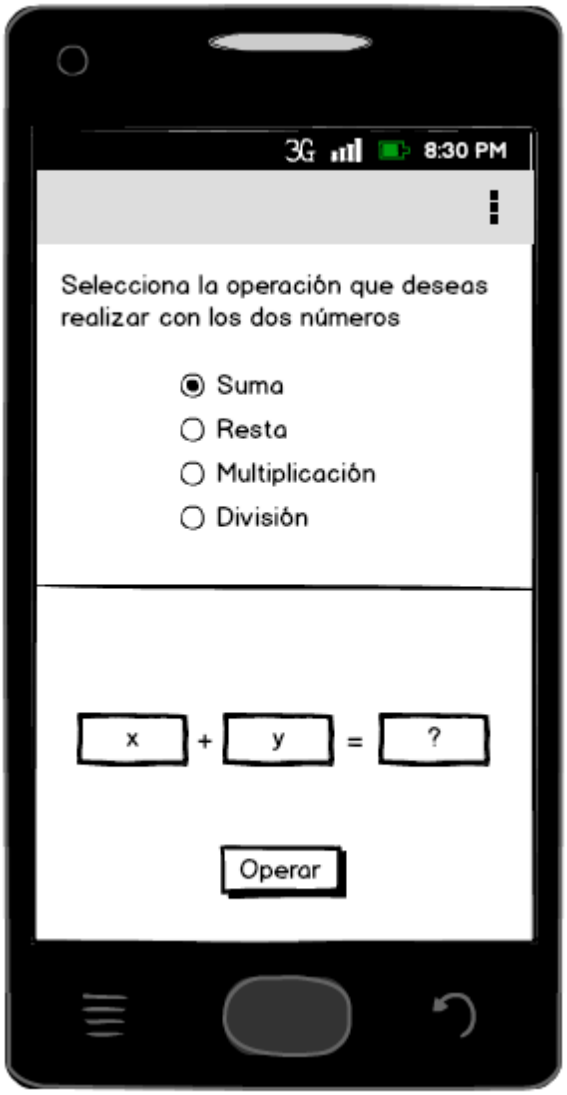
Selecciona el lugar donde deseas guardarlo y luego presiona **Finish**.



Paso 2: Diseña el layout de la actividad

Antes de comenzar a programar el comportamiento de *MainActivity* primero diseñaremos su contenido visual.

En este caso creé el siguiente boceto para tener una visión general de lo que haremos:

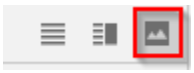


Con ello ya tenemos claro de qué forma acomodar los views que necesarios dentro del archivo `res/layout/activity_main.xml`. Este archivo trae contenido prefabricado. Elimínalo y déjalo así:

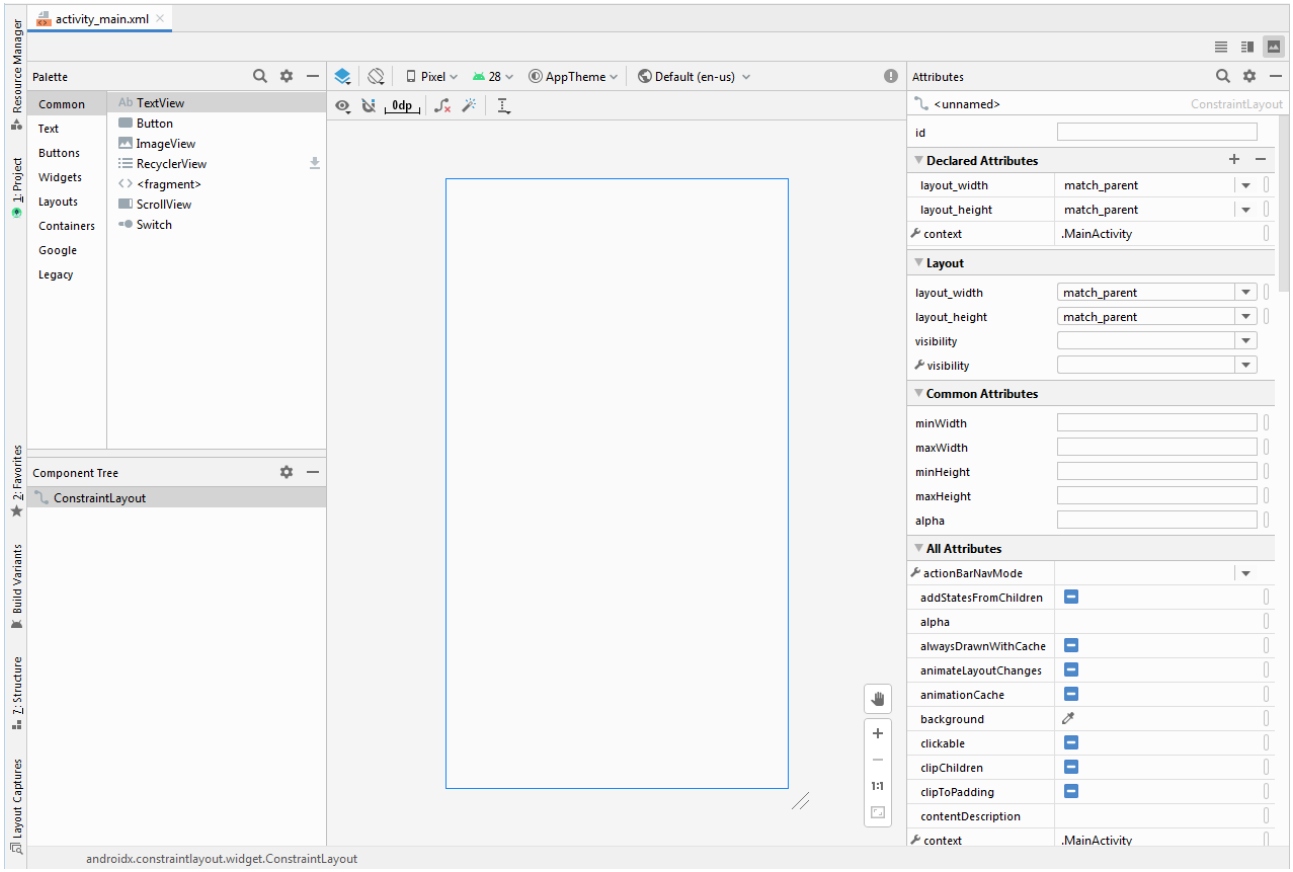
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
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

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

Ahora intentemos replicar el boceto. Las clases, el editor y la dinámica pueden parecer compleja pero son más intuitivos de lo que imaginas.

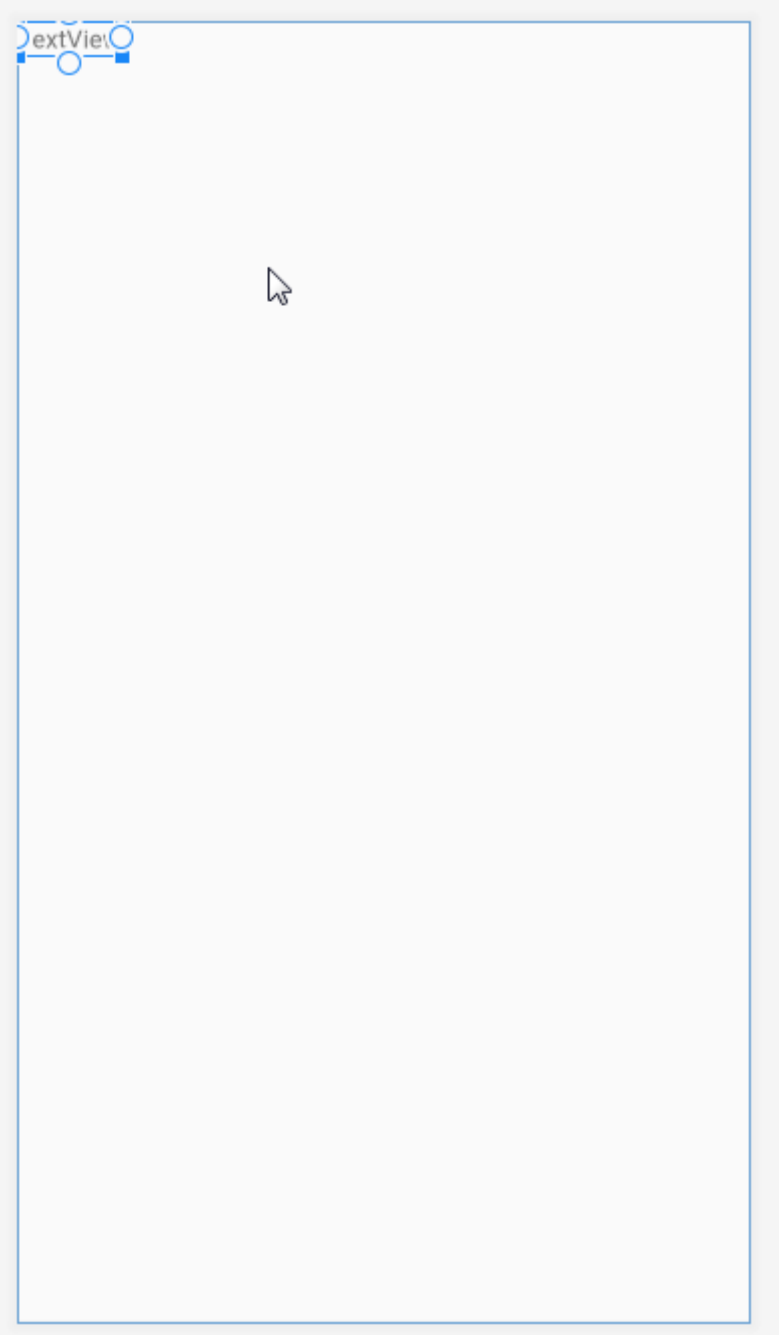
El primer texto que se muestra en la pantalla es un `TextView` con una cadena informativa que guardaremos en `res/values/strings.xml`. Para agregarlo, pasa al modo editor presionando el siguiente botón:



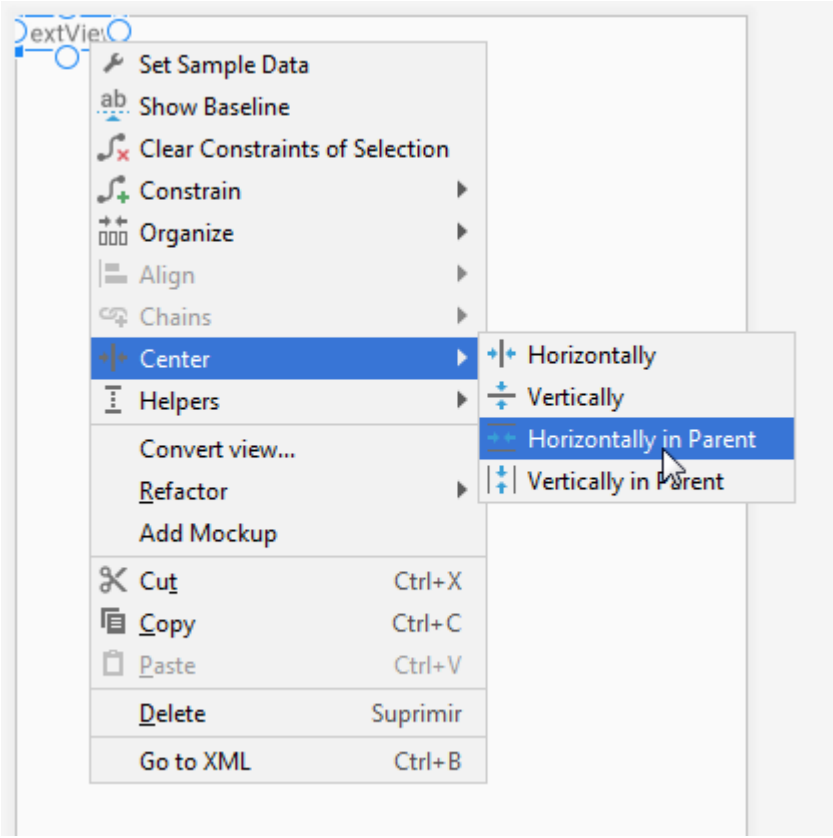
Ahí verás el lienzo en el centro; los views que puedes agregar a la izquierda (**Palette**); y los atributos del ítem que selecciones a la derecha (**Attributes**):



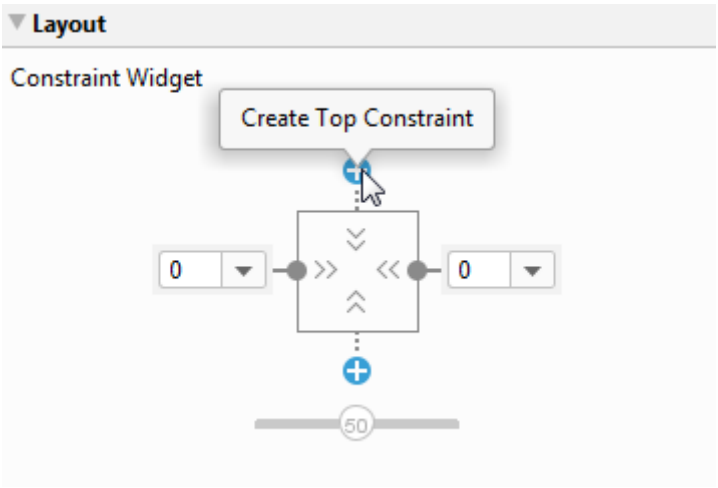
Sabiendo esto, arrastra el elemento **Common** > **TextView** hacia el lienzo o si deseas al interior del elemento **ConstraintLayout** en **Component Tree**:



El `ContraintLayout` te da la libertad de ubicar, redimensionar y colindar tus views de la forma que desees. Presiona click derecho sobre él y centralo con **Center** > **Horizontally in Parent**.

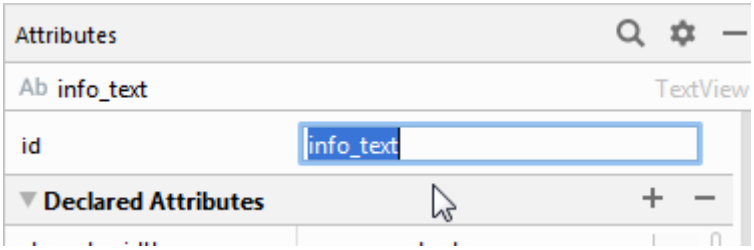


Ve al panel derecho y haz que su parte superior se ajuste a la parte superior del layout presionando el botón con signo más:

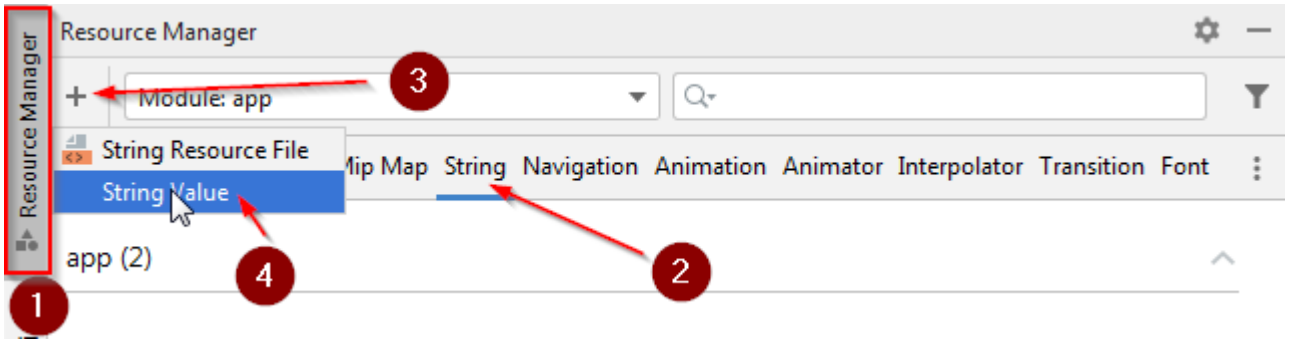


Ahora cambia el atributo `id` del *TextView* en el panel de propiedades para ajustar el nombre del elemento a las convenciones de código que use tu equipo de desarrollo. En mi caso particular uso notación `lower_case` e Inglés (usar Español está bien, pero recuerda que los acentos son problemáticos en los compiladores).

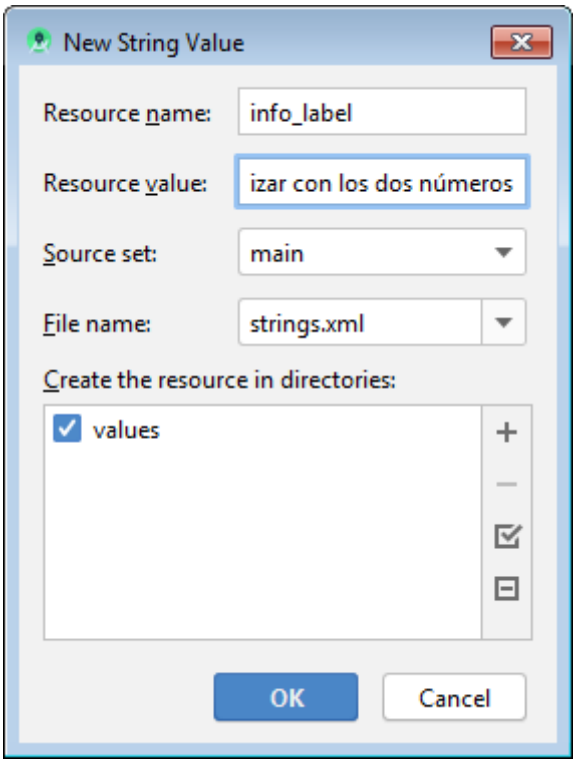
Siguiendo este estándar asignaré el id de `info_text`.



A continuación crearemos un nuevo recurso String en la ventana **Resources Manager** > **String**:

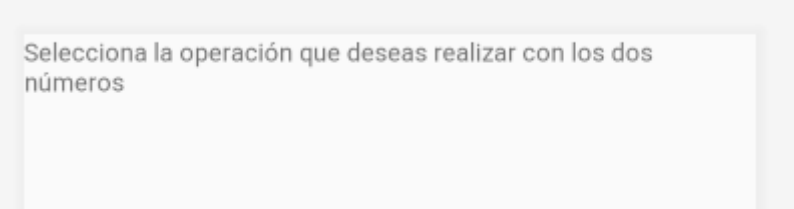


Lo llamaremos `info_label` y asignaremos el texto que tenemos en el boceto con **Resource Value**:



Reemplazaremos el texto que trae como predeterminado y añadiremos el string usando el operador de referencia `@string`.

Al revisar el lienzo se proyectará el texto:

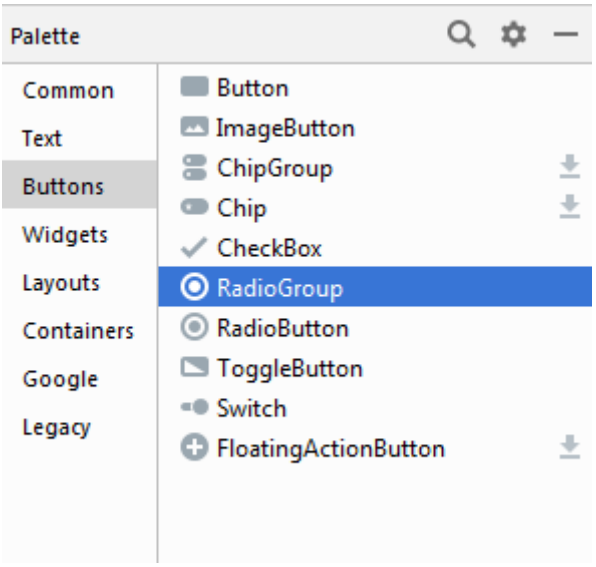


Fácil, ¿cierto?

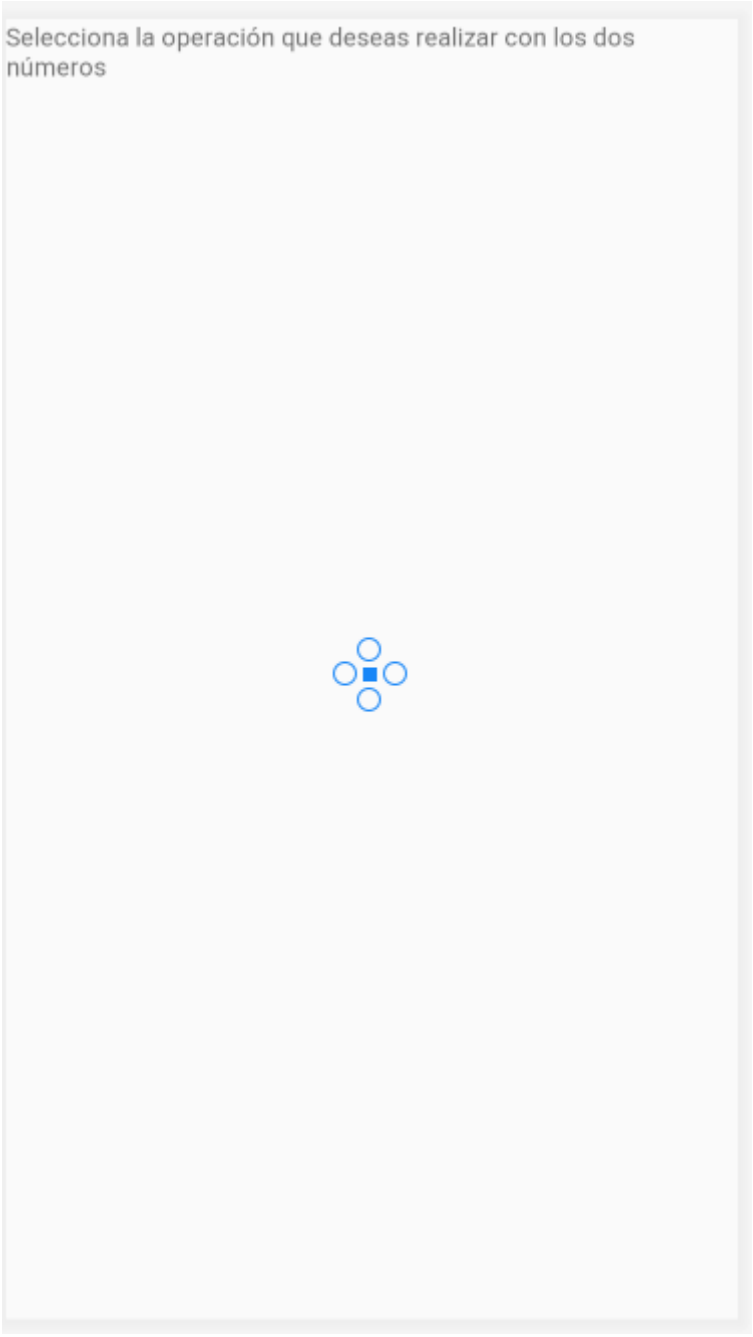
Sigamos con las operaciones.

Tarea 2.2: Añadir un RadioGroup

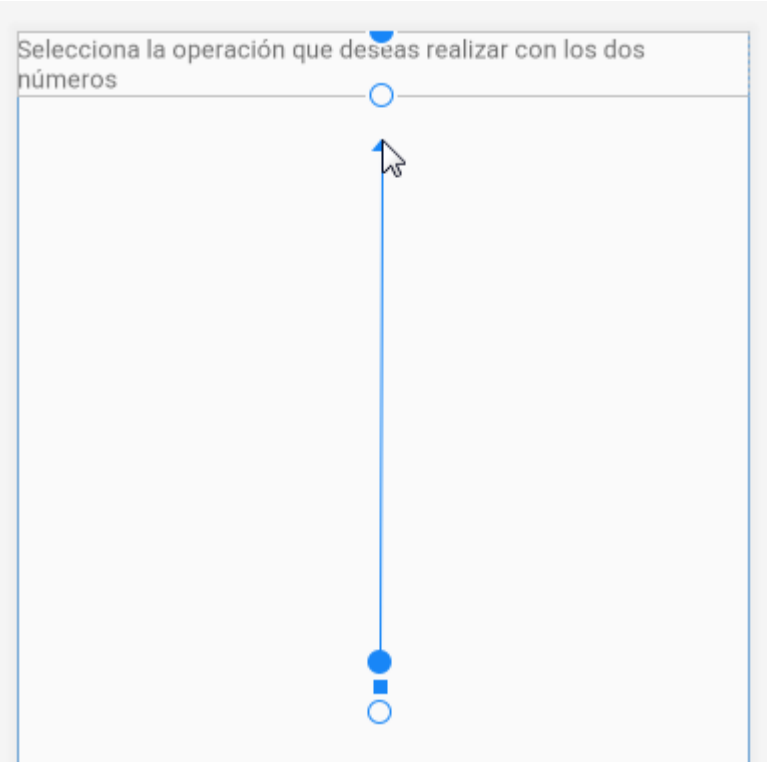
El siguiente elemento que añadiremos será un `RadioGroup` para generar exclusión entre varios `RadioButtons`.



Arrastra el elemento **Buttons** > **RadioGroup** hacía el centro del lienzo en un lugar visible.

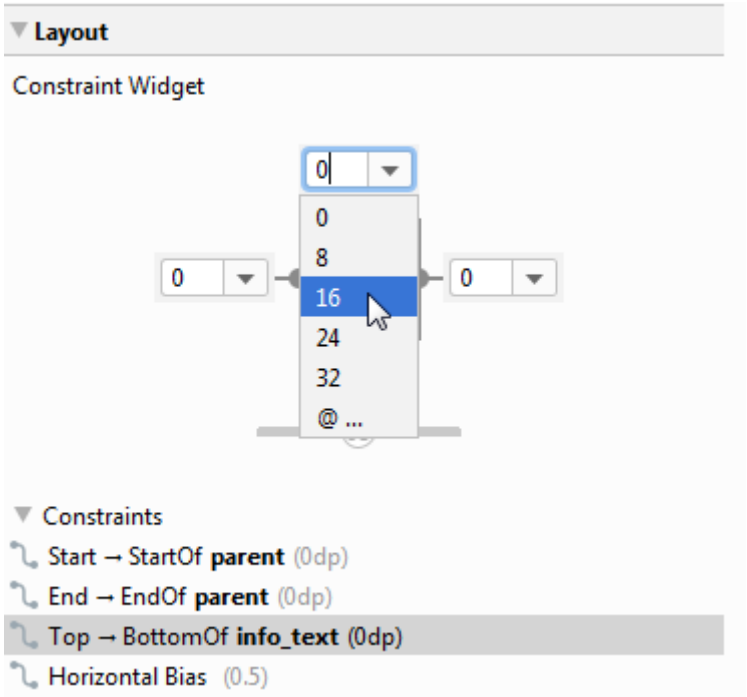


Haz que su borde superior se ajuste al borde inferior del texto arrastrando desde el globo superior al inferior; el lienzo mostrará una flecha motivando la interacción:



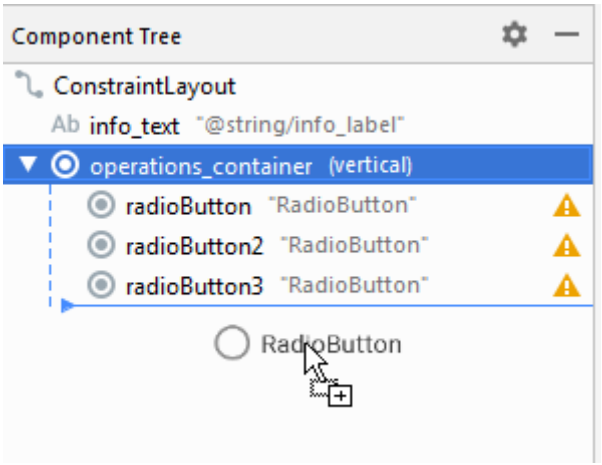
Céntralo en el padre para establecer sus restricciones horizontales y asígnale `operations_container` como id.

Ahora en la sección **Layout** de los atributos, agrega una margen superior de 16 puntos; despliega el menú en el borde superior del rectángulo representativo:

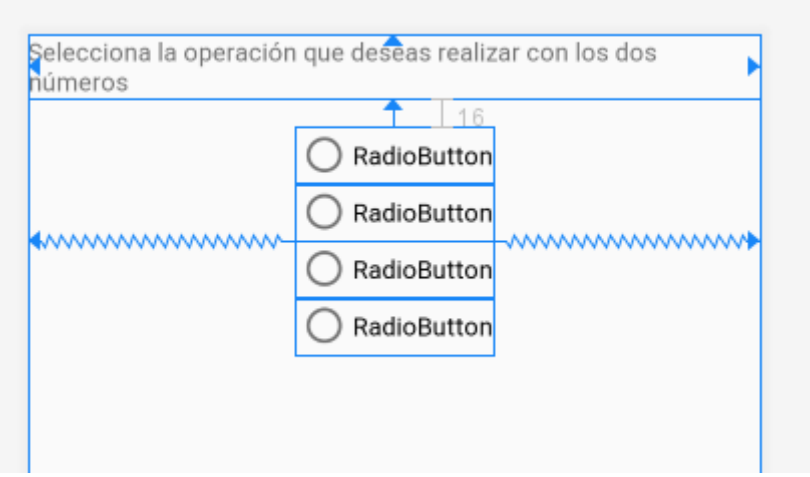


Tarea 2.3: Añadir RadioButtons

El siguiente movimiento es añadir cada `RadioButton` al interior del `RadioGroup`. Es decir, arrastra 4 **Buttons** > **RadioButton** hacia la ventana **Component Tree** para convertirlos en hijos del grupo:



El lienzo mostrará:

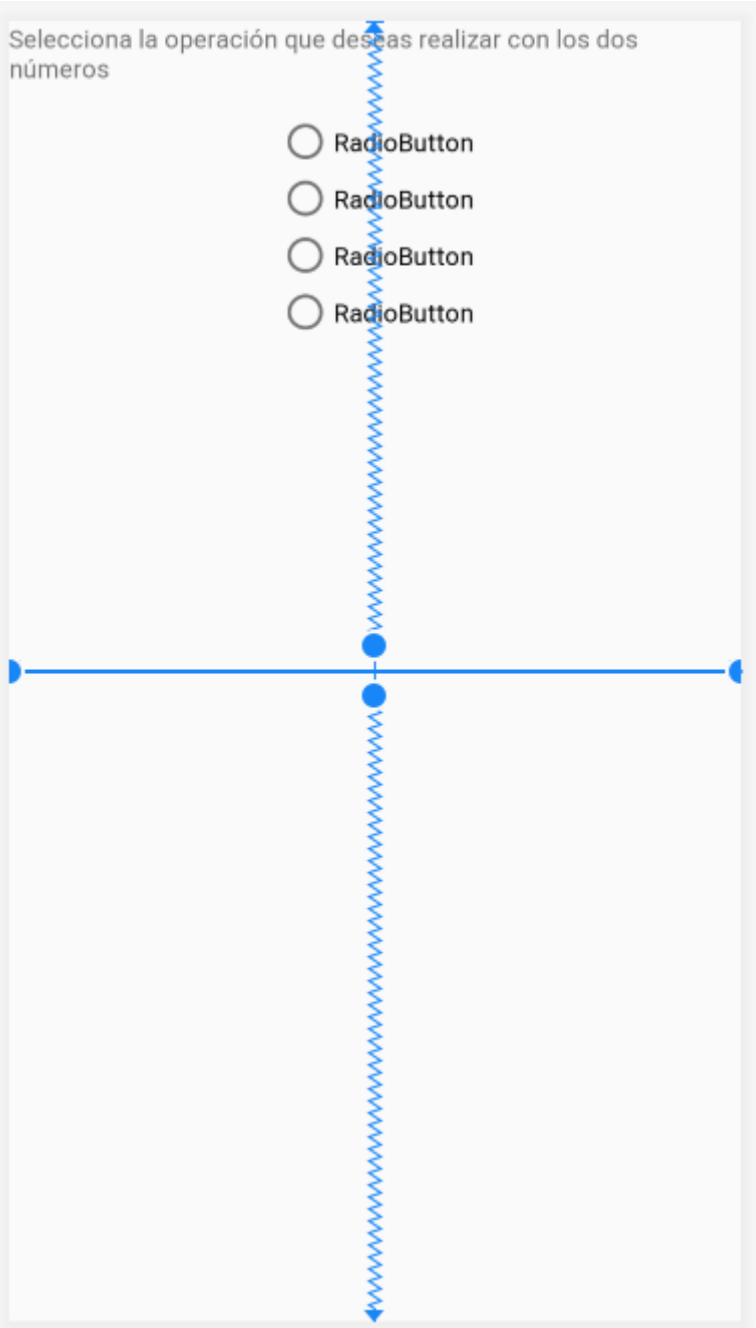


Asigna los ids: `addition_option`, `subtraction_option`, `multiplication_option` y `division_option`. Luego añade 4 strings con los valores `"Suma"`, `"Resta"`,`"Multiplicación"` y `"División"`. Concluye asignándolas a cada radio en el orden en que fueron declaradas.

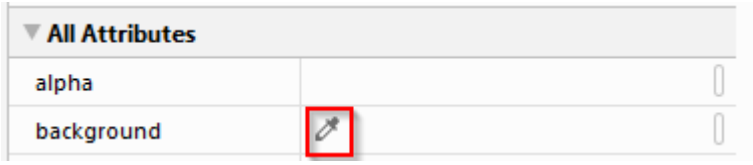
Tarea 2.4: Añadir Línea Divisoria

La línea de división en el boceto será un `View` personalizado.

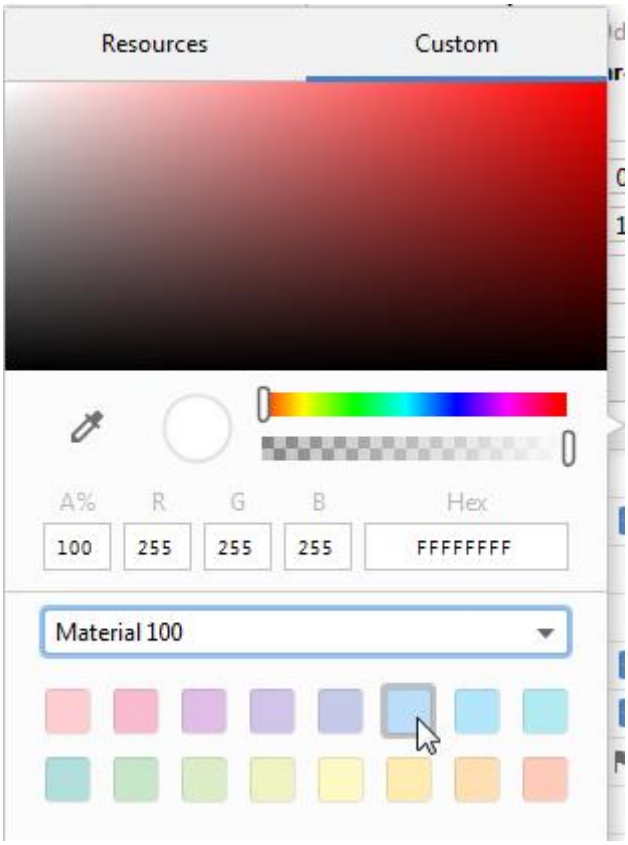
Arrastra un **Widgets** > **View** en un lugar del lienzo que sea visible. Cambia su ancho (`layout_width`) al valor `match_constraint` y su alto (`layout_height`) a 1dp. Luego centralo vertical y horizontalmente:



Ahora cambiemos su color en el atributo background:



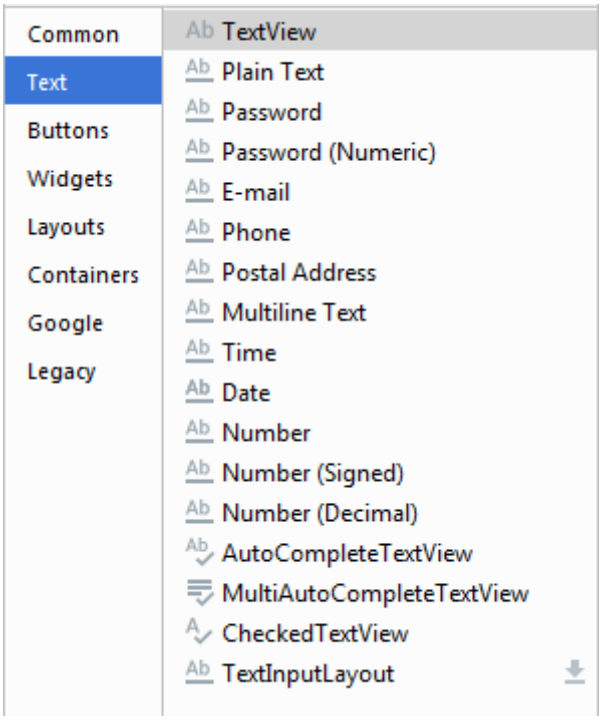
Presionar el gotero mostrará un diálogo con varias opciones de personalización en **Custom**, o bien usar **Resources** para asignar algún recurso de color de `res/values/colors.xml`:



Tarea 2.5: Añadir Edit Texts

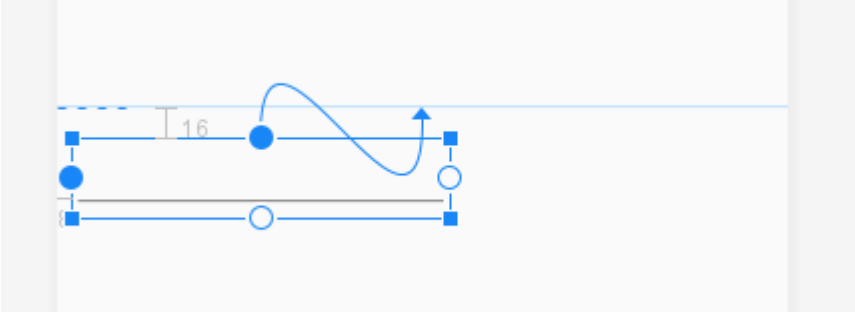
El usuario tendrá a su disposición campos de texto para ingresar 2 números enteros.

Estos se encuentran en la sección **Texts** de la paleta. Verás que hay varias opciones en cuanto al formato de los datos que soportan, como fecha, email, números, nombres, etc.

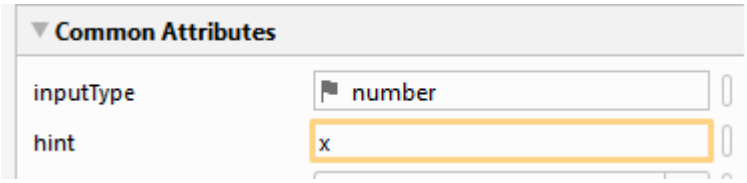


Todos son el mismo componente `EditText`, solo se diferencian en el valor de un atributo llamado `android:inputType`, el cual recibe una serie de constantes que se refieren al tipo de entrada aceptada.

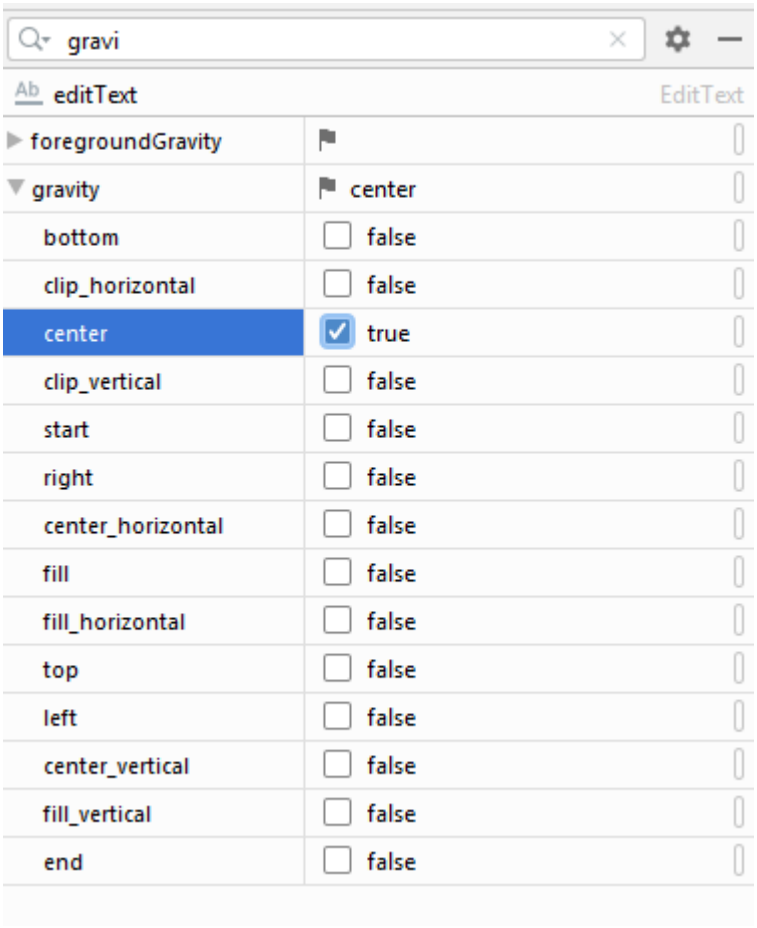
Ya tenemos claro que los campos que usaremos solo aceptarán números (**Number**); Arrastra el primero campo, haz que su borde superior se ajuste al divisor y su borde izquierdo al padre:



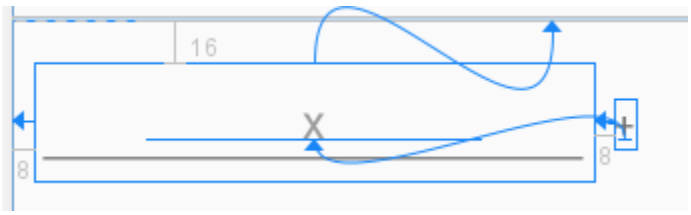
Ahora añade un texto informativo para el usuario con el atributo `android:hint` ubicado en el panel de propiedades. Recuerda que el boceto tenía la cadena `"x"` en este campo:



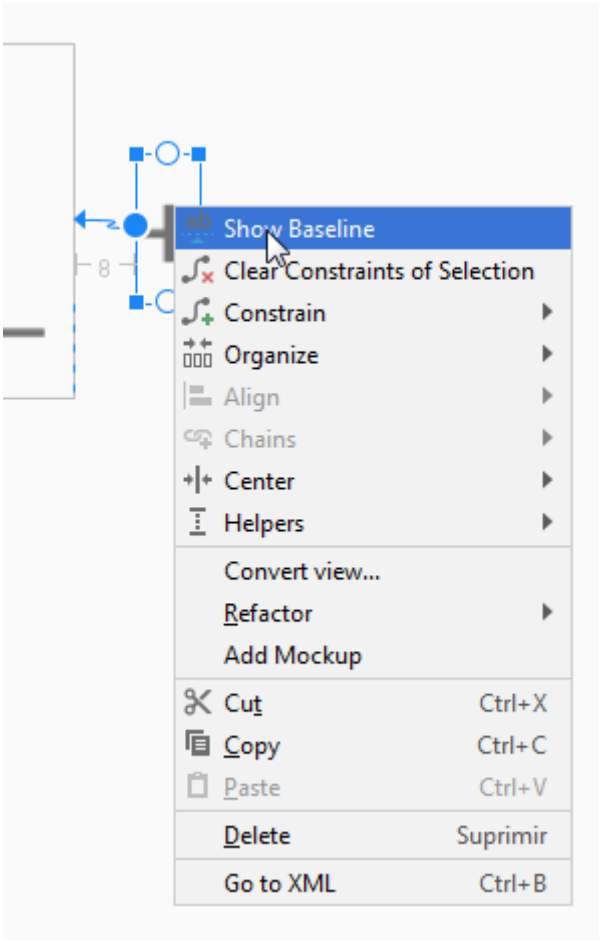
Centra el texto con `android:gravity="center_horizontal"`. Para encontrarlo usa la opción de búsqueda. Marca la bandera `center`:



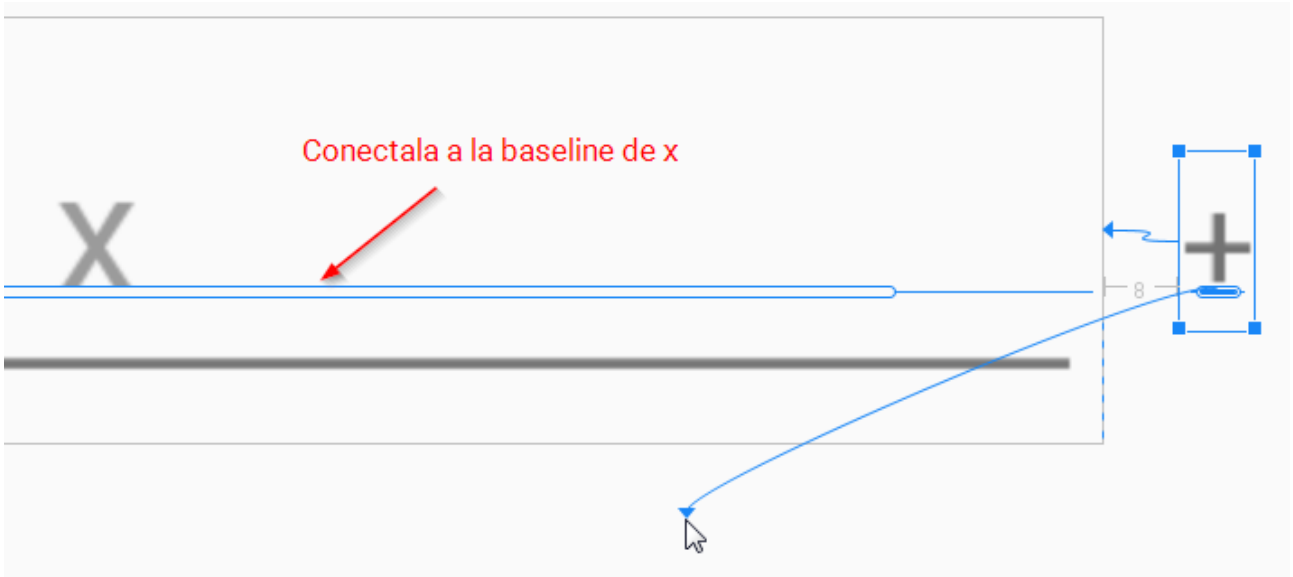
En seguida añade un `TextView` para ubicar el operador matemático correspondiente.



La idea es ajustar su borde izquierdo al derecho del numero x con un margen considerable (8dp puse yo). Para ajustar la línea inferior del texto del operador al texto del campo aplica **Click Derecho** > **Show Baseline**:

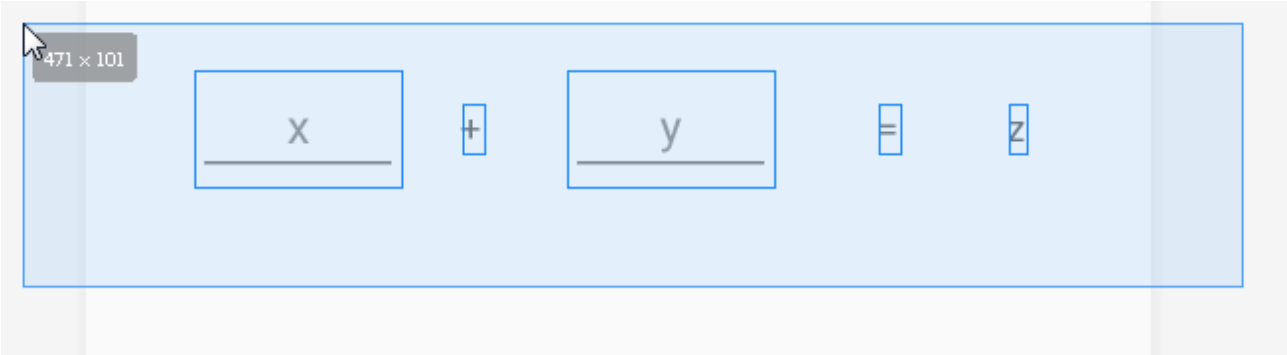


Saldrá un rectángulo, del cual podrás arrastrar la restricción hacia el campo de texto (haz zoom con **Ctrl+rueda del mouse** si es necesario):

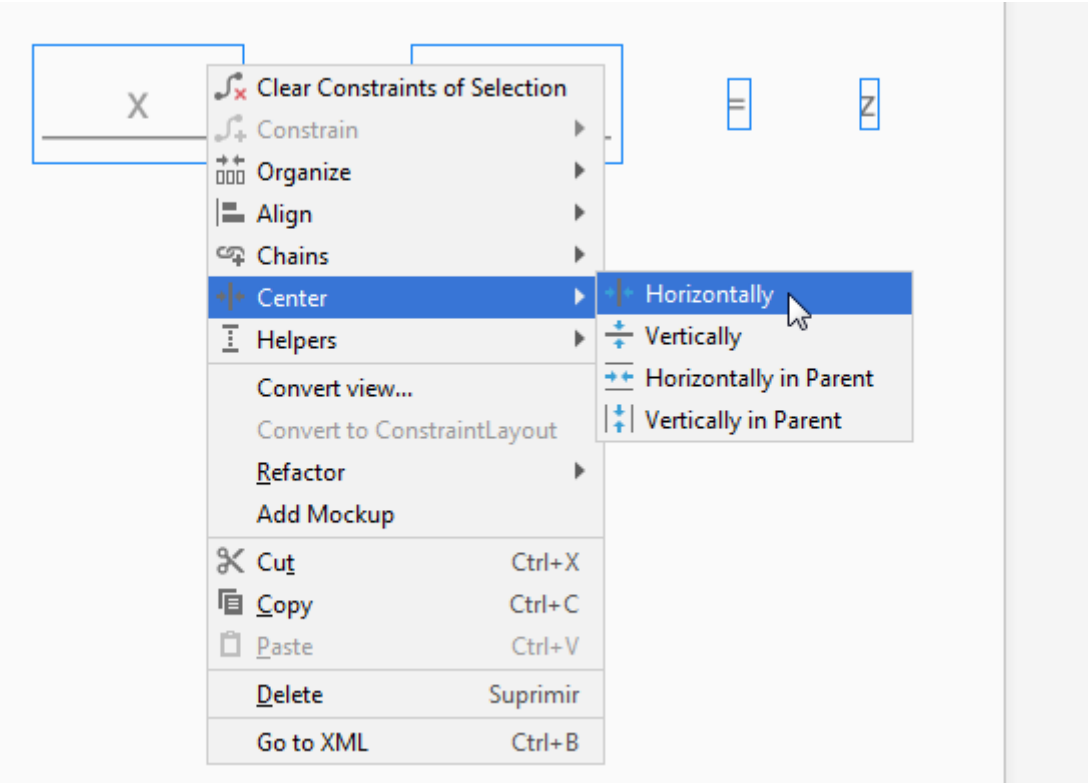


Ahora bien, teniendo claras las referencias de límites en los views, prueba ubicar todos los views en la operación como se muestra en el diseño. Modifica los atributos como más te parezca y añade tu toque personal.

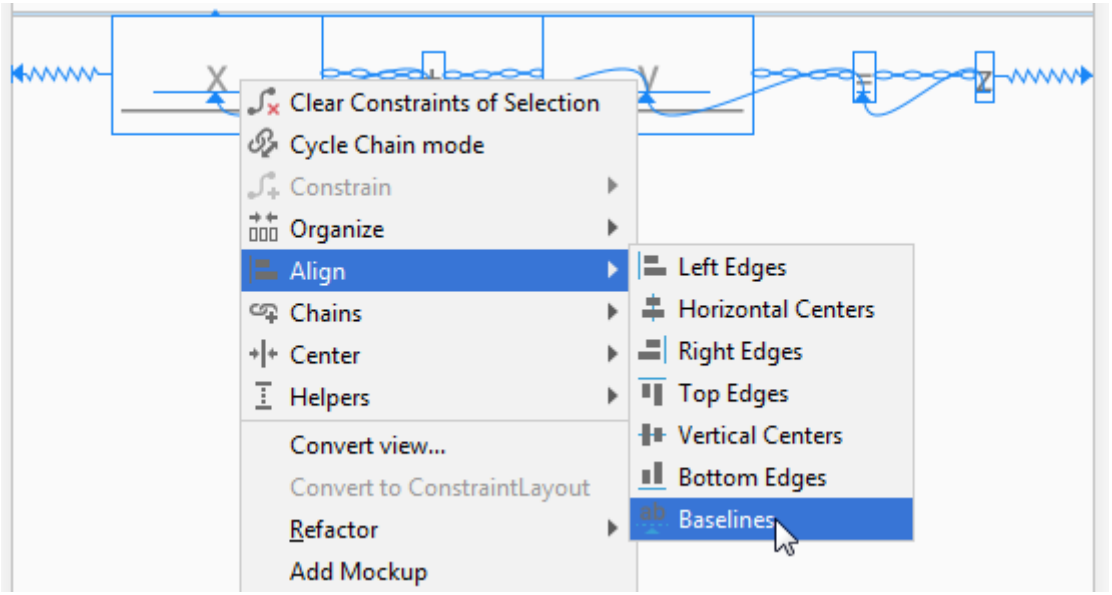
Una buena idea sería primero ubicar todos los views de la operación, seleccionarlos en conjunto:



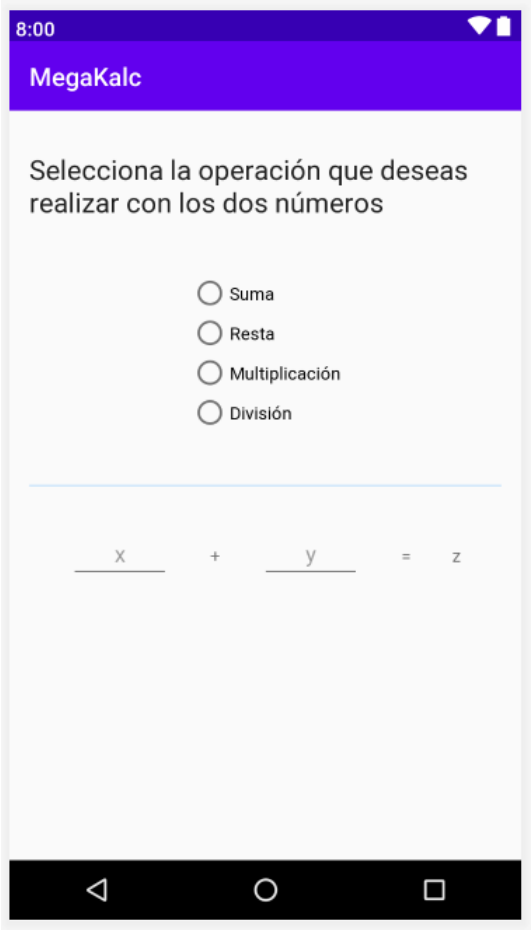
Y luego centrarlos horizontalmente. De esa forma crearás una cadena horizontal que afecte el desplazamiento y ancho de cada uno:



Igualmente la alineación:



La idea es que dejes el diseño de estar forma:



Y si se te complica tan solo abre el código descargable y copia el contenido de `activity_main.xml`.

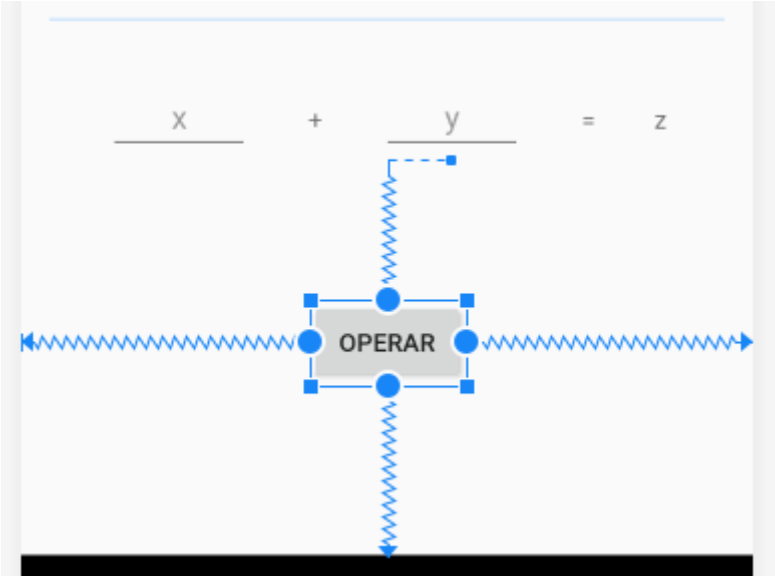
Tarea 2.6: Añadir botón para operar

Finalmente añadiremos el botón que permitirá calcular el resultado de los dos números que el usuario ha introducido al formulario.

Arrástralo debajo de la operación desde **Buttons** > **Button**.

Aplica las siguientes instrucciones:

- Alinea su borde inferior con el borde inferior del padre
- Céntralo horizontalmente
- Alinea su borde superior el borde inferior de algún view de la operación
- Asignale un string con el texto del boceto



Finalmente la definición XML del archivo `activity_main.xml` sería la siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<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"
    android:padding="16dp"
    tools:context=".MainActivity">

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

```
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:text="@string/info_label"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<RadioGroup
    android:id="@+id/operations_container"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toTopOf="@+id/divisor"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/info_text">

    <RadioButton
        android:id="@+id/addition_option"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/addition_option" />

    <RadioButton
        android:id="@+id/subtraction_option"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/subtraction_option" />

    <RadioButton
        android:id="@+id/multiplication_option"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/multiplication_option" />

    <RadioButton
        android:id="@+id/division_option"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/division_option" />
</RadioGroup>

<View
    android:id="@+id/divisor"
    android:layout_width="0dp"
    android:layout_height="1dp"
    android:background="#BBDEFB"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<EditText
    android:id="@+id/number_x_field"
    android:layout_width="80dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:ems="10"
    android:gravity="center"
```

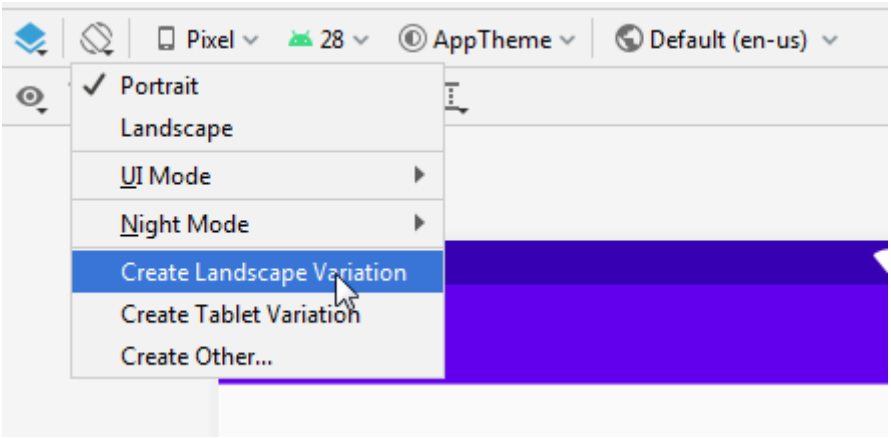
```
        android:hint="x"
        android:inputType="number"
        app:layout_constraintEnd_toStartOf="@+id/operator_symbol"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/divisor" />

    <TextView
        android:id="@+id/operator_symbol"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="+"
        app:layout_constraintBaseline_toBaselineOf="@+id/number_x_field"
        app:layout_constraintEnd_toStartOf="@+id/number_y_field"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/number_x_field" />

    <EditText
        android:id="@+id/number_y_field"
        android:layout_width="80dp"
        android:layout_height="wrap_content"
        android:ems="10"
        android:gravity="center"
        android:hint="y"
        android:inputType="number"
        app:layout_constraintBaseline_toBaselineOf="@+id/operator_symbol"
        app:layout_constraintEnd_toStartOf="@+id/equals_symbol"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/operator_symbol" />

    <TextView
        android:id="@+id/equals_symbol"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="="
        app:layout_constraintBaseline_toBaselineOf="@+id/number_y_field"
        app:layout_constraintEnd_toStartOf="@+id/result_text"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/number_y_field" />

    <TextView
        android:id="@+id/result_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBaseline_toBaselineOf="@+id/equals_symbol"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/equals_symbol"
        tools:text="z" />

    <Button
        android:id="@+id/calculate_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/calculate_text"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/number_y_field" />

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

Paso 3: Adaptar el layout a la orientación Landscape

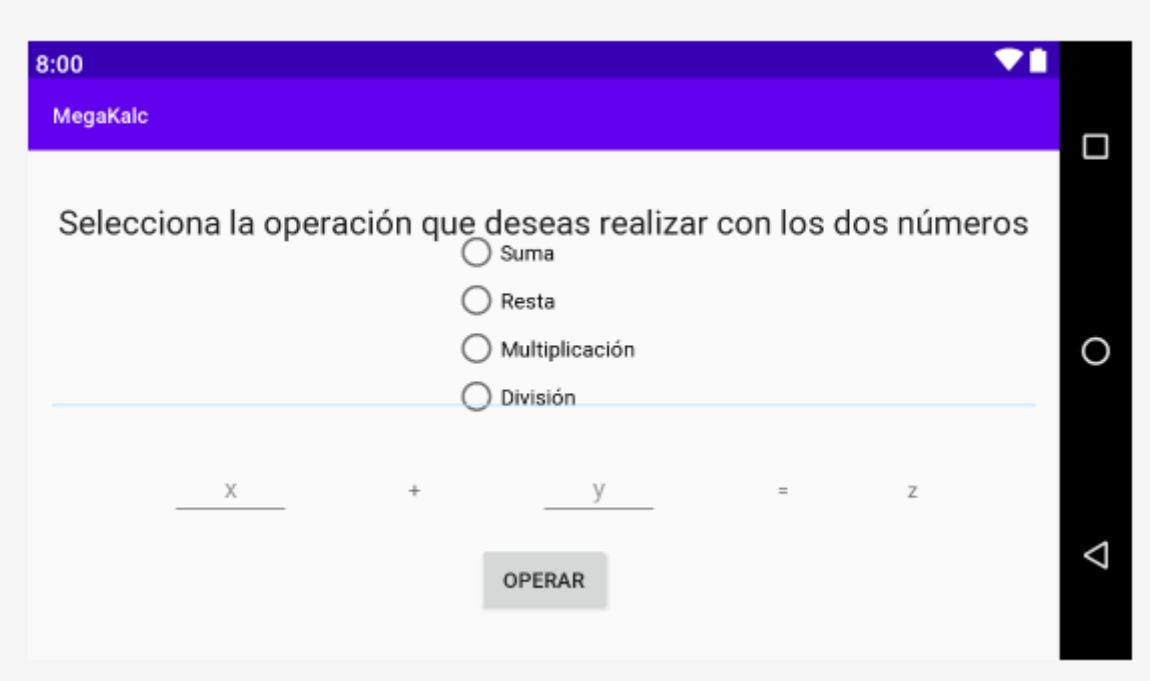
En esta sección veremos cómo crear un layout especial para cuando los dispositivos roten hacia la orientación landscape.

Para ello se debe crear una carpeta con un calificador del tipo `layout-land`, la cual contendrá el layout adaptado.

Si bien, esto puede realizarse manualmente, es más fácil realizarlo con la accesibilidad de Android Studio. Solo debes ubicarte en el layout original que deseas migrar a landscape y presionar el icono de configuración del layout:



Al presionarlo se despliegan varias opciones, pero la que hará el trabajo se denomina **Create Landscape Variation**. Si la ejecutas se creará automáticamente el directorio para layouts **Landscape** y se conservarán los mismo views que tienes actualmente.



Con este procedimiento se nota que el diseño **Portrait** que se había creado no es efectivo en orientación horizontal. Así que dividiremos los contenidos de la aplicación *MegaKalc* en dos secciones verticales. Del lado izquierdo irán los radios y en el lado derecho los campos de texto y el botón.

Tarea 3.1: Cambios

Las siguientes son las modificaciones que debes hacer para dividir en dos secciones el layout:

Divisor:

- Intercambia el ancho por el alto de la línea divisoria

Texto informativo:

- Alinea el borde derecho con el borde izquierdo del divisor
- Asigna a su ancho a `match_constraint`

Grupo de opciones:

- Alinea el borde derecho al divisor
- El borde inferior al padre

Número x:

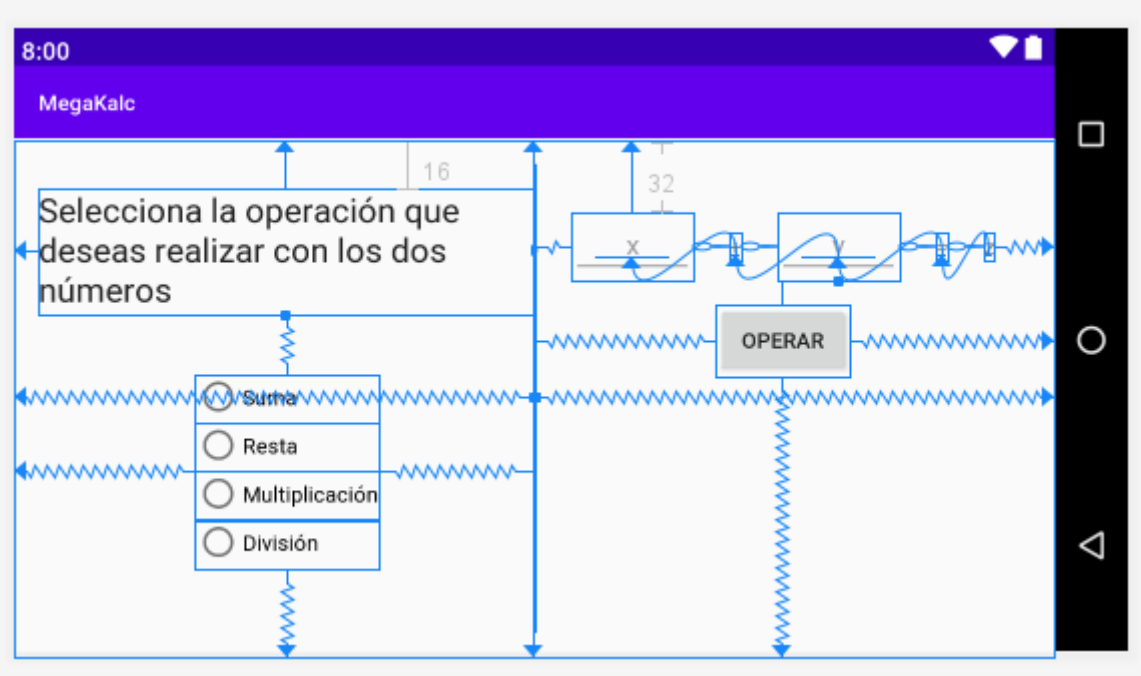
- Alinea el borde superior con el padre
- Alinea borde izquierdo con divisor

Botón:

- Alinea su borde izquierdo con el divisor

Si alteras el numero x la formula se orientará automáticamente.

El resultado de `res/layout-land/activity_main.xml` se vería así:



Ese es el primer diseño que se me ocurrió, no estás obligado a realizar lo mismo. Sería excelente que practicaras las orientaciones de distintas formas, esto te ayudará a comprender la ubicación de los bordes y límites del `ConstraintLayout`.

El código de este layout sería el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<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"
android:padding="16dp"
tools:context=".MainActivity">

<TextView
    android:id="@+id/info_text"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
    android:text="@string/info_label"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    app:layout_constraintEnd_toStartOf="@+id/divisor"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<RadioGroup
    android:id="@+id/operations_container"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/divisor"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/info_text">

    <RadioButton
        android:id="@+id/addition_option"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/addition_option" />

    <RadioButton
        android:id="@+id/subtraction_option"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/subtraction_option" />

    <RadioButton
        android:id="@+id/multiplication_option"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/multiplication_option" />

    <RadioButton
        android:id="@+id/division_option"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/division_option" />
</RadioGroup>

<View
    android:id="@+id/divisor"
    android:layout_width="1dp"
    android:layout_height="0dp"
    android:background="#BBDEFB"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

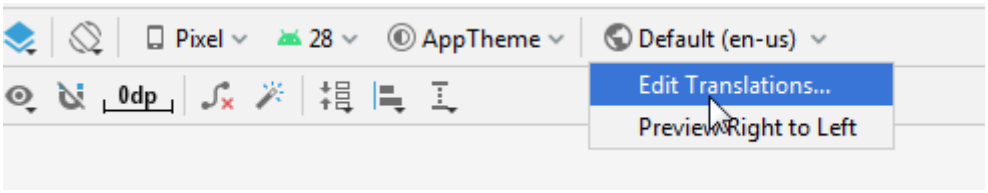
```
<EditText
    android:id="@+id/number_x_field"
    android:layout_width="80dp"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:ems="10"
    android:gravity="center"
    android:hint="x"
    android:inputType="number"
    app:layout_constraintEnd_toStartOf="@+id/operator_symbol"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/divisor"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/operator_symbol"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="+"
    app:layout_constraintBaseline_toBaselineOf="@+id/number_x_field"
    app:layout_constraintEnd_toStartOf="@+id/number_y_field"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/number_x_field" />

<EditText
    android:id="@+id/number_y_field"
    android:layout_width="80dp"
    android:layout_height="wrap_content"
    android:ems="10"
    android:gravity="center"
    android:hint="y"
    android:inputType="number"
    app:layout_constraintBaseline_toBaselineOf="@+id/operator_symbol"
    app:layout_constraintEnd_toStartOf="@+id/equals_symbol"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/operator_symbol" />

<TextView
    android:id="@+id/equals_symbol"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="="
    app:layout_constraintBaseline_toBaselineOf="@+id/number_y_field"
    app:layout_constraintEnd_toStartOf="@+id/result_text"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/number_y_field" />

<TextView
    android:id="@+id/result_text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintBaseline_toBaselineOf="@+id/equals_symbol"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toEndOf="@+id/equals_symbol"
    tools:text="z" />

<Button
    android:id="@+id/calculate_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="16dp"
```

```
        android:text="@string/calculate_text"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/divisor"
        app:layout_constraintTop_toBottomOf="@+id/number_y_field"
        app:layout_constraintVertical_bias="0.0" />

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

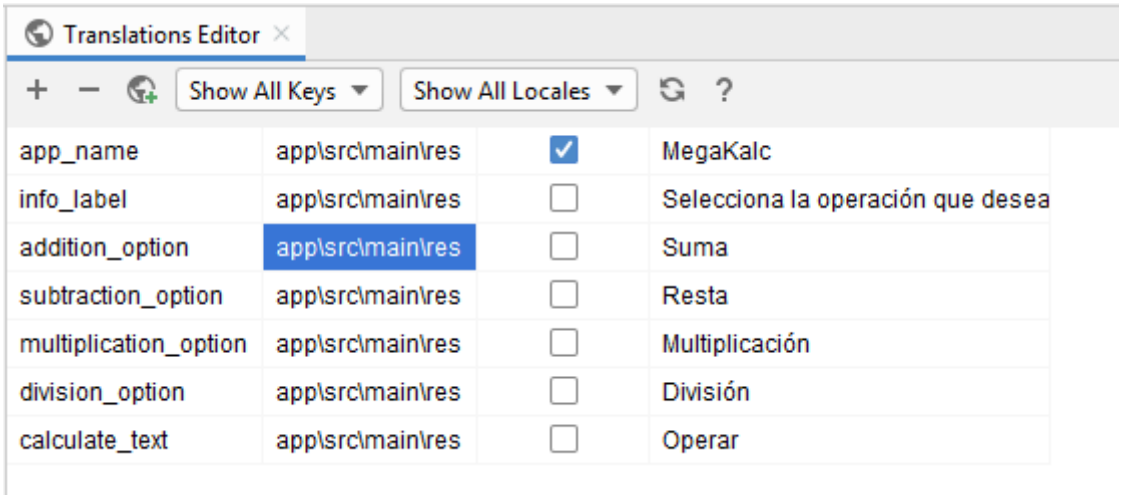
Paso 4: Soportar strings en otro idioma

La aplicación *MegaKalc* soportará el uso del idioma ingles en todas aquellas cadenas que lo requieran. Ya se había estudiado que existen calificadores para determinar el lenguaje de cobertura en los recursos de la aplicación, así que los emplearemos en la carpeta **values**.

Para crear el nuevo recurso que se ajuste al idioma iremos a la previsualización del layout y seleccionaremos **Edit Transalations...**



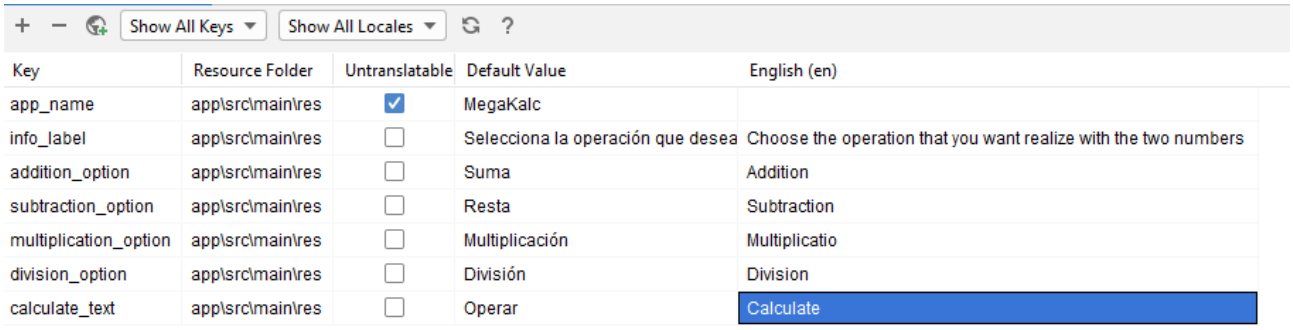
Al presionar esta opción aparecerá el editor:



Las columnas corresponden al string, su ubicación, una bandera que indica si no debe traducirse y el valor.

Para añadir una traducción al inglés, presionamos el planeta:

Esto creará el archivo `res/values-en/strings.xml`; El objetivo es asignar el texto en inglés para cada recurso:



De esa forma nuestro caso de uso soportará dispositivos con inglés en su configuración de idiomas.

Paso 5: Programar comportamientos

El último paso es añadir las acciones de cálculo necesarias para que el resultado de la operación sea exitosa.

Guardar referencia de RadioGroup

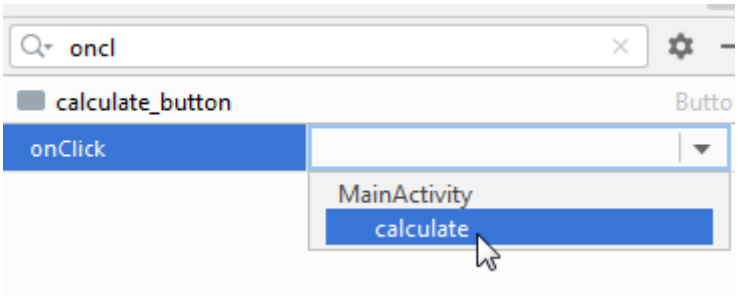
En primer lugar abre `MainActivity` y obtén las referencias de los views en `onCreate()`. Este también es el lugar ideal para usar el método `check()` y marcar un radio por defecto.

```
public class MainActivity extends AppCompatActivity {

    // Views
    RadioGroup mOperationsContainer;
    private EditText mNumberX;
    private EditText mNumberY;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Obtener radio group
        mOperationsContainer = findViewById(R.id.operations_container);
        mOperationsContainer.check(R.id.addition_option);
        // Obtener los campos de edición
        mNumberX = findViewById(R.id.number_x_field);
        mNumberY = findViewById(R.id.number_y_field);
    }

}
```

Método onClick() personalizado

Ejecutar acciones cada que se presione el botón puede lograse asigndo una escucha anónima con `setOnClickListener()` al botón. Sin embargo también podemos declarar un método publico void que reciba una instancia View y asignar su firma en el atributo `android:onclick` del layout.

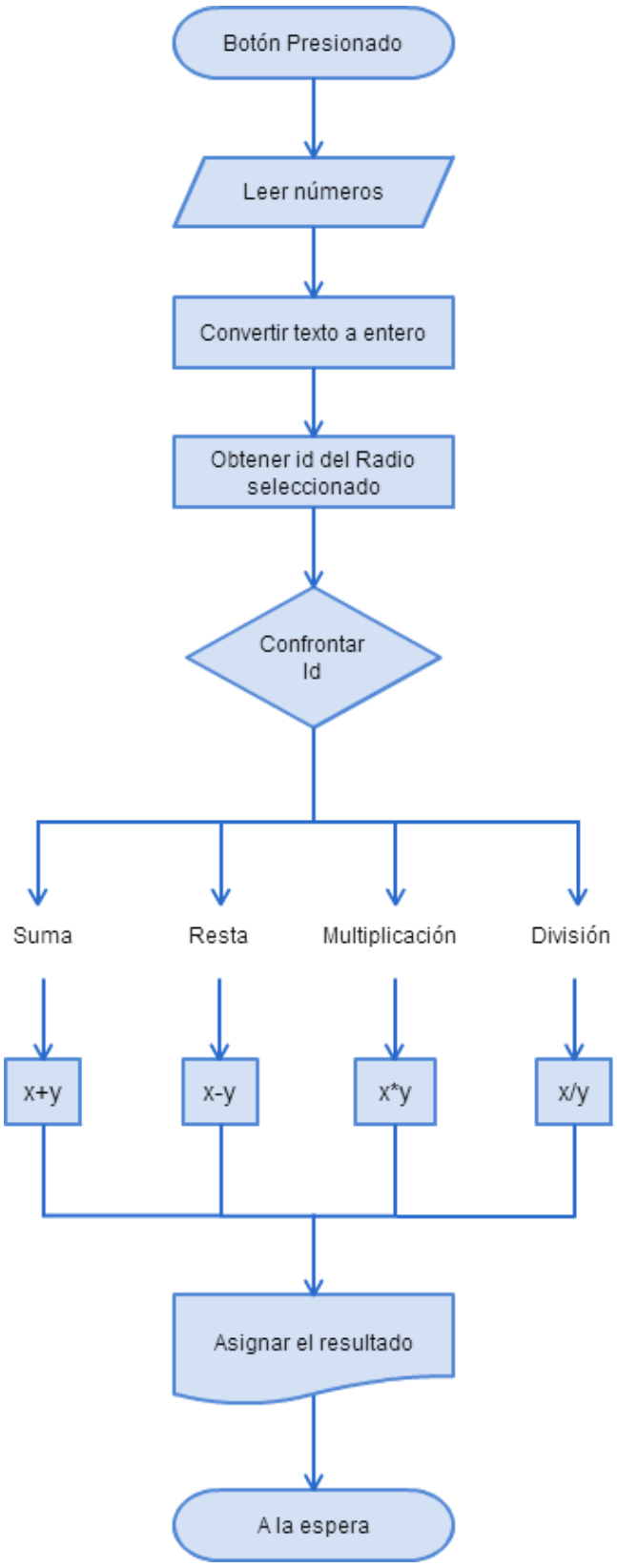
Si vas al editor, verás que en este atributo se te permite asignar `calculate()`:



Es solo declarar el método `calculate()` en la actividad.

```
public void calculate(View v) {

}
```

Con respecto a las acciones en su interior me gustaría que vieras este algoritmo:



¿Sencillo cierto?

Codifiquemos cada paso del diagrama anterior:

Leer números y convertir a enteros

Usa `getText().toString()` para obtener el texto de un campo:

```
// Variables locales
int a, b, c = 0;

// Convertir el texto a enteros
a = Integer.parseInt(mNumberX.getText().toString());
b = Integer.parseInt(mNumberY.getText().toString());
```

El casting lo realizas con el método `Integer.parseInt()`.

Obtener ID de la opción seleccionada y calcular

Obtener el identificador del radio seleccionado requiere el uso de `getCheckedRadioButtonId()`. Debido a que son 4 radios, es ideal usar una sentencia `switch` para comparar el id obtenido con cada caso:

```
// Calcular resultado
switch (mOperationsContainer.getCheckedRadioButtonId()) {
    case R.id.addition_option:
        c = a + b;
        break;
    case R.id.subtraction_option:
        c = a - b;
        break;
    case R.id.multiplication_option:
        c = a * b;
        break;
    case R.id.division_option:
        c = a / b;
        break;
}
```

Finalmente operamos los dos números dentro de cada case según sea la opción elegida por el usuario.

Asignar el resultado

El resultado final lo asignamos con el método `setText()`.

```
// Asignar el resultado
((TextView) findViewById(R.id.result_text))
        .setText(String.valueOf(c));
```

Cambiar operador al seleccionar un radio diferente

Otro pequeño caso de uso sería actualizar la vista cuando el usuario seleccionado un radio.

Usa `OnCheckedChangeListener` para la gestión del cambio de selección dentro del grupo.

Solo es implementar el método `onCheckedChanged()` de la interfaz para cambiar la cadena del **text view** que representa el operador.

Ve a `onCreate()` y asigna la escucha anónima:

```
mOperationsContainer.setOnCheckedChangeListener(new
RadioGroup.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(RadioGroup group, int checkedId) {
        TextView operador = findViewById(R.id.operator_symbol);

        switch (mOperationsContainer.getCheckedRadioButtonId()) {
            case R.id.addition_option:
                operador.setText("+");
                break;
            case R.id.subtraction_option:
                operador.setText("-");
                break;
            case R.id.multiplication_option:
                operador.setText("*");
                break;
            case R.id.division_option:
                operador.setText("/");
                break;
        }
    }
});
```

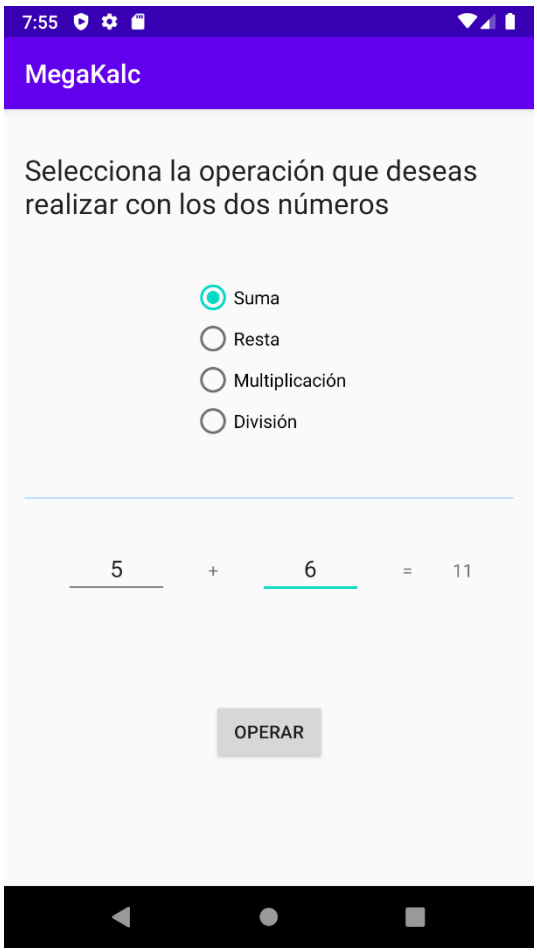
Completando el código

Finalizando el ejercicio deberías tener `MainActivity.java` con el siguiente contenido:

```
public class MainActivity extends AppCompatActivity {

    // Views
    RadioGroup mOperationsContainer;
    private EditText mNumberX;
    private EditText mNumberY;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Obtener radio group
        mOperationsContainer = findViewById(R.id.operations_container);
        mOperationsContainer.check(R.id.addition_option);
        // Obtener los campos de edición
        mNumberX = findViewById(R.id.number_x_field);
        mNumberY = findViewById(R.id.number_y_field);

        mOperationsContainer.setOnCheckedChangeListener(new
RadioGroup.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(RadioGroup group, int checkedId) {
```

```
                TextView operador = findViewById(R.id.operator_symbol);

                switch (mOperationsContainer.getCheckedRadioButtonId()) {
                    case R.id.addition_option:
                        operador.setText("+");
                        break;
                    case R.id.subtraction_option:
                        operador.setText("-");
                        break;
                    case R.id.multiplication_option:
                        operador.setText("*");
                        break;
                    case R.id.division_option:
                        operador.setText("/");
                        break;
                }
            }
        });
    }

    public void calculate(View v) {
        // Variables locales
        int a, b, c = 0;

        // Convertir el texto a enteros
        a = Integer.parseInt(mNumberX.getText().toString());
        b = Integer.parseInt(mNumberY.getText().toString());

        // Calcular resultado
        switch (mOperationsContainer.getCheckedRadioButtonId()) {
            case R.id.addition_option:
                c = a + b;
                break;
            case R.id.subtraction_option:
                c = a - b;
                break;
            case R.id.multiplication_option:
                c = a * b;
                break;
            case R.id.division_option:
                c = a / b;
                break;
        }

        // Asignar el resultado
        ((TextView) findViewById(R.id.result_text))
                .setText(String.valueOf(c));
    }

}
```

Ejecuta la app y prueba cada una de las operaciones.


7:55
MegaKalc
Selecciona la operación que deseas realizar con los dos números
Suma
Resta
Multiplicación
División
5
+
6
=
11
OPERAR

¿Qué tal te fue?

¿Se te hizo más fácil de comprender la programación Android?

Cuando alguien interesado en iniciar al desarrollo Android requiere mi ayuda, me enfoco en proporcionarle estas tres experiencias:

- Eliminar de su mente el mito de que el Desarrollo Android es muy complicado.
- Aumentar su comprensión del entorno que usaremos a lo largo de nuestras vidas como desarrolladores Android.
- Invitarle a jugar unos pocos minutos con la vista y manejo de eventos de Android para que vea lo sencillo que es crear interacciones.

Y está guía debería ayudarte a vivir esos tres momentos.

Aquí está que hacer ahora:

- Envíame un mensaje personal a mi dirección de correo james@hermosaprogramacion.com para dejarme saber qué piensas de este ebook.
- Práctica, práctica y práctica. Malcolm Gladwell dice que para dominar un campo con experticia se requieren 10.000 horas, tú ya llevas un par, así que ve por el resto.
- Lee uno a uno los tutoriales que hay en mi página de contenidos de desarrollo Android. Allí encontrarás todo lo que he publicado y seguiré publicando.
- Complementa tus conocimientos con mis plantillas y ebooks.
- Seguir mi página de Facebook donde también notifico sobre mis nuevos tutoriales.
- Comparte en Twitter o Comparte en Facebook
- Date una palmadita en la espalda. Leíste un ebook de 68 páginas :)

¡Espero te sean de utilidad mis contenidos!

¡Saludos!
James