

27 DE SEPTIEMBRE DE 2024

**INFORME DE INVESTIGACION: ABOUT FRAGMENTS, CREATE A FRAGMENT,  
FRAGMENT MANAGER, FRAGMENT TRANSACTIONS, ANIMATE TRANSITIONS  
BETWEEN FRAGMENTS, FRAGMENT LIFECYCLE, SAVING STATE WITH FRAGMENTS  
Y COMMUNICATE WITH FRAGMENTS**

SERGIO JUNIOR RUBEN DUARTE VANEGAS

# Indice

<u>Fragmentos</u> .....	3
<u>Modularidad</u> .....	3
<u>Cómo crear un fragmento</u> .....	4
<u>Cómo configurar tu entorno</u> .....	4
<u>Cómo crear una clase de fragmento</u> .....	5
<u>Cómo agregar un fragmento a una actividad</u> .....	5
<u>Cómo agregar un fragmento a través de XML</u> .....	6
<u>Cómo agregar un fragmento de manera programática</u> .....	6
<u>Administrador de fragmentos</u> .....	8
<u>Accede a FragmentManager</u> .....	8
<u>Fragmentos secundarios</u> .....	10
<u>Cómo usar FragmentManager</u> .....	11
<u>Cómo realizar una transacción</u> .....	11
<u>Cómo buscar un fragmento existente</u> .....	12
<u>Consideraciones especiales para fragmentos secundarios y del mismo nivel</u> .....	13
<u>Cómo brindar compatibilidad con varias pilas de actividades</u> .....	14
<u>Cómo proporcionar dependencias a tus fragmentos</u> .....	15
<u>Cómo realizar pruebas con FragmentFactory</u> .....	17
<u>Transacciones de fragmentos</u> .....	17
<u>Cómo permitir el reordenamiento de los cambios de estado de los fragmentos</u> .....	18
<u>Cómo agregar y quitar fragmentos</u> .....	18
<u>La confirmación es asíncrona</u> .....	19
<u>El orden de las operaciones es significativo</u> .....	20
<u>Cómo mostrar y ocultar vistas de fragmentos</u> .....	20
<u>Cómo conectar y desconectar fragmentos</u> .....	20
<u>Cómo navegar entre fragmentos con animaciones</u> .....	21
<u>Cómo configurar animaciones</u> .....	22
<u>Cómo establecer transiciones</u> .....	25
<u>Cómo usar transiciones de elementos compartidos</u> .....	26
<u>Cómo posponer transiciones</u> .....	29
<u>Cómo usar transiciones de elementos compartidos con una RecyclerView</u> .....	31
<u>Ciclo de vida de los fragmentos</u> .....	32

<u>Los fragmentos y el administrador de fragmentos .....</u>	33
<u>Estados del ciclo de vida de los fragmentos y devoluciones de llamada .....</u>	34
<u>Transiciones de estado ascendentes .....</u>	34
<u>Fragmento CREADO .....</u>	35
<u>Fragmento CREADO y vista INICIALIZADA .....</u>	36
<u>Fragmento y vista CREADOS .....</u>	36
<u>Fragmento y vista COMENZADOS .....</u>	36
<u>Fragmento y vista REANUDADOS .....</u>	37
<u>Transiciones de estado descendentes .....</u>	37
<u>Fragmento y vista COMENZADOS .....</u>	37
<u>Fragmento y vista CREADOS .....</u>	37
<u>Fragmento CREADO y vista DESTRUIDA .....</u>	38
<u>Fragmento DESTRUIDO .....</u>	38
<u>Cómo guardar un estado con fragmentos .....</u>	38
<u>Estado de vistas .....</u>	41
<u>NonConfig .....</u>	42
<u>Cómo comunicarse con fragmentos .....</u>	43
<u>Cómo compartir datos mediante un ViewModel .....</u>	43
<u>Cómo compartir datos con la actividad del host .....</u>	43
<u>Cómo compartir los datos entre fragmentos .....</u>	44
<u>Cómo compartir datos entre un fragmento superior y uno secundario .....</u>	46
<u>Cómo definir el alcance de un ViewModel según el gráfico de Navigation .....</u>	46
<u>Cómo obtener resultados con la API de resultados de fragmentos .....</u>	47
<u>Cómo pasar resultados entre fragmentos .....</u>	47
<u>Cómo probar los resultados de los fragmentos .....</u>	49
<u>Cómo pasar resultados entre fragmentos superiores y secundarios .....</u>	50
<u>Cómo recibir los resultados en la actividad del host .....</u>	52

# Fragmentos

Un Fragment representa una parte reutilizable de la IU de tu app. Un fragmento define y administra su propio diseño, tiene su propio ciclo de vida y puede administrar sus propios eventos de entrada. Los fragmentos no pueden existir por sí solos. Deben estar *alojados* por una actividad u otro fragmento. La jerarquía de vistas del fragmento forma parte de la jerarquía de vistas del host o está *conectada* a ella.

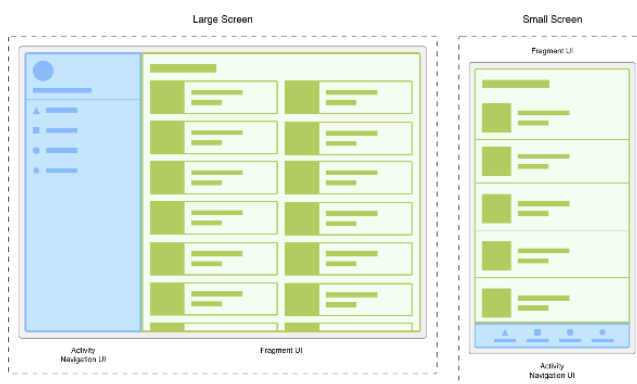
★ **Nota:** Algunas bibliotecas de [Android Jetpack](#), como [Navigation](#), [BottomNavigationView](#) y [ViewPager2](#), están diseñadas para funcionar con fragmentos.

## Modularidad

Los fragmentos introducen la modularidad y la capacidad de reutilización en la IU de tu actividad, ya que te permiten dividir la IU en fragmentos separados. Las actividades son un lugar ideal para colocar elementos globales en la interfaz de usuario de tu app, como un panel lateral de navegación. En cambio, los fragmentos son más adecuados para definir y administrar la IU de una sola pantalla o de una parte de ella.

Tomemos como ejemplo una app que responda a varios tamaños de pantalla. En pantallas más grandes, te recomendamos que la app muestre un panel lateral de navegación estático y una lista en un diseño de cuadrícula. En pantallas más pequeñas, te recomendamos que la app muestre una barra de navegación inferior y una lista en un diseño lineal.

Administrar estas variaciones en la actividad es difícil de controlar. Separar los elementos de navegación del contenido puede hacer que ese proceso sea más fácil de manejar. La actividad se encarga de mostrar la IU de navegación correcta, mientras que el fragmento muestra la lista con el diseño adecuado.



**Figura 1:** Dos versiones de la misma pantalla en pantallas de tamaños diferentes. En la parte izquierda, una pantalla grande contiene un panel lateral de navegación controlado por la actividad y una lista de cuadrícula controlada por el fragmento. A la derecha, una pantalla pequeña contiene una barra de navegación inferior controlada por la actividad y una lista lineal controlada por el fragmento.

Dividir tu IU en fragmentos te permite modificar la apariencia de la actividad con más facilidad durante el tiempo de ejecución. Mientras la actividad está en el estado de ciclo de vida `STARTED` o en uno superior, puedes agregar, reemplazar o quitar fragmentos. Puedes mantener un registro de esos cambios en una pila de actividades administrada por la actividad, lo que permite que se reviertan los cambios.

Puedes usar varias instancias de la misma clase de fragmento dentro de la misma actividad, en varias actividades o incluso como elemento secundario de otro fragmento. Con eso en mente, solo debes proporcionar un fragmento con la lógica necesaria para administrar su propia IU. Debes evitar que un fragmento dependa de otro, así como la manipulación del fragmento desde otro.

## Cómo crear un fragmento

Un fragmento representa una parte modular de la interfaz de usuario dentro de una actividad. Un fragmento tiene su propio ciclo de vida, recibe sus propios eventos de entrada, y tú puedes agregar o quitar fragmentos mientras se ejecuta la actividad que lo contiene.

En este documento, se describe cómo crear un fragmento e incluirlo en una actividad.

## Cómo configurar tu entorno

Los fragmentos requieren una dependencia en la biblioteca de fragmentos de AndroidX. Para incluir esta dependencia, debes agregar el repositorio de Maven de Google al archivo `settings.gradle` de tu proyecto.

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        ...
    }
}
```

Para incluir la biblioteca de AndroidX Fragment a tu proyecto, agrega las siguientes dependencias al archivo `build.gradle` de tu app:

```
dependencies {
    val fragment_version = "1.8.3"

    // Java language implementation
    implementation("androidx.fragment:fragment:$fragment_version")
    // Kotlin
    implementation("androidx.fragment:fragment-ktx:$fragment_version")
}
```

## Cómo crear una clase de fragmento

Si deseas crear un fragmento, extiende la clase `Fragment` de `AndroidX` y anula sus métodos para insertar la lógica de tu app, de manera similar a cómo crearías una clase `Activity`. Para crear un fragmento mínimo que defina su propio diseño, proporciona el recurso de diseño de tu fragmento en el constructor básico, como se muestra en el siguiente ejemplo:

```
class ExampleFragment : Fragment(R.layout.example_fragment)
```

La biblioteca de fragmentos también proporciona clases básicas de fragmentos más especializadas:

### `DialogFragment`

Muestra un diálogo flotante. Usar esta clase a fin de crear un diálogo es una buena alternativa al uso de los métodos auxiliares de diálogo en la clase `Activity`, ya que los fragmentos administran automáticamente la creación y la limpieza del `Dialog`. Consulta [Cómo mostrar diálogos con `DialogFragment`](#) para obtener más información.

### `PreferenceFragmentCompat`

Muestra una jerarquía de objetos `Preference` en forma de lista. Puedes usar `PreferenceFragmentCompat` a fin de crear una pantalla de configuración para tu app.

## Cómo agregar un fragmento a una actividad

En general, tu fragmento debe estar incorporado dentro de una `FragmentActivity` de `AndroidX` para contribuir con una parte de la IU al diseño de esa actividad. `FragmentActivity` es la clase básica de `AppCompatActivity`, de modo que si ya creas una subclase de `AppCompatActivity` a fin de proporcionar retrocompatibilidad en tu app, no es necesario que cambies la clase básica de tu actividad.

Puedes agregar tu fragmento a la jerarquía de vistas de la actividad; para ello, define el fragmento en el archivo de diseño de tu actividad, o bien define un contenedor de fragmentos en el archivo de diseño de la actividad y, luego, agrega el fragmento de forma programática desde tu actividad. En cualquier caso, debes agregar una `FragmentManager` que defina la ubicación donde se debería colocar el fragmento dentro de la jerarquía de vistas de la actividad. Te recomendamos que siempre uses una `FragmentManager` como contenedor de fragmentos, ya que `FragmentManager` incluye correcciones específicas para fragmentos que no proporcionan otros grupos de vistas, como `FrameLayout`.

## Cómo agregar un fragmento a través de XML

Para agregar un fragmento de forma declarativa al XML de diseño de tu actividad, usa un elemento `FragmentContainerView`.

A continuación, puedes ver un ejemplo de diseño de actividad que contiene un único `FragmentContainerView`:

```
<!-- res/layout/example_activity.xml -->
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.example.ExampleFragment" />
```

El atributo `android:name` especifica el nombre de clase del `Fragment` cuya instancia se creará. Cuando el diseño de la actividad aumenta, se crea una instancia del fragmento especificado, se llama a `onInflate()` en el fragmento cuya instancia se acaba de crear y se crea una `FragmentManager` para agregar el fragmento al `FragmentManager`.

★ **Nota:** Siempre debes usar `setReorderingAllowed(true)` cuando realices una `FragmentManager`. Para obtener más información sobre las transacciones reordenadas, consulta [Transacciones de fragmentos](#).

## Cómo agregar un fragmento de manera programática

Para agregar un fragmento de manera programática al diseño de tu actividad, debes incluir una `FragmentContainerView` que funcione como un contenedor de fragmentos, como se muestra en el siguiente ejemplo:

```
<!-- res/layout/example_activity.xml -->
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

A diferencia del enfoque XML, el atributo `android:name` no se usa en esta `FragmentContainerView`, por lo que no se crea automáticamente una instancia de ningún fragmento específico. En su lugar, se usa una `FragmentManager` a fin de crear una instancia de un fragmento y agregarlo al diseño de la actividad.

Mientras tu actividad se está ejecutando, puedes realizar transacciones de fragmentos, como agregar, quitar o reemplazar un fragmento. En tu `FragmentActivity`, puedes obtener una instancia del `FragmentManager`, que se puede usar para crear una `FragmentTransaction`. Luego, puedes crear una instancia de tu fragmento en el método `onCreate()` de tu actividad usando `FragmentManager.add()` y pasando el ID de `ViewGroup` del contenedor de tu diseño y la clase del fragmento que quieras agregar. A continuación, confirma la transacción, como se muestra en el siguiente ejemplo:

```
class ExampleActivity : AppCompatActivity(R.layout.example_activity) {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        if (savedInstanceState == null) {
            supportFragmentManager.commit {
                setReorderingAllowed(true)
                add<ExampleFragment>(R.id.fragment_container_view)
            }
        }
    }
}
```

★ **Nota:** Siempre debes usar `setReorderingAllowed(true)` cuando realices una `FragmentTransaction`. Para obtener más información sobre las transacciones reordenadas, consulta [Transacciones de fragmentos](#).

En el ejemplo anterior, ten en cuenta que la transacción de fragmentos solo se crea cuando `savedInstanceState` es `null`. Esto garantiza que el fragmento se agregue solo una vez, cuando se crea la actividad por primera vez. Cuando se produce un cambio de configuración y se vuelve a crear la actividad, `savedInstanceState` ya no es `null`, y no es necesario agregar el fragmento por segunda vez, ya que este se restablecerá automáticamente desde el `savedInstanceState`.

Si tu fragmento requiere algunos datos iniciales, puedes pasarle argumentos proporcionando un `Bundle` en la llamada a `FragmentManager.add()`, como se muestra a continuación:

```
class ExampleActivity : AppCompatActivity(R.layout.example_activity) {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        if (savedInstanceState == null) {
            val bundle = bundleOf("some_int" to 0)
            supportFragmentManager.commit {
                setReorderingAllowed(true)
                add<ExampleFragment>(R.id.fragment_container_view, args = bundle)
            }
        }
    }
}
```

Los argumentos `Bundle` se pueden recuperar desde tu fragmento llamando a `requireArguments()`, y se pueden usar los métodos `get` `Bundle` apropiados para recuperar cada argumento.



```
class ExampleFragment : Fragment(R.layout.example_fragment) {  
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {  
        val someInt = requireArguments().getInt("some_int")  
        ...  
    }  
}
```

## Administrador de fragmentos

★ **Nota:** Te recomendamos que uses la [biblioteca de Navigation](#) para administrar la navegación de tu app. El framework sigue las prácticas recomendadas para trabajar con fragmentos, la pila de actividades y el administrador de fragmentos. Para ver más información sobre Navigation, consulta [Cómo comenzar a usar el componente Navigation](#) y [Migra al componente Navigation](#).

FragmentManager es la clase responsable de realizar acciones en los fragmentos de tu app, como agregarlos, quitarlos o reemplazarlos, así como agregarlos a la pila de actividades.

Es posible que nunca interactúes con FragmentManager directamente si usas la biblioteca de Jetpack Navigation, ya que funciona con FragmentManager por ti. Sin embargo, cualquier app que utilice fragmentos usa FragmentManager en algún punto, por lo que es importante comprender qué es y cómo funciona.

En esta página, se abarca lo siguiente:

- Cómo acceder a FragmentManager
- Cuál es la función de FragmentManager en relación con tus actividades y fragmentos
- Cómo administrar la pila de actividades con FragmentManager
- Cómo proporcionar datos y dependencias a tus fragmentos

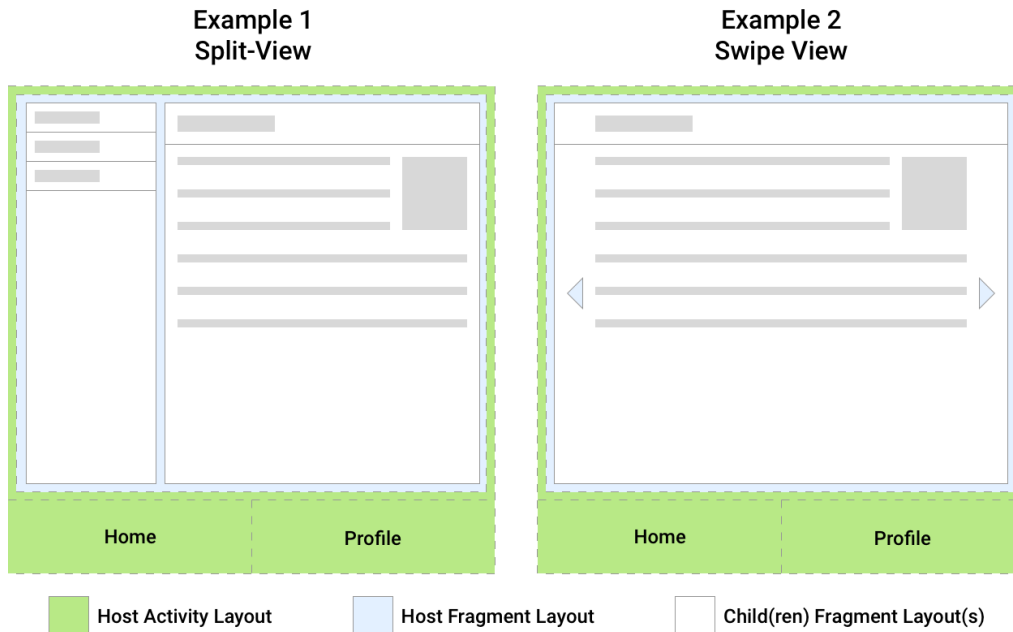
### Accede a FragmentManager

Puedes acceder a FragmentManager desde una actividad o un fragmento.

FragmentManager y sus subclases, como AppCompatActivity, tienen acceso a FragmentManager a través del método `getSupportFragmentManager()`.

Los fragmentos pueden alojar uno o más fragmentos secundarios. Dentro de un fragmento, puedes obtener una referencia a FragmentManager que administra los elementos secundarios del fragmento mediante `getChildFragmentManager()`. Si necesitas acceder a su host FragmentManager, puedes usar `getParentFragmentManager()`.

Veamos algunos ejemplos que muestran las relaciones entre los fragmentos, sus hosts y las instancias de FragmentManager asociadas con cada uno.

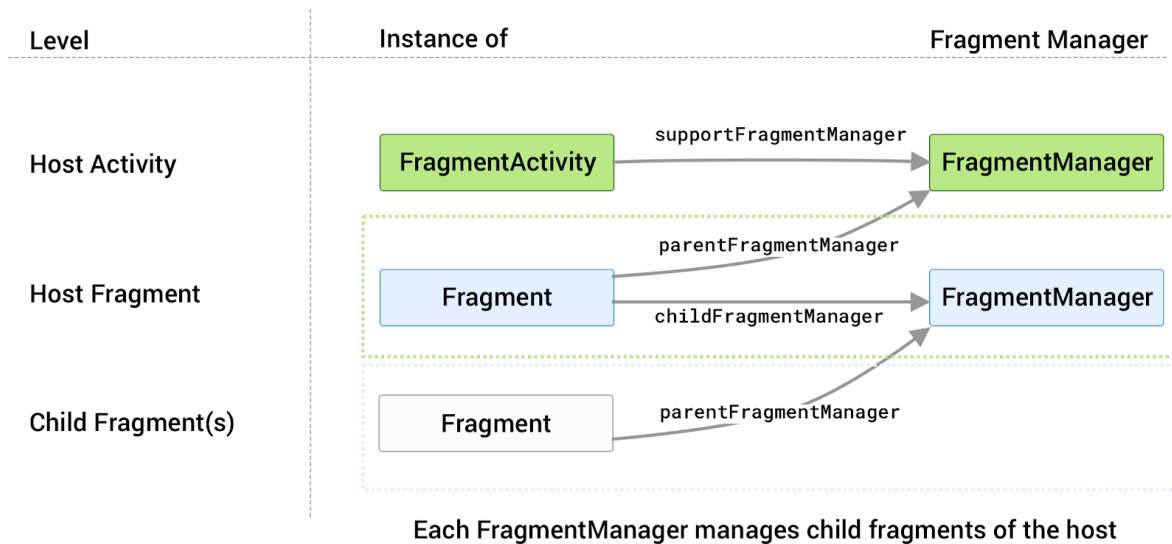


**Figura 1:** Dos ejemplos de diseño de la IU que muestran las relaciones entre los fragmentos y sus actividades de host

En la figura 1, se muestran dos ejemplos, cada uno de los cuales tiene un solo host de actividad. La actividad de host de ambos ejemplos muestra la navegación de nivel superior al usuario como una `BottomNavigationView` encargada de reemplazar el fragmento de host con diferentes pantallas de la app. Cada pantalla se implementa como un fragmento separado.

El fragmento de host del ejemplo 1 aloja dos fragmentos secundarios que conforman una pantalla con vista dividida. El fragmento de host del ejemplo 2 aloja un solo fragmento secundario que conforma el fragmento de la pantalla de una vista deslizante.

Dada esta configuración, se puede decir que cada host tiene un `FragmentManager` asociado que administra sus fragmentos secundarios. Esto se ilustra en la figura 2, junto con las asignaciones de propiedades entre `supportFragmentManager`, `parentFragmentManager` y `childFragmentManager`.



**Figura 2:** Cada host tiene su propio FragmentManager asociado que administra sus fragmentos secundarios

La propiedad FragmentManager adecuada a la cual hacer referencia depende del lugar en el que se encuentra el sitio de llamada en la jerarquía del fragmento, así como del administrador de fragmentos al que intentas acceder.

Una vez que tengas una referencia al FragmentManager, podrás usarla para manipular los fragmentos que se muestran al usuario.

## Fragmentos secundarios

En términos generales, tu app incluye una o varias actividades en el proyecto de la aplicación, y cada actividad debe representar un grupo de pantallas relacionadas. La actividad puede proporcionar un punto para colocar la navegación de nivel superior y un lugar para definir los objetos ViewModel y otro estado de vista entre fragmentos. Un fragmento representa un destino individual de tu app.

Si deseas mostrar varios fragmentos a la vez, como en una vista dividida o un panel, puedes usar fragmentos secundarios administrados por tu fragmento de destino y su administrador de fragmentos secundarios.

A continuación, se incluyen otros casos de uso para fragmentos secundarios:

- Diapositivas de pantalla, con un ViewPager2 en un fragmento superior para administrar una serie de vistas de fragmentos secundarios
- Subnavegación dentro de un conjunto de pantallas relacionadas
- Jetpack Navigation usa fragmentos secundarios como destinos individuales. Una actividad aloja un único NavHostFragment superior y llena su espacio con diferentes fragmentos secundarios de destino a medida que los usuarios navegan por tu app

## Cómo usar FragmentManager

FragmentManager administra la pila de actividades del fragmento. En el tiempo de ejecución, FragmentManager puede realizar operaciones de pila de actividades como agregar o quitar fragmentos en respuesta a interacciones del usuario. Cada conjunto de cambios se confirma como una sola unidad llamada FragmentTransaction. Para obtener un análisis más detallado sobre las transacciones de fragmentos, consulta la guía de transacciones de fragmentos.

Cuando el usuario presiona el botón Atrás en su dispositivo o cuando llamas a `FragmentManager.popBackStack()`, la transacción del fragmento que se encuentra en la parte superior de la pila se quita de ella. Si no hay más transacciones de fragmentos en la pila y no estás usando fragmentos secundarios, el evento Atrás aparecerá en la actividad. Si *estás* usando fragmentos secundarios, consulta las consideraciones especiales para fragmentos secundarios y del mismo nivel.

Cuando llamas a `addToBackStack()` en una transacción, esta puede incluir cualquier cantidad de operaciones, como agregar varios fragmentos o reemplazar fragmentos en múltiples contenedores.

Cuando se abre la pila de actividades, todas estas operaciones se revierten como una acción atómica única. Sin embargo, si confirmaste transacciones adicionales antes de la llamada a `popBackStack()` y *no* usaste `addToBackStack()` para la transacción, *no* se revertirán esas operaciones. Por lo tanto, dentro de una única `FragmentTransaction`, evita intercalar las transacciones que afectan la pila de actividades con aquellas que no lo hacen.

## Cómo realizar una transacción

Para mostrar un fragmento dentro de un contenedor de diseños, usa el FragmentManager a fin de crear una `FragmentTransaction`. Dentro de la transacción, puedes realizar una operación `add()` o `replace()` en el contenedor.

Por ejemplo, una `FragmentTransaction` simple puede verse de la siguiente manera:

```
supportFragmentManager.commit {  
    replace<ExampleFragment>(R.id.fragment_container)  
    setReorderingAllowed(true)  
    addToBackStack("name") // Name can be null  
}
```

En este ejemplo, `ExampleFragment` reemplaza al fragmento, si hay alguno, que se encuentra actualmente en el contenedor de diseño identificado por el ID de `R.id.fragment_container`. Proporcionar la clase del fragmento al método `replace()` permite que `FragmentManager` controle la creación de una instancia mediante su `FragmentManager`. Para obtener más información, consulta la sección [Cómo proporcionar dependencias a tus fragmentos](#).

`setReorderingAllowed(true)` optimiza los cambios de estado de los fragmentos involucrados en la transacción de modo que las animaciones y transiciones funcionen correctamente. Para obtener más información sobre la navegación con animaciones y transiciones, consulta [Transacciones de fragmentos](#) y [Cómo navegar entre fragmentos con animaciones](#).

Llamar a `addToBackStack()` confirma la transacción a la pila de actividades. Luego, el usuario puede revertir la transacción y recuperar el fragmento anterior presionando el botón Atrás. Si agregaste o

quitaste varios fragmentos dentro de una sola transacción, se descartarán todas las operaciones cuando se abra la pila de actividades. El nombre opcional proporcionado en la llamada a `addToBackStack()` te permite volver a esa transacción específica con `popBackStack()`.

Si no llamas a `addToBackStack()` cuando realizas una transacción que quita un fragmento, cuando se confirme la transacción, se destruirá el fragmento que se quitó, y el usuario no podrá volver a él. Si llamas a `addToBackStack()` cuando quitas un fragmento, ese fragmento pasa al estado `STOPPED` y, luego, a `RESUMED` cuando el usuario vuelva a él. Su vista *sí* se destruye en este caso. Para obtener más información, consulta [Ciclo de vida de los fragmentos](#).

## Cómo buscar un fragmento existente

Puedes obtener una referencia al fragmento actual dentro de un contenedor de diseños por medio de `findFragmentById()`. Usa `findFragmentById()` para buscar un fragmento por el ID dado cuando se lo aumenta desde XML o bien por el ID del contenedor cuando se lo agrega a una `FragmentManager`. Por ejemplo:

```
supportFragmentManager.commit {
    replace<ExampleFragment>(R.id.fragment_container)
    setReorderingAllowed(true)
    addToBackStack(null)
}
...
val fragment: ExampleFragment =
    supportFragmentManager.findFragmentById(R.id.fragment_container) as ExampleFragment
```

Como alternativa, puedes asignar una etiqueta única a un fragmento y obtener una referencia con `findFragmentByTag()`. Puedes asignar una etiqueta con el atributo XML `android:tag` en los fragmentos definidos de tu diseño o durante una operación `add()` o `replace()` dentro de una `FragmentManager`.

```
supportFragmentManager.commit {  
    replace<ExampleFragment>(R.id.fragment_container, "tag")  
    setReorderingAllowed(true)  
    addToBackStack(null)  
}  
...  
val fragment: ExampleFragment =  
    supportFragmentManager.findFragmentByTag("tag") as ExampleFragment
```

## Consideraciones especiales para fragmentos secundarios y del mismo nivel

Solo un `FragmentManager` puede controlar la pila de actividades del fragmento en cualquier momento. Si tu app muestra varios fragmentos del mismo nivel en la pantalla al mismo tiempo o si usa fragmentos secundarios, se designa un `FragmentManager` para controlar la navegación principal de tu app.

Para definir el fragmento de navegación principal dentro de una transacción de fragmento, llama al método `setPrimaryNavigationFragment()` en la transacción y pasa la instancia del fragmento cuyo `childFragmentManager` tiene el control principal.

Considera la estructura de navegación como una serie de capas, en la que la actividad es la capa más externa, que encierra cada capa de fragmentos secundarios debajo. Cada capa tiene un solo fragmento de navegación principal.

Cuando ocurre el evento Atrás, la capa más interna controla el comportamiento de navegación. Cuando esa capa no tenga más transacciones de fragmentos para mostrar, el control regresará a la siguiente capa, y este proceso se repetirá hasta llegar a la actividad.

Cuando se muestran dos o más fragmentos al mismo tiempo, solo uno de ellos es el fragmento de navegación principal. Configurar un fragmento como el de navegación principal quita la designación del fragmento anterior. Como se muestra en el ejemplo anterior, si estableces el fragmento de detalle como el fragmento de navegación principal, se quitará la designación del fragmento principal.

## Cómo brindar compatibilidad con varias pilas de actividades

En algunos casos, es posible que tu app necesite admitir varias pilas de actividades. Un ejemplo común es si tu app usa una barra de navegación inferior. `FragmentManager` te permite admitir varias pilas de actividades con los métodos `saveBackStack()` y `restoreBackStack()`, que te permiten intercambiar pilas de actividades guardando una pila y restableciendo una diferente.

★ **Nota:** De manera alternativa, puedes usar el componente [NavigationUI](#), que maneja automáticamente la compatibilidad con varias pilas de actividades para [navegación inferior](#).

`saveBackStack()` funciona de manera similar a llamar a `popBackStack()` con el parámetro opcional `name`: se resaltan las transacciones especificadas y todas las transacciones posteriores en la pila. La diferencia es que `saveBackStack()` guarda el estado de todos los fragmentos en las transacciones emergentes.

Por ejemplo, supongamos que agregaste un fragmento a la pila de actividades mediante la confirmación de una `FragmentTransaction` con `addToBackStack()`, como se muestra en el siguiente ejemplo:

```
supportFragmentManager.commit {  
    replace<ExampleFragment>(R.id.fragment_container)  
    setReorderingAllowed(true)  
    addToBackStack("replacement")  
}
```

En ese caso, puedes guardar esta transacción de fragmento y el estado de `ExampleFragment` llamando a `saveBackStack()`:

```
supportFragmentManager.saveBackStack("replacement")
```

★ **Nota:** Solo puedes usar `saveBackStack()` con transacciones que llamen a `setReorderingAllowed(true)`, de modo que las transacciones se puedan restablecer como una operación única y atómica.

Puedes llamar a `restoreBackStack()` con el mismo parámetro de nombre para restablecer todas las transacciones emergentes y todos los estados del fragmento guardado:

```
supportFragmentManager.restoreBackStack("replacement")
```

★ **Nota:** No puedes usar `saveBackStack()` ni `restoreBackStack()`, a menos que pases un nombre opcional para tus transacciones de fragmentos con `addToBackStack()`.

## Cómo proporcionar dependencias a tus fragmentos

Cuando agregas un fragmento, puedes crear una instancia del fragmento en forma manual y agregarla a la `FragmentManager`.

```
fragmentManager.commit {  
    // Instantiate a new instance before adding  
    val myFragment = ExampleFragment()  
    add(R.id.fragment_view_container, myFragment)  
    setReorderingAllowed(true)  
}
```

Cuando confirmas la transacción del fragmento, la instancia creada del fragmento es la instancia que se usará. No obstante, durante un cambio de configuración, tu actividad y todos sus fragmentos se destruyen y se vuelven a crear mediante los recursos de Android más aplicables. `FragmentManager` se encarga de todo esto por ti: recrea instancias de tus fragmentos, las adjunta al host y vuelve a crear el estado de la pila de actividades.

De forma predeterminada, `FragmentManager` usa una `FragmentFactory` que el framework proporciona para crear una instancia nueva de tu fragmento. Esta fábrica predeterminada usa el reflejo para encontrar e invocar un constructor sin argumentos para el fragmento. Eso significa que no puedes usarla a fin de proporcionar dependencias a tu fragmento. También significa que cualquier constructor personalizado que hayas usado para crear el fragmento la primera vez *no* se usará durante la recreación de forma predeterminada.

Para proporcionar dependencias a tu fragmento o usar cualquier constructor personalizado, crea una subclase de `FragmentFactory` personalizada y anula `FragmentFactory.instantiate`. Luego, puedes anular la fábrica predeterminada de `FragmentManager` con la tuya personalizada, que se usará para crear las instancias de los fragmentos.

Supongamos que tienes un `DessertsFragment` que se encarga de mostrar postres populares en tu ciudad natal y que `DessertsFragment` tiene una dependencia en una clase `DessertsRepository` que le proporciona la información que necesita para mostrar la IU correcta al usuario.

Podrías definir tu `DessertsFragment` de forma que requiera una instancia de `DessertsRepository` en su constructor.



```
class DessertsFragment(val dessertsRepository: DessertsRepository) : Fragment() {
    ...
}
```

Una implementación simple de tu FragmentFactory podría ser similar a la siguiente.

```
class MyFragmentFactory(val repository: DessertsRepository) : FragmentFactory() {
    override fun instantiate(classLoader: ClassLoader, className: String): Fragment =
        when (loadFragmentClass(classLoader, className)) {
            DessertsFragment::class.java -> DessertsFragment(repository)
            else -> super.instantiate(classLoader, className)
        }
}
```

En este ejemplo, se crean subclases de FragmentFactory, lo que anula el método instantiate() a fin de proporcionar una lógica de creación de fragmentos personalizada para un DessertsFragment. El comportamiento predeterminado de FragmentFactory controla mediante super.instantiate() otras clases de fragmentos.

Luego, puedes designar MyFragmentFactory como la fábrica que se usará cuando construyas los fragmentos de tu app configurando una propiedad en el FragmentManager. Debes establecer esta propiedad antes del objeto super.onCreate() de tu actividad para asegurarte de que se usará MyFragmentFactory cuando vuelvas a crear los fragmentos.

```
// Inside your test
val dessertRepository = mock(DessertsRepository::class.java)
launchFragment<DessertsFragment>{factory = MyFragmentFactory(dessertRepository)}.onFragment {
    // Test Fragment logic
}
```

Configurar la FragmentFactory en la actividad anula la creación de fragmentos en toda la jerarquía de fragmentos de la actividad. En otras palabras, el childFragmentManager de cualquier fragmento secundario que agregues usa la fábrica de fragmentos personalizada, a menos que se la anule en un nivel inferior.

## Cómo realizar pruebas con FragmentFactory

En una arquitectura de actividad única, prueba tus fragmentos de forma aislada usando la clase `FragmentScenario`. Dado que no puedes basarte en la lógica de `onCreate` personalizada de tu actividad, puedes pasar la `FragmentFactory` como un argumento en la prueba de fragmentos, como se muestra en el siguiente ejemplo:

```
// Inside your test
val dessertRepository = mock(DessertsRepository::class.java)
launchFragment<DessertsFragment>(factory = MyFragmentFactory(dessertRepository)).onFragment {
    // Test Fragment logic
}
```

## Transacciones de fragmentos

Durante el tiempo de ejecución, un `FragmentManager` puede agregar, quitar, reemplazar y realizar otras acciones con fragmentos en respuesta a la interacción del usuario. Cada conjunto de cambios de fragmento que confirmes se denomina *transacción*, y puedes especificar qué hacer dentro de la transacción mediante las API que proporciona la clase `FragmentTransaction`. Puedes agrupar varias acciones en una sola transacción; por ejemplo, una transacción puede agregar o reemplazar varios fragmentos. Esta agrupación puede ser útil para cuando se muestran varios fragmentos del mismo nivel en la misma pantalla, como en el caso de las vistas divididas.

Puedes guardar cada transacción en una pila de actividades administrada por el `FragmentManager`, lo que le permite al usuario navegar hacia atrás por los cambios realizados en el fragmento, de forma similar a navegar hacia atrás por las actividades.

Puedes obtener una instancia de `FragmentTransaction` desde el `FragmentManager` si llamas a `beginTransaction()`, como se muestra en el siguiente ejemplo:

```
val fragmentManager = ...
val fragmentTransaction = fragmentManager.beginTransaction()
```

La llamada final en cada `FragmentTransaction` debe confirmar la transacción. La llamada `commit()` señala al `FragmentManager` que todas las operaciones se agregaron a la transacción.

```
val fragmentManager = ...
// The fragment-ktx module provides a commit block that automatically
// calls beginTransaction and commit for you.
fragmentManager.commit {
    // Add operations here
}
```

## Cómo permitir el reordenamiento de los cambios de estado de los fragmentos

Cada `FragmentManager` debe usar `setReorderingAllowed(true)`:

```
supportFragmentManager.commit {  
    ...  
    setReorderingAllowed(true)  
}
```

Para lograr la compatibilidad del comportamiento, la marca de orden no está habilitada de forma predeterminada. Sin embargo, es necesaria para permitir que `FragmentManager` ejecute correctamente tu `FragmentManager`, en especial cuando opera en la pila de actividades y ejecuta animaciones y transiciones. Habilitar la marca garantiza que, si se ejecutan varias transacciones, los fragmentos intermedios (es decir, los que se agregan y, luego, se reemplazan de inmediato) no experimenten cambios del ciclo de vida, y que sus animaciones o transiciones no se ejecuten. Ten en cuenta que esta marca afecta tanto la ejecución inicial de la transacción como la reversión de la transacción con `popBackStack()`.

## Cómo agregar y quitar fragmentos

Para agregar un fragmento a un `FragmentManager`, llama a `add()` en la transacción. Este método recibe el ID del *contenedor* del fragmento, así como el nombre de la clase del fragmento que deseas agregar. El fragmento agregado pasa al estado `RESUMED`. Te recomendamos que el *contenedor* sea una `FragmentManager` que forme parte de la jerarquía de vistas.

Para quitar un fragmento del `host`, llama a `remove()` y pasa una instancia de fragmento que se recuperó del administrador de fragmentos a través de `findFragmentById()` o `findFragmentByTag()`. Si previamente se agregó la vista del fragmento a un contenedor, la vista se quitará del contenedor en este momento. El fragmento que se quita pasa al estado `DESTROYED`.

Usa `replace()` para reemplazar un fragmento existente en un contenedor por una instancia de la nueva clase de fragmento que proporciones. Llamar a `replace()` equivale a llamar a `remove()` con un fragmento en un contenedor y agregar un fragmento nuevo al mismo contenedor.

En el siguiente fragmento de código, se muestra cómo puedes reemplazar un fragmento por otro:

```
// Create new fragment  
val fragmentManager = // ...  
  
// Create and commit a new transaction  
fragmentManager.commit {  
    setReorderingAllowed(true)  
    // Replace whatever is in the fragment_container view with this fragment  
    replace<ExampleFragment>(R.id.fragment_container)  
}
```

En este ejemplo, una instancia nueva de `ExampleFragment` reemplaza el fragmento, si hay alguno, que se encuentra en ese momento en el contenedor de diseño identificado por `R.id.fragment_container`.

★ **Nota:** Te recomendamos que uses siempre las operaciones de fragmentos que toman una **Class** en lugar de una instancia de fragmento a fin de garantizar que los mismos mecanismos de creación del fragmento también se usarán para restablecer el fragmento desde un estado guardado. Consulta el [administrador de fragmentos](#) para obtener más información.

De forma predeterminada, los cambios realizados en una `FragmentTransaction` no se agregan a la pila de actividades. Para guardar esos cambios, puedes llamar a `addToBackStack()` en la `FragmentTransaction`. Si deseas obtener más información, consulta el administrador de fragmentos.

## La confirmación es asíncrona

Llamar a `commit()` no realiza la transacción inmediatamente. En cambio, la transacción se programará para ejecutarse en el subproceso de IU principal en cuanto pueda hacerlo. Sin embargo, si es necesario, puedes llamar a `commitNow()` a fin de ejecutar la transacción del fragmento en tu subproceso de IU de inmediato.

Ten en cuenta que `commitNow` no es compatible con `addToBackStack`. Como alternativa, puedes ejecutar todas las `FragmentTransactions` pendientes enviadas por las llamadas de `commit()` aún no ejecutadas llamando a `executePendingTransactions()`. Este enfoque es compatible con `addToBackStack`.

Para la gran mayoría de los casos de uso, solo necesitas `commit()`.

## El orden de las operaciones es significativo

El orden en el que realizas operaciones en una `FragmentManager` es significativo, en especial cuando se usa `setCustomAnimations()`. Este método aplica las animaciones determinadas a todas las operaciones de fragmentos que la siguen.

```
supportFragmentManager.commit {  
    setCustomAnimations(enter1, exit1, popEnter1, popExit1)  
    add<ExampleFragment>(R.id.container) // gets the first animations  
    setCustomAnimations(enter2, exit2, popEnter2, popExit2)  
    add<ExampleFragment>(R.id.container) // gets the second animations  
}
```

## Cómo limitar el ciclo de vida del fragmento

`FragmentTransactions` puede afectar el estado del ciclo de vida de los fragmentos individuales agregados dentro del alcance de la transacción. Cuando creas una `FragmentManager`, `setMaxLifecycle()` establece un estado máximo para el fragmento determinado. Por ejemplo, `ViewPager2` usa `setMaxLifecycle()` a fin de limitar los fragmentos fuera de la pantalla al estado `STARTED`.

## Cómo mostrar y ocultar vistas de fragmentos

Usa los métodos `show()` y `hide()` de `FragmentManager` para mostrar y ocultar la vista de fragmentos agregados a un contenedor. Estos métodos establecen la visibilidad de las vistas del fragmento *sin* afectar el ciclo de vida de este.

Si bien no necesitas usar una transacción de fragmento para activar o desactivar la visibilidad de las vistas dentro de un fragmento, estos métodos resultan útiles en casos en los que quieres que se asocien los cambios del estado de visibilidad con las transacciones en la pila de actividades.

## Cómo conectar y desconectar fragmentos

El método `detach()` de `FragmentManager` desvincula el fragmento de la IU, lo que destruye su jerarquía de vistas. El fragmento permanece en el mismo estado (`STOPPED`) que tenía cuando se coloca en la pila de actividades. Esto significa que se quitó el fragmento de la IU, pero aún lo administra el administrador de fragmentos.

El método `attach()` vuelve a conectar un fragmento que se había desconectado con anterioridad. Esto hace que su jerarquía de vistas se vuelva a crear, se conecte a la IU y se muestre.

Dado que una `FragmentTransaction` se trata como un conjunto único de operaciones atómicas, las llamadas a `detach` y `attach` en la misma instancia del fragmento y en la misma transacción se cancelan entre sí. De esta manera, evitan la destrucción de la IU del fragmento y su inmediata recreación. Si deseas desconectar y, luego, reconectar un fragmento, usa transacciones independientes, separadas por `executePendingOperations()` si usas `commit()`.

★ **Nota:** Los métodos `attach()` y `detach()` no están relacionados con los métodos `onAttach()` y `onDetach()` de `Fragment`. Para obtener más información sobre estos métodos de `Fragment`, consulta el [Ciclo de vida de los fragmentos](#).

## Cómo navegar entre fragmentos con animaciones

La API de `Fragment` proporciona dos formas de usar efectos de movimiento y transformaciones para conectar fragmentos visualmente durante la navegación. Una de ellas es el framework de animación, que usa tanto `Animation` como `Animator`. La otra es el framework de transición, que incluye transiciones de elementos compartidos.

★ **Nota:** En este tema, usamos el término *animación* para describir los efectos del framework de animación, y el término *transición* para describir los del framework de transición. Ambos frameworks son mutuamente excluyentes y no deben usarse al mismo tiempo.

Puedes especificar efectos personalizados para los fragmentos que entran y los que salen, así como para las transiciones de elementos compartidos entre fragmentos.

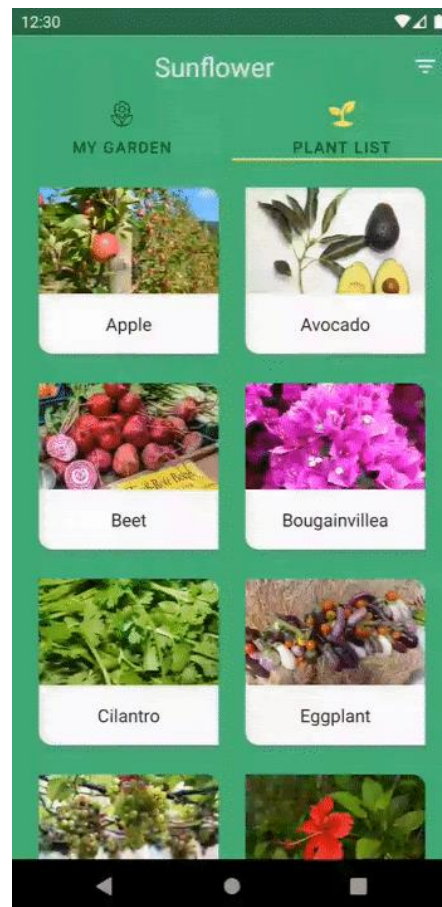
- Un efecto de *entrada* determina cómo un fragmento ingresa a la pantalla. Por ejemplo, puedes crear un efecto para deslizar el fragmento desde el borde de la pantalla cuando navegas a él.
- Un efecto de *salida* determina cómo un fragmento sale de la pantalla. Por ejemplo, puedes crear un efecto para atenuar el fragmento cuando salgas de él.
- Una *transición de elementos compartidos* determina cómo una vista que se comparte entre dos fragmentos se mueve entre ellos. Por ejemplo, una imagen que se muestra en una `ImageView` en el fragmento A transiciona al fragmento B una vez que B se vuelve visible.

## Cómo configurar animaciones

Primero, debes crear animaciones para tus efectos de entrada y salida, que se ejecutan cuando navegas a un fragmento nuevo. Puedes definir animaciones como recursos de animación de interpolación. Estos recursos te permiten definir la manera en la que los fragmentos deberán rotar, estirarse, desaparecer en forma gradual y moverse durante la animación. Por ejemplo, tal vez quieras que el fragmento actual desaparezca en forma gradual y el nuevo fragmento se deslice desde el borde derecho de la pantalla, como se muestra en la Figura 1.

**Figura 1:** Animaciones de entrada y de salida. El fragmento actual desaparece en forma gradual mientras que el fragmento siguiente se desliza desde la derecha.

Estas animaciones se pueden definir en el directorio `res/anim`:



```
<!-- res/anim/fade_out.xml -->
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="@android:integer/config_shortAnimTime"
    android:interpolator="@android:anim/decelerate_interpolator"
    android:fromAlpha="1"
    android:toAlpha="0" />
```

```
<!-- res/anim/slide_in.xml -->
<?xml version="1.0" encoding="utf-8"?>
<translate xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="@android:integer/config_shortAnimTime"
    android:interpolator="@android:anim/decelerate_interpolator"
    android:fromXDelta="100%"
    android:toXDelta="0%" />
```

★ **Nota:** Te recomendamos que uses transiciones para los efectos que involucran más de un tipo de animación, ya que existen problemas conocidos con el uso de instancias de AnimationSet anidadas.

También puedes especificar las animaciones de los efectos de entrada y de salida que se ejecutan al mostrar la pila de actividades, lo que puede suceder cuando el usuario presiona los botones Arriba o Atrás. Estas son las animaciones `popEnter` y `popExit`. Por ejemplo, cuando un usuario regresa a una pantalla anterior, quizás quieras que el fragmento actual se deslice fuera del borde derecho de la pantalla y que el fragmento anterior aparezca en forma gradual.

**Figura 2:** Animaciones `popEnter` y `popExit`. El fragmento actual se desliza fuera de la pantalla hacia la derecha mientras que el fragmento anterior aparece en forma gradual

Estas animaciones se pueden definir de la siguiente manera:





```
<!-- res/anim/slide_out.xml -->
<translate xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="@android:integer/config_shortAnimTime"
    android:interpolator="@android:anim/decelerate_interpolator"
    android:fromXDelta="0%"
    android:toXDelta="100%" />
```

```
<!-- res/anim/fade_in.xml -->
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="@android:integer/config_shortAnimTime"
    android:interpolator="@android:anim/decelerate_interpolator"
    android:fromAlpha="0"
    android:toAlpha="1" />
```

Una vez que hayas definido tus animaciones, utilízalas llamando a `FragmentManager.setCustomAnimations()` y pasando los recursos de animación por su ID de recurso, como se muestra en el siguiente ejemplo:

```
val fragment = FragmentB()
supportFragmentManager.commit {
    setCustomAnimations(
        enter = R.anim.slide_in,
        exit = R.anim.fade_out,
        popEnter = R.anim.fade_in,
        popExit = R.anim.slide_out
    )
    replace(R.id.fragment_container, fragment)
    addToBackStack(null)
}
```

★ **Nota:** `FragmentManager.setCustomAnimations()` aplica las animaciones personalizadas a todas las operaciones futuras de fragmentos en la `FragmentManager`. Las operaciones anteriores de la transacción no se ven afectadas.

## Cómo establecer transiciones

También puedes usar transiciones para definir los efectos de entrada y de salida. Estas transiciones se pueden definir en archivos de recursos XML. Por ejemplo, quizás desees que el fragmento actual desaparezca en forma gradual y que el nuevo fragmento se deslice desde el borde derecho de la pantalla. Estas transiciones se pueden definir de la siguiente manera:

```
<!-- res/transition/fade.xml -->
<fade xmlns:android="http://schemas.android.com/apk/res/android"
      android:duration="@android:integer/config_shortAnimTime"/>
```

```
<!-- res/transition/slide_right.xml -->
<slide xmlns:android="http://schemas.android.com/apk/res/android"
       android:duration="@android:integer/config_shortAnimTime"
       android:slideEdge="right" />
```

Una vez que hayas definido tus transiciones, aplícalas llamando a `setEnterTransition()` en el fragmento que entra y a `setExitTransition()` en el que sale, y pasando los recursos de transición aumentados por su ID de recurso, como se muestra en el siguiente ejemplo:

```
class FragmentA : Fragment() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val inflater = TransitionInflater.from(requireContext())
        exitTransition = inflater.inflateTransition(R.transition.fade)
    }
}

class FragmentB : Fragment() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val inflater = TransitionInflater.from(requireContext())
        enterTransition = inflater.inflateTransition(R.transition.slide_right)
    }
}
```

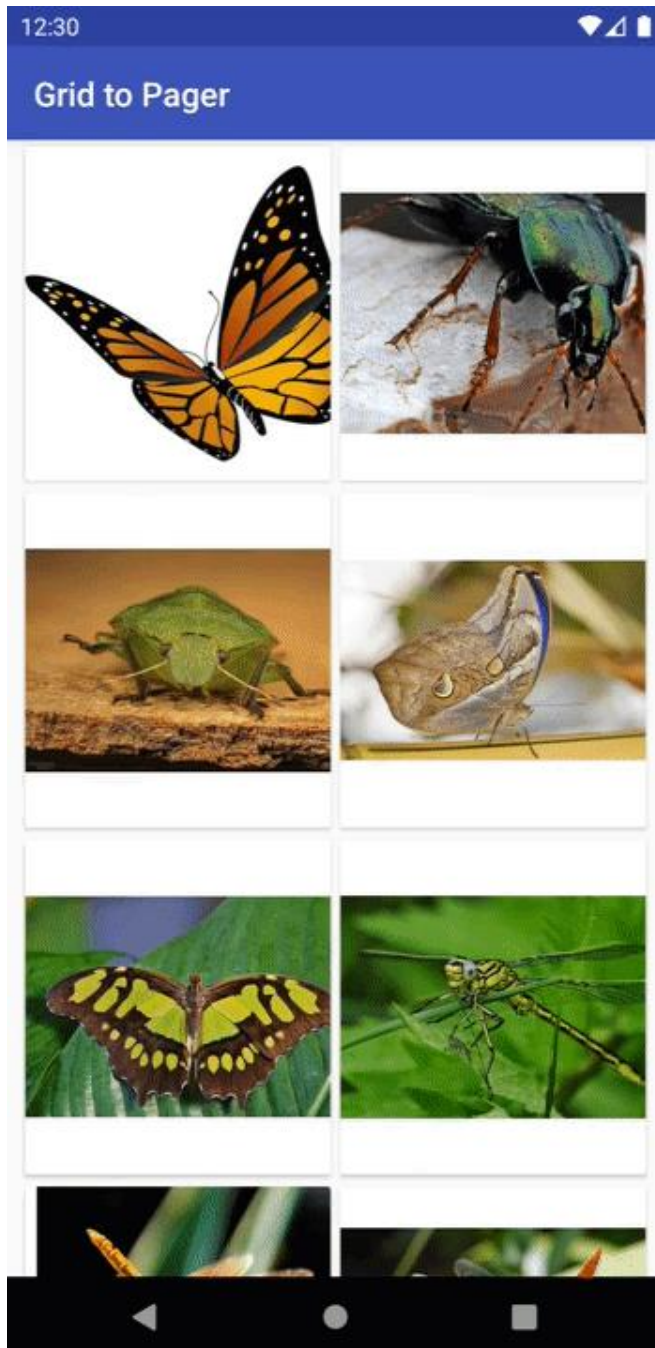
Los fragmentos admiten las transiciones de AndroidX. Si bien los fragmentos también admiten transiciones de framework, te recomendamos que uses las transiciones de AndroidX, ya que son compatibles con el nivel de API 14 y versiones posteriores, y contienen correcciones de errores que no estaban presentes en las versiones anteriores de las transiciones del framework.

## Cómo usar transiciones de elementos compartidos

Las transiciones de elementos compartidos forman parte del framework de transición y determinan cómo se mueven las vistas correspondientes entre dos fragmentos durante una transición de fragmentos. Por ejemplo, tal vez desees que una imagen que se muestra en una `ImageView` del fragmento A transicione al fragmento B una vez que B se vuelva visible, como se muestra en la Figura 3.

**Figura 3:** Transición de fragmentos con un elemento compartido

A grandes rasgos, aquí te explicamos cómo realizar una transición de fragmentos con elementos compartidos:



1. Asigna un nombre de transición único para cada vista de elementos compartidos.
2. Agrega vistas de elementos compartidos y nombres de transición a la `FragmentManager`.
3. Establece una animación de transición de elementos compartidos.

En primer lugar, debes asignar un nombre de transición único a cada vista de elementos compartidos para permitir que las vistas se mapeen de un fragmento al siguiente. Establece un nombre de transición en los elementos compartidos de cada diseño de fragmento mediante `ViewCompat.setTransitionName()`, que proporciona compatibilidad con el nivel de API 14 y versiones posteriores. A modo de ejemplo, el nombre de transición de una `ImageView` en los fragmentos A y B se puede asignar de la siguiente manera:

```

class FragmentA : Fragment() {
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        ...
        val itemImageView = view.findViewById<ImageView>(R.id.item_image)
        ViewCompat.setTransitionName(itemImageView, "item_image")
    }
}

class FragmentB : Fragment() {
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        ...
        val heroImageView = view.findViewById<ImageView>(R.id.hero_image)
        ViewCompat.setTransitionName(heroImageView, "hero_image")
    }
}

```

★ **Nota:** Para las apps que solo admiten el nivel de API 21 y versiones posteriores, puedes usar el atributo `android:transitionName` dentro de tu diseño XML a fin de asignar nombres de transición a vistas específicas.

Para incluir tus elementos compartidos en la transición de fragmento, `FragmentManager` debe conocer la manera en que las vistas de cada elemento compartido se mapean de un fragmento al siguiente. Agrega todos tus elementos compartidos a la `FragmentManager` llamando a `FragmentManager.addSharedElement()` y pasando la vista y el nombre de transición de la vista correspondiente del fragmento siguiente, como se muestra en el ejemplo a continuación:

```

val fragment = FragmentB()
supportFragmentManager.commit {
    setCustomAnimations(...)
    addSharedElement(itemImageView, "hero_image")
    replace(R.id.fragment_container, fragment)
    addToBackStack(null)
}

```

Para especificar cómo será la transición de tus elementos compartidos de un fragmento al siguiente, debes configurar una transición de *entrada* en el fragmento hasta el que se va a navegar. Llama a `Fragment.setSharedElementEnterTransition()` en el método `onCreate()` del fragmento, como se muestra en el siguiente ejemplo:

```
class FragmentB : Fragment() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        sharedElementEnterTransition = TransitionInflater.from(requireContext())
            .inflateTransition(R.transition.shared_image)
    }
}
```

La transición de `shared_image` se define de la siguiente manera:

```
<!-- res/transition/shared_image.xml -->
<transitionSet>
    <changeImageTransform />
</transitionSet>
```

Todas las subclases de `Transition` se admiten como transiciones de elementos compartidos. Si deseas crear una `Transition` personalizada, consulta [Cómo crear una animación de transición personalizada](#). `changeImageTransform`, que se usa en el ejemplo anterior, es una de las traslaciones precompiladas disponibles que puedes usar. Puedes encontrar subclases adicionales de `Transition` en la referencia de la API de la clase `Transition`.

De forma predeterminada, la transición de entrada del elemento compartido también se usa como la transición de *retorno* para los elementos compartidos. La transición de retorno determina cómo los elementos compartidos vuelven al fragmento anterior cuando se quita la transacción de la pila de actividades. Si quieres especificar una transición de retorno diferente, puedes usar `Fragment.setSharedElementReturnTransition()` en el método `onCreate()` del fragmento.

## Cómo posponer transiciones

En algunos casos, es posible que debas posponer la transición de tu fragmento durante un breve período. Por ejemplo, es posible que debas esperar hasta que se midan y se muestren todas las vistas en el fragmento que entra para que Android pueda capturar con precisión sus estados de inicio y fin para la transición.

Además, tal vez se deba posponer tu transición hasta que se hayan cargado algunos datos necesarios. Por ejemplo, quizás debas esperar hasta que las imágenes se hayan cargado para los elementos compartidos. De lo contrario, la transición puede resultar molesta si una imagen termina de cargarse durante la transición o después de ella.

A fin de posponer una transición, primero debes asegurarte de que la transacción del fragmento permita reordenar los cambios de estado de los fragmentos. Para permitir el reordenamiento de cambios de estado de fragmentos, llama a `FragmentManager.setReorderingAllowed()`, como se muestra en el siguiente ejemplo:

```
val fragment = FragmentB()
supportFragmentManager.commit {
    setReorderingAllowed(true)
    setCustomAnimation(...)
    addSharedElement(view, view.transitionName)
    replace(R.id.fragment_container, fragment)
    addToBackStack(null)
}
```

Para posponer la transición de entrada, llama a `Fragment.postponeEnterTransition()` en el método `onViewCreated()` del fragmento que ingresa:

```
class FragmentB : Fragment() {
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        ...
        postponeEnterTransition()
    }
}
```

Una vez que hayas cargado los datos y estés listo para comenzar la transición, llama a `Fragment.startPostponedEnterTransition()`. En el siguiente ejemplo, se usa la biblioteca de Glide a fin de cargar una imagen en una `ImageView` compartida, lo que pospone la transición correspondiente hasta que se complete la carga de las imágenes.

```
class FragmentB : Fragment() {
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        ...
        Glide.with(this)
            .load(url)
            .listener(object : RequestListener<Drawable> {
                override fun onLoadFailed(...): Boolean {
                    startPostponedEnterTransition()
                    return false
                }

                override fun onResourceReady(...): Boolean {
                    startPostponedEnterTransition()
                    return false
                }
            })
            .into(headerImage)
    }
}
```

Cuando la conexión a Internet de un usuario sea lenta, es posible que necesites que la transición pospuesta comience después de un período determinado en lugar de esperar a que se carguen todos los datos. En estos casos, puedes llamar a `Fragment.postponeEnterTransition(long, TimeUnit)` en el método `onViewCreated()` del fragmento que ingresa y pasar la duración y la unidad de tiempo. La transición pospuesta se iniciará automáticamente una vez transcurrido el tiempo especificado.

## Cómo usar transiciones de elementos compartidos con una RecyclerView

Las transiciones de entrada pospuestas no deben comenzar hasta que se hayan medido y mostrado todas las vistas del fragmento que ingresa. Cuando usas una RecyclerView, debes esperar a que se carguen los datos y que los elementos de RecyclerView estén listos para dibujar antes de iniciar la transición. Por ejemplo:

```
class FragmentA : Fragment() {
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        postponeEnterTransition()

        // Wait for the data to load
        viewModel.data.observe(viewLifecycleOwner) {
            // Set the data on the RecyclerView adapter
            adapter.setData(it)
            // Start the transition once all views have been
            // measured and laid out
            (view.parent as? ViewGroup)?.doOnPreDraw {
                startPostponedEnterTransition()
            }
        }
    }
}
```

Observa que se establece un `ViewTreeObserver.OnPreDrawListener` en el elemento superior de la vista del fragmento. Esto ocurre a los efectos de garantizar que se hayan medido y mostrado todas las vistas del fragmento que, por lo tanto, están listas para dibujarse antes de iniciar la transición de entrada pospuesta.

★ **Nota:** Cuando usas una transición de elementos compartidos de un fragmento que usa una **RecyclerView** a otro fragmento, **debes** seguir posponiendo el fragmento con una **RecyclerView** para asegurarte de que la transición de retorno de elementos compartidos funcione de manera correcta cuando regreses a la **RecyclerView**.

Otro punto que debes tener en cuenta cuando usas transiciones de elementos compartidos con una RecyclerView es que no puedes establecer el nombre de transición en el diseño XML del elemento de la RecyclerView porque una cantidad arbitraria de elementos comparten ese diseño. Se debe asignar un nombre de transición único de modo que la animación de transición use la vista correcta.

Puedes asignar un nombre de transición único a cada elemento compartido asignándolos cuando se vincule el ViewHolder. Por ejemplo, si los datos de cada elemento incluyen un ID único, se lo podría usar como nombre de transición, como se muestra en el siguiente ejemplo:



```

class ExampleViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    val image = itemView.findViewById<ImageView>(R.id.item_image)

    fun bind(id: String) {
        ViewCompat.setTransitionName(image, id)
        ...
    }
}

```

## Ciclo de vida de los fragmentos

Cada instancia de Fragment tiene su propio ciclo de vida. Cuando un usuario navega por tu app e interactúa con ella, los fragmentos pasan por varios estados durante su ciclo de vida a medida que se agregan, se quitan e ingresan a la pantalla o salen de ella.

Para administrar el ciclo de vida, Fragment implementa LifecycleOwner, lo que expone un objeto Lifecycle al que puedes acceder a través del método `getLifecycle()`.

En la enumeración `Lifecycle.State`, se representa cada estado posible de Lifecycle.

- INITIALIZED
- CREATED
- STARTED
- RESUMED
- DESTROYED

Si compilas Fragment sobre Lifecycle, puedes usar las técnicas y clases disponibles para administrar ciclos de vida con componentes optimizados para estos. Uno de estos componentes te permite mostrar la ubicación del dispositivo en la pantalla. Ese componente podría comenzar a escuchar automáticamente cuando el fragmento se active y detenerse cuando este pase a un estado inactivo.

Como alternativa al uso de un LifecycleObserver, la clase de Fragment incluye métodos de devolución de llamada que corresponden a cada uno de los cambios en el ciclo de vida de un fragmento. Se incluyen los siguientes: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()` y `onDestroy()`.

La vista de un fragmento tiene un Lifecycle separado que se administra de forma independiente del Lifecycle del fragmento. Los fragmentos mantienen un LifecycleOwner para su vista, al que se puede acceder con `getViewLifecycleOwner()` o `getViewLifecycleOwnerLiveData()`. Tener acceso al Lifecycle de la vista es útil para situaciones en las que un componente optimizado para ciclos de vida solo deba realizar trabajos mientras exista la vista de un fragmento, como observar LiveData que solo debe mostrarse en la pantalla.

En este tema, se analiza en detalle el ciclo de vida de `Fragment`, se explican algunas de las reglas que determinan el estado del ciclo de vida de un fragmento y se muestra la relación entre los estados de Lifecycle y las devoluciones de llamada de ciclo de vida de los fragmentos.

## Los fragmentos y el administrador de fragmentos

Cuando se crea una instancia de un fragmento, este comienza en el estado `INITIALIZED`. Para que un fragmento pase por el resto de su ciclo de vida, se lo debe agregar a un `FragmentManager`. El `FragmentManager` se encarga de determinar el estado que debe tener su fragmento y, luego, de pasarlo a ese estado.

Más allá del ciclo de vida del fragmento, `FragmentManager` también se encarga de adjuntar fragmentos a su actividad de host y de separarlos cuando el fragmento ya no esté en uso. La clase `Fragment` tiene dos métodos de devolución de llamada, `onAttach()` y `onDetach()`, que puedes anular para realizar trabajos cuando se produce cualquiera de estos eventos.

Cuando se agrega el fragmento a un `FragmentManager`, se invoca la devolución de llamada `onAttach()` y se la adjunta a su actividad de host. En este punto, el fragmento está activo y el `FragmentManager` administra su estado de ciclo de vida. Asimismo, los métodos de `FragmentManager` como `findFragmentById()` muestran este fragmento.

Siempre se llama a `onAttach()` antes de cualquier cambio de estado de ciclo de vida.

Cuando se quita el fragmento de un `FragmentManager`, se invoca la devolución de llamada `onDetach()` y se la desconecta de su actividad de host. El fragmento ya no está activo y ya no se puede recuperar con `findFragmentById()`.

Siempre se llama a `onDetach()` después de cualquier cambio de estado de ciclo de vida.

Ten en cuenta que estas devoluciones de llamada no están relacionadas con los métodos `attach()` ni `detach()` de `FragmentTransaction`. Para obtener más información sobre estos métodos, consulta Transacciones de fragmentos.




**Precaución:** Evita reutilizar instancias de `Fragment` después de quitarlas del `FragmentManager`. Si bien el fragmento controla su propia limpieza de estado interna, podrías pasar de manera involuntaria tu estado a la instancia reutilizada.

## Estados del ciclo de vida de los fragmentos y devoluciones de llamada

A la hora de determinar el estado del ciclo de vida de un fragmento, `FragmentManager` tiene en cuenta lo siguiente:

- `FragmentManager` determina el estado máximo de un fragmento. Un fragmento no puede avanzar más allá del estado de su `FragmentManager`.
- Como parte de una `FragmentTransaction`, puedes configurar un estado de ciclo de vida máximo para un fragmento con `setMaxLifecycle()`.
- El estado de ciclo de vida de un fragmento nunca puede ser mayor que su elemento superior. Por ejemplo, una actividad o un fragmento superior deben iniciarse antes que sus fragmentos secundarios. Del mismo modo, los fragmentos secundarios deben detenerse antes de que lo hagan sus fragmentos o actividades superiores.

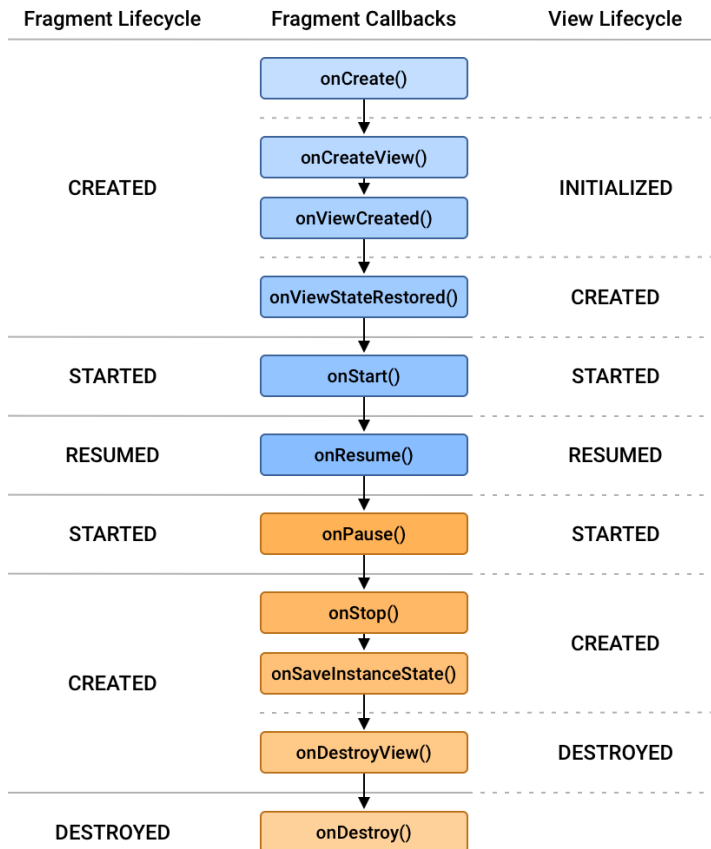
 **Precaución:** Evita el uso de la etiqueta `<fragment>` cuando agregues un fragmento por medio de XML, ya que la etiqueta `<fragment>` permite que un fragmento se mueva más allá del estado de su `FragmentManager`. En su lugar, siempre usa `FragmentContainerView`.

En la Figura 1, se muestra cada uno de los estados de Lifecycle del fragmento y la forma en la que se relacionan con las devoluciones de llamada de su ciclo de vida y con el Lifecycle de su vista.

A medida que un fragmento avanza por su ciclo de vida, se mueve de manera ascendente y descendente por sus estados. Por ejemplo, un fragmento que se agrega a la parte superior de la pila de actividades se mueve de manera ascendente de `CREATED` a `STARTED` y a `RESUMED`. En cambio, cuando un fragmento se quita de la pila de actividades, se mueve de manera descendente por esos estados, de `RESUMED` a `STARTED`, luego a `CREATED` y, por último, a `DESTROYED`.

### Transiciones de estado ascendentes

Cuando un fragmento se mueve por los estados de su ciclo de vida, primero llama a la devolución de llamada de ciclo de vida asociada a su estado nuevo. Una vez que finaliza la devolución de llamada, se transmite el `Lifecycle.Event` correspondiente a los observadores por el Lifecycle del fragmento, seguido del Lifecycle de la vista del fragmento, si se creó su instancia.



**Figura 1:** Estados de Lifecycle del fragmento y su relación tanto con las devoluciones de llamada del ciclo de vida del fragmento como con el Lifecycle de la vista del fragmento.

## Fragmento CREADO

Cuando tu fragmento alcanza el estado CREATED, significa que se lo agregó a un `FragmentManager` y ya se llamó al método `onAttach()`.

Este sería el lugar adecuado para restablecer el estado guardado asociado con el fragmento a través de su `SavedStateRegistry`. Ten en cuenta que la vista del fragmento aún *no* se creó, y cualquier estado asociado con la vista del fragmento debe restablecerse solo después de que se haya creado la vista.

Esta transición invoca la devolución de llamada `onCreate()`. Esta también recibe un argumento `savedInstanceState Bundle` que contiene cualquier estado guardado con anterioridad por `onSaveInstanceState()`. Ten en cuenta que `savedInstanceState` tiene un valor `null` la primera vez que se crea el fragmento, pero siempre es no nulo para las recreaciones posteriores, incluso si no anulas `onSaveInstanceState()`. Consulta [Cómo guardar estados con fragmentos](#) para obtener más detalles.

## Fragmento CREADO y vista INICIALIZADA

El Lifecycle de la vista del fragmento solo se crea cuando tu Fragment proporciona una instancia válida de View. En la mayoría de los casos, puedes usar los constructores de fragmentos que toman un `@LayoutId`, lo que aumenta automáticamente la vista en el momento apropiado. También puedes anular `onCreateView()` para aumentar o crear de manera programática la vista del fragmento.

Siempre y cuando se cree una instancia de la vista del fragmento con una View no nula, esa View se establece en el fragmento y se puede recuperar con `getView()`. Luego, el `getViewLifecycleOwnerLiveData()` se actualiza con el LifecycleOwner recientemente INITIALIZED, que corresponde a la vista del fragmento. La devolución de llamada del ciclo de vida de `onViewCreated()` también se llama en ese momento.

Este es el lugar adecuado para configurar el estado inicial de la vista a fin de comenzar a observar instancias de LiveData, cuyas devoluciones de llamada actualizan la vista del fragmento, y también para configurar los adaptadores en cualquier instancia de RecyclerView o ViewPager2, en la vista del fragmento.

## Fragmento y vista CREADOS

Después de crear la vista del fragmento, se restablece el estado anterior de la vista, si existe, y el Lifecycle de la vista pasa al estado CREATED. El propietario del ciclo de vida de la vista también emite el evento `ON_CREATE` a sus observadores. Aquí debes restablecer cualquier estado adicional asociado con la vista del fragmento.

Esta transición también invoca la devolución de llamada `onViewStateRestored()`.

## Fragmento y vista COMENZADOS

Se recomienda que vincules los componentes optimizados para ciclos de vida al estado STARTED de un fragmento, ya que este estado garantiza que la vista del fragmento, si se creó alguna, esté disponible y que sea seguro realizar una `FragmentTransaction` en el `FragmentManager` secundario del fragmento. Si la vista del fragmento no es nula, el Lifecycle de la vista del fragmento pasa a STARTED inmediatamente después de que el Lifecycle del fragmento pase a STARTED.

Cuando el fragmento pasa a STARTED, se invoca la devolución de llamada `onStart()`.



**Nota:** Los componentes como [ViewPager2](#) establecen el Lifecycle máximo de los fragmentos fuera de la pantalla en STARTED.

## Fragmento y vista REANUDADOS

Cuando el fragmento sea visible, habrán finalizado todos los efectos de Animator y Transition, y el fragmento estará listo para la interacción del usuario. El Lifecycle del fragmento pasa al estado RESUMED y se invoca la devolución de llamada onResume().

La transición a RESUMED es la señal correspondiente que indica que el usuario ahora puede interactuar con el fragmento. Los fragmentos que no tengan el estado RESUMED no deben establecer manualmente el foco en sus vistas ni intentar controlar la visibilidad del método de entrada.

## Transiciones de estado descendentes

Cuando un fragmento se mueve hacia un estado inferior del ciclo de vida, el Lifecycle de la vista del fragmento, si se creó una instancia de esta, emite el Lifecycle.Event correspondiente a los observadores, seguido del Lifecycle del fragmento. Después de que se emite el evento del ciclo de vida de un fragmento, este último llama a la devolución de llamada de ciclo de vida asociada.

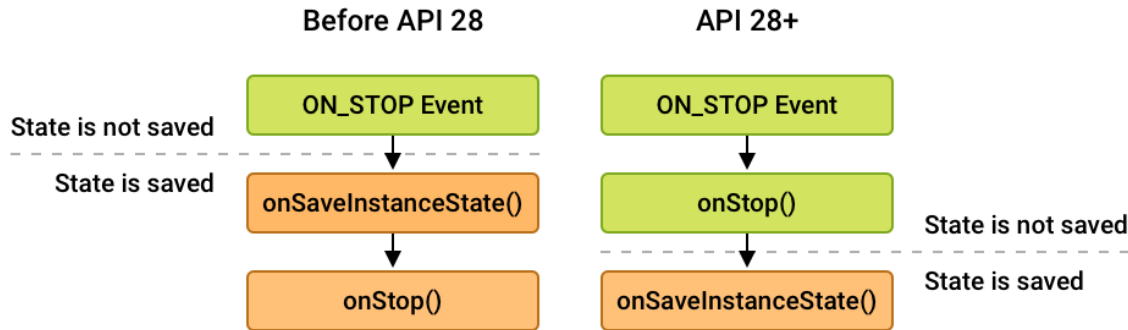
## Fragmento y vista COMENZADOS

Cuando el usuario comienza a dejar el fragmento y mientras este aún está visible, los Lifecycle del fragmento y de su vista vuelven a pasar al estado STARTED y a emitir el evento ON\_PAUSE a sus observadores. Luego, el fragmento invoca su devolución de llamada onPause().

## Fragmento y vista CREADOS

Una vez que el fragmento ya no esté visible, los Lifecycle del fragmento y de su vista pasarán al estado CREATED y emitirán el evento ON\_STOP a sus observadores. Esta transición de estado se activa no solo cuando se detiene la actividad o el fragmento superiores, sino también cuando alguno de ellos guarda un estado. Este comportamiento garantiza que el evento ON\_STOP se invoque antes de que se guarde el estado del fragmento. Esto hace que el evento ON\_STOP sea el último punto en el que es seguro realizar una FragmentTransaction en el FragmentManager secundario.

Como se muestra en la Figura 2, el orden de la devolución de llamada `onStop()` y el guardado del estado con `onSaveInstanceState()` difieren según el nivel de API. En todos los niveles de API anteriores a la API 28, `onSaveInstanceState()` se invoca antes de `onStop()`. En el caso de los niveles de API 28 y posteriores, el orden de llamada está invertido.



**Figura 2:** Diferencias en el orden de las llamadas para `onStop()` y `onSaveInstanceState()`.

## Fragmento CREADO y vista DESTRUIDA

Una vez que se completaron todas las animaciones y transiciones de salida y que la vista del fragmento se separó de la ventana, el Lifecycle de la vista del fragmento pasa al estado `DESTROYED` y emite el evento `ON_DESTROY` a sus observadores. Luego, el fragmento invoca su devolución de llamada `onDestroyView()`. En este punto, la vista del fragmento alcanzó el final de su ciclo de vida y `getViewLifecycleOwnerLiveData()` muestra un valor null.

Asimismo, se deben quitar todas las referencias a la vista del fragmento, lo que permitirá la recolección de elementos no utilizados.

## Fragmento DESTRUIDO

Si se quita el fragmento, o si se destruye el `FragmentManager`, el Lifecycle del fragmento pasa al estado `DESTROYED` y envía el evento `ON_DESTROY` a sus observadores. Luego, el fragmento invoca su devolución de llamada `onDestroy()`. En este punto, el fragmento alcanzó el final de su ciclo de vida.

## Cómo guardar un estado con fragmentos

Varias operaciones del sistema Android pueden afectar el estado de tu fragmento. Para garantizar que se guarde el estado del usuario, el framework de Android guarda y restablece automáticamente los fragmentos y la pila de actividades. Por lo tanto, debes asegurarte de guardar y restablecer también todos los datos de tu fragmento.

En la siguiente tabla, se describen las operaciones que hacen que el fragmento pierda su estado, junto con la posibilidad de que los diferentes tipos de estado se conserven a través de esos cambios. Los tipos de estado que se mencionan en la tabla son los siguientes:

- Variables: Variables locales en el fragmento
- View State: Cualquier dato **que pertenezca a una o más vistas** del fragmento
- SavedState: Datos inherentes a esta instancia del fragmento que se deben guardar en onSaveInstanceState()
- NonConfig: Datos extraídos de una fuente externa, como un servidor o repositorio local, o datos creados por el usuario que se envían al servidor una vez confirmados

A menudo, las *Variables* se tratan de la misma manera que el *SavedState*, pero la tabla siguiente distingue entre los dos para mostrar el efecto de las distintas operaciones en cada uno.

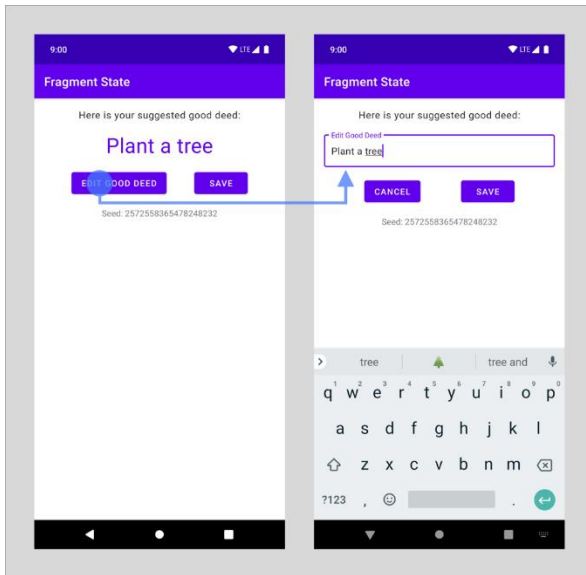
Operación	Variables	View State	SavedState	NonConfig
Agregado a la pila de actividades	✓	✓	x	✓
Cambio de configuración	x	✓	✓	✓
Cierre o recreación del proceso	x	✓	✓	✓*
Quitado, no agregado a la pila de actividades	x	x	x	x
Host terminado	x	x	x	x

\* Se puede retener el estado NonConfig luego del cierre del proceso mediante el módulo de estado guardado para ViewModel.

**Tabla 1:** Varias operaciones destructivas de fragmentos y los efectos que tienen en diferentes tipos de estados

Veamos un ejemplo específico. Considera una pantalla que genera una string aleatoria, la muestra en una TextView y proporciona la opción de editar la string antes de enviarla a un amigo:





**Figura 1:** App de generación de textos aleatorios que demuestra varios tipos de estado

Para este ejemplo, supongamos que una vez que el usuario presiona el botón de edición, la app muestra una vista de `EditText` en la que el usuario puede editar el mensaje. Si este hace clic en **CANCELAR**, la vista de `EditText` se debe borrar y su visibilidad se debe establecer en `View.GONE`. Tal pantalla podría requerir la administración de cuatro datos para garantizar una experiencia sin interrupciones:

Datos	Tipo	Tipo de estado	Descripción
<code>seed</code>	<code>Long</code>	<code>NonConfig</code>	Seed utilizada para generar un nuevo random good deed. Generada cuando se crea el <code>ViewModel</code> .
<code>randomGoodDeed</code>	<code>String</code>	<code>SavedState + Variable</code>	Generado cuando se crea el fragmento por primera vez. <code>randomGoodDeed</code> se guarda para garantizar que los usuarios vean el mismo random good deed, incluso después del cierre del proceso y de su recreación.
<code>isEditing</code>	<code>Boolean</code>	<code>SavedState + Variable</code>	Marca booleana establecida en <code>true</code> cuando el usuario comienza a editar. Se guarda <code>isEditing</code> para garantizar que la parte de edición de la pantalla permanezca visible cuando se vuelva a crear el fragmento.
Texto editado	<code>Editable</code>	View State (propiedad de <code>EditText</code> )	El texto editado en la vista de <code>EditText</code> . La vista <code>EditText</code> guarda este texto para garantizar que no se pierdan los cambios en curso del usuario.

**Tabla 2:** Estados que debe administrar la app de generación de textos aleatorios

En las siguientes secciones, se describe cómo administrar adecuadamente el estado de tus datos mediante operaciones destructivas.

## Estado de vistas

Las vistas se encargan de administrar su propio estado. Por ejemplo, cuando una vista acepta la entrada del usuario, es su responsabilidad guardar y restablecer esa entrada para controlar los cambios de configuración. Todas las vistas proporcionadas por el framework de Android tienen su propia implementación de `onSaveInstanceState()` y `onRestoreInstanceState()`, por lo que no necesitas administrar el estado de la vista en tu fragmento.

★ **Nota:** A fin de garantizar un manejo adecuado durante los cambios de configuración, debes implementar `onSaveInstanceState()` y `onRestoreInstanceState()` en las vistas personalizadas que crees.

Por ejemplo, en la situación anterior, la string editada se conserva en un `EditText`. Un `EditText` reconoce el valor del texto que muestra, así como otros detalles, como el principio y el final del texto seleccionado.

Una vista necesita un ID para retener su estado. Ese ID debe ser único dentro del fragmento y de su jerarquía de vistas. **Las vistas que no tienen un ID no pueden retener su estado.**

```
<EditText
    android:id="@+id/good_deed_edit_text"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Como se mencionó en la Tabla 1, las vistas guardan y restablecen su `ViewState` a través de todas las operaciones que no quitan el fragmento ni destruyen el host.

Tu fragmento se encarga de administrar pequeñas cantidades de estado dinámico que son fundamentales para su funcionamiento. Puedes retener datos serializados con facilidad mediante `Fragment.onSaveInstanceState(Bundle)`. Similar a `Activity.onSaveInstanceState(Bundle)`, los datos que colocas en el paquete se retienen a través de cambios de configuración y del cierre y la recreación de los procesos, y están disponibles en los métodos `onCreate(Bundle)`, `onCreateView(LayoutInflater, ViewGroup`

❗ **Precaución:** Solo se llama a `onSaveInstanceState(Bundle)` cuando la actividad host del fragmento llama a su propio `onSaveInstanceState(Bundle)`.

★ **Sugerencia:** Cuando usas un `ViewModel`, puedes guardar el estado directamente dentro de `ViewModel` con un `SavedStateHandle`. Para obtener más información, consulta el [módulo de estado guardado para ViewModel](#).

Siguiendo con el ejemplo anterior, `randomGoodDeed` es la acción que se muestra al usuario y `isEditing` es una marca para determinar si el fragmento muestra u oculta el `EditText`. Este estado guardado debe conservarse por medio de `onSaveInstanceState(Bundle)`, como se muestra en el siguiente ejemplo:

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putBoolean(IS_EDITING_KEY, isEditing)
    outState.putString(RANDOM_GOOD_DEED_KEY, randomGoodDeed)
}
```

Para restablecer el estado de `onCreate(Bundle)`, recupera el valor almacenado del paquete:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    isEditing = savedInstanceState?.getBoolean(IS_EDITING_KEY, false)
    randomGoodDeed = savedInstanceState?.getString(RANDOM_GOOD_DEED_KEY)
        ?: viewModel.generateRandomGoodDeed()
}
```

Como se mencionó en la Tabla 1, ten en cuenta que las variables se retienen cuando el fragmento se coloca en la pila de actividades. Tratarlas como estados guardados garantiza que se conserven a lo largo de todas las operaciones destructivas.

## NonConfig

Los datos de `NonConfig` deben colocarse fuera de tu fragmento, como en un `ViewModel`. En el ejemplo anterior, `seed` (nuestro estado de `NonConfig`) se genera en el `ViewModel`. La lógica para mantener su estado pertenece al `ViewModel`.

```
public class RandomGoodDeedViewModel : ViewModel() {
    private val seed = ... // Generate the seed

    private fun generateRandomGoodDeed(): String {
        val goodDeed = ... // Generate a random good deed using the seed
        return goodDeed
    }
}
```

La clase `ViewModel` permite de forma inherente que los datos sobrevivan a los cambios de configuración, como las rotaciones de pantalla, y que permanezcan en la memoria cuando el fragmento se coloca en la pila de actividades. Después del cierre y de la recreación del proceso, se recrea el `ViewModel` y se genera una nueva `seed`. Agregar un módulo de `SavedState` a tu `ViewModel` permite que el `ViewModel` retenga un estado simple a través del cierre y la recreación del proceso.

## Cómo comunicarse con fragmentos

Para reutilizar fragmentos, compílalos como componentes completamente independientes que definan su propio diseño y comportamiento. Una vez que hayas definido esos fragmentos reutilizables, podrás asociarlos con una actividad y conectarlos a la lógica de la aplicación para comprender la IU completa de la composición.

Para reaccionar de manera adecuada ante los eventos del usuario o compartir información de estado, a menudo, necesitas tener canales de comunicación entre una actividad y sus fragmentos, o entre dos o más fragmentos. Para que los fragmentos sigan siendo independientes, *no hagas* que los fragmentos se comuniquen directamente con otros fragmentos ni con su actividad de host.

La biblioteca de Fragment proporciona dos opciones para la comunicación: un ViewModel compartido y la API de resultados de fragmentos. La opción recomendada depende del caso de uso. Para compartir datos persistentes con las APIs personalizadas, usa un ViewModel. A fin de obtener un resultado único con los datos que se pueden colocar en un Bundle, debes usar la API de resultados de fragmentos.

En las siguientes secciones, se muestra cómo usar ViewModel y la API de Fragment Result a efectos de establecer una comunicación entre tus fragmentos y actividades.

## Cómo compartir datos mediante un ViewModel

ViewModel es una opción ideal cuando necesitas compartir datos entre varios fragmentos o entre fragmentos y su actividad del host. Los objetos de ViewModel almacenan y administran datos de IU. Para obtener más información sobre ViewModel, consulta la sección Descripción general de ViewModel.

## Cómo compartir datos con la actividad del host

En algunos casos, quizás necesites compartir datos entre fragmentos y su actividad del host. Por ejemplo, es posible que quieras activar o desactivar un componente global de IU basado en una interacción dentro de un fragmento.

Ten en cuenta el siguiente ItemViewModel:

```
class ItemViewModel : ViewModel() {
    private val mutableSelectedItem = MutableLiveData<Item>()
    val selectedItem: LiveData<Item> get() = mutableSelectedItem

    fun selectItem(item: Item) {
        mutableSelectedItem.value = item
    }
}
```

En este ejemplo, los datos almacenados se unen en una clase `MutableLiveData`. `LivData` es una clase de contenedor de datos observables optimizada para ciclos de vida. `MutableLiveData` permite cambiar su valor. Para obtener más información sobre `LivData`, consulta la Descripción general de `LivData`.

Tanto el fragmento como su actividad de host pueden recuperar una instancia compartida de un `ViewModel` que tenga alcance en la actividad pasando la actividad al constructor `ViewModelProvider`. `ViewModelProvider` controla la creación de la instancia de `ViewModel` o su recuperación si aquella ya existe. Ambos componentes pueden observar y modificar estos datos.

```
class MainActivity : AppCompatActivity() {
    // Using the viewModels() Kotlin property delegate from the activity-ktx
    // artifact to retrieve the ViewModel in the activity scope.
    private val viewModel: ItemViewModel by viewModels()
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        viewModel.selectedItem.observe(this, Observer { item ->
            // Perform an action with the latest item data.
        })
    }
}

class ListFragment : Fragment() {
    // Using the activityViewModels() Kotlin property delegate from the
    // fragment-ktx artifact to retrieve the ViewModel in the activity scope.
    private val viewModel: ItemViewModel by activityViewModels()

    // Called when the item is clicked.
    fun onItemClick(item: Item) {
        // Set a new item.
        viewModel.selectItem(item)
    }
}
```

**! Precaución:** Usa el alcance adecuado con `ViewModelProvider`. En el ejemplo anterior, se usa `MainActivity` como alcance tanto en `MainActivity` como en `ListFragment`, por lo que a ambos se les proporciona el mismo `ViewModel`. Si, en cambio, `ListFragment` se usa como alcance, proporciona un `ViewModel` diferente a `MainActivity`.

## Cómo compartir los datos entre fragmentos

A menudo, dos o más fragmentos de la misma actividad necesitan comunicarse entre sí. Por ejemplo, imagina un fragmento que muestra una lista y otro que permite al usuario aplicar varios filtros a esa lista. La implementación de este caso no es trivial si los fragmentos no se comunican directamente, pero ya no son independientes. Además, los dos fragmentos deben procesar la situación en la que el otro fragmento todavía no se haya creado o no esté visible.

Esos fragmentos pueden compartir un `ViewModel` mediante su alcance de actividad para administrar la comunicación. Cuando se comparte `ViewModel` de esta manera, los fragmentos no necesitan

conocer información acerca del otro, y la actividad no necesita hacer nada para facilitar la comunicación.

En el siguiente ejemplo, se muestra cómo dos fragmentos pueden usar un ViewModel compartido para la comunicación:

```
class ListViewModel : ViewModel() {
    val filters = MutableLiveData<Set<Filter>>()

    private val originalList: LiveData<List<Item>>() = ...
    val filteredList: LiveData<List<Item>> = ...

    fun addFilter(filter: Filter) { ... }

    fun removeFilter(filter: Filter) { ... }
}

class ListFragment : Fragment() {
    // Using the activityViewModels() Kotlin property delegate from the
    // fragment-ktx artifact to retrieve the ViewModel in the activity scope.
    private val viewModel: ListViewModel by activityViewModels()
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        viewModel.filteredList.observe(viewLifecycleOwner, Observer { list ->
            // Update the list UI.
        })
    }
}

class FilterFragment : Fragment() {
    private val viewModel: ListViewModel by activityViewModels()
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        viewModel.filters.observe(viewLifecycleOwner, Observer { set ->
            // Update the selected filters UI.
        })
    }

    fun onFilterSelected(filter: Filter) = viewModel.addFilter(filter)

    fun onFilterDeselected(filter: Filter) = viewModel.removeFilter(filter)
}
```

Ten en cuenta que ambos fragmentos usan su actividad de host como alcance del ViewModelProvider. Debido a que los fragmentos usan el mismo alcance, reciben la misma instancia del ViewModel, lo que les permite comunicarse entre sí.

**! Precaución:** El **ViewModel** permanecerá en la memoria hasta que el **ViewModelStoreOwner** que determina su alcance desaparezca de forma permanente. En una arquitectura de actividad única, si el alcance de **ViewModel** está determinado por la actividad, en esencia, se trata de un singleton. Después de crear una instancia del **ViewModel**, las llamadas posteriores para recuperar el **ViewModel** mediante el alcance de la actividad siempre muestran el mismo **ViewModel** junto con los datos existentes hasta que el ciclo de vida de la actividad finalice de forma permanente.

## Cómo compartir datos entre un fragmento superior y uno secundario

Cuando se trabaja con fragmentos secundarios, es posible que el fragmento superior y los secundarios necesiten compartir datos entre sí. Para compartir datos entre estos fragmentos, usa el fragmento superior como alcance de ViewModel, como se muestra en el siguiente ejemplo:

```
class ListFragment: Fragment() {
    // Using the viewModels() Kotlin property delegate from the fragment-ktx
    // artifact to retrieve the ViewModel.
    private val viewModel: ListViewModel by viewModels()
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        viewModel.filteredList.observe(viewLifecycleOwner, Observer { list ->
            // Update the list UI.
        })
    }
}

class ChildFragment: Fragment() {
    // Using the viewModels() Kotlin property delegate from the fragment-ktx
    // artifact to retrieve the ViewModel using the parent fragment's scope
    private val viewModel: ListViewModel by viewModels({requireParentFragment()})
    ...
}
```

## Cómo definir el alcance de un ViewModel según el gráfico de Navigation

Si usas la biblioteca de Navigation, también puedes determinar el alcance de ViewModel en relación con el ciclo de vida de la NavBackStackEntry de destino. Por ejemplo, el alcance de ViewModel podría ser la NavBackStackEntry del ListFragment:

```
class ListFragment: Fragment() {
    // Using the navGraphViewModels() Kotlin property delegate from the fragment-ktx
    // artifact to retrieve the ViewModel using the NavBackStackEntry scope.
    // R.id.list_fragment == the destination id of the ListFragment destination
    private val viewModel: ListViewModel by navGraphViewModels(R.id.list_fragment)

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        viewModel.filteredList.observe(viewLifecycleOwner, Observer { item ->
            // Update the list UI.
        })
    }
}
```

Si quieres obtener más información para determinar el alcance de un ViewModel en relación con una NavBackStackEntry, consulta [Cómo interactuar de manera programática con el componente Navigation](#).

## Cómo obtener resultados con la API de resultados de fragmentos

En algunos casos, es posible que quieras pasar un valor por única vez entre dos fragmentos o entre un fragmento y su actividad de host. Por ejemplo, puedes tener un fragmento que lee códigos QR y pasa los datos a un fragmento anterior.

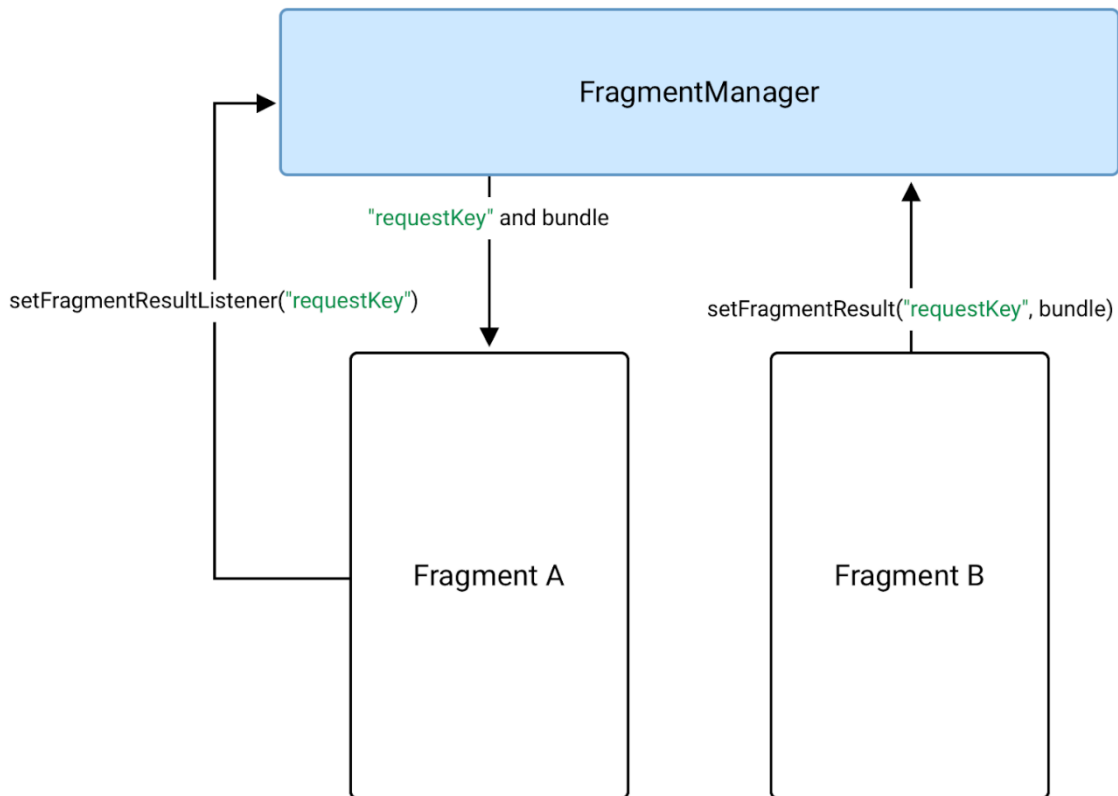
En la versión 1.3.0 de `Fragment` y versiones posteriores, cada `FragmentManager` implementa `FragmentManager.FragmentResultOwner`. Eso significa que un `FragmentManager` puede actuar como un almacenamiento central para los resultados de fragmentos. Este cambio permite que los componentes se comuniquen entre sí configurando los resultados de fragmentos y escuchando esos resultados sin que esos componentes tengan referencias directas entre sí.

## Cómo pasar resultados entre fragmentos

Para pasar datos al fragmento A desde el fragmento B, primero configura un objeto de escucha de resultados en el fragmento A (es decir, el que recibe el resultado). Llama a `setFragmentManagerResultListener()` en el `FragmentManager` del fragmento A, como se muestra en el siguiente ejemplo:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Use the Kotlin extension in the fragment-ktx artifact.
    setFragmentManagerResultListener("requestKey") { requestKey, bundle ->
        // We use a String here, but any type that can be put in a Bundle is supported.
        val result = bundle.getString("bundleKey")
        // Do something with the result.
    }
}
```





**Figura 1:** El fragmento B envía datos al fragmento A por medio de un FragmentManager.

En el fragmento B, que produce el resultado, debes establecer el resultado en el mismo FragmentManager con la misma requestKey. Puedes hacerlo con la API de setFragmentManager():

```
button.setOnClickListener {  
    val result = "result"  
    // Use the Kotlin extension in the fragment-ktx artifact.  
    setFragmentManager("requestKey", bundleOf("bundleKey" to result))  
}
```

Luego, el fragmento A recibe el resultado y ejecuta la devolución de llamada del objeto de escucha una vez que el fragmento está STARTED.

Únicamente puedes tener un solo objeto de escucha y resultado para una clave determinada. Si llamas a setFragmentManager() más de una vez para la misma clave y si el objeto de escucha no es STARTED, el sistema reemplazará los resultados pendientes con el resultado actualizado.

Si configuras un resultado sin un objeto de escucha correspondiente para recibirlo, el resultado se almacenará en el `FragmentManager` hasta que establezcas un objeto de escucha con la misma clave. Una vez que un objeto de escucha reciba un resultado y active la devolución de llamada `onFragmentResult()`, se borrará el resultado. Este comportamiento tiene dos implicaciones importantes:

- Los fragmentos sobre la pila de actividades no reciben resultados hasta que se resaltan y son `STARTED`.
- Si un fragmento que escucha un resultado es `STARTED` cuando se establece el resultado, se activa de inmediato la devolución de llamada del objeto de escucha.

★ **Nota:** Como los resultados del fragmento se almacenan en el nivel del `FragmentManager`, tu fragmento debe estar vinculado para llamar a `setFragmentManagerResultListener()` o `setFragmentManagerResult()` con el `FragmentManager` superior.

## Cómo probar los resultados de los fragmentos

Usa `FragmentScenario` para hacer llamadas de prueba a `setFragmentManagerResult()` y `setFragmentManagerResultListener()`. Crea un escenario para el fragmento en prueba usando `launchFragmentInContainer` o `launchFragment`, y luego llama de forma manual al método que no se está probando.

A fin de probar `setFragmentManagerResultListener()`, crea una situación con el fragmento que hace la llamada a `setFragmentManagerResultListener()`. Luego, llama a `setFragmentManagerResult()` de forma directa y verifica el resultado:

Para probar `setFragmentManagerResult()`, cree una situación con el fragmento que hace la llamada a `setFragmentManagerResult()`. A continuación, llama a `setFragmentManagerResultListener()` de forma directa y verifica el resultado:

```
@Test
fun testFragmentManagerResultListener() {
    val scenario = launchFragmentInContainer<ResultListenerFragment>()
    scenario.onFragment { fragment ->
        val expectedResult = "result"
        fragment.parentFragmentManager.setFragmentManagerResult("requestKey", bundleOf("bundleKey" to expectedResult))
        assertThat(fragment.result).isEqualTo(expectedResult)
    }
}

class ResultListenerFragment : Fragment() {
    var result : String? = null
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // Use the Kotlin extension in the fragment-ktx artifact.
        setFragmentManagerResultListener("requestKey") { requestKey, bundle ->
            result = bundle.getString("bundleKey")
        }
    }
}
```

```

@Test
fun testFragmentResult() {
    val scenario = launchFragmentInContainer<ResultFragment>()
    lateinit var actualResult: String?
    scenario.onFragment { fragment ->
        fragment.parentFragmentManager
            .setFragmentResultListener("requestKey") { requestKey, bundle ->
                actualResult = bundle.getString("bundleKey")
            }
    }
    onView(withId(R.id.result_button)).perform(click())
    assertThat(actualResult).isEqualTo("result")
}

class ResultFragment : Fragment(R.layout.fragment_result) {
    override fun onCreateView(view: View, savedInstanceState: Bundle?) {
        view.findViewById(R.id.result_button).setOnClickListener {
            val result = "result"
            // Use the Kotlin extension in the fragment-ktx artifact.
            setFragmentResult("requestKey", bundleOf("bundleKey" to result))
        }
    }
}

```

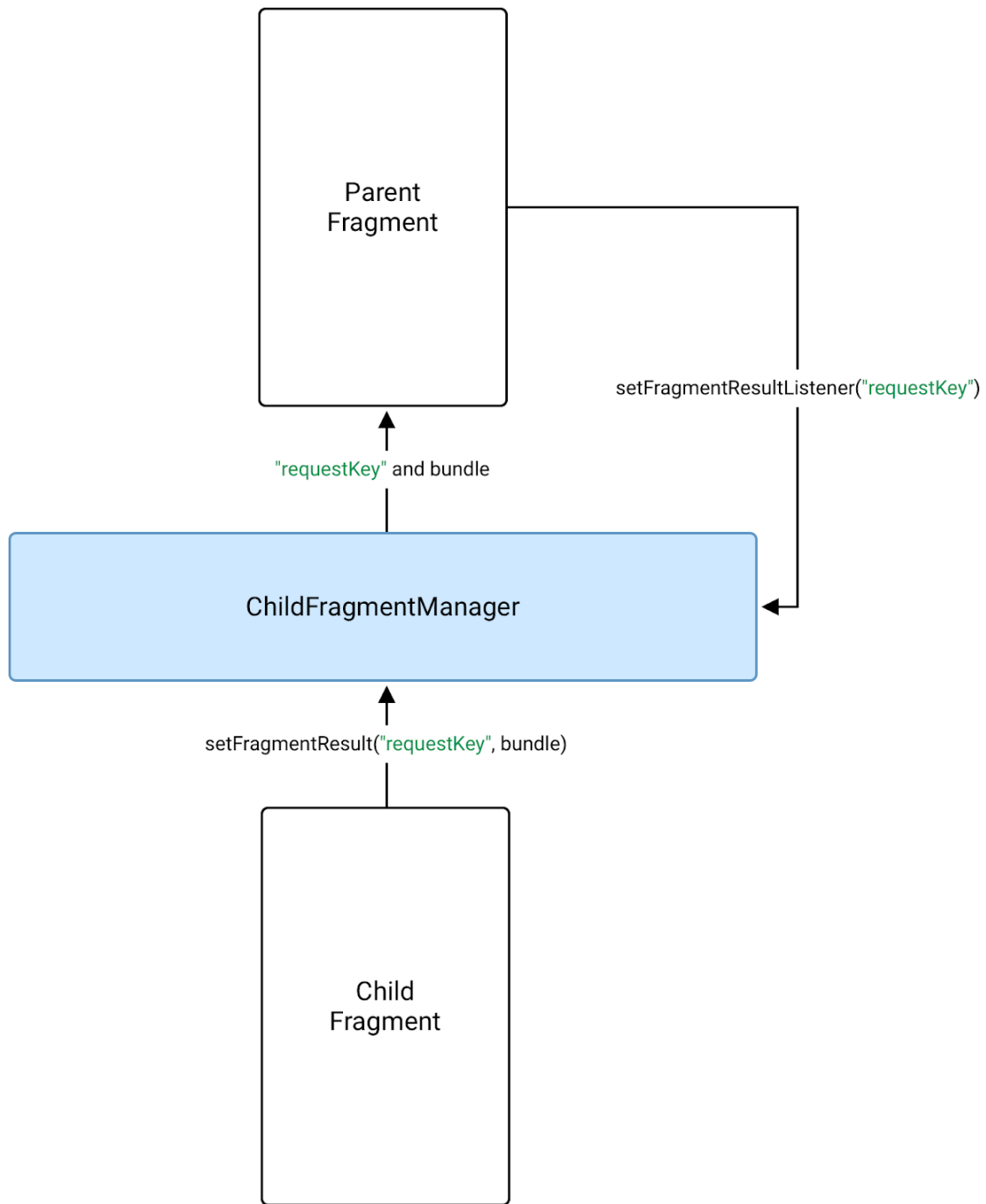
## Cómo pasar resultados entre fragmentos superiores y secundarios

Para pasar un resultado de un fragmento secundario a uno superior, usa `getChildFragmentManager()` del fragmento superior en lugar de `getParentFragmentManager()` cuando llames a `setFragmentResultListener()`.

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // Set the listener on the child fragmentManager.
    childFragmentManager.setFragmentResultListener("requestKey") { key, bundle ->
        val result = bundle.getString("bundleKey")
        // Do something with the result.
    }
}

```



**Figura 2:** Un fragmento secundario puede usar FragmentManager para enviar un resultado a su superior

El fragmento secundario establece el resultado en su FragmentManager. Luego, el superior recibe el resultado una vez que el fragmento está STARTED:

```
button.setOnClickListener {
    val result = "result"
    // Use the Kotlin extension in the fragment-ktx artifact.
    setFragmentResult("requestKey", bundleOf("bundleKey" to result))
}
```

## Cómo recibir los resultados en la actividad del host

Para recibir un resultado de fragmento en la actividad del host, configura un objeto de escucha de resultados en el administrador de fragmentos mediante `getSupportFragmentManager()`.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        supportFragmentManager
            .setFragmentResultListener("requestKey", this) { requestKey, bundle ->
                // We use a String here, but any type that can be put in a Bundle is supported.
                val result = bundle.getString("bundleKey")
                // Do something with the result.
            }
    }
}
```