

**MP_0489. Programación
multimedia y dispositivos móviles
UF2. Programación de
aplicaciones para dispositivos móviles**

2.2. Material Design

Índice

☰	Objetivos	3
☰	Introducción	4
☰	Colores	6
☰	Tipografía	12
☰	Layout	15
☰	Componentes y patrones	18
☰	Movimiento	27
☰	Animaciones	30
☰	Resumen	38

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

- 1 Descubrir la normativa *Material Design*.
- 2 Conocer los principios de Material Design en colores, tipografía, *layout*, componentes y animaciones.
- 3 Implementar estos principios en las aplicaciones a desarrollar.

¡Ánimo y adelante!

Introducción

Material Design es una normativa de diseño que Google creó en 2014. Su objetivo es una experiencia de usuario unificada en plataformas y dispositivos.

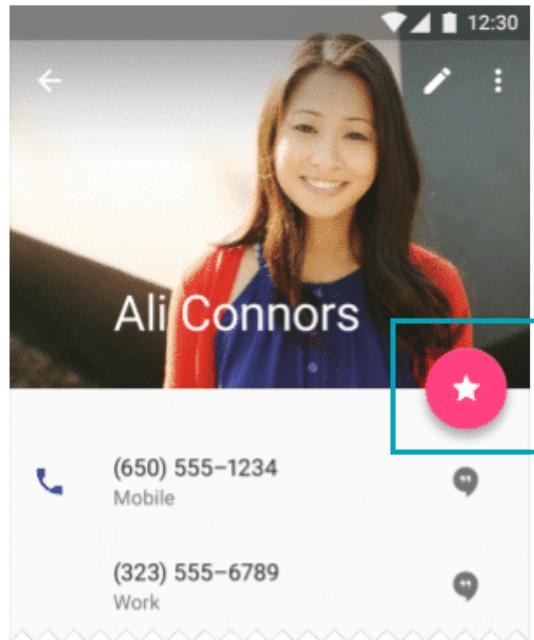
Material Design incluye un **conjunto de pautas para el estilo, diseño, movimiento y otros aspectos del diseño en la aplicación**. Las directrices completas están disponibles en sus especificaciones.

Principios de *Material Design*

En *Material Design* los elementos de la aplicación se comportan como materiales del mundo real: proyectan sombras, ocupan espacio e interactúan entre sí.

Implica elecciones de color deliberadas, imágenes de borde a borde, tipografía a gran escala y espacios en blanco intencionados que crean una interfaz audaz y gráfica.

Enfatiza las acciones de usuario en la aplicación para que el usuario sepa de inmediato qué hacer y cómo hacerlo, y resalta las cosas con las que los usuarios pueden interactuar, como los **botones**, los campos *EditText* y los *switches*.



En este diseño, por ejemplo, el botón de acción flotante se resalta con un acento de color rosa.

Movimiento

Es importante que las animaciones y otros movimientos en la aplicación sean significativos, es decir, usar movimientos para reforzar la idea de que el usuario es lo más importante de la aplicación.

Por ejemplo, **diseñar la aplicación de modo que la mayoría de los movimientos sean provocados por las acciones del usuario**, no por eventos fuera de su control. También podemos usar el movimiento para enfocar la atención del usuario, darle *feedback* o resaltar un elemento de la aplicación.



Cuando la aplicación presenta un objeto al usuario, **debemos asegurarnos de que el movimiento no interrumpe la experiencia del usuario**. Por ejemplo, no debería tener que esperar a que se complete una animación o transición.

Colores

Tres de los pilares del *Material Design* son los colores, la tipografía y los layout.

En el siguiente apartado nos centraremos en el uso de los **colores**.



Los principios de *Material Design* incluyen el uso de colores llamativos.

Cuando creamos un proyecto en Android Studio, se selecciona una combinación de colores de *Material Design* de muestra y se aplica al tema. En `values/colors.xml` se definen tres elementos `<color>`:

1. `ColorPrimary`.
2. `ColorPrimaryDark`.
3. `ColorAccent`.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <!-- Indigo. -->
    <color name="colorPrimaryDark">#303F9F</color>
    <!-- A darker shade of indigo. -->
    <color name="colorAccent">#FF4081</color>
    <!-- A shade of pink. -->
</resources>
```

En `values/styles.xml` los tres colores definidos se aplican al tema predeterminado, que implementa los colores en algunos elementos de la aplicación de forma predeterminada:

1

`colorPrimary`

Es utilizado por varias vistas de forma predeterminada. Por ejemplo, en el tema `AppTheme`, `colorPrimary` se utiliza como color de fondo para la barra de acción.

2

`ColorPrimaryDark`

Se usa en áreas que necesitan un ligero contraste con su color primario como, por ejemplo, la barra de estado.

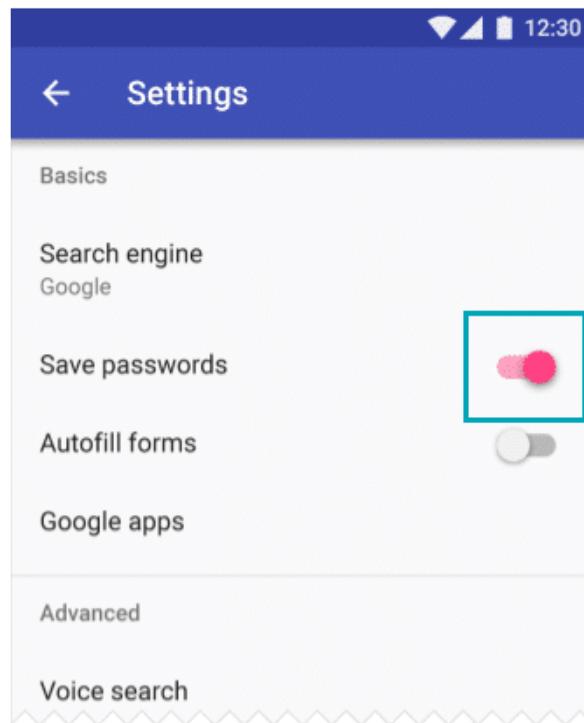
3

`ColorAccent`

Se utiliza como color de resaltado para varias vistas. También se usa para los interruptores en la posición "on", los botones de acción y otras acciones.

En este diseño, el fondo de la barra de acción usa `colorPrimary` (índigo), la barra de estado usa `colorPrimaryDark` (un tono más oscuro que el índigo), y el interruptor en la posición "on" usa `colorAccent` (un tono de rosa).

El interruptor en la posición "on" se resalta con un color rosado.



Paleta de colores

Veamos cómo usar la paleta de colores de *Material Design* en nuestra aplicación:

- 1 Elegimos un color primario para la aplicación de la paleta de colores de *Material Design* y copiamos su valor hexadecimal en el elemento `colorPrimary`, en `colors.xml`.
- 2 Elegimos un tono más oscuro de este color y copiamos su valor hexadecimal en el elemento `colorPrimaryDark`.
- 3 Elegimos un color destacado oscuro que comienza con una "A" y copiamos su valor hexadecimal en el elemento `colorAccent`.
- 4 Si necesitamos más colores, creamos elementos `<color>` adicionales en el archivo `colors.xml`. Por ejemplo, podemos elegir una versión más ligera del índigo y crear un elemento `<color>` adicional llamado `colorPrimaryLight`.

```
<color name="colorPrimaryLight">#9FA8DA</color>
```

Para usar este color, lo llamamos así:
`@color/colorPrimaryLight.`

Cambiar los valores en colors.xml cambia automáticamente los colores de las Views de la aplicación, porque los colores se aplican al tema en styles.xml.

- Si el tema se hereda de `Theme.AppCompat`, el sistema asume que estamos utilizando un fondo oscuro. Por lo tanto, todo el texto es casi blanco por defecto.
- Si el tema se hereda de `Theme.AppCompat.Light`, el texto es casi negro, porque el tema tiene un fondo claro.
- Si usamos el tema `Theme.AppCompat.Light.DarkActionBar`, el texto en la barra de acción es casi blanco, para contrastar con el fondo oscuro de la barra de acción. El resto del texto en la aplicación es casi negro, para contrastar con el fondo claro.

Es conveniente usar el contraste de color para crear una separación visual entre los elementos de la aplicación. Así, el color de su `colorAccent` debe llamar la atención sobre los elementos clave de la interfaz de usuario, como los botones de acción y los `switches` en la posición "on".

Opacidad

La aplicación puede mostrar texto con diferentes grados de opacidad para transmitir la importancia relativa a su información.

Podemos establecer el atributo `android:textColor` utilizando cualquiera de estos formatos:

- "#rgb".
- "#rrggb".
- "#argb".
- "#aarrggb".

Para establecer la opacidad del texto usamos el "#argb" o "#aarrggb" e incluimos un valor para el canal alfa. El canal alfa es el *a* o el *aa* al comienzo del valor de *textColor*.

- El valor máximo de opacidad, FF en hexadecimal, hace que el color sea completamente opaco.
- El valor mínimo, 00 en hexadecimal, hace que el color sea transparente.

Determinar qué número hexadecimal usamos en el canal alfa

1

Decidimos qué nivel de opacidad queremos utilizar, como porcentaje. El nivel de opacidad utilizado para el texto depende de si el fondo es oscuro o claro.

2

Multiplicamos ese porcentaje, como valor decimal, por 255. Por ejemplo, si necesitamos un texto primario que sea opaco al 87%, multiplicamos 0.87×255 . El resultado es 221.85.

3

Redondeamos el resultado al número entero más cercano: 222.

4

Usamos un convertidor hexadecimal para convertir el resultado a hex: DE. Si el resultado es un solo valor, ponemos un 0 delante.

En el siguiente código XML, el fondo del texto es oscuro y el color del texto primario es 87% blanco (`deffffff`). Los dos primeros números del código de color (de) indican la opacidad.

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Hello World!"  
    android:textSize="45dp"  
    android:background="@color/colorPrimaryDark"  
    android:textColor="#defffff"/>
```

Tipografía

Como sabes, los pilares del *Material Design* son los colores, la tipografía y los layout.

En este apartado nos centraremos en el uso de la **tipografía**.

Tipo de letra

Roboto es la tipografía estándar de *Material Design* en Android. Roboto tiene seis pesos:

Roboto Thin
Roboto Light
Roboto Regular
Roboto Medium
Roboto Bold
Roboto Black
Roboto Thin Italic
Roboto Light Italic
Roboto Italic
Roboto Medium Italic
Roboto Bold Italic
Roboto Black Italic

1. **Fino.**
2. **Ligero.**
3. **Regular.**
4. **Medio.**
5. **Negrita.**
6. **Negro.**

Estilos de fuente

Android proporciona estilos y tamaños de fuente predefinidos que podemos utilizar en la aplicación. Estos estilos y tamaños se desarrollaron para equilibrar la densidad del contenido y la comodidad de lectura en condiciones típicas.

Los tamaños de tipo se especifican con **sp** (píxeles escalables) para habilitar los modos de tipografía mayor para la accesibilidad.



i Ojo! No es conveniente usar demasiados tipos de tamaños y estilos diferentes juntos en un diseño.

Display 4

Light 112sp

Display 3

Regular 56sp

Display 2

Regular 45sp

Display 1

Regular 34sp

Headline

Regular 24sp

Title

Medium 20sp

Subheading

Regular 16sp (Device), Regular 15sp (Desktop)

Body 2

Medium 14sp (Device), Medium 13sp (Desktop)

Body 1

Regular 14sp (Device), Regular 13sp (Desktop)

Caption

Regular 12sp

Button

MEDIUM (ALL CAPS) 14sp

Para usar uno de estos estilos predefinidos en una *View*, establecemos el atributo *android:textAppearance*. Este atributo define la apariencia predeterminada del texto: su color, tipo de letra, tamaño y estilo.

Por ejemplo, para hacer que un *TextView* aparezca en el estilo Display 3, agregaremos el siguiente atributo al *TextView* en XML:

```
android:textAppearance="@style/TextAppearance.AppCompat.Display3"
```

Layout

Ya conocemos el uso de colores y tipografías en *Material Design*. ¿Qué nos falta?

En el siguiente apartado nos centraremos en el uso de los **layouts**.

Métricas y líneas clave

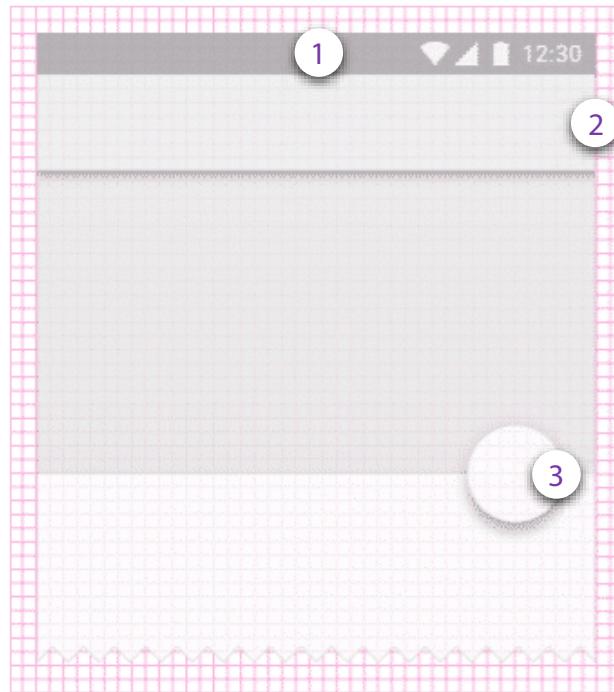
Los componentes de las plantillas de *Material Design* destinados a dispositivos móviles, tabletas y al escritorio se alinean con una cuadrícula cuadrada de 8dp.



Un dp es un píxel independiente de la densidad, una unidad abstracta basada en la densidad de la pantalla.

Es similar a un sp, pero sp se puede escalar según la preferencia de tamaño de fuente del usuario, por eso se prefiere cuando se busca la máxima accesibilidad.

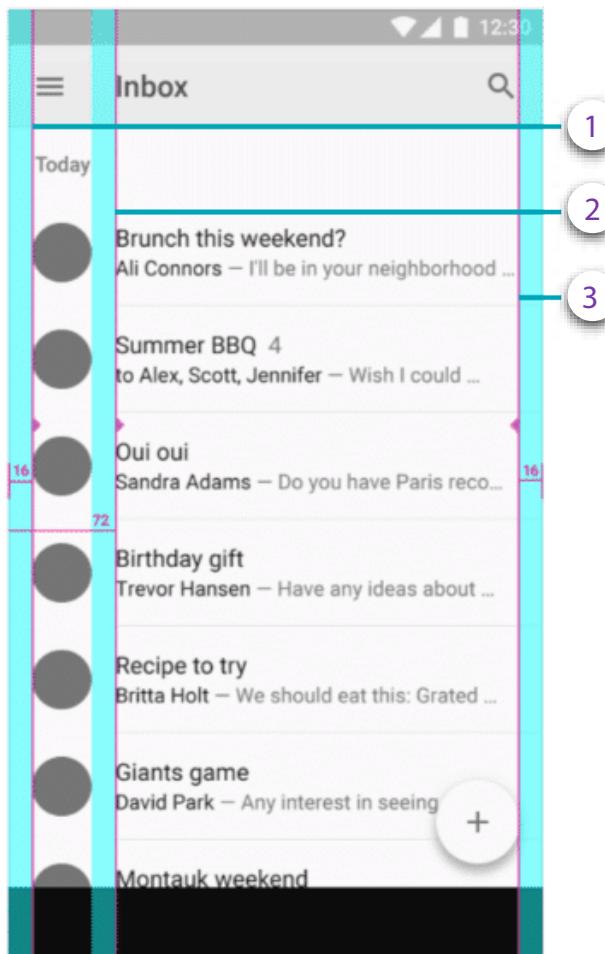
La cuadrícula de 8dp guía la ubicación de los elementos en su diseño. Cada cuadrado en la cuadrícula es **8dp x 8dp**, por lo que la altura y el ancho de cada elemento en el diseño es un múltiplo de 8dp.



1. La barra de estado en este diseño es 24dp de altura, es decir, la altura de tres cuadrados de cuadrícula.
2. La barra de herramientas tiene una altura de 56dp, la altura de siete cuadrículas.
3. Uno de los márgenes de contenido de la derecha es 16dp desde el borde de la pantalla, es decir, el ancho de dos cuadrículas.

Los **iconos** en las barras de herramientas se alinean con una cuadrícula cuadrada de 4dp en lugar de una cuadrícula cuadrada de 8dp, por lo que sus dimensiones son múltiplos de 4dp.

Las **líneas clave** son contornos en una cuadrícula de diseño que determinan la ubicación del texto y los iconos. Por ejemplo, las líneas clave marcan los bordes de los márgenes en un diseño.



1. Línea clave que muestra el margen izquierdo para el borde de la pantalla, que, en este caso, es 16dp.
2. Línea clave que muestra el margen izquierdo para el contenido asociado con un ícono o avatar, 72dp.
3. Línea clave que muestra el margen derecho para el borde de la pantalla, 16dp.

La tipografía de *Material Design* se alinea con una cuadrícula de base de 4dp, que es una cuadrícula formada únicamente por líneas horizontales.

Componentes y patrones

Los botones y muchas otras vistas utilizadas por Android se ajustan por defecto a los principios de *Material Design*.

La guía de *Material Design* incluye **componentes y patrones en los que puede desarrollarse para ayudar a los usuarios a intuir cómo funcionan los elementos** en su interfaz, incluso si son inexpertos en esta aplicación.

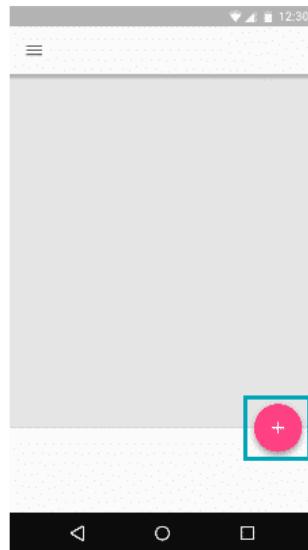
Floating action buttons (FABs)



Un *Floating action button* (FAB) es un ícono en un círculo que flota "sobre" la interfaz de usuario, y sirve para alentar al usuario a pulsarlo.

En el enfoque cambia ligeramente de color, y parece que se levanta cuando se selecciona.

Cuando se toca, **puede contener acciones relacionadas**.



Un FAB de tamaño normal.

Para implementar un FAB, usamos el widget *FloatingActionButton* y establecemos los atributos FAB en su diseño XML. **Por ejemplo:**

```
<android.support.design.widget.FloatingActionButton  
    android:id="@+id/addNewItemFAB"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:src="@drawable/ic_plus_sign"  
    app:fabSize="normal"  
    app:elevation="10%" />
```

El atributo *fabSize* establece el tamaño de FAB. Puede ser

- "normal" (56dp).
- "mini" (40dp).
- "auto", que cambia según el tamaño de la ventana.

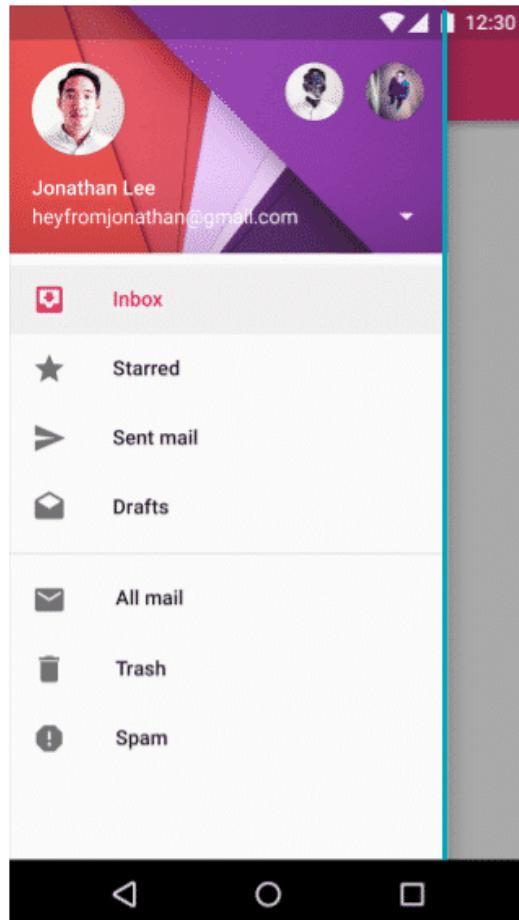
La elevación del FAB es la distancia entre su superficie y la profundidad de su sombra.

Navigation drawer



Navigation drawer es un panel que se desliza desde la izquierda y contiene destinos de navegación para la aplicación.

Un *Navigation drawer* abarca la altura de la pantalla, y todo lo que hay detrás de él es visible, pero está oscurecido.



Un Navigation drawer "abierto".

Para implementarlo, usamos las API de *DrawerLayout* disponibles en la biblioteca de soporte.

En XML, usamos un objeto *DrawerLayout* como *view principal* del *layout*. Dentro, agregamos dos vistas:

- Una para el diseño principal cuando el *Navigation drawer* está oculto.
- Otra para el contenido del *Navigation drawer*.

Por ejemplo, el siguiente diseño tiene dos vistas secundarias: un *FrameLayout* para contener el contenido principal y un *ListView* para el *Navigation drawer*.

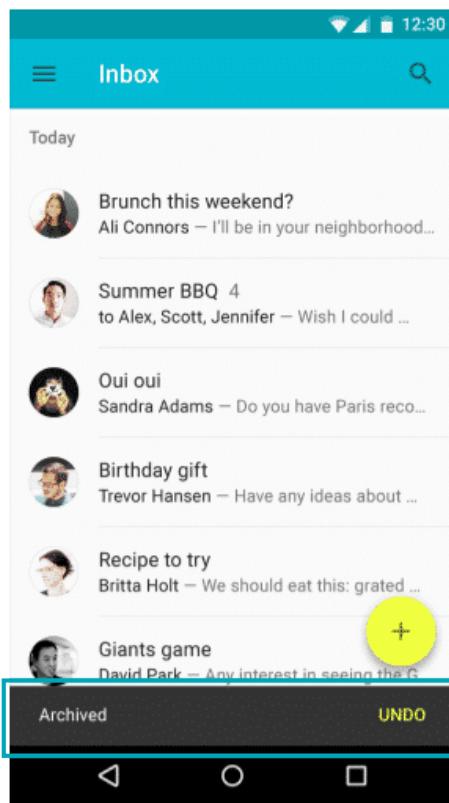
```
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <!-- The main content view -->
    <FrameLayout
        android:id="@+id/content_frame"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
    <!-- The navigation drawer -->
    <ListView android:id="@+id/left_drawer"
        android:layout_width="240dp"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:choiceMode="singleChoice"
        android:divider="@android:color/transparent"
        android:dividerHeight="0dp"
        android:background="#111"/>
</android.support.v4.widget.DrawerLayout>
```

Snackbar



Un **Snackbar** proporciona información breve sobre una operación a través de un mensaje en una barra horizontal en la pantalla.

Contiene una sola línea de texto directamente relacionada con la operación realizada. Un **Snackbar** **no puede contener iconos**.



Snackbar.

Los **snackbars** desaparecen automáticamente después de un tiempo de espera, o después de una interacción del usuario en otra parte de la pantalla.

Podemos asociar un **Snackbar** con cualquier tipo de *view*. Sin embargo, si asociamos el **Snackbar** con un *CoordinatorLayout*, el **Snackbar** tiene características adicionales:

- El usuario puede descartar el *Snackbar* al deslizarlo.
- El *layout* mueve algunos otros elementos de la interfaz de usuario cuando aparece el *Snackbar*. Por ejemplo, si el diseño tiene un FAB, el diseño mueve el FAB hacia arriba cuando muestra el *Snackbar*, en lugar de dibujar el *Snackbar* sobre la FAB.

Para crear un objeto *Snackbar* usamos el método *Snackbar.make()*. Especificamos el ID de la view del *CoordinatorLayout* que se utilizará para el *Snackbar*, el mensaje que muestra el *Snackbar* y el tiempo que debe transcurrir para mostrar el mensaje.

Por ejemplo, esta declaración crea el *Snackbar* y llama a *show()* para mostrarlo al usuario:

```
Snackbar.make(findViewById(R.id.myCoordinatorLayout), R.string.email_sent,
```

Tabs

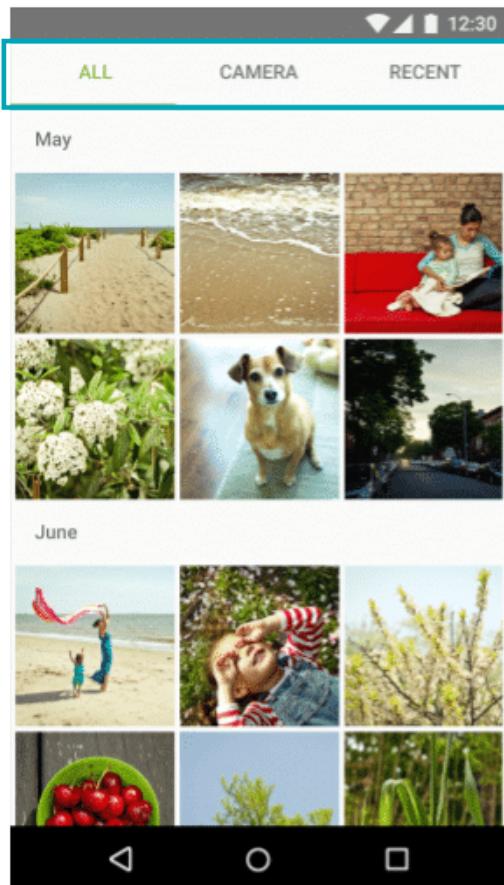


Las pestañas o *Tabs* se usan para organizar el contenido a un alto nivel. Por eso es mejor usar las etiquetas del *tab* cortas e informativas.

Por ejemplo, el usuario puede usar pestañas para cambiar entre *Views*, conjuntos de datos o aspectos funcionales de la aplicación. Las pestañas se muestran como una sola fila sobre el contenido asociado.

Podemos usar las *tabs* con *swipe views*, en las que los usuarios navegan entre las pestañas con un gesto de dedo horizontal.

Si las pestañas usan *swipe views*, no es bueno vincularlas con contenido que también se pueda deslizar.



Tres tabs.

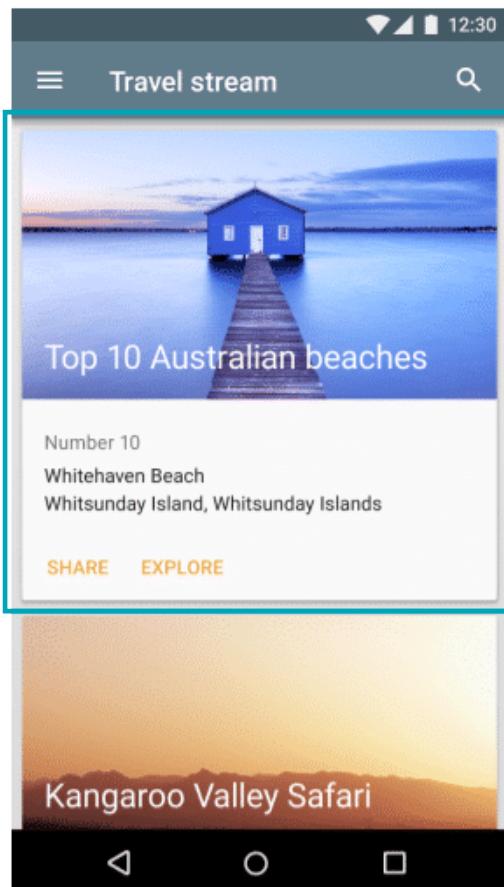
Cards



Un *card* es una “hoja” que sirve como punto de entrada para obtener información más detallada.

Cada *card* cubre solo un tema, y puede contener una foto, un texto y un enlace.

Puede mostrar contenido que contenga elementos de diferentes tamaños, como fotos con textos de distinta longitud.



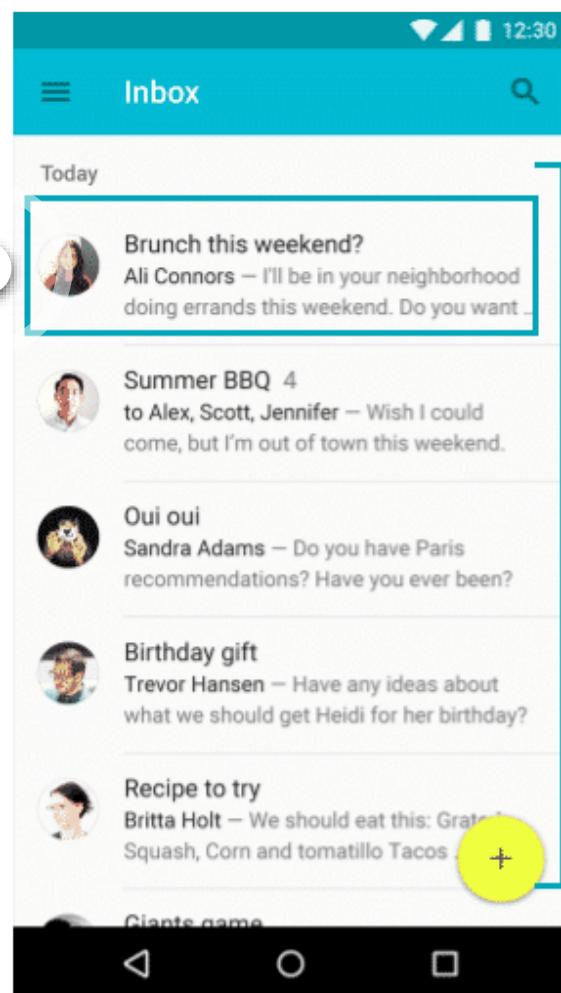
Card.

Lists



Una *list* es una sola columna continua de filas de igual ancho.

Cada fila funciona como un contenedor para un pieza. Las piezas tienen contenido y pueden variar en altura dentro de una lista.



1. Una pieza dentro de la lista.
2. Una lista con filas de igual ancho, cada una con una pieza.

Movimiento

El movimiento en *Material Design* se utiliza para describir las relaciones espaciales, la funcionalidad y la intención.

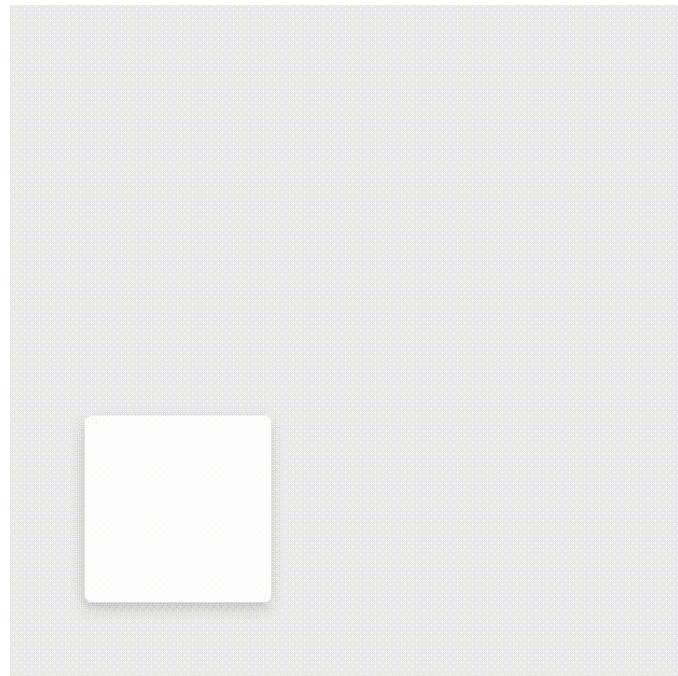
El movimiento muestra **cómo se organiza una aplicación y qué puede hacer.**

Características del movimiento en *Material Design*

El movimiento en *Material Design* debe ser:

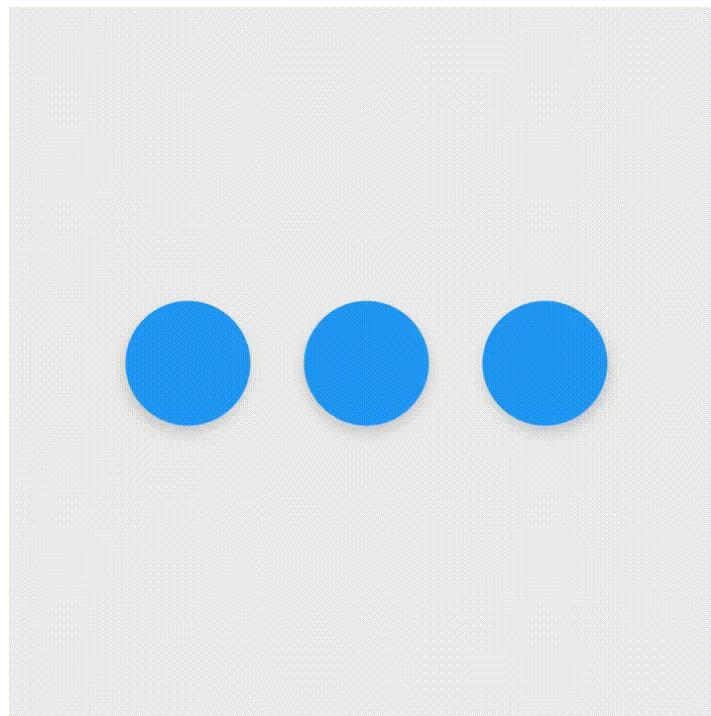
- **Rápido**
Responde rápidamente a la entrada del usuario precisamente donde el usuario la activa.

- **Natural**
Está inspirado por fuerzas del mundo natural. Por ejemplo, la gravedad inspira el movimiento de un elemento a lo largo de un arco, en lugar de en una línea recta.



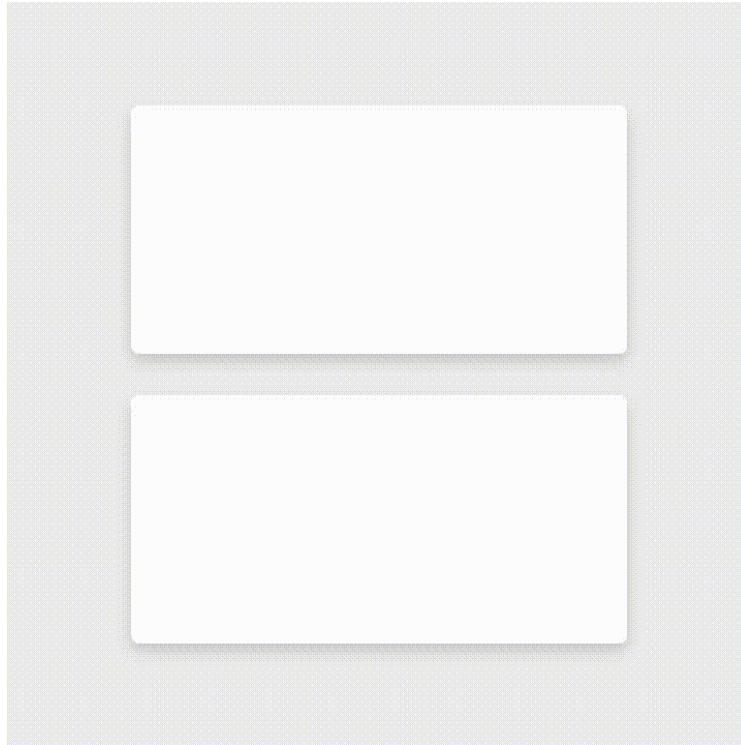
- **Consciente**

Material Design es consciente de su entorno, incluido el usuario y el material que lo rodea. Los objetos pueden ser atraídos a otros objetos en la interfaz de usuario y responden adecuadamente a la intención del usuario. A medida que los elementos pasan a la *View*, su movimiento se coreografía de una manera que define sus relaciones.



- **Con intención**

El movimiento guía el enfoque del usuario al lugar correcto en el momento adecuado. Puede comunicar diferentes señales, como si una acción está o no está disponible, por ejemplo.



Para poner en práctica estos principios en Android, utilizamos **animaciones** y **transiciones**, que te mostraremos en el siguiente apartado.

Animaciones

A continuación analizaremos más detalladamente cómo funcionan las animaciones y transiciones.

Comenzaremos por **crear animaciones**.

Formas de crear animaciones

Hay tres formas de crear animaciones en una aplicación:

1

Property animation

Cambia las propiedades de un objeto durante un período de tiempo específico. El sistema de *Property animation* se introdujo en Android 3.0 (nivel de API 11). Es más flexible que *view animation* y también ofrece más funciones.

2

View animation

Calcula la animación utilizando puntos de inicio, puntos finales, rotación y otros aspectos de la animación. El sistema de *view animation* es más antiguo que el de *property animation* y solo se puede utilizar para las Views. Es relativamente fácil de configurar y ofrece capacidades suficientes para la mayoría de los casos.

3

Drawable animation

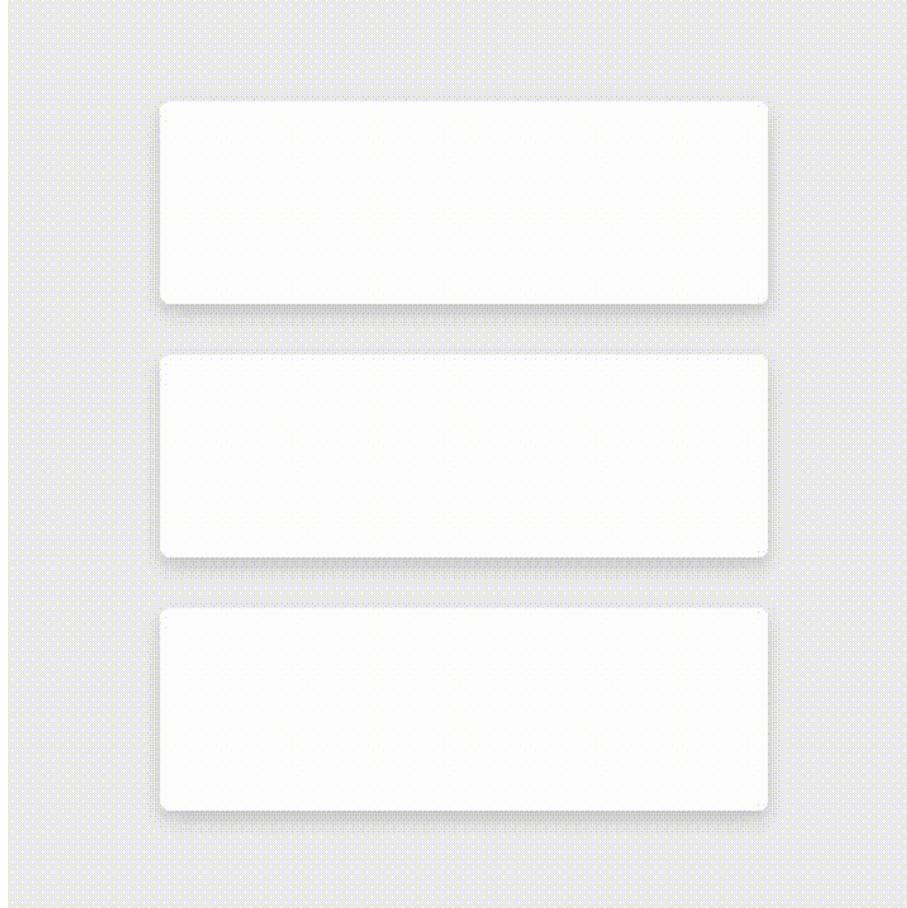
Nos permite cargar una serie de recursos *drawables* uno tras otro para crear una animación. La *drawable animation* es útil si desea animar cosas que son más fáciles de representar con recursos *drawables*, como, por ejemplo, una progresión de imágenes de mapa de bits.

Los temas en *Material Design* proporcionan algunas animaciones predeterminadas para comentarios táctiles y *Activity transitions*. Las API de animación, además, nos permiten crear animaciones personalizadas para *touch feedback* en los controles de la interfaz de usuario, cambios en el estado de visualización y *Activity transitions*.

Animaciones *Touch feedback*

Nos proporciona una confirmación visual instantánea en el punto de contacto cuando un usuario interactúa con un elemento de la interfaz de usuario. Las animaciones de *touch feedback* predeterminadas para los botones usan la clase *RippleDrawable*, que realiza transiciones entre diferentes estados con un efecto de onda.

En este **ejemplo**, las ondas de tinta se expanden hacia afuera desde el punto de contacto para confirmar la entrada del usuario. El card "levanta" y proyecta una sombra para indicar un estado activo:



En la mayoría de los casos, aplicamos la onda en la vista XML especificando el fondo de la vista de las siguientes maneras:

1

Para una onda limitada:

```
?android:attr/selectableItemBackground
```

2

Para una onda que se extiende más allá de la View:

```
?android:attr/selectableItemBackgroundBorderless
```

También podemos definir un *RippleDrawable* como un recurso XML utilizando el elemento <ripple>.

Podemos asignar un color a los objetos *RippleDrawable*. Para **cambiar el color de touch feedback** predeterminado, usamos el atributo *android:colorControlHighlight* del tema.

Circular reveal

Una animación *reveal* muestra u oculta un grupo de elementos de la interfaz de usuario al animar los límites de recorte de una vista. *Circular reveal* revela u oculta una vista animando un círculo de recorte.



Un círculo de recorte es un **círculo que recorta u oculta la parte de una imagen que está fuera del círculo**.

Para animar un círculo de recorte usamos el **método *ViewAnimationUtils.createCircularReveal()***.

Por ejemplo, observa **cómo revelar una View que anteriormente era invisible**:

```
// previously invisible view
View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;

// get the final radius for the clipping circle
float finalRadius = (float) Math.hypot(cx, cy);

// create the animator for this view (the start radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy, 0, finalRa-
dius);

// make the view visible and start the animation
myView.setVisibility(View.VISIBLE);
anim.start();
```

A continuación, te explicamos **cómo ocultar una View previamente visible mediante una circular reveal**:

```
// previously visible view
final View myView = findViewById(R.id.my_view);

// get the center for the clipping circle
int cx = myView.getWidth() / 2;
int cy = myView.getHeight() / 2;

// get the initial radius for the clipping circle
float initialRadius = (float) Math.hypot(cx, cy);

// create the animation (the final radius is zero)
Animator anim =
    ViewAnimationUtils.createCircularReveal(myView, cx, cy, initialRadius, 0);

// make the view invisible when the animation is done
anim.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator animation) {
        super.onAnimationEnd(animation);
        myView.setVisibility(View.INVISIBLE);
    }
});

// start the animation
anim.start();
```

Activity transitions

Son animaciones que proporcionan conexiones visuales entre diferentes estados en su **interfaz de usuario**. Pueden especificar animaciones personalizadas para las transiciones de entrada y salida , y para las transiciones de elementos compartidos entre actividades.

- Una transición de entrada determina cómo las *Views* entran en la escena en una actividad. Por ejemplo, en una transición de explosión las *views* entran en la escena desde el exterior y vuelan hacia el centro de la pantalla.
- Una transición de salida determina cómo las *Views* salen de la escena en una actividad. Por ejemplo, en una transición de salida de explosión, las vistas salen de la escena alejándose del centro.
- Una transición de elementos compartidos determina cómo las vistas que se comparten entre dos actividades hacen la transición entre estas actividades. Por ejemplo, si dos actividades tienen la misma imagen en diferentes posiciones y tamaños, la transición del elemento compartido *changeImageTransform* traduce y escala la imagen sin problemas entre estas actividades.

Para usar estas transiciones configuramos los atributos de transición en un elemento **<style>** en el XML.

El siguiente ejemplo crea un tema llamado *BaseAppTheme* que hereda uno de los temas de *Material Design*. Este tema usa los tres tipos de transiciones de actividad:

```
<style name="BaseAppTheme" parent="android:Theme.Material">
    <!-- enable window content transitions -->
    <item name="android:windowActivityTransitions">true</item>

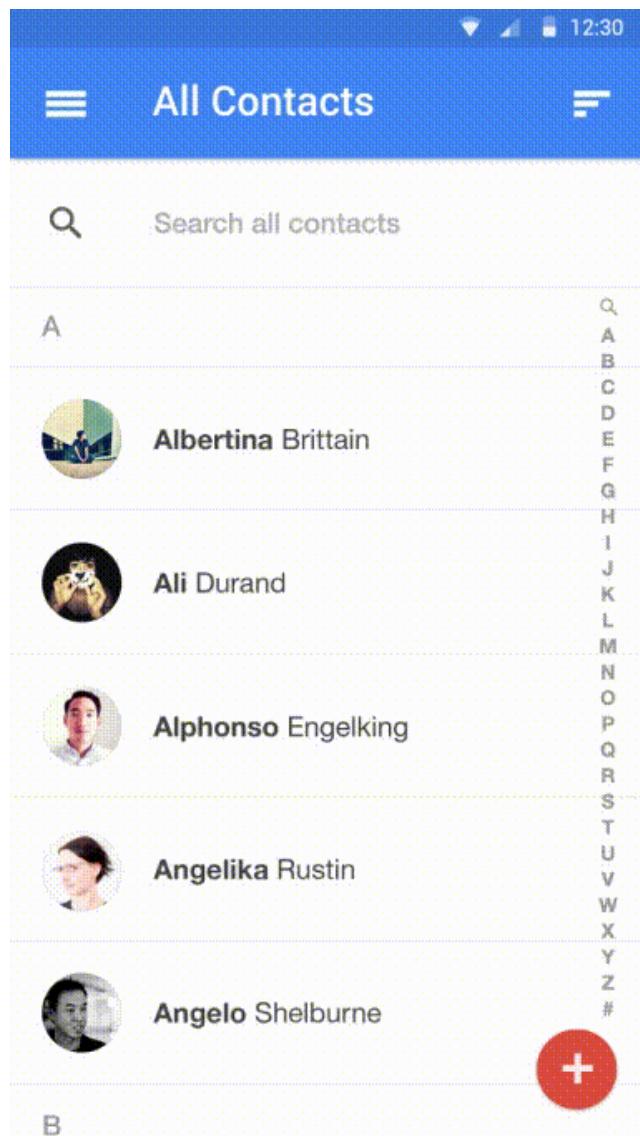
    <!-- specify enter and exit transitions -->
    <item name="android:windowEnterTransition">@transition/explode</item>
    <item name="android:windowExitTransition">@transition/explode</item>

    <!-- specify shared element transitions -->
    <item name="android:windowSharedElementEnterTransition">
        @transition/change_image_transform</item>
    <item name="android:windowSharedElementExitTransition">
        @transition/change_image_transform</item>
```

La transición *change_image_transform* en este ejemplo se define de la siguiente manera:

```
<!-- res/transition/change_image_transform.xml -->
<!-- (see also Shared Transitions below) -->
<transitionSet xmlns:android="http://schemas.android.com/apk/res/android">
    <changeImageTransform/>
</transitionSet>
```

El elemento *changeImageTransform* corresponde a la clase *ChangeImageTransform*.



Para especificar transiciones en su código, llamamos a los siguientes métodos con un objeto *Transition*:

- *Window.setEnterTransition()*.
- *Window.setExitTransition()*.
- *Window.setSharedElementEnterTransition()*.
- *Window.setSharedElementExitTransition()*.

Para iniciar una actividad que utiliza transiciones, usamos el **método *ActivityOptions.makeSceneTransitionAnimation()***.

Movimiento curvo

En Android 5.0 (nivel API 21) y superior, podemos definir patrones de movimiento curvo para animaciones. Para hacer esto, usamos la clase *PathInterpolator*, que interpola la ruta de un objeto basándose en una curva de Bézier o un objeto de Path.

El interpolador especifica una curva de movimiento en un cuadrado de 1x1, con puntos de anclaje en (0,0) y (1,1) y puntos de control, que especifica usando los argumentos del constructor. También puede definir un interpolador de ruta como un recurso XML:

```
<pathInterpolator xmlns:android="http://schemas.android.com/apk/res/android"  
    android:controlX1="0.4"  
    android:controlY1="0"  
    android:controlX2="1"  
    android:controlY2="1"/>
```

El sistema proporciona recursos XML para las tres curvas básicas en la especificación de *Material Design*:

- @interpolator/fast_out_linear_in.xml
- @interpolator/fast_out_slow_in.xml
- @interpolator/linear_out_slow_in.xml

Para usar un objeto *PathInterpolator*, lo pasamos al método *Animator.setInterpolator()* .

La clase *ObjectAnimator* tiene constructores que puede usar para animar coordenadas a lo largo de una ruta, usando dos o más propiedades a la vez. Por ejemplo, el siguiente código utiliza un objeto de *Path* para animar las propiedades X e Y de una *View*:

```
ObjectAnimator mAnimator;  
mAnimator = ObjectAnimator.ofFloat(view, View.X, View.Y, path);  
...  
mAnimator.start()
```

Resumen

Hemos terminado la lección, repasemos los puntos más importantes que hemos tratado.

- A lo largo de esta unidad hemos conocido la **normativa de diseño Material Design**, y su importancia en el aspecto de las aplicaciones Android sobre las que estamos trabajando.
- Hemos aprendido cómo manejar **colores, tipografía y composición**.
- Además, hemos conocido algunos de sus componentes: **Floating action buttons, Navigation drawer, Snackbar, Tabs, Cards y Lists**. Y aprendido cómo implementarlos en nuestras aplicaciones.
- Para finalizar, hemos descubierto el **movimiento** y cómo podemos implementarlo a través de **animaciones y transiciones**.

