Developing Android Apps

Source: <https://classroom.udacity.com/courses/ud851>

# Lesson 01 - Create Project Sunshine

## Creating a project

Crear un Proyecto es sencillo, es solo seleccionar New Project, darle un nombre al app, el company domain el cual es generalmente el nombre nombre del dominio de internet de la empresa. Este nombre debe ser único globalmente a través de todos los paquetes instalado en Android.

Luego se seleccionan los canales donde uno quiere correr el app, como tablets, teléfonos, etc. Y por último el tipo de activity.

## Min and Target Version

Estas versiones se pueden verificar al seleccionar el app folder en el tree view del proyecto y seleccionar: Files > Project Structure. En el tab Modules, hay un Default Config que tiene el Application Id, el Version Code que es reconocido por Google Play. Y esta Tambien el Target SDK Version ye l Minimum SDK version.

El Min SDK version es la versión mínima en la que el app puede correr. Esto es como un filtro que le permite a Google Play mostrar u ocultar el app en dispositivos que tengan una versión menor a la especificada aquí.

## Android Software Stack

La base del Stack de Android es un kernel de Linux que maneja el hardware, drivers y alimentación de energía. Luego están las librerías de C/C++ y el Android Runtime. El app corre dentro de su propia instancia en el runtime usando las clases y servicios provistos por el Application Framework. Encima de eso esta el Application Layer que incluye nuestra app y todas las demás instaladas.

### Gradle

Gradle es el Sistema para generar builds en Android Studio. En cada Proyecto hay varios scripts de gradle generados automáticamente. Al correr el Proyecto, si algo ha cambiado, los scripts se corren.

Una tarea de gradle representa una pieza atómica de trabajo para un build. Uno puede correr tareas de build en la línea de comando de gradle. Para empezar, se debe navegar al root del Proyecto y correr el commando:

./gradlew tasks

En el archivo build.gradle se incluyen dependencias del Proyecto, en Android Studio se pueden ver varios de estos archivos, uno para el Proyecto y otro para los módulos por ejemplo.

### Android Debug Bridge

El ADB es una utilidad de línea de comando incluida en el SDK que permite hacer muchas cosas como listar los dispositivos conectados, iniciar el app, correr el emulador, etc.

## Activities, Packages and Layout

Las apps son colecciones de componentes conectados. Hay 4 tipos de componentes en un app:

* Activity
* Service
* Content Provider
* Broadcast Receiver

Los componentes se registran en el Android Manifest.

Las actividades son las responsables de crear las ventanas que el app usa para dibujar y recibir eventos del sistema. Desde la perspectiva del usuario, las actividades están ligadas y Android las mantiene en una cola para preservar el contexto. Al presionar el botón de back, Android retorna la actividad anterior en la cola o la devuelve al “launcher”. En el Manifest se registra la actividad que funcionara como launcher.

Cada actividad tiene un layout que se encarga de decir que es lo que se va a desplegar y como se va a desplegar. Las actividades están escritas en Java y extienden la clase Activity.

Las actividades tienen un archivo XML que se ubica en la carpeta res > layout. Cada layout puede tener varios views. En el folder java, se encuentran las clases que utilizan las vistas. En el método **onCreate** generalmente se pone el método **setContentView** que es el que carga el layout que uno especifique.

## Android Layouts Primer

Existen 2 categorías principales de views. La primera es sobre los compontes UI, como TextView, EditText, ImageView y otros que son interactivos generalmente.

La segunda son los Container View o llamados layouts, que extienden la clase ViewGroup, como:

* **LinearLayout**: despliegan vistas en una única columna o fila
* **RelativeLayout**: despliega vistas posicionadas de forma relativa hacia los demás y al layout.
* **FrameLayout**: contiene una única vista hija.
* **ScrollView**: un FrameLayout que está diseñado para permitirle al usuario hacer scroll a través del contenido de la vista.
* **ConstraintLayout**: Posiciona las vistas de forma flexible.

Estos layouts pueden anidarse.

### XML Attributes

Las vistas tienen atributos XML que controlan las propiedades de la vista, como el texto, la altura y ancho.

La altura y el ancho deben especificarse para todas las vistas, se pueden especificar en pixels o mejor aun en **dps** (density-independent pixels). Generalmente por un tema de responsiveness, se usan los valores “wrap\_content” o “match\_parent”

### Asociar los XML Layouts a las Actividades

Para asociar una actividad de Java con un XML Layout se usa el método setContentView en el método onCreate de la actividad y pasando una referencia del layout como R.layout.name-of-layout.

### La Clase R

Esta es generada al compilar el app y crea constantes que permite identificar los contenidos del folder res.

Para agregar un id a una vista en el XML se agrega lo siguiente:

**android:id="@+id/id-del-view"**

Los ids son identificadores numéricos que se usan para localizar los views pero que se puede representar con una constante. El @ le indica a las herramientas que no trate lo que se encuentra entre paréntesis como un string literal, en lugar de eso, que busque en los contenidos dentro de los recursos de Android. El signo de más indica que cree el id si aún no existe. La palabra id indica que se está creando un id.

En el activity, para poder encontrar una vista en específico, se usa el método **findViewById**, por ejemplo:

mWeatherTextView = (TextView) findViewById(R.id.tv\_weather\_data);

## Responsive Design

Las apps en android permiten el diseño responsive a través del uso de diferentes layouts, como el linear, el constraint y el grid layout.

*FrameLayout* es recomendado para layouts simples cuando se tiene solo 1 hijo.

*LinearLayout* son buenos para apilar vistas vertical u horizontalmente.

*ConstraintLayout* es poderoso, pero más complicado que los otros. En este se puede posicionar vistas de forma relativa al contenedor o a las vistas hermanas.

## Setting Text

Hay varias formas de setear el texto en los views, por ejemplo, para un text view se puede usar el método **append** o bien el **setText**. El append va a agregando a lo que ya tiene el view, mientras que el setText sobreescribe el texto.

# Lesson 02 – Connect to the Internet

## Logging

Android Studio cuenta con una Ventana de logging para ver mensajes sobre la aplicación. Cuando uno loguea mensajes, existen diferentes niveles: WTF, ERROR, WARN, INFO, DEBUG y VERBOSE

Siendo Verbose el de más baja importancia. Error, Warn e Info se suelen dejar para las versiones de release. Para loguear algo, la sintaxis suele ser:

Log.***x***(String TAG, String message);

Donde tag aunque puede ser cualquier texto, suele usarse para el nombre de la clase

X es el nivel, y suele ser:

* w para WTF, que no debería usarse y al ocurrir, detiene el device y envía un reporte de error
* e para error
* w para warn
* i para info
* d para debug
* v para verbose. Esta suele usarse para información muy granular

## Resources

### El directorio res

En este folder es donde uno debe agregar cosas como imágenes, strings y layouts. Hay más información aquí: <https://developer.android.com/guide/topics/resources/providing-resources.html>

Algunos folders de recursos importantes son:

* *values*: archivos XML que tienen valores simples
* *drawable*: archivos como imágenes o bitmaps
* *layouts*: los XML layouts del app
* *color*: XMLs que definen listas de colores
* *mipmap*: archivos para los iconos del launcher
* *menu*: XMLs que definen los menús del app
* *xml*: XMLs arbitrarios, como para configuraciones

### Trabajando con strings.xml

En java, se puede obtener un String que se haya guardado en res > values > strings.xml al llamar al método ***getString*** pasando el resource ID. Por ejemplo, si en el strings.xml se tiene esto:

<string name="today">Today</string>

Y se quisiera obtener ese valor en Java, se podría hacer así:

String myString = getString(R.string.today);

En un XML se puede acceder un String usando el accessor @string, por ejemplo:

<TextView text=”@string/today” />

## Menús

Los menús son recursos XML que tienen ítems y cada ítem tiene un ***orderInCategory*** que indica el orden de aparición de cada ítem en referencia a los otros ítems. A partir de la versión Honeycomb se agregó un control UX llamado **action bar** que se ubica en la parte de arriba de la pantalla, es decir, se le indica que se quiere mostrar como un botón. También permite especificar menu items como acciones y si no hay mucho espacio, esas acciones aparecen como botones. Para eso los ítems llevan una propiedad llamada **app:showAsAction=”ifRoom”.** Esta propiedad lleva app en lugar de Android para mantener la compatibilidad con Gingerbread y la librería de soporte.

<menu xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto">  
 <item android:id="@+id/action\_search"  
 android:orderInCategory="1"  
 app:showAsAction="ifRoom"  
 android:title="@string/search" />  
</menu>

Para crear un menú, se sobre escribe el método **onCreateOptionsMenu** dentro de la actividad. Se pueden manejar la selección de elementos del menú al sobre escribir el método **onOptionsItemSelected**

En el método **onCreateOptionsMenu** se “infla” el menú usando el método getMenuInflater().inflate():

@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
 getMenuInflater().inflate(R.menu.*main*, menu );  
 return true;  
}

El método **onOptionsItemSelected** se usa para poder trabajar con los elementos que se seleccionen de un menú. Primero se debe obtener el id del ítem seleccionado. Luego se puede usar un ***Toast*** para poder visualizar que el botón se presionó:

@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
 int menuItemThatWasSelected = item.getItemId();  
 if(menuItemThatWasSelected == R.id.*action\_search*){  
 Context context = MainActivity.this;  
 String message = "Search clicked!";  
 Toast.*makeText*(context, message, Toast.*LENGTH\_LONG*).show();  
 }

return super.onOptionsItemSelected(item);  
}

Otra forma que existe de asignar una acción a un menú item es hacerlo desde el propio onCreateOptionsMenu. Al “inflar” el menú, se puede seleccionar de una vez un menuItem y si lo que hace ese menuItem es ejecutar un Intent, se puede usar el método setIntent, por ejemplo:

private Intent createShareForecastIntent() {

Intent shareIntent = ShareCompat.IntentBuilder.from(this)

.setType("text/plain")

.setText(mForecast + FORECAST\_SHARE\_HASHTAG)

.getIntent();

return shareIntent;

}

@Override

public boolean onCreateOptionsMenu(Menu menu) {

getMenuInflater().inflate(R.menu.detail, menu);

MenuItem menuItem = menu.findItem(R.id.action\_share);

menuItem.setIntent(createShareForecastIntent());

return true;

}

En el ejemplo anterior se tiene un método que devuelve un intent cargado para compartir un texto, en el onCreateOptionsMenu se obtiene uno de los menuItems y se setea este método mediante el setIntent.

## Toast

Los toast son mensajes a pantalla en forma de popups. Para crear un Toast, se usa el metodo ***makeToast*** de la clase estática Toast el cual recibe un context, que se puede obtener del activity, por ejemplo, MainActivity.this, se le da un mensaje y se especifica por cuanto tiempo se quiere mostrar. Y se llama al método show para que lo muestre.

## Build a URL

Para construir una URL, lo recomendado es usar constantes para evitar errores o typos. También, hacer uso de la clase **URI.Builder** del framework., que ayuda a construir URLs de manera fácil sin tener que preocuparse por los componentes del URI como agregar ampersands o realizar el encoding de caracteres inválidos.

Por ejemplo, al tener definidas algunas constantes como el Base URL, los parámetros (PARAM\_QUERY, PARAM\_SORT) se puede tener un método como el siguiente:

public static URL buildUrl(String githubSearchQuery) {  
 // *TODO (1) Fill in this method to build the proper GitHub query URL* Uri builtUri = Uri.*parse*(*GITHUB\_BASE\_URL*).buildUpon()  
 .appendQueryParameter(*PARAM\_QUERY*, githubSearchQuery)  
 .appendQueryParameter(*PARAM\_SORT*, *sortBy*)  
 .build();  
 //The Uri needs to be parsed as a Java URL  
 URL url = null;  
 try {  
 url = new URL(builtUri.toString());  
 } catch (MalformedURLException e){  
 e.printStackTrace();  
 }  
   
 return url;  
}

La primera parte construye una Uri (android Uri) a partir de los valores de las constantes y el query string que se envía como parte del método, mientras que la segunda parte crea una URL (Java URL) a partir de esta Uri.

Primero se parsea la url base, luego se crea un **Uri.builder** con el método **buildUpon**, luego se agregan los query parametros y por último se produce el Uri con el método **build**.

Este método genera un tipo Android Uri, pero el método para llamar la url ocupa un tipo Java URL, por lo que se puede crear usando un new URL y el valor obtenido del Android Uri.

Nota:

Para obtener el texto de un EditText, se utiliza el metodo getText()

## Fetching an HTTP Request

Para poder hacer un llamado a un servicio, por ejemplo, un http request, hay varias formas, una de ellas es usar un objeto ***HttpURLConnection*** el cual se puede cargar con el método **openConnection** del objeto URL. Esto devuelve un http connection pero esto aun no llama al método http.

Luego se debe obtener el input stream y leerlo, lo cual se puede hacer de muchas formas en Java, una de ellas es usando el objeto **Scanner** que es fácil y relativamente rápido. Al usar el delimitador “\A” se fuerza al objeto a leer el contenido entero del stream hasta la siguiente secuencia de tokens. Esto ayuda en varias cosas:

* Almacena los datos, es decir, no solo obtiene los datos, sino que automáticamente asigna y desasigna los buffers a como se necesiten.
* Maneja el encoding de los caracteres: transforma de UTF-8 (default de JSON y JS) a UTF-16, el formato usado por Android.

Hay muchas formas de hacer esto, al igual que hay librerías que simplifican el networking de Android mientras agregan funcionalidad a la cola http.

Ejemplo de un método que llama a un servicio y devuelve un string con los resultados:

public static String getResponseFromHttpUrl(URL url) throws IOException {  
 HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();  
 try {  
 InputStream in = urlConnection.getInputStream();  
  
 Scanner scanner = new Scanner(in);  
 scanner.useDelimiter("\\A");  
  
 boolean hasInput = scanner.hasNext();  
 if (hasInput) {  
 return scanner.next();  
 } else {  
 return null;  
 }  
 } finally {  
 urlConnection.disconnect();  
 }  
}

## Permisos

Antes de poder realizar cualquier llamado se debe pedir permiso. Las apps tienen su propio y único Linux user ID, y cada una corre dentro de su propia instancia del Android runtime. Esto hace que sus archivos, procesos y recursos sean inaccesibles por otras apps.

Cada app tiene un manifiesto donde se declaran los permisos que estas van a necesitar. No para todos los permisos se despliega un dialogo al instalar el app, pero si para los más sensibles al usuario. Como Buena práctica se debería pedir un numero mínimo de permisos.

Información sobre permisos en Android:

<http://developer.android.com/guide/topics/security/permissions.html>

### Tipos de Permisos

#### Install-time Permissions

Da acceso limitado a datos restringidos al app y permite al app ejecutar acciones restringidas que afectan de forma mínima al Sistema u otras apps.

#### Runtime Permissions

O permisos peligrosos, le dan al app acceso adicional a datos restringidos y le permiten ejecutar acciones restringidas que afectan sustancialmente al Sistema y otras apps. Estos necesitan ser solicitados y muestran un cuadro de dialogo.

#### Special Permissions

Corresponden a operaciones particulares del app. Solo la Plataforma y OEMs pueden definirlos

### Declarar Permisos

Para declarar los permisos, se deben incluir el element **<uses-permission>** apropiado en el manifiesto, por ejemplo, para un app que requiere acceso a la cámara se puede usar esta línea:

<manifest ...>  
    **<uses-permission android:name="android.permission.CAMERA"/>**  
    <application ...>  
        ...  
    </application>  
</manifest>

Algunos permisos requieren el uso especifico de hardware. En la mayoría de los casos el hardware es opcional, por lo que es mejor declararlo como opcional usando el android:required en false en el **<uses-feature>** como este:

<manifest ...>  
    <application>  
        ...  
    </application>  
    **<uses-feature android:name="android.hardware.camera"  
                  android:required="false"** />  
<manifest>

Si no se declara el feature como false, Android asume que es requerido para el app y eso puede prevenir que algunos dispositivos instalen el app si no cuentan con el hardware. Pero se puede determinar si el hardware está disponible o no. Ver más en:

<https://developer.android.com/training/permissions/declaring>

### Solicitar Permisos

Si se necesita usar un recurso o solicitar información fuera del “sandbox” del app, es necesario declarar un permiso y solicitar el acceso: ([https://developer.android.com/guide/topics/permissions/overview#workflow](https://developer.android.com/guide/topics/permissions/overview" \l "workflow) )

Después de la versión 6.0 (API 23) uno debe solicitar permisos peligrosos en tiempo de ejecución.

Ver más aquí: <https://developer.android.com/training/permissions/requesting>

## Thread Basics

Android dispara una excepción cuando se intenta conectar a internet en el hilo principal: Android soporta múltiples hilos o bien, multitasking, pero cada app también se puede dividir en múltiples hilos de ejecución. En dispositivos modernos estos hilos pueden correrse en diferentes núcleos.

Android maneja para cada app una Interfaz Simple de Hilos que es la responsable de obtener eventos de varios sensores y setear el siguiente frame a dibujar. Para poder correr a 60FPS, todas las operaciones deberían durar menos de 17 ms. Networking toma segundos, lo que supone que, si se maneja en el hilo principal, el app se congelaría. Después de 5 segundos Android pregunta al usuario si desea cerrar el app.

Por lo tanto, para networking se debe correr en un hilo secundario y hacer cambios en la UI a través de tareas asíncronas.

## AsyncTask

Permite correr una tarea en un hilo background mientras publica resultados en el hilo del UI.

El hilo de UI maneja una cola de mensajes que pueden venir de otros hilos.

Un AsynTask es una clase genérica que recibe tipos parametrizados en su constructor. Los principales tipos usados en un AsyncTask son:

* **Params**: el tipo que se envía a la tarea para la ejecución
* **Progress**: tipo publicado para actualizar el progreso durante la ejecución en background
* **Result**: el tipo de resultado de la ejecución.

Estos 3 parámetros corresponden a las 3 principales funciones que se pueden sobre escribir en un AsyncTask:

* doInBackground
* onProgressUpdate
* onPostExecute
* onPreExecute (esta es la primera que se ejecuta)

Para ejecutar un AsyncTask se llama al método Execute y esto pasa:

1. En el UI Thread se ejecuta el onPreExecute por si uno quiere inicializar algo en el UI antes de que se corra la tarea en background
2. Luego se corre el doInBackground en otro hilo. Es el único método que se debe sobre escribir y lleva los parámetros que uno envía al Execute.
3. Si uno quiere actualizar algo en el UI, uno puede llamar al publishProgress que cause que se llame al onProgressUpdate con los parámetros de progreso. Este método se puede llamar tantas veces como se necesite desde el doInBackground.
4. Cuando la tarea en background finaliza se ejecuta el onPostExecute

Mas información aquí: <https://developer.android.com/reference/android/os/AsyncTask.html>

La clase AsyncTask debe ser extendida, es decir, uno debe crear una clase que extienda AsyncTask y sobre escrirbla al menos el método doInBackground. Ejemplo:

 private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {  
     protected Long doInBackground(URL... urls) {  
         int count = urls.length;  
         long totalSize = 0;  
         for (int i = 0; i < count; i++) {  
             totalSize += Downloader.downloadFile(urls[i]);  
             publishProgress((int) ((i / (float) count) \* 100));  
             // Escape early if cancel() is called  
             if (isCancelled()) break;  
         }  
         return totalSize;  
     }  
  
     protected void onProgressUpdate(Integer... progress) {  
         setProgressPercent(progress[0]);  
     }  
  
     protected void onPostExecute(Long result) {  
         showDialog("Downloaded " + result + " bytes");  
     }  
 }

Para llamar a esta función, solo se necesita el siguiente código:

 new DownloadFilesTask().execute(url1, url2, url3);

Otro detalle a tomar en cuenta es que el parámetro del metodo doInBackground se declara asi URL… urls, esto quiere decir que son variables Java que técnicamente son pasados como arrays. Por eso al obtener el url se usa urls[0].

En el hilo principal de UI se corre el método execute, que también llama al onPreExecute, luego en el hilo background se ejecuta el doInBackground que puede llamar al publisProgress en varias ocasiones y esto ocasionaría que en el main UI se ejecute el onProgressUpdate, por último, al terminar la tarea en el hilo background, se llamaría el método onPostExecute en el main UI.

Si en este punto se comentara el permiso de Internet que se concedió previamente, ahora el error que debería aparecer el logcat es sobre SecurityException.

## Puliendo el App

En este punto, se pueden agregar ciertas cosas al app para hacerla más user friendly. Por ejemplo, poner un mensaje de error si algo falla con el request o bien, mostrar un progress bar para que el usuario sepa que la aplicación está trabajando en obtener los resultados.

Para esto se puede agregar un FrameLayout que envuelva al ScrollView que tiene los resultados y agregar otro TextView para el mensaje de error y un ProgressBar, estos dos últimos con un visibility = “invisible”. Este FrameLayout tendrá el ancho y alto como match\_parent para que adopte todo el espacio que queda de la pantalla.

La forma de ocultar un view desde código suele ser asi:

mSearchResultsTextView.setVisibility(View.*INVISIBLE*);

## JSON Format

JSON es el principal método para intercambiar información por web porque su formato es sintácticamente idéntico al que se usa en JS para crear objetos. En comparación con XML, es más compacto, más fácil de escribir y permite la declaración de arreglos.

Como se lee un JSON en Android

* Primero, se inicializa un objeto JSONObject con el JSON string
* Se obtiene la parte del JSON que interesa, usando por ejemplo: .getJSONObject(“name”);
* Se obtienen valores atraves del getString

Por ejemplo, teniendo el siguiente JSON

{

"temp": {

"min":"11.34",

"max":"19.01"

},

"weather": {

"id":"801",

"condition":"Clouds",

"description":"few clouds"

},

"pressure":"1023.51",

"humidity":"87"

}

Para obtener el valor de la propiedad condition, se podría obtener asi:

String **getCondition**(String jsonString) {

JSONObject forecast = **new** JSONObject(jsonString);

JSONObject weather = forecast.getJSONObject("weather");

**return** weather.getString("condition");

}

# Lesson 03 – RecyclerView

## ¿Porque usar RecyclerViews?

Cuando se tiene una aplicación con diferentes vistas las cuales pueden tener imágenes, eventos ligados como una lista de los pronósticos del tiempo. Aunque se vean solo algunos datos, el app debe crear estas vistas para todos los datos a mostrar. Además, las imágenes a usar deben estar almacenadas en memoria. Para aplicaciones con listas extensas, esto se vuelve impráctico. Para cada elemento se debería manejar las interacciones como asignar los click handlers y todo este setup involucra retrasos en la experiencia del usuario y aún peor, puede causar que la aplicación se quede sin memoria.

Para eso está el RecyclerView. En lugar de crear ítems al momento de hacer scroll, lo cual puede causar glitches, se mantienen algunos datos en una cola o una papelera de reciclaje, para reusar. Cuando se está por hacer un scroll, el RecyclerView retorna uno de los ítems previamente creados y el código enlaza el item view con el nuevo contenido. Los ítems que se quedan por fuera de pantalla al hacer scroll, vuelven a ser puesto en la cola para su reusó

## ¿Como Usarlo?

Casi todas las partes del RecyclerView son modulares. El RV tiene un **Adapter** que provee las nuevas vistas cuando se ocupan. El adapter es usado también para ligar data a las vistas de algún Data Source. El adapter envía las vistas al RecyclerView en un objeto llamado **ViewHolder**, el cual tiene una referencia a la vista raíz para el ítem. Se espera que uno lo use para guardar en cache las vistas representadas en el layout. De esta forma, el findViewByID se llama solo por cada vista cuando el nuevo ítem es creado y no cada vez que quieras popular las vistas en el ítem con datos.

El **Layout Manager** luego le dice al RV como mostrar todas esas vistas. Que puede ser haciendo scroll vertical, horizontal o incluso como listas en un grid.

Como se basa en un Data Source, es fácil filtrar ítems.

## ViewHolders

Es importante usar ViewHolders para almacenar en cache los objetos view que uno va a llenar con la data o imágenes. Cuando el RV es poblado por primera vez, uno llama al findViewById por cada vista que va a mostrar data desde el Adaptor. Esta búsqueda puede ser costosa. Por lo que es mejor hacerlo solo una vez y almacenar en cache estas vistas en un View Holder, pudiendo accesarlos luego sin tener que buscarlos repetidamente.

## Creando el RecyclerView

En las dependencias del build.gradle, agregar la dependencia especifica al RecyclerView:

Ejemplo:

dependencies **{** implementation fileTree(dir: 'libs', include: ['\*.jar'])  
 implementation 'com.android.support:appcompat-v7:25.1.0'  
 // COMPLETED (1) Add RecyclerView dependency  
 implementation 'com.android.support:recyclerview-v7:25.1.0'  
  
**}**

Cuando se hacen cambios en el build.gradle se debe sincronizar el proyecto.

Un RecyclerView en el layout se ve básicamente así:

<android.support.v7.widget.RecyclerView  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:id="@+id/rv\_numbers" ></android.support.v7.widget.RecyclerView>

El ViewHolder es el objeto que guardara las referencias a los ítems que se mostraran en el RV.

#### Nota Sobre DP y SP

SP y DP son medidas que automáticamente escalan para tener el mismo tamaño físico aproximado con respecto a la densidad de los pixeles en la pantalla.

La conversión de DP es la siguiente:

|  |
| --- |
| **px = dp \* (dpi / 160)** |

SP funciona parecido a DP pero escala basado en las preferencias del usuario. Específicamente en el tamaño de texto del usuario.

Un ejemplo de un ViewHolder es el siguiente, una clase que extiende a RecyclerView.ViewHolder

class NumberViewHolder extends RecyclerView.ViewHolder {  
  
 TextView listItemNumberView;  
  
 public NumberViewHolder(View itemView) {  
 super(itemView);  
 listItemNumberView = (TextView)itemView.findViewById(R.id.*tv\_item\_number*);  
 }  
  
 void bind(Integer listIndex){  
 listItemNumberView.setText(String.*valueOf*(listIndex));  
 }  
  
}

En el constructor del ViewHolder se obtiene una referencia del item\_number usando el findViewByID del itemView. Una de las cosas más importantes que hace el ViewHolder es almacenar estas referencias en cache que serán modificadas en el adapter.

Luego se creó un método llamado bind el cual recibe un integer y lo setea como texto al listItemNumberView usando el método valueOf de String para no tener problemas al castear un integer a un string. Si se usa solo setText y se le pasa un integer, Android asume que se le está pasando una referencia de un recurso string del strings.xml que podría o mostrar un texto incorrecto o botar el app.

#### Nota sobre Gravity

En un FrameLayout, al tener un TextView, si se quiere alinear el view al inicio del padre y verticalmente en el centro se usa la propiedad Gravity, usando dos valores separados por el pipe:

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:padding="16dp">  
 <TextView  
 android:id="@+id/tv\_item\_number"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_gravity="center\_vertical|start"  
 android:fontFamily="monospace"  
 android:textSize="42sp"></TextView>  
</FrameLayout>

## RecyclerView y Adapter

Cuando ya se tiene un RecyclerView y un ViewHolder, se necesita un Adapter para poder conectar el RecyclerView a un DataSource. El Adapter es llamado por el RecyclerView para crear nuevos ítems en forma de ViewHolders. El Adapter también llena o liga estos ítems con la data y retorna información sobre la data como cuantos ítems hay en un dado Data Source.

Esta data puede venir de un ArrayList, un JSON o cualquier data source que uno pueda modelar. El adapter requiere que uno sobre escriba 3 funciones:

1. **onCreateViewHolder**: este es llamado cuando el RecyclerView instancia una nueva instancia de ViewHolder.
2. **onBindViewHolder**: es llamado cuando el RecyclerView quiere poblar el view con data del modelo, ligándolo al data source.
3. **getItemCount**: retorna el número de ítems en nuestro data source

El proceso es más o menos así:

* Al inicio, el RecyclerView está vacío, por lo que le pide al Adapter el número de ítems que va a desplegar, esto lo puede hacer muchas veces durante el proceso de layout.
* El RecyclerView le va a pedir al Adapter que cree objetos ViewHolder y durante el proceso, que infle los ítem views individuales con su correspondiente XML al llamar al onCreateViewHolder. Creará tantos ViewHolders como sea necesario.
* El onCreateViewHolder es el responsable de crear las vistas, ya sea inflar los ítem views desde XML o crearlos en código. Y luego retorna un ViewHolder asociado con esta nueva vista.
* Después de que cada ViewHolder es creado, el RecyclerView llama al onBindViewHolder para poblar cada ítem con los datos. Cuando uno se desplaza (scroll), el RecyclerView reusara esos ViewHolders pidiéndole al Adapter que ligue nuevos datos a ellos.

En el método onCreateViewHolder, uno puede “inflar” los layouts usando el método **inflate** de la clase statica **inflater**, este método toma un ID de un layout en XML, luego lo infla o convierte en una colección de view groups o views que los representan en código Java, el ultimo parámetro, un boolean es para decidir si el layout debe ser agregado inmediatamente al padre o no. Este método retorna una vista, la cual se pasa en el constructor del ViewHolder.

Ejemplo de Adapter y ViewHolder:

La clase GreenAdapter tiene una clase propia para el ViewHolder en este ejemplo

/\*  
 \* limitations under the License.  
 \*/  
package com.example.android.recyclerview;  
  
import android.content.Context;  
import android.support.v7.widget.RecyclerView;  
import android.view.LayoutInflater;  
import android.view.View;  
import android.view.ViewGroup;  
import android.widget.TextView;  
  
*/\*\*  
 \* We couldn't come up with a good name for this class. Then, we realized  
 \* that this lesson is about RecyclerView.  
 \*  
 \* RecyclerView... Recycling... Saving the planet? Being green? Anyone?  
 \* #crickets  
 \*  
 \* Avoid unnecessary garbage collection by using RecyclerView and ViewHolders.  
 \*  
 \* If you don't like our puns, we named this Adapter GreenAdapter because its  
 \* contents are green.  
 \*/*// COMPLETED (4) From GreenAdapter, extend RecyclerView.Adapter<NumberViewHolder>  
public class GreenAdapter extends RecyclerView.Adapter<GreenAdapter.NumberViewHolder> {  
  
 // Add a private int variable called mNumberItems  
 private int mNumberItems;  
  
 // Create a constructor for GreenAdapter that accepts an int as a parameter for numberOfItems  
 // Store the numberOfItems parameter in mNumberItems  
 public GreenAdapter(int numberOfItems){  
 mNumberItems = numberOfItems;  
 }  
  
 // Override the onCreateViewHolder method  
 // Create and return a new NumberViewHolder within this method  
 @Override  
 public NumberViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {  
 Context context = parent.getContext();  
 int layoutIdForListItem = R.layout.*number\_list\_item*;  
 LayoutInflater inflater = LayoutInflater.*from*(context);  
 boolean shouldAttachToParentImmediately = false;  
  
 View view = inflater.inflate(layoutIdForListItem, parent, shouldAttachToParentImmediately);  
 NumberViewHolder viewHolder = new NumberViewHolder(view);  
 return viewHolder;  
 }  
  
  
 // Override onBindViewHolder  
 // Within onBindViewHolder, call holder.bind and pass in the position  
 @Override  
 public void onBindViewHolder(NumberViewHolder holder, int position) {  
 holder.bind(position);  
 }  
  
 // Override getItemCount and return the number of items to display  
 @Override  
 public int getItemCount() {  
 return mNumberItems;  
 }  
  
 */\*\*  
 \* Cache of the children views for a list item.  
 \*/* class NumberViewHolder extends RecyclerView.ViewHolder {  
  
 // Will display the position in the list, ie 0 through getItemCount() - 1  
 TextView listItemNumberView;  
  
 */\*\*  
 \* Constructor for our ViewHolder. Within this constructor, we get a reference to our  
 \* TextViews and set an onClickListener to listen for clicks. Those will be handled in the  
 \* onClick method below.  
 \** ***@param*** *itemView The View that you inflated in  
 \* {****@link*** *GreenAdapter#onCreateViewHolder(ViewGroup, int)}  
 \*/* public NumberViewHolder(View itemView) {  
 super(itemView);  
  
 listItemNumberView = (TextView) itemView.findViewById(R.id.*tv\_item\_number*);  
 }  
  
 */\*\*  
 \* A method we wrote for convenience. This method will take an integer as input and  
 \* use that integer to display the appropriate text within a list item.  
 \** ***@param*** *listIndex Position of the item in the list  
 \*/* void bind(int listIndex) {  
 listItemNumberView.setText(String.*valueOf*(listIndex));  
 }  
 }  
}

El Adapter extiende la clase RecyclerView.Adapter y recibe la clase NumberViewHolder como parámetro. Con esto se asegura trabajar con el view holder correcto.

Se tiene una variable para el numero de ítems que se llena en el constructor.

En el método onCreateViewHolder se debe retornar un ViewHolder, para eso primero se obtiene el contexto del viewGroup , se obtiene también el id del layout a inflar (number\_list\_item). Se usa el LayoutInflater el cual primero debe obtener el layoutinflater del contexto dado, es decir, del contexto del viewgroup.

Después infla el layout usando el id del layout a inflar, el viewGroup y un parámetro para indicar si se debe o no agregar de una vez al padre. Este método devuelve una vista.

La vista devuelva por el inflater, se usa para crear el viewHolder y luego retornarlo como valor del método.

En el método onBindViewHolder, se utiliza el método bind del viewholder para mostrar en el mismo item el índice o posición del item.

Por último, el getItemCount retorna la cantidad de ítems que ya se almacena en la variable mNumberItems.

## LayoutManager

El último elemento que hay que setear para enlazar el RecyclerView y el adapter es el LayoutManager. Mientras que el ViewHolder determina como se despliegan las entradas individualmente, el LayoutManager determina como se despliega la colección de ítems.

El LayoutManager determina cuando reciclar ítem views que ya no están visibles para el usuario.

Android viene con 3 implementaciones del LayoutManager:

1. LinearLayoutManager: permite hacer scroll vertical u horizontalmente
2. GridLayoutManager: una subclase de LinearLayoutManager, acomoda los ítems en un grid que también puede hacer scroll vertical u horizontal.
3. StaggedGridLayoutManager: despliega un conjunto de ítems que generalmente tienen dimensiones variantes

Ejemplo de cómo ligar el LayoutManager al RecyclerView:

public class MainActivity extends AppCompatActivity {  
  
 private static final int *NUM\_LIST\_ITEMS* = 100;  
   
 private GreenAdapter mAdapter;  
 private RecyclerView mNumberList;  
   
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
   
 mNumberList = (RecyclerView)findViewById(R.id.*rv\_numbers*);  
  
 LinearLayoutManager layoutManager = new LinearLayoutManager(this);  
 mNumberList.setLayoutManager(layoutManager);  
   
 // Designate that the contents of the RecyclerView won't change an item's size  
 mNumberList.setHasFixedSize(true);   
 // Creates a new instance of GreenAdapter  
 mAdapter = new GreenAdapter(*NUM\_LIST\_ITEMS*);  
 // Set the GreenAdapter on the RecyclerView  
 mNumberList.setAdapter(mAdapter);  
   
 }  
}

En el MainActivity se crea una variable para el Adapter y una para el RecyclerView, además se tiene una variable estática para la cantidad de ítems.

En el onCreate, se obtiene la referencia del RecyclerView (mNumberList).

Se crea un **LineaLayoutManager** y este se le asigna al RecyclerView.

Se especifica que el **setHasFixedSize** en true que permite algunas optimizaciones en el UI, como evitar invalidar el layout completo cuando el contenido del adapter cambie.

Se crea un nuevo adapter y por último se conecta el adapter al recycler view

## Respondiendo a los Clicks en los Items

Una de las ventajas del RecyclerView es que facilita el manejo de los clicks en los ítems, aunque no trae un click listener especifico. Aunque es sencillo, Por ejemplo:

Se pude crear una interfaz al cual se le puede agregar un método para que maneje el evento click en un list item. En el adapter se pude agregar una variable del tipo de la interfaz para almacenar una referencia al listener del click del list item.

El constructor del adapter también necesita llevar un parámetro de este tipo lo que permite usar el adapter como un componente con un click handler externo como con la actividad. La variable que se agregó al adapter para manejar el onClickListener se inicializa con el listener que se pasa en el constructor.

Ahora que el adapter tiene acceso al listener, se pasa al view holder para que la vista pueda invocarlo. Por lo que el view holder ahora implementa a View.OnClickListener. Se hace un override del método onClick y se obtiene la posición del adapter, que es el ítem que fue clickeado y luego invoca el onClickListener de la otra clase.

Por último, en el adapter, se necesita setear el setOnClickListener en el itemView en el constructor a this.

En el MainActivity se debe implementar el onListItemClick que recibe el index del item cliqueado, en este método se lanza un toast para mostrar un mensaje sobre el número de item cliqueado.

## Recapitulando

En resumen, el RecyclerView se usa para optimizar recursos al reutilizar vistas o bien, reciclarlas al tener solamente que actualizar el contenido de las mismas, pero dándole la impresión al usuario de que se tienen muchas vistas cargadas.

El RecyclerView es el contenedor para el UI.

Layout: uno puede crear uno o usar uno de los que ya provee Android. El layout generalmente representa solo un ítem, que se va a repetir.

Adapter: maneja la colección de datos y los objetos ViewHolder

ViewHolder: toma un elemento de la lista y llena los componentes en el row layout individual. Maneja las vistas en la lista. El RecyclerView solo crea los suficientes para llenar la pantalla.

Se necesita una colección de objetos Java.

Se necesita agregar la dependencia al build.gradle que generalmente viene siendo así:

dependencies **{**   
 implementation 'com.android.support:recyclerview-v7:28.0.0'  
**}**

Algunas cosas que generalmente se hacen en el onCreate:

* Obtener acceso al LinearLayout
* Invocar el hasFixedSize
* Establecer el LayoutManager (crear una nueva instancia de LinearLayoutManager, por ejemplo)
* Setear el Item Animator
* Asociar el Adapter y el RecyclerView

Adapter

Maneja la colección de objetos y las operaciones de UI

* Obtiene un layout que representa el row actual, lo pasa al ViewHolder para que este lo llene.
* Liga los datos con el ViewHolder
* Crea vistas para los ítems, y reemplaza el contenido de algunas vistas con nuevos data ítems.
* Provee un contador de ítems.

Métodos del Adapter que se usan mucho:

* notify() para volver a ligar una colección al adapter.
* notifyDataSetChanged(): notifica al adapter que la estructura del dataset ha cambiado (se han agregado rows o se han eliminado).

ViewHolder

* Este opera sobre todos los rows, pero solo sobre un ítem a la vez.
* Extiende la clase RecyclerView.ViewHolder
  + Obtiene los componentes de la vista dentro del layout que representan cada fila.
  + Llena los componentes de la vista con la información que representa la posición actual de la colección del adapter.

# Lesson 04 - Intents

## Intents Framework

Al tener varias actividades en un app, en lugar de comunicarse una con la otra, Android facilita la comunicación usando objetos mensajeros llamados intents. Un Intent le permite a un app solicitar que una acción tome lugar, que puede ser cualquier cosa como mostrar otra actividad, desplegar una imagen o hacer una llamada.

Vienen siendo como un sobre, el cual incluye a que componente se le quiere enviar el mensaje junto con la información que uno quiere enviar.

Para moverse entre 2 activities, se llama al método **startActivity** que toma un Intent como parámetro.

Existen muchos constructores para los intents. La documentación de Android ofrece toda la información sobre estos.

## Explicit Intents

Los intents explícitos son aquellos donde uno especifica el componente que se va a invocar desde la actividad.

Cada actividad extiende a context, por lo que cuando se necesite enviar el contexto al intent se puede usar, por ejemplo, MainActivity.this. Para obtener una referencia de una clase, por ejemplo, de ChildActivity, se usa la propiedad class:

Context context = MainActivity.this;  
  
// COMPLETED (1) Store ChildActivity.class in a Class object called destinationActivity  
Class destinationActivity = ChildActivity.class;  
  
// COMPLETED (2) Create an Intent to start ChildActivity  
Intent intent = new Intent(context, destinationActivity);  
  
// COMPLETED (3) Replace the Toast with code to start ChildActivity  
startActivity(intent);

## Pasando Información entre Actividades

Para pasar información entre intents, al tener ya un objeto de tipo **Intent**, se puede usar el método **putExtra**, el cual tiene diferentes sobrecargas. Puede enviar datos primitivos, como caracteres, números o strings.

Intent startChildActivityIntent = new Intent(context, destinationActivity);  
  
// COMPLETED (2) Use the putExtra method to put the String from the EditText in the Intent  
startChildActivityIntent.putExtra(Intent.*EXTRA\_TEXT*, textEntered);  
  
/\*  
 \* Once the Intent has been created, we can use Activity's method, "startActivity"  
 \* to start the ChildActivity.  
 \*/  
startActivity(startChildActivityIntent);

El valor EXTRA\_TEXT es una constante publica definida en la clase Intent, que se puede usar cuando solo se pasan algunos pocos datos.

Para obtener los datos, en la actividad que los va a recibir, se obtiene la instancia de la actividad que inicio la actividad actual con el método **getIntent**().

El método **getStringExtra** permite extraer los datos que han sido pasados a través del putExtra. Antes de obtener la información, se chequea si el intent lleva información extra con el método **hasExtra**.

/\* Typical usage of findViewById... \*/  
mDisplayText = (TextView) findViewById(R.id.*tv\_display*);  
  
// COMPLETED (3) Use the getIntent method to store the Intent that started this Activity in a variable  
Intent intentThatStartedThisActivity = getIntent();  
  
// COMPLETED (4) Create an if statement to check if this Intent has the extra we passed from MainActivity  
if(intentThatStartedThisActivity.hasExtra(Intent.*EXTRA\_TEXT*)){  
 // COMPLETED (5) If the Intent contains the correct extra, retrieve the text  
 String textEntered = intentThatStartedThisActivity.getStringExtra(Intent.*EXTRA\_TEXT*);  
  
 // COMPLETED (6) If the Intent contains the correct extra, use it to set the TextView text  
 mDisplayText.setText(textEntered);  
}

## Implicit Intents

Cuando no se sabe qué actividad o componente se va a llamar para completar la acción, se llama un intente implícito. Estos indican una acción a ser ejecutada y data opcional, como elegir un contacto o marcar un número, sin especificar que aplicación va a ejecutar la acción. Si se tiene solo una app que pueda completar la acción, esta se ejecuta automáticamente, pero si hay más de una, Android permitirá al usuario elegir cual usar.

La documentación de Google es la principal fuente de información para trabajar con Android. Por ejemplo, para crear un intent de uso común, se puede buscar common intents en la documentación, ahí hay ejemplos de cómo validar un intent antes de llamarlo, como con el método **resolveActivity**. Con el resolveActivity se le pide a Android que verifique si hay un app que pueda resolver la actividad, si devuelve algo, entonces se da por hecho que si existe al menos un app:

private void openWebPage(String webUrl){  
 // Use Uri.parse to parse the String into a Uri  
 Uri pageToOpen = Uri.*parse*(webUrl);  
 // Create an Intent with Intent.ACTION\_VIEW and the webpage Uri as parameters  
 Intent intent = new Intent(Intent.*ACTION\_VIEW*, pageToOpen);  
 // Verify that this Intent can be launched and then call startActivity  
 if(intent.resolveActivity(getPackageManager()) != null){  
 startActivity(intent);  
 }  
}

La mayoría de los intents incluyen dos parámetros en el constructor, un action y la data.

## URIs

Para algunos intents como cuando se abre una página web o un mapa se necesitan objetos Uri.

Un URI o Uniform Resource Identifier es un string que identifica un recurso. El nombre formal de una dirección web es URL o Uniform Resource Locator. Una URL es un URI que identifica una web o un recurso de red. Pero una URI como geo puede identificar también una ubicación.

Una URI tiene los siguientes componentes

* **Scheme**: describe que tipo de recurso se está apuntando. Los más populares son http, https, mailto, FTP, file y geo pero hay más. Dependiendo del tipo de recurso, puede venir seguido de 2 slash y un authority que indica un username y password opcional y un puerto.
* **Path to Resouce**: si hay un authority, se pone un slash entre este y el path. Por ejemplo:

En <https://www.audacity.com/courses/all> El authority es [www.audacity.com](http://www.audacity.com/) y el path es courses/all.

mailto es uno de los esquemas que no requiere authority y trabaja con solo el path.

* **Query**: no tiene un formato requerido, pero debe tener un signo de pregunta al inicio. Lo normal es que se usen key-value pairs para definir los query parameters.
* **Fragment**: Precedido por un hash e indica data secundaria que el path usara.

## Creando un Map Intent

Para utilizar un intent que abra un map, generalmente se usa una dirección que debe estar estructurada como un GO URI, la cual consiste del esquema geo, una latitud, una longitud y parámetros opcionales que indican la dirección de la calle o negocio, y el nivel de zoom. Es importante notar que, si hay un query parameter para la calle o negocio, el lat y long deben ir en 0

Ejemplo:

Uri gmmIntentUri = Uri.parse("geo:0,0?q=1600 Amphitheatre Parkway, Mountain+View, California");  
Intent mapIntent = new Intent(Intent.ACTION\_VIEW, gmmIntentUri);  
mapIntent.setPackage("com.google.android.apps.maps");  
startActivity(mapIntent);

Es menos propenso a errores utilizar el Uri.Builder de Android.

public void onClickOpenAddressButton(View v) {  
 // Store an address in a String  
 String address = "1600 Amphitheatre Parkway, CA";  
   
 // Use Uri.Builder with the appropriate scheme and query to form the Uri for the address  
 Uri.Builder builder = new Uri.Builder();  
 builder.scheme("geo")  
 .path("0,0")  
 .query(address);  
 Uri addressUri = builder.build();  
   
 // Replace the Toast with a call to showMap, passing in the Uri from the previous step  
 showMap(addressUri);  
}

## Sharing Intent

Este intent es el que permite compartir data a través de diferentes aplicaciones. Para esto hace falta pensar que tipo de data se va a compartir, el número de archivos y otros detalles, pero Android provee una clase llamada **ShareCompat** para poder trabajar con esto.

ShareCompat y su clase interna **IntentBuilder** abstraen todas esas decisiones para que uno pueda encadenar los bits que uno necesita e ignorar aquellos que no.

Cada archivo que se comparte por internet tiene dos identificadores, está el Media Type que generalmente se conoce como MIME o Multipurpose Internet Mail Extension. Un Media Type string consiste de:

**Top-level type name / subtype name [; parameters]**

Ejemplo:

text/html; charset=UTF-8

es un texto, específicamente un texto HTMl y tiene un carácter set encoding de UTF-8

Para utilizar un Share Intent, uno puede utilizar la clase **IntentBuilder** de ShareCompat para generar el mime type a partir de diferentes strings

private void shareText(String message){  
 // Create a String to hold the mimeType   
 String mimeType = "text/plain";  
 // Create a title for the chooser window that will pop up  
 String title = "Choose an app to share your message";  
 // Use ShareCompat.IntentBuilder to build the Intent and start the chooser  
 ShareCompat.IntentBuilder.*from*(this)  
 .setType(mimeType)  
 .setChooserTitle(title)  
 .setText(message)  
 .startChooser();  
  
}

Esta clase ShareCompat.IntentBuilder puede usar el **startChooser** una vez que se han seteado algunos valores para iniciar el cuadro de dialogo o bien se puede utilizar otro como el getIntent para devolver un intent y verificar por uno mismo si hay aplicaciones que puedan resolver dicha actividad.

## Agregar un Up Button

Para que una actividad tenga un button Up, es decir, un botón que permite volver a la pantalla de atrás uno debe definir este comportamiento en el Manifest.xml, básicamente se le define cual es el parentActivityName y se agrega un meta data donde se especifica el Parent Activity, así:

<activity android:name=".DetailActivity"  
 android:parentActivityName=".MainActivity">  
 <meta-data  
 android:name="android.support.PARENT\_ACTIVITY"  
 android:value=".MainActivity" ></meta-data>  
</activity>

En el ejemplo anterior, se agregó una actividad llamada DetailActivity que se muestra desde el MainActivity. Al ingresar al DetailActivity, se muestra un botón de back para poder volver al MainActivity gracias a esta configuración.

Otra forma de hacer esto, cuando una actividad se inicia a partir de otra, con el **startActivityForResult**, en este caso se puede usar el método **finishActivity()** desde la actividad y esta la cerrara y mostrara la actividad previa.

Si la actividad se inició con **startActivity**() también se puede cerrar con el método **finish**() y esto hará lo mismo.

# Lesson 05 - Lifecycle

## Android Activity Lifecycle

El ciclo de vida de una app en Android conlleva algunos de los siguientes métodos:

1. **onCreate**: donde uno construye y se arma el UI.
2. Una vez creada, hay otros métodos que se pueden disparar como el **onStart**, el cual se ejecuta cuando la actividad se vuelve visible.
3. El método **onResume** se ejecuta cuando la actividad gana el foco y se vuelve el app de primer plano activa.
4. El **onPause** sucede cuando el app pierde el foco
5. Seguido por el **onStop** cuando la aplicación ya no está visible.
6. Finalmente, el **onDestroy** se ejecuta cuando la aplicación ha sido destruida.

Cuando el app arranca, pasa por los primeros tres estados rápidamente (Create, Start, Resume). Durante el ciclo de vida completo de un app, desde que se crea hasta que se destruye, el app pasa de un estado activo a pausado muchas veces.

El app esta Activa y Visible cuando está recibiendo entradas del usuario y no hay otras activities oscureciéndola.

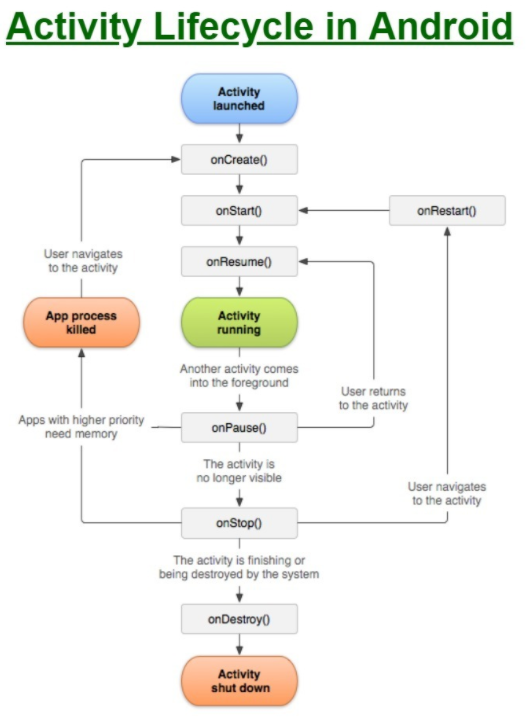
onPause es llamado cuando la actividad es parcialmente oscurecida, como cuando otra actividad de otra app recibe el foco.

Uno no debe pausar ningún proceso que este dibujando el UI.

El lifetime Visible continua cuando el app esta visible y termina cuando es completamente oscurecida por otra app. En ese punto, el app es movida al background. Por lo que se llama al onStop.

Por lo que tanto el onCreate yel onDestroy se van a llamar al menos una vez cuando se use el app, pero pueden llamarse varias veces.

El siguiente es un diagrama de los procesos:



Al rotar el device, la actividad se destruye y se recrea, por lo que el ciclo de vida comienza con el onPause y termina con un onResume. El comportamiento por default es destruir y recrear actividad cuando una configuración del dispositivo cambia.

## Salvar y Restaurar el Instance State

El método onSaveInstanceState permite persistir el estado entre actividades. Este método recibe un Bundle como parámetro que es un mecanismo de almacenaje de llave-valor que se usa para salvar la data que uno quiera. Pero no se puede pasar cualquier cosa. Los bundles tienen que pasarse entre procesos o serializados a un archivo.

Si uno necesita pasar un tipo complejo a un bundle, estos deberían implementar la interfaz parceable. **Parceable** contiene las instrucciones sobre como producir un objeto en un flujo de data y luego recrear el objeto a partir de ese flujo de data.

Un ejemplo es este:

@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
  
 mLifecycleDisplay = (TextView) findViewById(R.id.*tv\_lifecycle\_events\_display*);  
  
 // *If savedInstanceState is not null and contains LIFECYCLE\_CALLBACKS\_TEXT\_KEY, set that text on our TextView* if(savedInstanceState != null)  
 {  
 if(savedInstanceState.containsKey(*LIFECYCLE\_CALLBACKS\_TEXT\_KEY*)) {  
 String savedContent = savedInstanceState.getString(*LIFECYCLE\_CALLBACKS\_TEXT\_KEY*);  
 mLifecycleDisplay.setText(savedContent);   
 }  
 }  
 logAndAppend(*ON\_CREATE*);  
}

En el onCreate se obtiene el *savedInstanceState* el cual preserva lo que se haya guardado ahí, en este caso se verifica que tiene algo y si posee la llave LIFECYCLE\_CALLBACKS\_TEXT\_KEY. Si es así, se recupera el valor de esta y se llena de nuevo el TextView mLifecycleDisplay.

Por otro lado, se sobre escribe el método **onSaveInstanceState** para guardar este valor. Es importante definir que existen varias sobrecargas de este método.

@Override  
public void onSaveInstanceState(Bundle outState) {

// Call super.onSaveInstanceState  
 super.onSaveInstanceState(outState);

// Call logAndAppend with the ON\_SAVE\_INSTANCE\_STATE String  
 logAndAppend(*ON\_SAVE\_INSTANCE\_STATE*);  
 // Put the text from the TextView in the outState bundle  
 String lifecycleDisplayTextViewContent = mLifecycleDisplay.getText().toString();  
 outState.putString(*LIFECYCLE\_CALLBACKS\_TEXT\_KEY*, lifecycleDisplayTextViewContent);  
  
 logAndAppend(lifecycleDisplayTextViewContent);  
}

Cuando se presiona el botón de home y se re-lanza el app, se obtiene un paso extra, **onRestart**. Esto significa que la actividad no fue desplegada momentáneamente y luego fue desplegada de nuevo, sin ser recreada.

## Preparing for Termination

Cuando el dispositivo se está quedando sin memoria u otros recursos, Android comienza a matar las apps que están en background que más recursos estén consumiendo. Desde el punto de perspectiva del sistema, onPause y onStop son señales de que el app podría ser terminada en cualquier momento, por lo que, para proveer una buena experiencia al usuario, es recomendable limpiar recursos de forma ordenada, como cerrar conexiones abiertas o sockets.

Los valores de los *EditText* se preservan automáticamente por el framework de Android durante un cambio de configuración como una rotación de pantalla, pero los TextView regulares no.

## AsyncTask y Loaders

Que pasa cuando el dispositivo está rotando mientras se hace una transacción de red. Cuando se destruye y recrea una actividad, la aplicación continúa corriendo, eso quiere decir que todos los hilos que estaban corriendo continúan su proceso.

Si la aplicación crea una transacción en el onCreate que ejecuta un AsyncTask y se rota el dispositivo o se hace algo más que cause que se reinicie la actividad, la nueva actividad va a crear otra asyncTask en la operación background, causando un uso extra de la red ya que ambos hilos estarían corriendo en paralelo y al usuario le tomaría más tiempo ver los resultados. Lo peor es que esos hilos entregan el resultado a un callback que es parte de la actividad y esos asyncTask mantienen a todos los activities “zombis” anteriores vivos mientras los hilos siguen corriendo causando presión extra en la memoria.

Los Loaders llegan a solucionar este problema. Se agregaron a Android desde Honeycomb. Los loaders proveen un framework que lleva a cabo cargas de datos asíncronas y están registrados por un ID con un componente llamado **LoaderManager**, el cual les permite vivir más allá del ciclo de vida de la actividad de la cual están asociados, previniendo cargas duplicadas en paralelo. Por lo que si se quiere cargar data en un hilo background se puede usar una implementación de un loader llamado **AsyncTaskLoader**.

El AsyncTaskLoader implementa la misma funcionalidad que un AsynTask pero como es un loader, su ciclo de vida es diferente. Con un AsyncTaskLoader, una vez que se ha rotado el dispositivo, el loader manager se asegura que el loader que está corriendo esté conectado al equivalente del onPostExecute del AsyncTaskLoader, la funcion **onLoadFinished**. El hilo sigue corriendo en la función background y una vez que finaliza, la actividad recibe una notificación a través del onLoadFinished.

Los AsyncTaskLoaders son una buena opción porque manejan de manera correcta los cambios del ciclo de vida de la actividad, envían el resultado a la actividad activa actual y previenen la duplicación de hilos background y ayudan a eliminar la duplicación de activities zombis.

## Crear Loaders

Para crear un loader se necesita:

1. Crear un **Loader ID** (constante int).
2. Implementar los Loader Callbacks
3. Inicializar el Loader con el LoaderManager.

Para trabajar con un AsyncTask en el ejemplo del githubSearchQuery, lo primero a hacer es implementar LoaderManager.LoaderCallbacks<String> en la clase de la actividad y sus respectivos métodos.

Luego se crea la constante para el ID del loader.

En el makeGithubSearchQuery() se crea un bundle que contenga el search query.

Se llena el onCreateLoader que retornara un AsyncTaskLoader<String> y ahí mismo que sobreescriba el **loadInBackground** y el **onStartLoading** donde se comprueban si los argumentos vienen nulos para hacer un return.

Al setear los args del **onCreateLoader** como final, se pueden accesar desde la clase interna.

Luego se mueve el código del onPreExecute al **onStartLoading** para setear el loading indicator como visible.

Luego se llena el método **loadInBackground** donde se obtiene el query url del bundle que se creó y si este query es nulo o vacío se retorna nulo. Después se copia el try..catch del asyncTask para hacer en el loadInBackground. Si se obtiene una excepción, se devuelve null.

Se copia el código del onPostExecute al **onLoadFinished** donde se valida, si el usuario no ingreso nada entonces no se hace la búsqueda.

Finalmente se usa el loaderManager para correr el loader dentro del makeGithubSearchQuery. Empezando por obtener el support library versión del loader manager (getSupportLoaderManager()).

Luego se intenta obtener nuestro loader. Si el loader no existe, se inicializa y si no, se reinicia.

Por último se copia el código que despliega el JSON del savedInstanceState.

Ejemplo:

// implement LoaderManager.LoaderCallbacks<String> on MainActivity  
public class MainActivity extends AppCompatActivity  
 implements LoaderManager.LoaderCallbacks<String> {  
  
 private static final String *TAG* = MainActivity.class.getSimpleName();  
  
 /\* A constant to save and restore the URL that is being displayed \*/  
 private static final String *SEARCH\_QUERY\_URL\_EXTRA* = "query";  
  
 // Create a constant int to uniquely identify your loader. Call it GITHUB\_SEARCH\_LOADER  
 private static final int *GITHUB\_SEARCH\_LOADER* = 126;

En el método onCreate se inicializa el loader usando un bundle nulo:

getSupportLoaderManager().initLoader(*GITHUB\_SEARCH\_LOADER*, null, this);

En el método makeGithubSearchQuery, donde se obtiene el texto a buscar, se construye la URL, ahora se crea un bundle para guardar dicha url e inicializar el Loader:

private void makeGithubSearchQuery() {  
 String githubQuery = mSearchBoxEditText.getText().toString();  
  
 // If no search was entered, indicate that there isn't anything to search for and return  
 if(githubQuery == null || githubQuery.equals("")) {  
 mUrlDisplayTextView.setText("No query entered, nothing to search for");  
 return;  
 }  
  
 URL githubSearchUrl = NetworkUtils.*buildUrl*(githubQuery);  
 mUrlDisplayTextView.setText(githubSearchUrl.toString());  
  
 // Create a bundle called queryBundle and add the string value of the URL  
 Bundle bundle = new Bundle();  
 bundle.putString(*SEARCH\_QUERY\_URL\_EXTRA*, githubSearchUrl.toString());  
  
 // Call getSupportLoaderManager and store it in a LoaderManager variable  
 LoaderManager loaderManager = getSupportLoaderManager();  
 // Get our Loader by calling getLoader and passing the ID we specified  
 Loader<String> githubSearchLoader = loaderManager.getLoader(*GITHUB\_SEARCH\_LOADER*);  
 // If the Loader was null, initialize it. Else, restart it.  
 if(githubSearchLoader == null){  
 Log.*d*(*TAG*, "makeGithubSearchQuery: initializing Loader");  
 loaderManager.initLoader(*GITHUB\_SEARCH\_LOADER*, bundle, this);  
 }  
 else{  
 Log.*d*(*TAG*, "makeGithubSearchQuery: restarting Loader");  
 loaderManager.restartLoader(*GITHUB\_SEARCH\_LOADER*, bundle, this);  
 }  
}

Uno de los métodos a sobre escribir del LoaderManager es el **onCreateLoader**, donde se retorna un **AsyncTaskLoader**<String> y en esta clase anónima interna se sobre escribe el **onStartLoading** donde se revisan los argumentos y se fuerza un load, básicamente este es come el onPreExecute del AsyntTask.

Luego está el **loadInBackground**, donde se obtiene el URL del bundle que se creó en el onCreate y se ejecuta la petición para obtener el response.

// Override onCreateLoader  
@RequiresApi(api = Build.VERSION\_CODES.*HONEYCOMB*)  
@Override  
public Loader<String> onCreateLoader(int id, final Bundle args) {  
 // Return a new AsyncTaskLoader<String> as an anonymous inner class with this as the constructor's parameter  
 return new AsyncTaskLoader<String>(this) {  
 @Override  
 protected void onStartLoading() {  
 // If args is null, return.  
 if(args == null) {  
 return;  
 }  
 // Show the loading indicator  
 mLoadingIndicator.setVisibility(View.*VISIBLE*);  
 // Force an asynchronous load instead of calling onStartLoading  
 forceLoad();  
 }  
 // Override loadInBackground  
 @Override  
 public String loadInBackground() {  
 // Get the String for our URL from the bundle passed to onCreateLoader  
 String searchQueryUrlString = args.getString(*SEARCH\_QUERY\_URL\_EXTRA*);  
 // If the URL is null or empty, return null  
 if(searchQueryUrlString == null || TextUtils.*isEmpty*(searchQueryUrlString)){  
 return null;  
 }  
 // Copy the try / catch block from the AsyncTask's doInBackground method  
 try {  
 // Get an URL string from the URL passed in the bundle and return the response  
 URL githubUrl = new URL(searchQueryUrlString);  
 return NetworkUtils.*getResponseFromHttpUrl*(githubUrl);  
 } catch (IOException e) {  
 e.printStackTrace();  
 return null;  
 }  
 }  
 };  
}

Otro método que se sobre escribe es el **onLoadFinished**, que es cuando la ejecución de la tarea en background ha terminado y aquí se muestran los resultados en el TextView correspondiente.

// Override onLoadFinished  
@Override  
public void onLoadFinished(Loader<String> loader, String data) {  
 // Hide the loading indicator  
 mLoadingIndicator.setVisibility(View.*INVISIBLE*);  
  
 // If the data is not null or empty, show the result or an error otherwise  
 if (data != null && !data.equals("")) {  
 showJsonDataView();  
 mSearchResultsTextView.setText(data);  
 } else {  
 showErrorMessage();  
 }  
}

El otro método que se sobre escribe es **onLoaderReset** pero a ese no hace falta hacerle nada para el ejemplo.

Con este código no hace falta la clase GithubQueryTask que se creó en las clases anteriores.

## Caching with Loaders

Los loaders están atados al ciclo de vida de la aplicación y estos manejan los cambios de configuración automáticamente, como con la rotación. Están diseñados para recargarse si el usuario se sale de la actividad y luego regresa. Se puede evitar la carga extra al guardar en cache y reenviar el resultado existente.

Para esto se puede usar el método **deliverResult**, el cual es un paso intermedio entre el **loadInBackground** y el **onLoadFinished** y se puede utilizar para hacer uso de cache y evitar hacer la carga de background.

En el ejemplo se crea una variable para almacenar en cache, en el AsyncTaskLoader.

Se sobre escribe el método deliverResult y ahí se guarda el valor en la variable de cache

En el onStartingLoading se revisa si la variable tiene datos y si es así, se envían esos datos, si no, se hace el load:

public Loader<String> onCreateLoader(int id, final Bundle args) {  
 return new AsyncTaskLoader<String>(this) {  
  
 // Create a String member variable called mGithubJson that will store the raw JSON  
 String mGithubJson;  
  
 @Override  
 protected void onStartLoading() {  
  
 /\* If no arguments were passed, we don't have a query to perform. Simply return. \*/  
 if (args == null) {  
 return;  
 }  
  
 // If mGithubJson is not null, deliver that result. Otherwise, force a load  
 if(mGithubJson != null)  
 {  
 deliverResult(mGithubJson);  
 }  
 else {  
 /\*  
 \* When we initially begin loading in the background, we want to display the  
 \* loading indicator to the user  
 \*/  
 mLoadingIndicator.setVisibility(View.*VISIBLE*);  
 forceLoad();  
 }  
 }

……

En este caso la variable mGithubJson es la que se usara para almacenar en cache los datos.

// Override deliverResult and store the data in mGithubJson  
// Call super.deliverResult after storing the data  
@Override  
public void deliverResult(String data) {  
 mGithubJson = data;  
 super.deliverResult(data);  
}

El bundle no siempre es necesario que lleva data, si los datos se pueden obtener en el mismo loadInBackground por ejemplo.

# Lesson 06 - Preferences

## Visualizer

Cuando se solicitan permisos, una vez que el usuario ha respondido a esa solicitud, la respuesta a este request se puede manejar en un método llamado **onRequestPermissionResult**, el cual uno sobre escribe y dependiendo del código del permiso uno puede ejecutar las acciones que necesite.

En los siguientes ejemplos, se estará trabajando con una aplicación que muestra tres elementos (un cuadro, un triángulo y un círculo) que se mueven en una circunferencia alrededor del centro de la actividad y van marcando un path de acuerdo a la frecuencia de una pista de audio.

## Data Persistance

Antes de empezar a ver los settings, es importante hablar de la persistencia de los datos, que es guardar algunos datos en el dispositivo. Hay 5 formas de hacer esto:

|  |  |  |
| --- | --- | --- |
| ***Opción*** | ***Tipos de datos guardados*** | ***Cantidad de tiempo guardado*** |
| onSaveInstanceState | Usando key-value pairs.  Usualmente para guardar el estado durante eventos como rotación del app. | Mientras el app está abierta.  Al presionar el back button o se sale del app, esta información se pierde |
| SharedPreferences | Guarda key-value pairs en un archivo.  Las llaves son siempre strings y los valores son valores primitivos como strings o números. | Se mantiene entre uso de apps y reinicios de teléfono. |
| SQLite Database | Datos más complicados y organizados como textos, números, booleans, entre otros. | Se mantiene entre el uso de apps y reinicio del teléfono. |
| Internal / External Storage | Datos multimedia o más grandes | Se mantiene entre el uso de apps y reinicio del teléfono. |
| Servidores (por ejemplo, Firebase) | Datos que pueden ser accesados por múltiples dispositivos | Entre apps, reinicios de dispositivo, borrar el app, usar otro device, etc. |

## Preference Fragments

Como los SharedPreferences usualmente se usan para los settings de los apps, suelen usarse con otra parte del framework de Android que se usa para crear una IU para las activitties de settings, llamado Preference Fragments.

### Fragments

Los Fragments son clases que representan una pieza modular y reusable de una Actividad.

Por ejemplo, en un teléfono, un app podría mostrar una lista de libros y al darle click a uno se abriría una pantalla con la descripción del libro. En una Tablet por tener más espacio, se podría tener tanto la lista de los libros como la vista con la descripción del libro seleccionado. Entonces, en lugar de tener varias activities, se pueden tener en fragmentos y reusarlos.

En las pantallas de settings, generalmente se tiene un Activity que puede contener una o más subclases de fragmentos, llamada PreferenceFragment. Esta está hecha específicamente para trabajar con los settings.

Esta clase se llena a partir de un archivo XML de preferencias. Al cambiar un valor en el UI, este valor se actualiza automáticamente en el archivo XML.

***NOTA***

Cuando se crea un menú que abre otro activity, y se quiere que ese activity sea un hijo de quien la llama, en el Android Manifest se setean algunos valores a la definición del activity hijo, por ejemplo, un label, la propiedad parentActivityName y se agrega el meta-data donde se define el activity padre

<activity android:name=".VisualizerActivity"  
 android:launchMode="singleTop">  
 <intent-filter>  
 <action android:name="android.intent.action.MAIN" />   
 <category android:name="android.intent.category.LAUNCHER" />  
 </intent-filter>  
</activity>  
<activity android:name=".SettingsActivity"  
 android:label="@string/action\_settings"  
 android:parentActivityName=".VisualizerActivity">  
 <meta-data android:name="android.support.PARENT\_ACTIVITY"  
 android:value=".VisualizerActivity"></meta-data>  
</activity>

El android:label permite definir una etiqueta para la actividad la cual es desplegada en pantalla cuando se despliega la actividad, generalmente junto con el icono. Si no se especifica, se toma el label del app.

El launchMode = “singleTop” que se pone al VisualizerActivity se asegura de que cuando uno navegue de vuelta del SettingsActivity a esta actividad no rehaga el VisualizerActivity.

En la actividad de settings, se puede obtener una referencia del action bar con el método **getSupportActionBar**() y si no es nulo, llamar al **setDisplayHomeAsUpEnabled**(true), lo cual haría que al presionar el back button este se vaya hasta el home del app, es decir, a la actividad principal.

@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_settings*);  
 ActionBar actionBar = this.getSupportActionBar();  
 if(actionBar != null){  
 actionBar.setDisplayHomeAsUpEnabled(true);  
 }  
  
}  
  
@Override  
public boolean onOptionsItemSelected(MenuItem item) {  
 int id = item.getItemId();  
 // Check if the home button was pressed  
 if(id == android.R.id.*home*){  
 // Navigate up and finish the current activity  
 NavUtils.*navigateUpFromSameTask*(this);  
 }  
 return super.onOptionsItemSelected(item);  
}

El método **NavUtils.navigateUpFromSameTask** permite navegar hacia arriba en la cola de actividades y cerrar la actividad actual mientras el padre y la actividad actual estén corriendo bajo la misma tarea.

Para trabajar con un preference fragment, si se usa la versión 7 del support library, en el build.gradle se debe agregar la dependencia:

dependencies **{** implementation fileTree(dir: 'libs', include: ['\*.jar'])  
 implementation 'com.android.support:appcompat-v7:28.0.0'   
 implementation 'com.android.support:preference-v7:28.0.0'  
**}**

Y si se esta usando androidx

implementation "androidx.legacy:legacy-preference-v14:1.0.0"

implementation "androidx.preference:preference:1.1.1"

Para trabajar con preferences, se ocupa un archivo XML, el cual se puede agregar manualmente. Cualquier tag de preferencias de XML es siempre un <PreferenceScreen>. Dentro de este tag, se pueden anidar otros tags de preferencias o de otros tipos, lo cual genera una jerarquía de preferencias.

Por ejemplo, si se quiere usar un checkbox para mostrar o no el bass circle de la actividad, se usa un CheckBoxPreference. En este tag, se definen 5 atributos:

* defaultValue: el valor por defecto para el setting
* key: el key para identificar el setting
* summaryOff: descripción cuando el valor es off
* summaryOn: descripción cuando el valor es on
* title: el titulo a mostrar

<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">  
 <CheckBoxPreference  
 android:defaultValue="true"  
 android:key="show\_bass"  
 android:summaryOff="Hidden"  
 android:summaryOn="Shown"  
 android:title="Show Bass"></CheckBoxPreference>  
</PreferenceScreen>

En la clase que se va a usar para el Fragment de los settings, en este caso llamada SettingsPreference, se extiende la clase **PreferenceFragmentCompat**. El método que hay que implementar, el **onCreatePreferences** uno agrega las preferencias usando el método **addPreferencesFromResource** y enviando el id del XML que se creó.

public class SettingsFragment extends PreferenceFragmentCompat {  
 @Override  
 public void onCreatePreferences(Bundle bundle, String s) {  
 addPreferencesFromResource(R.xml.*pref\_visualizer*);  
 }  
}

Para mostrar esto en el activity, se quita el layout por defecto y se agrega un fragment tag

Activity\_settings.xml

<fragment xmlns:android="http://schemas.android.com/apk/res/android"  
 android:id="@+id/activity\_settings"  
 android:name="android.example.com.visualizerpreferences.SettingsFragment"  
 android:layout\_height="match\_parent"  
 android:layout\_width="match\_parent">  
</fragment>

Ademas para que el fragment se pueda visualizar, es necesario agregar un item en los styles para el preference. Por ejemplo asi:

Style.xml

<style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">  
 <!-- Customize your theme here. -->  
 <item name="colorPrimary">@color/colorPrimary</item>  
 <item name="colorPrimaryDark">@color/colorPrimaryDark</item>  
 <item name="colorAccent">@color/colorAccent</item>  
 <!-- *TODO (6) Add a theme for the preference* -->  
 <item name="preferenceTheme">@style/PreferenceThemeOverlay</item>  
</style>

## Reading from SharedPreferences

Al guardar los valores de los preferences usando la actividad de settings, estos valores se actualizan en un archivo. Para poder leer este archivo y usar esos valores en el app se necesita:

1. Obtener una instancia de shared preferences usando **PreferenceManager.getDefaultSharedPreferences**. Hay otras opciones para esto, pero solo si se usan multiples archivos de preferencias en lugar del default.
2. Con esta instancia, hay varios métodos get para obtener la data del archivo, dependiendo del tipo de dato. Para esto se ocupa solo el key con el que se guardó el valor. Estos tipos no tienen como verificar el tipo, así que, si usa un getInt para obtener un valor string o de otro tipo, el app va a crashear.

// Get a reference to the default shared preferences from the PreferenceManager class  
SharedPreferences preferences = PreferenceManager.*getDefaultSharedPreferences*(this);  
// Get the value of the show\_bass checkbox preference and use it to call setShowBass  
mVisualizerView.setShowBass(preferences.getBoolean("show\_bass",true));

1. El código para escribir directamente al shared preferences es similar al de leer, solo que se obtiene una referencia usando **sharedPreferences.edit()**. Y se usa un método put según el tipo de dato.

## Usando Recursos

Lo ideal es tener en recursos todos los keys y los string a mostrar en los settings. En los recursos se puede usar una propiedad llamada **translatable** que es boolean y si se pone en false, al usar localization, estos valores no van a traducirse, esto es más que nada para keys, los cuales no hace falta traducir.

<!-- Key for SharedPreference Show Bass -->  
<string name="pref\_show\_bass\_key" translatable="false">show\_bass</string>  
  
<!-- Value in SharedPreference for Show Bass title -->  
<string name="pref\_show\_bass\_title">Show Bass</string>

Incluso los booleanos que se usan para valores por default, se pueden guardar en archivos de recursos:

// Get all of the values from shared preferences to set it up  
SharedPreferences sharedPreferences = PreferenceManager.*getDefaultSharedPreferences*(this);  
mVisualizerView.setShowBass(sharedPreferences.getBoolean(getString(R.string.*pref\_show\_bass\_key*),  
 getResources().getBoolean(R.bool.*pref\_show\_bass\_default*) ));

## Preference Change Listener

En el ciclo de vida de un app, cuando se navega de una actividad a otra, los métodos onPause y onStop se llaman. Cuando se vuelve a la actividad, el onStart y onResume se llaman. El onCreate no se llama a no ser que uno rote la actividad o reabra la actividad después de cerrarla.

Una opción para leer las preferencias es hacerlo en el onStart y onResume también, pero hay otra mejor forma. El objeto sharedPreferences permite que uno registre un objeto que implemente **onSharedPreferenceChangeListener**. Este objeto se llama cuando un valor en los SharedPreferences cambia. De esta forma, uno puede actualizar la UI solo y solo si una preferencia en el archivo cambio.

Para esto se necesita:

1. Determinar la actividad a llamar cuando la preferencia cambia.
2. Implementar **onSharedPreferenceChangeListener** en la actividad, específicamente un método llamado **onSharedPreferenceChange**
3. Ligar el onSharedPreferenceChangeListener con el objeto **sharedPreferenceObject** que debe escuchar. Para eso se usa el .**registerOnSharedPreferenceChangeListener**
4. Deregistrar el listener cuando la actividad se apague usando .**unregisterOnSharedPreferenceListener**.

Ejemplo:

// Implement OnSharedPreferenceChangeListener  
public class VisualizerActivity extends AppCompatActivity implements SharedPreferences.OnSharedPreferenceChangeListener {

El metodo a implmentar:

// Override the onSharedPreferenceChanged method and update the show bass preference  
@Override  
public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key) {  
 // Check what preference changed and update the object in UI  
 if(key.equals(getString(R.string.*pref\_show\_bass\_key*))){  
 mVisualizerView.setShowBass(sharedPreferences.getBoolean(key,  
 getResources().getBoolean(R.bool.*pref\_show\_bass\_default*)));  
 }  
}

Registrar el listener en el onCreate:

private void setupSharedPreferences() {  
 // Get all of the values from shared preferences to set it up  
 **SharedPreferences sharedPreferences = PreferenceManager.*getDefaultSharedPreferences*(this);** mVisualizerView.setShowBass(sharedPreferences.getBoolean(getString(R.string.*pref\_show\_bass\_key*),  
 getResources().getBoolean(R.bool.*pref\_show\_bass\_default*)));  
 mVisualizerView.setShowMid(true);  
 mVisualizerView.setShowTreble(true);  
 mVisualizerView.setMinSizeScale(1);  
 mVisualizerView.setColor(getString(R.string.*pref\_color\_red\_value*));  
 // Register the listener  
 sharedPreferences.registerOnSharedPreferenceChangeListener(this);  
}

Desregistrar el listener en el onDestroy:

// Override onDestroy and unregister the listener  
@Override  
protected void onDestroy() {  
 super.onDestroy();  
 **PreferenceManager.*getDefaultSharedPreferences*(this)  
 .unregisterOnSharedPreferenceChangeListener(this);**}

## List Preference

Los ListPreference permiten elegir una opción de una lista. Entre los atributos que los diferencian de los otros, esta:

* **entries**: un arreglo con los labels que se ven en la lista
* **entryValues**: un arreglo con los valores que se almacenan en los sharedPreferences

Para los arrays, se puede crear otro archivo arrays.xml donde se definen los diferentes valores en array que se necesiten para estos preferences y otros.

Ejemplo:

Valores en strings para el título, key, labels y valores de las opciones

<!-- Add string values for the list preference -->  
<!--Key for the color preference-->  
<string name="pref\_color\_key">color</string>  
<!-- Label for the color preference option-->  
<string name="pref\_color\_title">Shape Color</string>  
<!-- Label for red color preference -->  
<string name="pref\_color\_label\_red">Red</string>  
<!-- Label for blue color preference -->  
<string name="pref\_color\_label\_blue">Blue</string>  
<!-- Label for green color preference -->  
<string name="pref\_color\_label\_green">Green</string>  
  
<!-- Value in SharedPreferences for red color option -->  
<string name="pref\_color\_red\_value" translatable="false">red</string>  
<!-- Value in SharedPreferences for blue color option -->  
<string name="pref\_color\_blue\_value" translatable="false">blue</string>  
<!-- Value in SharedPreferences for green color option -->  
<string name="pref\_color\_green\_value" translatable="false">green</string>

Arrays.xml:

<resources>  
 <!-- Array for the color labels -->  
 <array name="pref\_color\_option\_labels">  
 <item>@string/pref\_color\_label\_red</item>  
 <item>@string/pref\_color\_label\_blue</item>  
 <item>@string/pref\_color\_label\_green</item>  
 </array>  
 <!-- Array for the color values -->  
 <array name="pref\_color\_option\_keys">  
 <item>@string/pref\_color\_red\_value</item>  
 <item>@string/pref\_color\_blue\_value</item>  
 <item>@string/pref\_color\_green\_value</item>  
 </array>  
</resources>

El pref\_visualizer.xml que es el que se usa para los settings tendría ahora un ListPreference asi:

<!-- A list preference -->  
<ListPreference  
 android:defaultValue="@string/pref\_color\_red\_value"  
 android:entries="@array/pref\_color\_option\_labels"  
 android:entryValues="@array/pref\_color\_option\_keys"  
 android:key="@string/pref\_color\_key"  
 android:title="@string/pref\_color\_title"/>

Para cargar un color desde las preferencias, se puede tener un método como este:

private void loadColorFromPreference(SharedPreferences preferences){  
 mVisualizerView.setColor(preferences.getString(getString(R.string.*pref\_color\_key*),  
 getString(R.string.*pref\_color\_red\_value*));  
}

Y ese método se llamaría desde el onCreate y en el listener

## Preference Summary

Se refiere a un texto que pueda mostrar el valor del setting, en el caso de los list preferences, al ser una lista, es más difícil saber cuál valor seleccionó el usuario. Para eso se puede hacer que el fragmento de settings actualice el summary programáticamente. Para esto se puede:

* Implementar **OnSharedPreferencesChangeListener** en el fragmento (SettingsFragment)
* Obtener la referencia de las preferencias, usando **getPreferenceScreen().getSharedPreferences()**
* Se puede obtener los PreferenceScreen para saber el total de preferencias en pantalla y poder iterar sobre ellos. Al hacer esto se puede comprobar el tipo de preferencia
* Si la preferencia es un ListPreference, se puede obtener el label asociado con el valor usando el **findIndexOfValue** y **getEntries**(), ambos del objeto Preference. El getEntries obtiene todos los labels mientras el findIndexOfValue obtiene el índice del arreglo según el key
* Para que el valor de summary cambie al cambiar el valor también se debe registrar y desregistrar el listener en el SettingsFragment.

Clase SettingsFragment:

package android.example.com.visualizerpreferences;  
  
/\*  
 \* Copyright (C) 2016 The Android Open Source Project  
 \*  
 \* Licensed under the Apache License, Version 2.0 (the "License");  
 \* you may not use this file except in compliance with the License.  
 \* You may obtain a copy of the License at  
 \*  
 \* http://www.apache.org/licenses/LICENSE-2.0  
 \*  
 \* Unless required by applicable law or agreed to in writing, software  
 \* distributed under the License is distributed on an "AS IS" BASIS,  
 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
 \* See the License for the specific language governing permissions and  
 \* limitations under the License.  
 \*/  
  
import android.content.SharedPreferences;  
import android.content.SharedPreferences.OnSharedPreferenceChangeListener;  
import android.os.Bundle;  
import android.support.v7.preference.CheckBoxPreference;  
import android.support.v7.preference.EditTextPreference;  
import android.support.v7.preference.ListPreference;  
import android.support.v7.preference.Preference;  
import android.support.v7.preference.PreferenceFragmentCompat;  
import android.support.v7.preference.PreferenceScreen;  
import android.widget.Toast;  
  
// Implement OnSharedPreferenceChangeListener  
public class SettingsFragment extends PreferenceFragmentCompat implements OnSharedPreferenceChangeListener {  
  
 @Override  
 public void onCreatePreferences(Bundle bundle, String s) {  
  
 // Add visualizer preferences, defined in the XML file in res->xml->pref\_visualizer  
 addPreferencesFromResource(R.xml.*pref\_visualizer*);  
  
 // Get the preference screen, get the number of preferences and iterate through  
 // all of the preferences if it is not a checkbox preference, call the setSummary method  
 // passing in a preference and the value of the preference  
 SharedPreferences sharedPreferences = getPreferenceScreen().getSharedPreferences();  
 // Get the preferences screen to get the total of items  
 PreferenceScreen preferenceScreen = getPreferenceScreen();  
 int count = preferenceScreen.getPreferenceCount();  
 for (int i=0; i<count; i++){  
 Preference pref = preferenceScreen.getPreference(i);  
 // if preference is not a checkbox  
 if( !(pref instanceof CheckBoxPreference)){  
 // Get the value or key of the preference  
 String value = sharedPreferences.getString(pref.getKey(), "");  
 setPreferenceSummary(pref, value);  
 }  
 }  
 }  
 // Override onSharedPreferenceChanged and, if it is not a checkbox preference,  
 // call setPreferenceSummary on the changed preference  
 @Override  
 public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key) {  
 // Get the preference by the key  
 Preference preference = findPreference(key);  
 if(null != preference){  
 // if the preference is not a checkbox  
 if(!(preference instanceof CheckBoxPreference)){  
 String value = sharedPreferences.getString(preference.getKey(), "");  
 setPreferenceSummary(preference, value);  
 }  
 }  
 }  
  
 // Method which takes a Preference and String value as parameters.  
 // This method should check if the preference is a ListPreference and, if so, find the label  
 // associated with the value. You can do this by using the findIndexOfValue and getEntries methods  
 // of Preference.  
 private void setPreferenceSummary(Preference preference, String value){  
 // Check the type of the preference using instanceof  
 if(preference instanceof ListPreference){  
 ListPreference listPreference = (ListPreference)preference;  
 //Getting the index of the value passed in the parameters  
 int index = listPreference.findIndexOfValue(value);  
 if(index >= 0){  
 // Sets the summary for the preference  
 listPreference.setSummary(listPreference.getEntries()[index]);  
 }  
 }  
 }  
  
 // Register and unregister the OnSharedPreferenceChange listener (this class) in  
 // onCreate and onDestroy respectively.  
 @Override  
 public void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 getPreferenceScreen().getSharedPreferences()  
 .registerOnSharedPreferenceChangeListener(this);  
 }  
  
 @Override  
 public void onDestroy() {  
 super.onDestroy();  
 getPreferenceScreen().getSharedPreferences()  
 .unregisterOnSharedPreferenceChangeListener(this);  
 }  
}

## Edit Text Preference

Otro tipo de preference donde el usuario puede editar el valor en un edit text.

Trabajar con un EditTextPreference es parecido a un CheckBoxPreference pero con el detalle de que tampoco muestra el summary por defecto.

En el XML de los settings se agrega un EditTextPreference como este:

<EditTextPreference  
 android:key="@string/pref\_size\_key"  
 android:defaultValue="@string/pref\_size\_default"  
 android:title="@string/pref\_size\_label" />

En el VisualizerActivity, donde se usan los settings, se puede agregar un método para cargar el valor desde los SharedPreference:

// Loads the size preference from the SharedPreferences  
private void loadSizeFromPreference(SharedPreferences sharedPreferences){  
 // Get a float from the string value of the shared preferences for the size  
 float minSize = Float.*parseFloat*(sharedPreferences.getString( getString(R.string.*pref\_size\_key*),  
 getString(R.string.*pref\_size\_default*)));  
 mVisualizerView.setMinSizeScale(minSize);  
}

Este método se llamada desde el onCreate y también en el onSharedPreferenceChanged

// Updates the screen if the shared preferences change. This method is required when you make a  
// class implement OnSharedPreferenceChangedListener  
@Override  
public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key) {  
 if (key.equals(getString(R.string.*pref\_show\_bass\_key*))) {  
 mVisualizerView.setShowBass(sharedPreferences.getBoolean(key, getResources().getBoolean(R.bool.*pref\_show\_bass\_default*)));  
 } else if (key.equals(getString(R.string.*pref\_show\_mid\_range\_key*))) {  
 mVisualizerView.setShowMid(sharedPreferences.getBoolean(key, getResources().getBoolean(R.bool.*pref\_show\_mid\_range\_default*)));  
 } else if (key.equals(getString(R.string.*pref\_show\_treble\_key*))) {  
 mVisualizerView.setShowTreble(sharedPreferences.getBoolean(key, getResources().getBoolean(R.bool.*pref\_show\_treble\_default*)));  
 } else if (key.equals(getString(R.string.*pref\_color\_key*))) {  
 loadColorFromPreferences(sharedPreferences);  
 } else if(key.equals(getString(R.string.*pref\_size\_key*))){  
 loadSizeFromPreference(sharedPreferences);  
 }  
}

En el SettingsFragment También se verifica que cuando se llame al setPreferenceSummary, donde se actualizan los summary para los preferences, que se actualice el del EditTextPreference:

private void setPreferenceSummary(Preference preference, String value) {  
 // Don't forget to add code here to properly set the summary for an EditTextPreference  
 if (preference instanceof ListPreference) {  
 // For list preferences, figure out the label of the selected value  
 ListPreference listPreference = (ListPreference) preference;  
 int prefIndex = listPreference.findIndexOfValue(value);  
 if (prefIndex >= 0) {  
 // Set the summary to that label  
 listPreference.setSummary(listPreference.getEntries()[prefIndex]);  
 }  
 } else if( preference instanceof EditTextPreference){  
 preference.setSummary(value);  
 }  
}

El summary para este tipo de preference es sencillo, ya que no hay listas o arrays involucrados.

### Constraints

Para evitar que un usuario ingrese valores no permitidos, en este caso para el EditTextPreference, se puede usar el **PreferenceChangeListener**, que es diferente al SharedPreferenceChangeListener.

Diferencias:

* SharedPreferencesChangeListener se dispara después de que cualquier valor es salvado en el archivo SharedPreferences
* PreferenceChangeListener se dispara antes de que un valor sea salvado en el archivo SharedPreferences. Por esto se puede usar para validar el input.
* PreferenceChangeListener está ligado a una preferencia.

En la actividad se implementa **Preference.OnPreferenceChangeListener**, se sobre escribe el método onPreferenceChange(Preference preference, Object newValue)

Este método devuelve true o false dependiendo si la preferencia debe ser guardada.

Ejemplo:

// Implement OnPreferenceChangeListener  
public class SettingsFragment extends PreferenceFragmentCompat implements  
 OnSharedPreferenceChangeListener, Preference.OnPreferenceChangeListener {

En el onCreatePreferences se adjunta el listener a la preferencia:

@Override  
public void onCreatePreferences(Bundle bundle, String s) {  
  
 // Add visualizer preferences, defined in the XML file in res->xml->pref\_visualizer  
 addPreferencesFromResource(R.xml.*pref\_visualizer*);  
  
 SharedPreferences sharedPreferences = getPreferenceScreen().getSharedPreferences();  
 PreferenceScreen prefScreen = getPreferenceScreen();  
 int count = prefScreen.getPreferenceCount();  
  
 // Go through all of the preferences, and set up their preference summary.  
 for (int i = 0; i < count; i++) {  
 Preference p = prefScreen.getPreference(i);  
 // You don't need to set up preference summaries for checkbox preferences because  
 // they are already set up in xml using summaryOff and summary On  
 if (!(p instanceof CheckBoxPreference)) {  
 String value = sharedPreferences.getString(p.getKey(), "");  
 setPreferenceSummary(p, value);  
 }  
 }  
 // Add the OnPreferenceChangeListener specifically to the EditTextPreference  
 Preference preference = findPreference(getString(R.string.*pref\_size\_key*));  
 preference.setOnPreferenceChangeListener(this);  
}

Se sobre escribe el método **onPreferenceChange** y aquí se hacen las validaciones correspondientes para el campo:

// Override onPreferenceChange. This method should try to convert the new preference value  
// to a float; if it cannot, show a helpful error message and return false. If it can be converted  
// to a float check that that float is between 0 (exclusive) and 3 (inclusive). If it isn't, show  
// an error message and return false. If it is a valid number, return true.  
@Override  
public boolean onPreferenceChange(Preference preference, Object newValue) {  
 //Create a toast for the error message  
 Toast error = Toast.*makeText*(getContext(), R.string.*err\_size\_invalid*, Toast.*LENGTH\_LONG* );  
   
 // Validate that the size preference is the one to be updated  
 String sizeKey = getString(R.string.*pref\_size\_key*);  
 if(preference.getKey().equals(sizeKey)){  
 String stringSize = (String)newValue;  
 try{  
 float size = Float.*parseFloat*(stringSize);  
 if(size >3 || size <= 0){  
 error.show();  
 return false;  
 }  
 }  
 catch (NumberFormatException ex){  
 error.show();  
 return false;  
 }  
 }  
 return true;  
}

### Finalizar la actividad

Hay un par de métodos propios de la clase Activity en Android para finalizar la actividad

* **finish()**: destruye la actividad y no se puede acceder a ella hasta que se crea de nuevo
* **onBackPressed()**: se llama cuando la actividad ha detectado que el usuario presiono el back key. La implementación default simplemente finaliza la actividad actual y vuelve a la actividad o fragmento anterior.

### PreferenceManager

Para obtener los defaultSharedPreferences en una clase que sea diferente al Settings Fragment es decir, que no extienda al PreferenceFragmentCompat, se puede utilizar el objeto **PreferenceManager**

Por ejemplo:

public static String getPreferredWeatherLocation(Context context) {  
 // COMPLETED (1) Return the user's preferred location  
  
 SharedPreferences sharedPreferences = PreferenceManager.*getDefaultSharedPreferences*(context);  
 String location = context.getString(R.string.*pref\_location\_key*);  
 String defaultLocation = context.getString(R.string.*pref\_location\_default*);  
 return sharedPreferences.getString(location, defaultLocation);  
}

## Adding Fragments to a View

Cuando se tiene la opción de usar Fragments para reutilizar código, a parte del Preference Fragment, hay otras dos opciones, un DialogFragment y los Fragments como tal.

Para poder usar un fragment se debe comenzar por agregar las dependencias:

dependencies **{**  
 implementation 'androidx.fragment:fragment:1.2.5'

**}**

Se puede utilizar el wizard para agregar un Fragment o bien, hacerlo a mano. Lo primero sería crear un layout para el fragment con los elementos que se necesite. Luego, en la actividad donde se va a mostrar dicho Fragment, hay que agregar un contenedor:

<androidx.fragment.app.FragmentContainerView  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content"  
 android:id="@+id/fc\_discover\_movies" />

Si se quiere cargar el Fragment desde el XML habría que agregar otra propiedad al FragmentContainerView que es el name, donde se pondría el nombre del fragment. Si se quiere hacer de forma programada, se ocupa un **transaction** en la clase del activity. Ejemplo:

@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
 // Only when the savedInstance is null to ensure the fragment is added only once  
 if(savedInstanceState == null){  
 getSupportFragmentManager().beginTransaction()  
 .setReorderingAllowed(true)  
 .add(R.id.*fc\_discover\_movies*, DiscoverMovie.class, null)  
 .commit();  
 }  
}

En el onCreate de la actividad se valida primero que se haga cuando el savedInstanceState sea nulo para crear el fragment solo una vez. Cuando ocurre un cambio de configuración y la actividad es recreada, savedInstanceState ya no es nulo, pero no se necesita recrear el fragmento ya que este es automáticamente restablecido desde el savedInstanceState.

Se usa el ***getSupportFragmentManager*** para poder iniciar la transacción con el beginTransaction(), luego se llama al setReordingAllowed con true, se llama al **add** para cargar el fragment usando el id del fragment container, la clase del Fragment a cargar y un bundle, en este caso nulo porque no se van a pasar parámetros al Fragment, pero si se pasara, se puede crear un bundle, llenarlo y pasarlo.

El ***setReorderingAllowed*** se recomienda ponerlo siempre en true, ya que, al estar en true, se asegura que, si múltiples transacciones están corriendo juntas, cualquier fragmento intermediario no va a ir a través de cambios del ciclo de vida o sus animaciones o transacciones se van a ejecutar ([https://developer.android.com/guide/fragments/transactions#reordering](https://developer.android.com/guide/fragments/transactions" \l "reordering) ).

Información de la documentación oficial:

[https://developer.android.com/guide/fragments/create#java](https://developer.android.com/guide/fragments/create" \l "java)

### Fragment Lifecycle

El cliclo de vida de un fragment es un poco diferente al de un Activity. El estado en el que se encuentra un Fragment es determinado por su **FragmentManager**. El estado del ciclo de vida de un Fragment nunca puede ser mayor al de su padre.

Estos son los estados y Callbacks que se pueden usar para dichos estados en un Fragment:

fragment lifecycle states and their relation both the fragment's
            lifecycle callbacks and the fragment's view lifecycle

Más información aquí: <https://developer.android.com/guide/fragments/lifecycle>

Para obtener las referencia de los views del fragment y poder trabajar con ellos, se debe optar por el callback onViewCreated() y no el onCreate() o el onCreateView(), ya que en esta este momento es cuando la vista del fragmento ha sido realmente creado. Este mismo método ofrece un objeto view para poder trabajar con él. Por lo que en este método hay un par de formas de obtener la referencia de los views, usando el objeto view o bien el método getView(). Por ejemplo:

@Override  
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {  
 super.onViewCreated(view, savedInstanceState);  
  
 // Get the instance reference of the controls  
 // The getView can only be used in this method  
 *mLoadingIndicator* = (ProgressBar)view.findViewById(R.id.*pb\_dm\_loading\_indicator*);  
 *mErrorMessage* = getView().findViewById(R.id.*tv\_dm\_error\_message\_display*);  
 mRecyclerView = getView().findViewById(R.id.*rv\_discover\_movies*);  
  
}

Es importante tener en cuenta que las variables que aquí se usan para referenciar a los views deberían removerse en el onDestroy del fragment para permitir que el garbage collector recolecte el fragment.

# Lesson 07 – Content Providers

Hay 4 componentes principales cuando se trabaja con Android que uno debe conocer bien: Activities, Services, Broadcast Receivers y Content Providers.

Un Content Provider es una clase que se sitúa entre una aplicación y su fuente de datos y su trabajo es proveer acceso sencillo a dicha fuente de datos. Un proveedor de contenido administra el acceso a un repositorio central de datos. Un proveedor forma parte de una aplicación para Android y a menudo proporciona su propia IU para trabajar con los datos. No obstante, los proveedores de contenido están principalmente orientados a que los usen otras aplicaciones que acceden al proveedor usando un objeto de cliente del proveedor.

## Ventajas del Content Provider

Los Content Provider ofrecen varias ventajas:

1. Es un nivel extra de abstracción que permite a los developer cambiar la fuente de datos sin la necesidad de cambiar el código de la aplicación que accede a esa fuente de datos.
2. Algunas clases importantes como Loaders y Cursor Adapters usan Content Providers. Si uno quiere usar un Loader uno necesita hacer que la data este accesible a través de un Content Provider.
3. Permite que muchas apps accedan, usen y modifiquen un data source de forma segura. De no usar un content provider, esa data se limitaría a tu app.

Los Content Providers son clases a las que se les puede hacer peticiones, por lo que se puede agregar código para “sanitizar” el código y proteger los datos cuando se necesite.

Mas info aquí: <https://developer.android.com/guide/topics/providers/content-provider-basics?hl=es-419>

## Content Provider Permissions

Generalmente, lo primero que se hace al trabajar con Content Providers es solicitar permisos de ser necesario.

Los permisos pueden ser de lectura o escritura, por ejemplo, esta es la solicitud de permisos para un content provider de una aplicación llamada DroidTermsExample para leer los términos conocidos por el usuario:

<!-- Add the correct permission to access the DroidTermsExample content provider -->  
<uses-permission android:name="com.example.udacity.droidtermsexample.TERMS\_READ" />

Los permisos son una característica de los Content Providers, los cuales están protegidos por el mismo sistema de permisos que despliegan un pop-up que le informan al usuario lo que hace el app.

## The ContentResolver

Después de obtener los permisos para el Content Provider, el segundo paso es obtener el ContentResolver.

Uno tiene muchos Content Providers en el teléfono. Cuando uno baja apps para almacenar data, estos usan Content Providers. Puede haber otras apps usando content providers en paralelo. Un Content Resolver actúa como un intermediario entre cada app y el content provider o providers a los que quiera acceder. Maneja la comunicación entre procesos y mantiene todo sincronizado y corriendo. Por lo que cuando uno quiere usar un Content Provider, lo tiene que hacer mediante un Content Resolver.

## El Método query()

Una vez que se ha obtenido el content resolver, se usa el método **query()**. Este método es parte de un set de 4

Con un content provider hay 4 métodos básicos que uno puede hacer:

1. Leer desde la data: query()
2. Agregar un row of rows a la data: insert()
3. Actualizar la data: update()
4. Borrar un row o rows desde la data delete()

## Use a URI

Un URI es usado para identificar o dar la localización de alguna data. Para poder trabajar con la data del teléfono también se ocupa un URI. Un ejemplo de URI en Android es:

content://com.example.udacity.droidtermsexample/terms

El content es el prefijo para Content Provider y es el estándar para iniciar URIs en una aplicación de Android.

El com.example.udacity.droidtermsexample es el Content Authority y especifica cual content provider usar.

La ultima parte especifica exactamente que data en el content provider uno está interesada en acceder. En este caso se trata de los términos

Para saber que content providers se pueden acceder, lo ideal es que el diseñador del content provider siga las convenciones de Android, que dictan:

* Crear un Contract class, que es lo que se usa también para diseñar base de datos. Estos contracts tienen la definición del uri y el authority así como algunos métodos.

El siguiente es un ejemplo de como se podría ver el código para obtener un resolver y llamar al método que permita borrar data de un content provider de una app llamada TVTimeContract:

ContentResolver resolver = getContentResolver();

resolver.delete(TVTimeContract.CONTENT\_URI, **null**, **null**);

Si uno necesita cambiar el content provider que uno esta tratando de acceder, lo que se necesita cambiar es el URI, sobre todo si la acción es la misma como leer.

Otro ejemplo usando uno de los Content Providers que provee Google, el del calendario, para poder cambiar el color del calendario:

resolver.update(CalendarContract.Calendars.CONTENT\_URI, values, null, null);

Para saber que contract usar, se puede ver la documentación de Android:

<https://developer.android.com/reference/android/provider/CalendarContract.Calendars>

Se usa la acción update porque lo que se quiere por ejemplo es cambiar el color del calendario. Se especifica CalendarContract.Calendars.CONTENT\_URI como lo que se quiere cambiar.

## Llamando al Content Provider

Al igual que con las llamadas a servicios, las llamadas a las bases de datos se deben hacer fuera del hilo principal. Hay muchas formas de manejar eso y una es usar un AsyncTask.

El método query provee un objeto Cursor que es el mismo tipo retornado por la clase de Android SQLite database. El cursor almacena toda la data que interesa.

Los cursores son iteradores que proveen acceso de lectura y escritura a los datos de un content provider.

La data viene en forma tabular. Generalmente hay una columna \_ID y luego mas según las que se provea el content provider.

Los parámetros que lleva el método query son un espejo de las partes de una consulta en SQL:

* *projection*: Usado para acortar las columnas o filtrarlas, por ejemplo, para obtener solo una columna.
* *selection*: especifica como filtrar las filas.
* *selection arguments*: especifica que filtrar de las filas
* *sort order*: determina el orden en el que deben ser devuelta la data

Ejemplos:

[https://developer.android.com/guide/topics/providers/content-provider-basics.html#Query](https://developer.android.com/guide/topics/providers/content-provider-basics.html" \l "Query)

## Trabajando con Cursores

Un cursor tiene una posición, que es la fila a la que esta apuntando en dicho momento. Cuando se obtiene el cursor del método query, la posición es **-1**, lo que indica que no está apuntando a ningún dato. Cuando se llama al método **moveToNext()**, el cursor se mueve al siguiente row y retorna true/false dependiendo si el movimiento fue exitoso o no. Al llegar al final, devolverá false porque ya no se puede mover más hacia abajo.

Hay otros métodos que permiten moverse a otras filas:

**moveToFirst()** mueve el cursor a la primera fila.

**getColumnIndex(String heading)** si uno quiere saber que valor hay en una columna de la fila actual, es importante saber el índice de la columna. Con este método se envía el nombre de la columna y se obtiene el índice de la misma. Los heading se encuentran en el contrato

**get<Type>(int ColumnIndex)** Retornan el valor en la columna con el índice especificado. Estos métodos son, por ejemplo, getString o getInteger.

**getCount()** devuelve el número de filas en el cursor.

**close()** se debe llamar cuando ya uno termino de trabajar con el cursor. Es importante siempre llamar a este método para evitar que haya problemas con la memoria.

Este es un ejemplo de cómo usar estos métodos:

int wordCol = cursor.getColumnIndex(DroidTermsExampleContract.*COLUMN\_WORD*);  
int defCol = cursor.getColumnIndex(DroidTermsExampleContract.*COLUMN\_DEFINITION*);  
  
while(cursor.moveToNext()){  
 String word = cursor.getString(wordCol);  
 String definition = cursor.getString(defCol);  
 Log.*v*("Cursor Example", word + " - " + definition);  
}  
cursor.close();

Un ejemplo de una clase para trabajar con cursores es esta:

public class MainActivity extends AppCompatActivity {  
  
 // The data from the DroidTermsExample content provider  
 private Cursor mData;  
  
 // The current state of the app  
 private int mCurrentState;  
  
 // The index of the definition and word column in the cursor  
 private int mDefCol, mWordCol;  
  
 private TextView mWordTextView, mDefinitionTextView;  
 private Button mButton;  
  
 // This state is when the word definition is hidden and clicking the button will therefore  
 // show the definition  
 private final int STATE\_HIDDEN = 0;  
  
 // This state is when the word definition is shown and clicking the button will therefore  
 // advance the app to the next word  
 private final int STATE\_SHOWN = 1;  
  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
  
 // Get the views  
 // COMPLETED (1) You'll probably want more than just the Button  
 mWordTextView = (TextView) findViewById(R.id.*text\_view\_word*);  
 mDefinitionTextView = (TextView) findViewById(R.id.*text\_view\_definition*);  
 mButton = (Button) findViewById(R.id.*button\_next*);  
  
 //Run the database operation to get the cursor off of the main thread  
 new WordFetchTask().execute();  
 }  
  
 */\*\*  
 \* This is called from the layout when the button is clicked and switches between the  
 \* two app states.  
 \** ***@param*** *view The view that was clicked  
 \*/* public void onButtonClick(View view) {  
  
 // Either show the definition of the current word, or if the definition is currently  
 // showing, move to the next word.  
 switch (mCurrentState) {  
 case STATE\_HIDDEN:  
 showDefinition();  
 break;  
 case STATE\_SHOWN:  
 nextWord();  
 break;  
 }  
 }  
  
 public void nextWord() {  
 // COMPLETED (3) Go to the next word in the Cursor, show the next word and hide the definition  
 // Note that you shouldn't try to do this if the cursor hasn't been set yet.  
 // If you reach the end of the list of words, you should start at the beginning again.  
  
 if (mData != null) {  
 // Move to the next position in the cursor, if there isn't one, move to the first  
 if (!mData.moveToNext()) {  
 mData.moveToFirst();  
 }  
 // Hide the definition TextView  
 mDefinitionTextView.setVisibility(View.*INVISIBLE*);  
  
 // Change button text  
 mButton.setText(getString(R.string.*show\_definition*));  
  
 // Get the next word  
 mWordTextView.setText(mData.getString(mWordCol));  
 mDefinitionTextView.setText(mData.getString(mDefCol));  
  
 mCurrentState = STATE\_HIDDEN;  
 }  
 }  
  
 public void showDefinition() {  
 // COMPLETED (4) Show the definition  
 if (mData != null) {  
 // Show the definition TextView  
 mDefinitionTextView.setVisibility(View.*VISIBLE*);  
  
 // Change button text  
 mButton.setText(getString(R.string.*next\_word*));  
  
 mCurrentState = STATE\_SHOWN;  
 }  
 }  
  
 @Override  
 protected void onDestroy() {  
 super.onDestroy();  
 // COMPLETED (5) Remember to close your cursor!  
 mData.close();  
 }  
  
 // Use an async task to do the data fetch off of the main thread.  
 public class WordFetchTask extends AsyncTask<Void, Void, Cursor> {  
  
 // Invoked on a background thread  
 @Override  
 protected Cursor doInBackground(Void... params) {  
 // Make the query to get the data  
  
 // Get the content resolver  
 ContentResolver resolver = getContentResolver();  
  
 // Call the query method on the resolver with the correct Uri from the contract class  
 Cursor cursor = resolver.query(DroidTermsExampleContract.*CONTENT\_URI*,  
 null, null, null, null);  
 return cursor;  
 }  
  
  
 // Invoked on UI thread  
 @Override  
 protected void onPostExecute(Cursor cursor) {  
 super.onPostExecute(cursor);  
 // COMPLETED (2) Initialize anything that you need the cursor for, such as setting up  
 // the screen with the first word and setting any other instance variables  
  
 //Set up a bunch of instance variables based off of the data  
  
 // Set the data for MainActivity  
 mData = cursor;  
 // Get the column index, in the Cursor, of each piece of data  
 mDefCol = mData.getColumnIndex(DroidTermsExampleContract.*COLUMN\_DEFINITION*);  
 mWordCol = mData.getColumnIndex(DroidTermsExampleContract.*COLUMN\_WORD*);  
  
 // Set the initial state  
 nextWord();  
 }  
 }

### Nota

Un objeto Cursor puede ser nulo por varias razones diferentes. Algunas son:

1. URI Invalida
2. El metodo query del ContentProvider retorna null
3. Una RemoteException

Tambien, se puede ejecutar un AsyncTask como un objeto dentro de un metodo, sin necesidad de crear una clase aparte, por ejemplo, en este metodo initialize se revisa si el content provider devuelve algun valor, si no tiene nada o da null llama al metodo startImmediateSync. Importante recordar siempre cerrar el cursor:

// (2) Create a synchronized public static void method called initialize

synchronized public static void initialize(final Context context){

// (3) Only execute this method body if sInitialized is false

if(*sInitialized*) return;

// (4) If the method body is executed, set sInitialized to true

*sInitialized* = true;

// (5) Check to see if our weather ContentProvider is empty

new AsyncTask<Void, Void, Void>(){

@Override

protected Void doInBackground(Void... voids) {

// Uri for the data

Uri queryUri = WeatherContract.WeatherEntry.*CONTENT\_URI*;

// Projection columns

String[] projection = {WeatherContract.WeatherEntry.*COLUMN\_WEATHER\_ID* };

// Select for the query

String selectStatement = WeatherContract.WeatherEntry.*getSqlSelectForTodayOnwards*();

Cursor cursor = context.getContentResolver().query(

queryUri,

projection,

selectStatement,

null,

null

);

// (6) If it is empty or we have a null Cursor, sync the weather now!

if(cursor == null || cursor.getCount() < 1){

*startImmediateSync*(context);

}

// Make sure the cursor is closed

cursor.close();

return null;

}

}.execute();

}

## CursorLoader

La mejor forma de cargar datos de manera asíncrona desde un Content Provider es con un **CursorLoader**.

Un CursorLoader es una subclase de **AsyncTaskLoader** que consulta un **ContentProvider**, via un **ContentResolver** y con un **URI** especifico y devuelve un Cursor con la data requerida. Este loader corre en un hilo background por lo que no bloquea el UI. Cando un CursorLoader está activo, esta ligado a una URI, y uno puede elegir monitorizar la URI en caso de cambios en la data, lo que significa que el CursorLoader puede enviar nuevos resultados siempre que el contenido de la base de datos cambie, y uno puede automáticamente actualizar el UI para reflejar el cambio.

Un ejemplo de esto es el ejercicio del app Sunshine donde se reemplazó el AsyncTaskLoader por un CursorLoader.

La clase del Adapter, donde se tiene un método swapCursor que recibe un cursor ya cargado desde el MainActivity y se encarga de desplegar los valores en el RecyclerView por medio del Adapter:

*/\*\**

*\* {****@link*** *ForecastAdapter} exposes a list of weather forecasts*

*\* from a {****@link*** *android.database.Cursor} to a {****@link*** *android.support.v7.widget.RecyclerView}.*

*\*/*

class ForecastAdapter extends RecyclerView.Adapter<ForecastAdapter.ForecastAdapterViewHolder> {

// Declare a private final Context field called mContext

final private Context mContext;

/\*

\* Below, we've defined an interface to handle clicks on items within this Adapter. In the

\* constructor of our ForecastAdapter, we receive an instance of a class that has implemented

\* said interface. We store that instance in this variable to call the onClick method whenever

\* an item is clicked in the list.

\*/

final private ForecastAdapterOnClickHandler mClickHandler;

*/\*\**

*\* The interface that receives onClick messages.*

*\*/*

public interface ForecastAdapterOnClickHandler {

void onClick(String weatherForDay);

}

// Declare a private Cursor field called mCursor

private Cursor mCursor;

// Add a Context field to the constructor and store that context in mContext

*/\*\**

*\* Creates a ForecastAdapter.*

*\**

*\** ***@param*** *clickHandler The on-click handler for this adapter. This single handler is called*

*\* when an item is clicked.*

*\*/*

public ForecastAdapter(ForecastAdapterOnClickHandler clickHandler, Context context) {

mClickHandler = clickHandler;

mContext = context;

}

*/\*\**

*\* This gets called when each new ViewHolder is created. This happens when the RecyclerView*

*\* is laid out. Enough ViewHolders will be created to fill the screen and allow for scrolling.*

*\**

*\** ***@param*** *viewGroup The ViewGroup that these ViewHolders are contained within.*

*\** ***@param*** *viewType If your RecyclerView has more than one type of item (which ours doesn't) you*

*\* can use this viewType integer to provide a different layout. See*

*\* {****@link*** *android.support.v7.widget.RecyclerView.Adapter#getItemViewType(int)}*

*\* for more details.*

*\** ***@return*** *A new ForecastAdapterViewHolder that holds the View for each list item*

*\*/*

@Override

public ForecastAdapterViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {

int layoutIdForListItem = R.layout.*forecast\_list\_item*;

LayoutInflater inflater = LayoutInflater.*from*(mContext);

boolean shouldAttachToParentImmediately = false;

View view = inflater.inflate(layoutIdForListItem, viewGroup, shouldAttachToParentImmediately);

return new ForecastAdapterViewHolder(view);

}

*/\*\**

*\* OnBindViewHolder is called by the RecyclerView to display the data at the specified*

*\* position. In this method, we update the contents of the ViewHolder to display the weather*

*\* details for this particular position, using the "position" argument that is conveniently*

*\* passed into us.*

*\**

*\** ***@param*** *forecastAdapterViewHolder The ViewHolder which should be updated to represent the*

*\* contents of the item at the given position in the data set.*

*\** ***@param*** *position The position of the item within the adapter's data set.*

*\*/*

@Override

public void onBindViewHolder(ForecastAdapterViewHolder forecastAdapterViewHolder, int position) {

// Move the cursor to the appropriate position

mCursor.moveToPosition(position);

// Generate a weather summary with the date, description, high and low

long date = mCursor.getLong(MainActivity.*DATE\_COL\_INDEX*);

String dateString = SunshineDateUtils.*getFriendlyDateString*(mContext, date, true);

// Get the weather id to form the description

int weatherId = mCursor.getInt(MainActivity.*WEATHER\_ID\_COL\_INDEX*);

String description = SunshineWeatherUtils.*getStringForWeatherCondition*(mContext, weatherId);

// Get the high and low temperatures from the cursor (in degrees celsius)

double highInCelsius = mCursor.getDouble(MainActivity.*MAX\_COL\_INDEX*);

double lowInCelsius = mCursor.getDouble(MainActivity.*MIN\_COL\_INDEX*);

String highAndLowString = SunshineWeatherUtils.*formatHighLows*(mContext,

highInCelsius, lowInCelsius);

String weatherSummary = dateString + " - " + description + " - " + highAndLowString;

// Display the summary that you created above

forecastAdapterViewHolder.weatherSummary.setText(weatherSummary);

}

*/\*\**

*\* This method simply returns the number of items to display. It is used behind the scenes*

*\* to help layout our Views and for animations.*

*\**

*\** ***@return*** *The number of items available in our forecast*

*\*/*

@Override

public int getItemCount() {

// If mCursor is null, return 0. Otherwise, return the count of mCursor

if(mCursor == null){

return 0;

}

return mCursor.getCount();

}

// Create a new method that allows you to swap Cursors. This is called by MainActivity after

// the load has finished and when the weather data is reset.

public void swapCursor(Cursor newCursor){

mCursor = newCursor;

// After the new Cursor is set, call notifyDataSetChanged to tell the RecyclerView to update

notifyDataSetChanged();

}

*/\*\**

*\* A ViewHolder is a required part of the pattern for RecyclerViews. It mostly behaves as*

*\* a cache of the child views for a forecast item. It's also a convenient place to set an*

*\* OnClickListener, since it has access to the adapter and the views.*

*\*/*

class ForecastAdapterViewHolder extends RecyclerView.ViewHolder implements View.OnClickListener {

final TextView weatherSummary;

ForecastAdapterViewHolder(View view) {

super(view);

weatherSummary = (TextView) view.findViewById(R.id.*tv\_weather\_data*);

view.setOnClickListener(this);

}

*/\*\**

*\* This gets called by the child views during a click.*

*\**

*\** ***@param*** *v The View that was clicked*

*\*/*

@Override

public void onClick(View v) {

// Instead of passing the String from the data array, use the weatherSummary text!

mClickHandler.onClick( weatherSummary.getText().toString());

}

}

}

La clase MainActivity, donde se crea el Cursor, en lugar de trabajar con un AsynTaskLoader, implementa un LoaderCallback<Cursor>. En el onCreateLoader se verifica el id del loader y retorna un CursorLoader cargado con la Uri para el content provider, el projection que son basicamente las columnas con las que uno quiere trabajar, un selection que se forma en una de las clases utilitarias y un order by.

En el onLoadFinished se obtiene un cursor ya cargado por el método anterior y este se pasa al adapter en el método swapCursor. En el onLoadReset también se llama al método swapCursor del adapter, pero en este caso se pasa null para invalidar los datos.

public class MainActivity extends AppCompatActivity implements

// Remove the implements declaration for SharedPreferences change listener and methods

// Implement LoaderCallbacks<Cursor> instead of String[]

ForecastAdapter.ForecastAdapterOnClickHandler,

LoaderManager.LoaderCallbacks<Cursor> {

private final String TAG = MainActivity.class.getSimpleName();

// Create a String array containing the names of the desired data columns from our ContentProvider

public static final String[] *mProjection* = {

WeatherContract.WeatherEntry.*COLUMN\_DATE*,

WeatherContract.WeatherEntry.*COLUMN\_WEATHER\_ID*,

WeatherContract.WeatherEntry.*COLUMN\_MIN\_TEMP*,

WeatherContract.WeatherEntry.*COLUMN\_MAX\_TEMP*

};

// Create constant int values representing each column name's position above

public static final int *DATE\_COL\_INDEX* = 0;

public static final int *WEATHER\_ID\_COL\_INDEX* = 1;

public static final int *MIN\_COL\_INDEX* = 2;

public static final int *MAX\_COL\_INDEX* = 3;

/\*

\* This ID will be used to identify the Loader responsible for loading our weather forecast. In

\* some cases, one Activity can deal with many Loaders. However, in our case, there is only one.

\* We will still use this ID to initialize the loader and create the loader for best practice.

\* Please note that 44 was chosen arbitrarily. You can use whatever number you like, so long as

\* it is unique and consistent.

\*/

private static final int *ID\_FORECAST\_LOADER* = 44;

private ForecastAdapter mForecastAdapter;

private RecyclerView mRecyclerView;

private int mPosition = RecyclerView.*NO\_POSITION*;

private ProgressBar mLoadingIndicator;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.*activity\_forecast*);

getSupportActionBar().setElevation(0f);

FakeDataUtils.*insertFakeData*(this);

/\*

\* Using findViewById, we get a reference to our RecyclerView from xml. This allows us to

\* do things like set the adapter of the RecyclerView and toggle the visibility.

\*/

mRecyclerView = (RecyclerView) findViewById(R.id.*recyclerview\_forecast*);

// COMPLETED (36) Remove the findViewById call for the error TextView

/\*

\* The ProgressBar that will indicate to the user that we are loading data. It will be

\* hidden when no data is loading.

\*

\* Please note: This so called "ProgressBar" isn't a bar by default. It is more of a

\* circle. We didn't make the rules (or the names of Views), we just follow them.

\*/

mLoadingIndicator = (ProgressBar) findViewById(R.id.*pb\_loading\_indicator*);

/\*

\* A LinearLayoutManager is responsible for measuring and positioning item views within a

\* RecyclerView into a linear list. This means that it can produce either a horizontal or

\* vertical list depending on which parameter you pass in to the LinearLayoutManager

\* constructor. In our case, we want a vertical list, so we pass in the constant from the

\* LinearLayoutManager class for vertical lists, LinearLayoutManager.VERTICAL.

\*

\* There are other LayoutManagers available to display your data in uniform grids,

\* staggered grids, and more! See the developer documentation for more details.

\*

\* The third parameter (shouldReverseLayout) should be true if you want to reverse your

\* layout. Generally, this is only true with horizontal lists that need to support a

\* right-to-left layout.

\*/

LinearLayoutManager layoutManager =

new LinearLayoutManager(this, LinearLayoutManager.*VERTICAL*, false);

/\* setLayoutManager associates the LayoutManager we created above with our RecyclerView \*/

mRecyclerView.setLayoutManager(layoutManager);

/\*

\* Use this setting to improve performance if you know that changes in content do not

\* change the child layout size in the RecyclerView

\*/

mRecyclerView.setHasFixedSize(true);

// Pass in this again as the ForecastAdapter now requires a Context

/\*

\* The ForecastAdapter is responsible for linking our weather data with the Views that

\* will end up displaying our weather data.

\*

\* Although passing in "this" twice may seem strange, it is actually a sign of separation

\* of concerns, which is best programming practice. The ForecastAdapter requires an

\* Android Context (which all Activities are) as well as an onClickHandler. Since our

\* MainActivity implements the ForecastAdapter ForecastOnClickHandler interface, "this"

\* is also an instance of that type of handler.

\*/

mForecastAdapter = new ForecastAdapter(this, this);

/\* Setting the adapter attaches it to the RecyclerView in our layout. \*/

mRecyclerView.setAdapter(mForecastAdapter);

// Call the showLoading method

showLoading();

/\*

\* Ensures a loader is initialized and active. If the loader doesn't already exist, one is

\* created and (if the activity/fragment is currently started) starts the loader. Otherwise

\* the last created loader is re-used.

\*/

getSupportLoaderManager().initLoader(*ID\_FORECAST\_LOADER*, null, this);

}

*/\*\**

*\* Uses the URI scheme for showing a location found on a map in conjunction with*

*\* an implicit Intent. This super-handy Intent is detailed in the "Common Intents" page of*

*\* Android's developer site:*

*\**

*\** ***@see*** *"http://developer.android.com/guide/components/intents-common.html#Maps"*

*\* <p>*

*\* Protip: Hold Command on Mac or Control on Windows and click that link to automagically*

*\* open the Common Intents page*

*\*/*

private void openPreferredLocationInMap() {

double[] coords = SunshinePreferences.*getLocationCoordinates*(this);

String posLat = Double.*toString*(coords[0]);

String posLong = Double.*toString*(coords[1]);

Uri geoLocation = Uri.*parse*("geo:" + posLat + "," + posLong);

Intent intent = new Intent(Intent.*ACTION\_VIEW*);

intent.setData(geoLocation);

if (intent.resolveActivity(getPackageManager()) != null) {

startActivity(intent);

} else {

Log.*d*(TAG, "Couldn't call " + geoLocation.toString() + ", no receiving apps installed!");

}

}

// Refactor onCreateLoader to return a Loader<Cursor>, not Loader<String[]>

*/\*\**

*\* Instantiate and return a new Loader for the given ID.*

*\**

*\** ***@param*** *id The ID whose loader is to be created.*

*\** ***@param*** *loaderArgs Any arguments supplied by the caller.*

*\**

*\** ***@return*** *Return a new Loader instance that is ready to start loading.*

*\*/*

@Override

public Loader<Cursor> onCreateLoader(int id, final Bundle loaderArgs) {

// If the loader requested is our forecast loader, return the appropriate CursorLoader

if(id == *ID\_FORECAST\_LOADER*){

// Uri for all rows of weather data in the weather table

Uri forecastQueryUri = WeatherContract.WeatherEntry.*CONTENT\_URI*;

// Select the rows we'd like to return

String selection = WeatherContract.WeatherEntry.*getSqlSelectForTodayOnwards*();

// Sort oder: ascending by date

String orderBy = *mProjection*[*DATE\_COL\_INDEX*] + " ASC";

return new CursorLoader(this,

forecastQueryUri,

*mProjection*,

selection,

null,

orderBy );

}

else{

throw new RuntimeException("Loader not implemented: " + id);

}

}

// Change onLoadFinished parameter to a Loader<Cursor> instead of a Loader<String[]>

*/\*\**

*\* Called when a previously created loader has finished its load.*

*\**

*\** ***@param*** *loader The Loader that has finished.*

*\** ***@param*** *data The data generated by the Loader.*

*\*/*

@Override

public void onLoadFinished(Loader<Cursor> loader, Cursor data) {

// Call mForecastAdapter's swapCursor method and pass in the new Cursor

mForecastAdapter.swapCursor(data);

// If mPosition equals RecyclerView.NO\_POSITION, set it to 0

if(mPosition == RecyclerView.*NO\_POSITION*){

mPosition = 0;

}

// Smooth scroll the RecyclerView to mPosition

mRecyclerView.smoothScrollToPosition(mPosition);

// If the Cursor's size is not equal to 0, call showWeatherDataView

if(data.getCount() != 0){

showWeatherDataView();

}

}

*/\*\**

*\* Called when a previously created loader is being reset, and thus making its data unavailable.*

*\* The application should at this point remove any references it has to the Loader's data.*

*\**

*\** ***@param*** *loader The Loader that is being reset.*

*\*/*

@Override

public void onLoaderReset(Loader<Cursor> loader) {

// Call mForecastAdapter's swapCursor method and pass in null

/\*

\* Since this Loader's data is now invalid, we need to clear the Adapter that is

\* displaying the data.

\*/

mForecastAdapter.swapCursor(null);

}

*/\*\**

*\* This method is for responding to clicks from our list.*

*\**

*\** ***@param*** *weatherForDay String describing weather details for a particular day*

*\*/*

@Override

public void onClick(String weatherForDay) {

Context context = this;

Class destinationClass = DetailActivity.class;

Intent intentToStartDetailActivity = new Intent(context, destinationClass);

intentToStartDetailActivity.putExtra(Intent.*EXTRA\_TEXT*, weatherForDay);

startActivity(intentToStartDetailActivity);

}

*/\*\**

*\* This method will make the View for the weather data visible and hide the error message and*

*\* loading indicator.*

*\* <p>*

*\* Since it is okay to redundantly set the visibility of a View, we don't need to check whether*

*\* each view is currently visible or invisible.*

*\*/*

private void showWeatherDataView() {

/\* First, hide the loading indicator \*/

mLoadingIndicator.setVisibility(View.*INVISIBLE*);

/\* Finally, make sure the weather data is visible \*/

mRecyclerView.setVisibility(View.*VISIBLE*);

}

// Create a method called showLoading that shows the loading indicator and hides the data

private void showLoading(){

mLoadingIndicator.setVisibility(View.*VISIBLE*);

mRecyclerView.setVisibility(View.*INVISIBLE*);

}

*/\*\**

*\* This is where we inflate and set up the menu for this Activity.*

*\**

*\** ***@param*** *menu The options menu in which you place your items.*

*\**

*\** ***@return*** *You must return true for the menu to be displayed;*

*\* if you return false it will not be shown.*

*\**

*\** ***@see*** *#onPrepareOptionsMenu*

*\** ***@see*** *#onOptionsItemSelected*

*\*/*

@Override

public boolean onCreateOptionsMenu(Menu menu) {

/\* Use AppCompatActivity's method getMenuInflater to get a handle on the menu inflater \*/

MenuInflater inflater = getMenuInflater();

/\* Use the inflater's inflate method to inflate our menu layout to this menu \*/

inflater.inflate(R.menu.*forecast*, menu);

/\* Return true so that the menu is displayed in the Toolbar \*/

return true;

}

*/\*\**

*\* Callback invoked when a menu item was selected from this Activity's menu.*

*\**

*\** ***@param*** *item The menu item that was selected by the user*

*\**

*\** ***@return*** *true if you handle the menu click here, false otherwise*

*\*/*

@Override

public boolean onOptionsItemSelected(MenuItem item) {

int id = item.getItemId();

if (id == R.id.*action\_settings*) {

startActivity(new Intent(this, SettingsActivity.class));

return true;

}

if (id == R.id.*action\_map*) {

openPreferredLocationInMap();

return true;

}

return super.onOptionsItemSelected(item);

}

}

### Nota sobre Intents

Cuando se crea un intent, es importante especificar el tipo de datos (MIME) ademas del URI. Por ejemplo, una actividad que puede mostrar imágenes probablemente no sea capaz de reproducir un archivo de audio aunque los formatos de URI sean similares. El MIME a veces se puede deducir del URI, especialmente cuando los datos son un URI content:. Un URI content: indica que los datos se encuentran en el dispositivo y son controlados por un **ContentProvider**, lo que hace que el sistema pueda ver el tipo de datos de MIME.

El método setData() se usa para establecer el URI de datos de un intent.

Mas información aquí: <https://developer.android.com/guide/components/intents-filters.html>

# Lesson 08 – Android Architecture Components

Los componentes de Android Architecture están hechos para facilitar a los developres muchas de las tareas comunes al momento de crear apps, como manejo de base de datos, buen uso de los recursos para evitar problemas de memoria y otros. Los principales componentes incluyen:

* Room: una librería para manejo de objetos relacionales de SQLite.
* LiveData: para observar cambios en la base de datos
* ViewModel: para guardar en cache data que necesita sobrevivir a cambios de configuración
* Lifecycle: permitir que objetos no-lifecycle se puedan usar como objetos del ciclo de vida

## Intro to Room

Los Content Providers son uno de los 4 principales componentes de las apps de Android. En los casos donde uno no se pueda beneficiar de dichas ventajas, se puede utilizar Room. Room es un ORM (object relational mapping library), esto es, una libreria que mapea los objetos de base de datos a objetos Java.

Caracteristicas de Room:

* Permite escribir menos “boilerplates”
* Validación de SQL en tiempo de compilacion
* Hecho para trabajar con LiveData y RxJava para data observation.

Room usa annotations y sus principales componentes son:

* @Entity: para definir tablas de base de datos.
* @DAO: para proveer un API para leer y escribir data
* @Database: que representa un soporte para base de datos. Esto incluye una lista de entities y DAOs y permite crear una nueva base de datos o adquirir una conexión a la base de datos en tiempo de ejecución.

## Creando una Entidad

Los Android Architecture Components son librerías, las cuales, para poder usarse necesitan una o varias dependencias en el build.gradle.

Ver: https://developer.android.com/jetpack/androidx/releases/room

Ejemplo:

def room\_version = "2.2.6"

implementation "androidx.room:room-runtime:$room\_version"

annotationProcessor "androidx.room:room-compiler:$room\_version"

// optional - RxJava support for Room

implementation "androidx.room:room-rxjava2:$room\_version"

// optional - Guava support for Room, including Optional and ListenableFuture

implementation "androidx.room:room-guava:$room\_version"

// optional - Test helpers

testImplementation "androidx.room:room-testing:$room\_version"

POJO (Plain Old Java Object) es un objeto de java simple y ordinario. Es una clase con sus miembros, setters, getters y constructor o constructores. Una clase que no debe extender otras clases, implementar interfaces o contener annotations.

Para tener un Entity, se puede tener un POJO y a este agregarle el annotation **@Entity**, con esto, Room generara una tabla con el nombre de la clase, sin embargo, es común que la tabla en la base de datos no tenga el mismo nombre que la clase, para esto se puede usar la propiedad **tableName** del annotation Entity.

Algunas veces, la clase tiene algunas propiedad que no queremos que formen parte de la tabla, para esto se pueden usar el annotation @**ignore**.

Otro requerimiento de Room es definir un primary key, el cual se establece con el annotation **@PrimaryKey**. En una tabla donde se maneje un id autogenerado, se puede usar la propiedad autoGenerate = true en el annotation PrimaryKey.

Una de las limitantes de Room es que solo puede usar un constructor. Si uno tiene dos o mas constructores, se puede usar el @ignore en los constructores que uno no quiere que Room utilice.

@ignore parece estar obsoleto, por lo que lo recomendado es usar la interfaz ignore: https://developer.android.com/reference/androidx/room/Ignore

En la siguiente url se puede encontrar un sqlite command cheat sheet para ver las operaciones comunes a utilizar en las operaciones de SQLite: https://d17h27t6h515a5.cloudfront.net/topher/2016/September/57ed880e\_sql-sqlite-commands-cheat-sheet/sql-sqlite-commands-cheat-sheet.pdf

**@ColumnInfo(name = “”)** permite identificar que en la base de datos, la propiedad se encuentra con otro nombre.

Ejemplo de Entity:

import androidx.room.Entity;

import androidx.room.Ignore;

import androidx.room.PrimaryKey;

import java.util.Date;

// Annotate the class with Entity. Use "task" for the table name

@Entity(tableName = "task")

public class TaskEntry {

// Annotate the id as PrimaryKey. Set autoGenerate to true.

@PrimaryKey(autoGenerate = true)

private int id;

private String description;

private int priority;

@ColumnInfo(name = "update\_at")

private Date updatedAt;

// Use the Ignore annotation so Room knows that it has to use the other constructor instead

@Ignore

public TaskEntry(String description, int priority, Date updatedAt) {

this.description = description;

this.priority = priority;

this.updatedAt = updatedAt;

}

public TaskEntry(int id, String description, int priority, Date updatedAt) {

this.id = id;

this.description = description;

this.priority = priority;

this.updatedAt = updatedAt;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public int getPriority() {

return priority;

}

public void setPriority(int priority) {

this.priority = priority;

}

public Date getUpdatedAt() {

return updatedAt;

}

public void setUpdatedAt(Date updatedAt) {

this.updatedAt = updatedAt;

}

}

## Creando un DAO

Los DAO son para accessar cada Entity. En general, son una interfaz que tiene la anotación **@Dao**. Estas pueden tener definidos varios métodos con las siguientes anotaciones:

* **@Query** para definir consultas. Esta anotación incluye el query.
* **@Insert** para definir métodos que agregan data a la base de datos,
* **@Update** para definir métodos que actualizan información en la base de datos. Esta anotación puede tener una propiedad **onConflict** que indica que hacer en caso que haya un conflicto al intentar actualizar.
* **@Delete** para definir métodos que eliminan información de la base de datos.

Ejemplo de Dao:

@Dao

public interface TaskDao {

@Query("SELECT \* FROM task ORDER BY priority")

List<TaskEntry> loadAllTasks();

@Insert

void insertTask(TaskEntry taskEntry);

@Update(onConflict = OnConflictStrategy.*REPLACE*)

void updateTask(TaskEntry taskEntry);

@Delete

void deleteTask(TaskEntry taskEntry);

}

## Creando la Base de Datos

Una forma estándar de crear una base de datos usando Room es crear una clase abstracta que extienda **RoomDatabase**.

El método **getInstance** va a retornar un AppDatabase usando el patrón Singleton. La instancia se crea con el método **Room.databaseBuilder** que recibe el contexto de la aplicación, la clase actual de la base de datos y un nombre para la base de datos. Este método crea la base de datos si aun no existe, si ya existe se crea una conexión a la base de datos. Al método al final se le encadena el método **build()**.

A esta clase hay que agregarle la anotación **@Database**, a esta se le define un atributo **entities** que especifica la lista de clases que se han definido como Entities. Cambien se le define la **version**, la cual se incrementa cuando se actualiza la base de datos y la propiedad **exportSchema** que es opcional pero si no se usa, se pone en false.

Se agrega también un método abstracto para el o los DAO definidos.

Example:

@Database(entities = {TaskEntry.class}, version = 1, exportSchema = false)

@TypeConverters(DateConverter.class)

public abstract class AppDatabase extends RoomDatabase {

private static final String *LOG\_TAG* = AppDatabase.class.getSimpleName();

private static final Object *LOCK* = new Object();

private static final String *DATABASE\_NAME* = "todolist";

private static AppDatabase *sInstance*;

public static AppDatabase getInstance(Context context) {

if (*sInstance* == null) {

synchronized (*LOCK*) {

Log.*d*(*LOG\_TAG*, "Creating new database instance");

*sInstance* = Room.*databaseBuilder*(context.getApplicationContext(),

AppDatabase.class, AppDatabase.*DATABASE\_NAME*)

.build();

}

}

Log.*d*(*LOG\_TAG*, "Getting the database instance");

return *sInstance*;

}

public abstract TaskDao taskDao();

}

## Type Converters

La documentación de SQLite especifica cuales son los data types que se pueden almacenar en una base de datos, que son:

* NULL
* INTEGER
* REAL
* TEXT
* BLOB

Datos como Date no se pueden guardar como tal, por lo que necesitan ser guardados como Text, Real o Integer. Room puede mapear objetos complejos como Date en estos tipos de datos. Uno puede definir clases para convertir tipos. Los métodos de estas clases usan un annotation **@TypeConverter**.

Una vez definida la clase, se agrega un annotation a la clase de la base de datos como esta:

**@TypeConverters(DateConverter.class)**, donde DateConverter es el nombre de la clase del converter.

## Nota Importante

Accesar una base de datos en el main thread puede consumir tiempo y bloquear el UI, por lo tanto, Room por defecto arroja un error si uno intenta accesar la base de datos desde el main thread.

Para efectos de practica unicamente, se puede usar el método allowMainThreadQueries en el momento de obtener la instancia de la base de datos para permitir esto, pero no se debe hacer en desarrollo.

## Threads and Runnables

Una forma de trabajar con las operaciones de la base de datos en un hilo diferente al main, es usar runnables. Un ejemplo es este:

Thread thread = new Thread(new Runnable() {

@Override

public void run() {

//Database logic

}

});

thread.start();

Aquí se crea un hilo que usa un Runnable, el cual ejecutara la lógica de la base de datos y en el caso de ser necesario, actualiza el UI, sin embargo, para actualizar el UI se usa un component de Android Architecture Components. Por ultimo, se inicia el hilo.

NOTA: Es importante acotar que esta solución es solo utemporal hasta aprender un nuevo componente de Android.

## Executors

Al usar hilos para ejecutar la lógica de la base de datos, se puede crear condiciones donde varios hilos lleguen a correr al mismo tiempo. Lo ideal es que todas las llamadas a la BD corran en el mismo hilo. Ahí es donde entran en juego los Executors.

Esta clase especifica un objeto que ejecuta tareas ejecutables enviadas. Se usa para evitar estar creando hilos para cada llamada a la base de datos.

Para el ejemplo, se crea una clase que sera un singleton que tiene 3 executors. El diskIO se asegura de que las transacciones en la base de datos se ejecuten en orden. El networkIO es un pool de 3 hilos que permite correr diferentes llamados de red simultáneamente mientras controla el numero de hilos. El mainThread executor usa la clase MainThreadExecutor que publica runnables usando un manejador asociado al looper principal.

Cuando se esta en un Activity, no se ocupa este main thread porque se puede usar el método runOnUIThread. Cuando se esta en una clase diferente y no se puede usar el método runOnUiTrhead, se puede acceder al main thread usando el executor.

Ejemplo de la clase AppExecutors:

public class AppExecutors {

// For Singleton instantiation

private static final Object *LOCK* = new Object();

private static AppExecutors *sInstance*;

private final Executor diskIO;

private final Executor mainThread;

private final Executor networkIO;

private AppExecutors(Executor diskIO, Executor networkIO, Executor mainThread) {

this.diskIO = diskIO;

this.networkIO = networkIO;

this.mainThread = mainThread;

}

public static AppExecutors getInstance() {

if (*sInstance* == null) {

synchronized (*LOCK*) {

*sInstance* = new AppExecutors(Executors.*newSingleThreadExecutor*(),

Executors.*newFixedThreadPool*(3),

new MainThreadExecutor());

}

}

return *sInstance*;

}

public Executor diskIO() {

return diskIO;

}

public Executor mainThread() {

return mainThread;

}

public Executor networkIO() {

return networkIO;

}

private static class MainThreadExecutor implements Executor {

private Handler mainThreadHandler = new Handler(Looper.*getMainLooper*());

@Override

public void execute(@NonNull Runnable command) {

mainThreadHandler.post(command);

}

}

}

Mas detalles de la clase Executor aqui: <https://developer.android.com/reference/java/util/concurrent/Executor.html>

Ejemplo de uso del diskIO para salvar un objeto TaskEntry:

public void onSaveButtonClicked() {

String description = mEditText.getText().toString();

int priority = getPriorityFromViews();

Date date = new Date();

// COMPLETED (4) Make taskEntry final so it is visible inside the run method

final TaskEntry taskEntry = new TaskEntry(description, priority, date);

// COMPLETED (2) Get the diskIO Executor from the instance of AppExecutors and

// call the diskIO execute method with a new Runnable and implement its run method

AppExecutors.*getInstance*().diskIO().execute(new Runnable() {

@Override

public void run() {

mDb.taskDao().insertTask(taskEntry);

finish();

}

});

}

La variable taskEntry debe ser declarada como final para poder ser usada dentro del run del diskIO.execute.

En el siguiente ejemplo también se usa el diskIO.execute, sin embargo, la parte donde se actualiza el UI se debe correr en otro runnable. Este approach es temporal, mas adelante se utilizara otro componente para hacer esto:

@Override

protected void onResume() {

super.onResume();

// Get the diskIO Executor from the instance of AppExecutors and

// call the diskIO execute method with a new Runnable and implement its run method

AppExecutors.*getInstance*().diskIO().execute(

new Runnable() {

@Override

public void run() {

// Move the logic into the run method and

// extract the list of tasks to a final variable

final List<TaskEntry> tasks = mDb.taskDao().loadAllTasks();

runOnUiThread(new Runnable() {

@Override

public void run() {

// Wrap the setTask call in a call to runOnUiThread

mAdapter.setTasks(tasks);

}

});

}

}

);

}

## Delete Task using Swipe

Una de las funciones que se usan mucho con las apps es deslizar un componente a un lado. Por ejemplo, con el app de Task o ToDo List, para eliminar una tarea, se hará swipe a la izquierda.

Para esto, se puede crear un **ItemTouchHelper** que se asocia al RecyclerView para habilitar el comportamiento de touch en el RecyclerView. A este helper se le sobre escribe el método onSwiped para poder trabajar con el swipe del elemento. Este método recibe dos parámetros: un ViewHolder para la fila y un int que describe la direccion del swipe.

/\*

Add a touch helper to the RecyclerView to recognize when a user swipes to delete an item.

An ItemTouchHelper enables touch behavior (like swipe and move) on each ViewHolder,

and uses callbacks to signal when a user is performing these actions.

\*/

new ItemTouchHelper(new ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.*LEFT* | ItemTouchHelper.*RIGHT*) {

@Override

public boolean onMove(RecyclerView recyclerView, RecyclerView.ViewHolder viewHolder, RecyclerView.ViewHolder target) {

return false;

}

// Called when a user swipes left or right on a ViewHolder

@Override

public void onSwiped(final RecyclerView.ViewHolder viewHolder, int swipeDir) {

// Here is where you'll implement swipe to delete

// Get the diskIO Executor from the instance of AppExecutors and

// call the diskIO execute method with a new Runnable and implement its run method

AppExecutors.*getInstance*().diskIO().execute(new Runnable() {

@Override

public void run() {

// get the position from the viewHolder parameter

int itemPosition = viewHolder.getAdapterPosition();

// Call deleteTask in the taskDao with the task at that position

List<TaskEntry> tasks = mAdapter.getTasks();

mDb.taskDao().deleteTask(tasks.get(itemPosition));

// Call retrieveTasks method to refresh the UI

retrieveTasks();

}

});

}

}).attachToRecyclerView(mRecyclerView);

## Queries with Parameters

Cuando se ocupa devolver elementos de la base de datos basados en algún parámetro, como un id por ejemplo, en el DAO (la interfaz) se agrega un método mas que se decora con el annotation @Query y al que se le define un query con el parámetro, por ejemplo:

// A Query method named loadTaskById that receives an int id and returns a TaskEntry Object

// The query for this method should get all the data for that id in the task table

@Query("SELECT \* FROM task WHERE id = :id")

TaskEntry loadTaskById(int id);

Los parámetros que se envían al método, se usan en el query con dos puntos (:) como prefijo y deben usar el mismo nombre.

Room hace validaciones en tiempo de compilación, lo que significa que si hay un error en las sentencias SQL, al momento de compilar se va a mostrar un error lo cual salva mucho tiempo y bugs en la versión final del app.

## LiveData and Observer Pattern

LiveData es una clase de contenedor de datos observable. LiveData se situá entre la base de datos y el UI. LiveData monitorea cambios en la BD y notifica a los observadores cuando los datos cambian. Esto es posible por el patrón Observer.

### El Patrón Observer

En este patrón, las clases Observers se suscriben a un “sujeto”. El sujeto (como un LiveData) mantendrá una lista de todos los observadores que están suscritos a el y les notifica cuando cualquier cambio relevante se da.

En el caso del LiveData, cuando se da un cambio, el método **setValue** se llama y esto dispara una llamada a un método en cada observador.

Algunas de las ventajas del LiveData son:

* Garantiza que el UI coincida con el estado de los datos.
* Evita fugas de memoria
* Si el ciclo de vida del observer esta inactivo este no recibe ningún evento de LiveData.
* Los componentes de IU solo observan datos relevantes y no detienen ni reanudan la observación. LiveData se ocupa de esto.
* Datos siempre actualizados
* Un activity o fragment que se vuelve a crear por cambios de configuración, como la rotación, recibe de inmediato los datos disponibles mas recientes.
* Compartir recursos.

Más información de LiveData: <https://developer.android.com/topic/libraries/architecture/livedata.html>

Mas información del Patrón Observable: https://en.wikipedia.org/wiki/Observer\_pattern

## Agregando LiveData

Para comenzar a usar LiveData hay que agregar las dependencias. Se puede ver en la documentación oficial. Para el ejemplo se va a usar tanto ViewModel como LiveData:

https://developer.android.com/jetpack/androidx/releases/lifecycle

dependencies {

    def lifecycle\_version = "2.3.0"

    def arch\_version = "2.1.0"

    // ViewModel

    implementation "androidx.lifecycle:lifecycle-viewmodel:$lifecycle\_version"

    // LiveData

    implementation "androidx.lifecycle:lifecycle-livedata:$lifecycle\_version"

    // Lifecycles only (without ViewModel or LiveData)

    implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle\_version"

    // Saved state module for ViewModel

    implementation "androidx.lifecycle:lifecycle-viewmodel-savedstate:$lifecycle\_version"

    // Annotation processor

    annotationProcessor "androidx.lifecycle:lifecycle-compiler:$lifecycle\_version"

    // alternately - if using Java8, use the following instead of lifecycle-compiler

    implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle\_version"

    // optional - helpers for implementing LifecycleOwner in a Service

    implementation "androidx.lifecycle:lifecycle-service:$lifecycle\_version"

    // optional - ProcessLifecycleOwner provides a lifecycle for the whole application process

    implementation "androidx.lifecycle:lifecycle-process:$lifecycle\_version"

    // optional - ReactiveStreams support for LiveData

    implementation "androidx.lifecycle:lifecycle-reactivestreams:$lifecycle\_version"

    // optional - Test helpers for LiveData

    testImplementation "androidx.arch.core:core-testing:$arch\_version"

}

Para el caso del ejemplo, se ocupa tanto el de LiveData como el de ViewModel y el del Annotation processor

En la interfaz DAO, se envuelve el método que consulta u obtiene los tasks con LiveData<>, de esta forma, podemos obtener notificaciones cuando se den cambios en los datos.

import androidx.lifecycle.LiveData;

import java.util.List;

@Dao

public interface TaskDao {

// Wrap the return type with LiveData

@Query("SELECT \* FROM task ORDER BY priority")

LiveData<List<TaskEntry>> loadAllTasks();

@Insert

void insertTask(TaskEntry taskEntry);

@Update(onConflict = OnConflictStrategy.*REPLACE*)

void updateTask(TaskEntry taskEntry);

@Delete

void deleteTask(TaskEntry taskEntry);

@Query("SELECT \* FROM task WHERE id = :id")

TaskEntry loadTaskById(int id);

}

Una de las ventajas del LiveData es que este por default corre fuera del main thread, lo que permite no tener que usar el Executor en el MainActivity para obtener dichos valores.

Al tener un objeto LiveData, se puede llamar al método **observe**, el cual requiere 2 parámetros:

* un lifecycle owner: algo que tiene un ciclo de vida, por ejemplo, el activity
* un observer: una interfaz de la que se necesita implementar el método **onChanged**.

El método onChaged recibe lo mismo que se envuelve con el LiveData, por ejemplo, un List<TaskEntry> y por lo tanto esto se puede usar para realizar las operaciones que se tenían en el runOnUiThread

private void retrieveTasks() {

// Extract all this logic outside the Executor and remove the Executor

Log.*d*(*TAG*, "Actively retrieving the tasks from the DataBase");

// Fix compile issue by wrapping the return type with LiveData

final LiveData<List<TaskEntry>> tasks = mDb.taskDao().loadAllTasks();

// Observe tasks and move the logic from runOnUiThread to onChanged

tasks.observe(this, new Observer<List<TaskEntry>>() {

@Override

public void onChanged(List<TaskEntry> taskEntries) {

mAdapter.setTasks(taskEntries);

}

});

}

Es importante recordar que para las operaciones insert, delete y update no hace falta usar observers, por lo tanto estos no requieren el uso de LiveData y por lo tanto, se sigue usando los executers.

La lógica para actualizar el UI se tiene en el método que devuelve las tareas o en si, que carga los datos de la tabla, y este corre en el main thread por default. Cada cambio en la base de datos va a disparar al método onChaged del observer, por lo que no hace falta hacer el llamado a este método cuando se eliminan tareas.

También se puede mover la llamada al retrieveTasks del onResume al onCreate

### Remove the Observer

El objeto LiveData tiene un método llamado removeObserver el cual recibe el objeto observer del LiveData. Este método permite quitar el observer de la lista de oberservers, esto sirve para casos donde uno no quiera estar escuchando cambios en la base de datos. Para el ejemplo siguiente, se podría haber dejado un executor pero cuando se vea el ViewModel se podrá tener un mayor entendimiento de los beneficios del LiveData.

Intent intent = getIntent();

if (intent != null && intent.hasExtra(*EXTRA\_TASK\_ID*)) {

mButton.setText(R.string.*update\_button*);

if (mTaskId == *DEFAULT\_TASK\_ID*) {

// populate the UI

mTaskId = intent.getIntExtra(*EXTRA\_TASK\_ID*, *DEFAULT\_TASK\_ID*);

// Extract all this logic outside the Executor and remove the Executor

// Fix compile issue by wrapping the return type with LiveData

final LiveData<TaskEntry> task = mDb.taskDao().loadTaskById(mTaskId);

// Observe tasks and move the logic from runOnUiThread to onChanged

task.observe(this, new Observer<TaskEntry>() {

@Override

public void onChanged(@Nullable TaskEntry taskEntry) {

// Remove the observer as we do not need it any more

task.removeObserver(this);

Log.*d*(*TAG*, "onChanged: Receiving database update from LiveData");

populateUI(taskEntry);

}

});

}

}

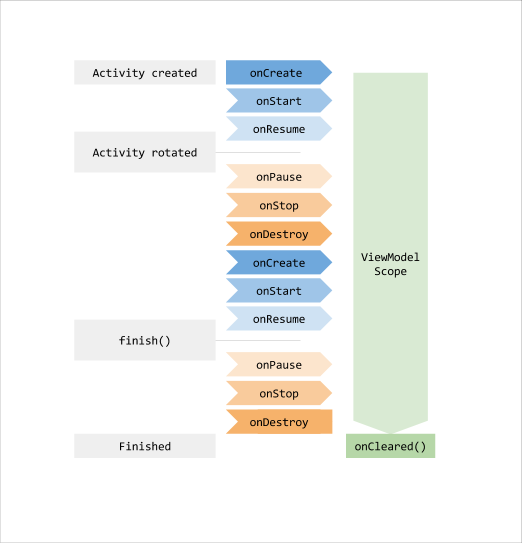
## ViewModel

LiveData permite que uno reciba notificaciones cuando la base de datos cambio sin la necesidad de esta haciendo consultas a la misma, sin embargo, igual se hace la consulta cuando se tiene cambios de configuración como la rotación del dispositivo.

El uso del OnSaveInstanceState es para guardar pequeños montos de información que se pueden serializar y deserializar, pero este no es el caso, ya que habría que consultar a la BD. Para esto esta ViewModel.

ViewModel permite persistir datos a través de cambios de configuración como la rotación.

El siguiente es el diagrama del ciclo de vida del ViewModel:



El ciclo de vida del ViewModel comienza cuando la actividad se crea y dura hasta que esta termina. Esto permite guardar en cache del ViewModel datos complejos. En ocasiones, se necesitan hacer llamadas asíncronas que sin el ViewModel se podría dar problemas de memoria si la actividad es destruida antes de que esta llamada finalice. Si se tiene el ViewModel, estas llamadas se pueden hacer hacia el mismo ViewModel y esto significa que no importa si el activity se destruye

Mas información de ViewModel: <https://developer.android.com/topic/libraries/architecture/viewmodel>

Información sobre Parcelable vs Serializable: https://medium.com/android-news/parcelable-vs-serializable-6a2556d51538

## Adding ViewModel

Para comenzar a trabajar con un ViewModel, una clase debe extender **AndroidViewModel**.

El constructor recibe un parámetro de tipo Application.

Ejemplo de ViewModel:

// Make this class extend AndroidViewModel and implement its default constructor

public class MainViewModel extends AndroidViewModel {

// Constant for logging

private static final String *TAG* = MainViewModel.class.getSimpleName();

// Add a tasks member variable for a list of TaskEntry objects wrapped in a LiveData

private LiveData<List<TaskEntry>> tasks;

// In the constructor use the loadAllTasks of the taskDao to initialize the tasks variable

public MainViewModel(@NonNull Application application) {

super(application);

AppDatabase database = AppDatabase.*getInstance*(this.getApplication());

Log.*d*(*TAG*, "Actively retrieving the tasks from the DataBase");

tasks = database.taskDao().loadAllTasks();

}

// Create a getter for the tasks variable

public LiveData<List<TaskEntry>> getTasks() {

return tasks;

}

}

Este viewmodel va a mantener en cache la información de los tasks, la cual obtiene desde el objeto de base de datos en el constructor.

En la clases donde se obtienen los datos de las tareas, por ejemplo, MainActivity, ya no seria necesario llamar directamente al loadAllTasks, sino que se obtendría el ViewModel llamando al provider del ViewModel de la actividad y pasando la clase del ViewModel como parámetro. Ahora se puede obtener el objeto LiveData usando getTasks del ViewModel:

// This method is not retrieving the tasks any more. Refactor to a more suitable name such as setupViewModel

private void setupViewModel() {

// Remove the logging and the call to loadAllTasks, this is done in the ViewModel now

// Declare a ViewModel variable and initialize it by calling ViewModelProviders.of

MainViewModel viewModel = ViewModelProviders.*of*(this).get(MainViewModel.class);

// Observe the LiveData object in the ViewModel

viewModel.getTasks().observe(this, new Observer<List<TaskEntry>>() {

@Override

public void onChanged(@Nullable List<TaskEntry> taskEntries) {

Log.*d*(*TAG*, "Updating list of tasks from LiveData in ViewModel");

mAdapter.setTasks(taskEntries);

}

});

}

\* ViewModelProviders.of parece estar obsolote, si se usa una versión mayor del framework, es necesario usar otro código como esteÑ

boardViewModel = ViewModelProvider(this).get(BoardViewModel::class.java)

## ViewModel Factory

Para el método loadTaskById, que carga un task especifico según el id que se le pase, si se va a crear un ViewModel, este necesita recibir el id también, para esto se necesita crear un ViewModelFactory.

Para trabajar con un ViewModelFactory, se crea una clase que extienda a **ViewModelProvider.NewInstanceFactory**.

En la clase, se pueden crear variables para el AppDatabase y otra para que reciba el id del Task y estas se inicializan en el constructor.

Se puede sobre escribir el método create para retornar un AddTaskViewModel nuevo que use nuestros parámetros en su constructor.

// Make this class extend ViewModel ViewModelProvider.NewInstanceFactory

public class AddTaskViewModelFactory extends ViewModelProvider.NewInstanceFactory {

// Add two member variables. One for the database and one for the taskId

private final AppDatabase mDb;

private final int mTaskId;

// Initialize the member variables in the constructor with the parameters received

public AddTaskViewModelFactory(AppDatabase mDb, int mTaskId) {

this.mDb = mDb;

this.mTaskId = mTaskId;

}

// Note: This can be reused with minor modifications

@Override

public <T extends ViewModel> T create(Class<T> modelClass) {

//noinspection unchecked

return (T) new AddTaskViewModel(mDb, mTaskId);

}

}

Esta clase AddTaskViewModel debe extender a ViewModel en lugar de AndroidViewModel ya que se esta usando un Factory para crearla. Ademas, el constructor recibiría un objeto AppDatabase y el id del task para así llamar al loadTaskById alli:

// Make this class extend ViewModel

public class AddTaskViewModel extends ViewModel {

// Add a task member variable for the TaskEntry object wrapped in a LiveData

private LiveData<TaskEntry> task;

// Create a constructor where you call loadTaskById of the taskDao to initialize the tasks variable

// Note: The constructor should receive the database and the taskId

public AddTaskViewModel(AppDatabase database, int taskId) {

task = database.taskDao().loadTaskById(taskId);

}

// Create a getter for the task variable

public LiveData<TaskEntry> getTask() {

return task;

}

}

Ahora, en el AddTaskActivity, donde se estaba llamando antes al loadTaskById, se puede crear un view model de AddTaskViewModel haciendo uso del factory y utilizando el observer desde ese viewModel:

Intent intent = getIntent();

if (intent != null && intent.hasExtra(*EXTRA\_TASK\_ID*)) {

mButton.setText(R.string.*update\_button*);

if (mTaskId == *DEFAULT\_TASK\_ID*) {

// populate the UI

mTaskId = intent.getIntExtra(*EXTRA\_TASK\_ID*, *DEFAULT\_TASK\_ID*);

// Declare a AddTaskViewModelFactory using mDb and mTaskId

AddTaskViewModelFactory factory = new AddTaskViewModelFactory(mDb, mTaskId);

// Declare a AddTaskViewModel variable and initialize it by calling ViewModelProviders.of

// for that use the factory created above AddTaskViewModel

final AddTaskViewModel viewModel = ViewModelProviders.*of*(this, factory).get(AddTaskViewModel.class);

// Observe the LiveData object in the ViewModel. Use it also when removing the observer

viewModel.getTask().observe(this, new Observer<TaskEntry>() {

@Override

public void onChanged(@Nullable TaskEntry taskEntry) {

viewModel.getTask().removeObserver(this);

Log.*d*(*TAG*, "Receiving database update from LiveData");

populateUI(taskEntry);

}

});

}

}

Al igual que en el caso del MainViewModel, el viewmodel se puede obtener al llamar al ViewModelProviders.of, pero en este caso se pasa tanto el activity como el factory.

## A “Lifecycle” Surprise

El flujo que se espera con la aplicación del ToDo list, es que al agregar una nueva tarea:

* Se guarde en la BD
* LiveData se actualice
* Observer sea notificado
* UI se actualice

Sin embargo, si la actividad AddTaskActivity no se cierra después de agregar una tarea nueva, lo esperado seria que se ejecuten las 2 primeras tareas, pero no las 2 ultimas, porque el observer esta en el MainActivity, no en el AddTaskActivity y por lo tanto el UI no debería tampoco actualizarse. Sin embargo, si se actualiza y esto es por el hecho de usar el Lifecycle de Android Architectura Components.

## Lifecycle Owners y LifecycleObservers

LiveData es un componente consciente del ciclo de vida. Gracias a esto, LiveData puede saber el estado de sus componentes asociados, como el activity. Cuando una actividad es iniciada o resumida, se considera como activa, y es hasta en ese momento donde sus observadores van a ser notificados.

Un beneficio también del LiveData es que sabrá cuando el estado del activity es “destruido” y cuando eso pasa, automáticamente des inscribe los observadores para evitar fugas de memoria.

Lifecycle tiene 2 interfaces:

* LifecycleOwners: objetos con un ciclo de vida como activities y fragments.
* LifecycleObserver: observan a los LifecycleOwners y reciben notificaciones en los cambios de ciclo de vida. LiveData es de este tipo. Cuando se llama al método observe se le indica cual objeto observar (activity o fragment).

Cualquier clase puede volverse un LifecycleOwner si implementa dicha clase.

Información adicional:

https://developer.android.com/topic/libraries/architecture/lifecycle.html

[https://developer.android.com/topic/libraries/architecture/lifecycle.html#use-cases](https://developer.android.com/topic/libraries/architecture/lifecycle.html" \l "use-cases)

## The Repository

Una recomendación para trabajar con base de datos en Android es mantener toda la lógica en un lugar en común, un Repositorio. Esto agrega mas flexibilidad a la arquitectura de la aplicación y ejemplifica el uso del Principio de “Single Responsbility” que dice que una clase debería tener solo una razón para cambiar.

Esta es una guía de como manejar la arquitectura del app: [https://developer.android.com/jetpack/guide#fetch-data](https://developer.android.com/jetpack/guide" \l "fetch-data)

Esto es un ejemplo de Repositorio para el app Sunshine:

package com.example.android.sunshine.data;

import android.arch.lifecycle.LiveData;

import android.util.Log;

import com.example.android.sunshine.AppExecutors;

import com.example.android.sunshine.data.database.ListWeatherEntry;

import com.example.android.sunshine.data.database.WeatherDao;

import com.example.android.sunshine.data.database.WeatherEntry;

import com.example.android.sunshine.data.network.WeatherNetworkDataSource;

import com.example.android.sunshine.utilities.SunshineDateUtils;

import java.util.Date;

import java.util.List;

/\*\*

\* Handles data operations in Sunshine. Acts as a mediator between {@link WeatherNetworkDataSource}

\* and {@link WeatherDao}

\*/

public class SunshineRepository {

private static final String LOG\_TAG = SunshineRepository.class.getSimpleName();

// For Singleton instantiation

private static final Object LOCK = new Object();

private static SunshineRepository sInstance;

private final WeatherDao mWeatherDao;

private final WeatherNetworkDataSource mWeatherNetworkDataSource;

private final AppExecutors mExecutors;

private boolean mInitialized = false;

private SunshineRepository(WeatherDao weatherDao,

WeatherNetworkDataSource weatherNetworkDataSource,

AppExecutors executors) {

mWeatherDao = weatherDao;

mWeatherNetworkDataSource = weatherNetworkDataSource;

mExecutors = executors;

// As long as the repository exists, observe the network LiveData.

// If that LiveData changes, update the database.

LiveData<WeatherEntry[]> networkData = mWeatherNetworkDataSource.getCurrentWeatherForecasts();

networkData.observeForever(newForecastsFromNetwork -> {

mExecutors.diskIO().execute(() -> {

// Deletes old historical data

deleteOldData();

Log.d(LOG\_TAG, "Old weather deleted");

// Insert our new weather data into Sunshine's database

mWeatherDao.bulkInsert(newForecastsFromNetwork);

Log.d(LOG\_TAG, "New values inserted");

});

});

}

public synchronized static SunshineRepository getInstance(

WeatherDao weatherDao, WeatherNetworkDataSource weatherNetworkDataSource,

AppExecutors executors) {

Log.d(LOG\_TAG, "Getting the repository");

if (sInstance == null) {

synchronized (LOCK) {

sInstance = new SunshineRepository(weatherDao, weatherNetworkDataSource,

executors);

Log.d(LOG\_TAG, "Made new repository");

}

}

return sInstance;

}

/\*\*

\* Creates periodic sync tasks and checks to see if an immediate sync is required. If an

\* immediate sync is required, this method will take care of making sure that sync occurs.

\*/

private synchronized void initializeData() {

// Only perform initialization once per app lifetime. If initialization has already been

// performed, we have nothing to do in this method.

if (mInitialized) return;

mInitialized = true;

// This method call triggers Sunshine to create its task to synchronize weather data

// periodically.

mWeatherNetworkDataSource.scheduleRecurringFetchWeatherSync();

mExecutors.diskIO().execute(() -> {

if (isFetchNeeded()) {

startFetchWeatherService();

}

});

}

/\*\*

\* Database related operations

\*\*/

public LiveData<List<ListWeatherEntry>> getCurrentWeatherForecasts() {

initializeData();

Date today = SunshineDateUtils.getNormalizedUtcDateForToday();

return mWeatherDao.getCurrentWeatherForecasts(today);

}

public LiveData<WeatherEntry> getWeatherByDate(Date date) {

initializeData();

return mWeatherDao.getWeatherByDate(date);

}

/\*\*

\* Deletes old weather data because we don't need to keep multiple days' data

\*/

private void deleteOldData() {

Date today = SunshineDateUtils.getNormalizedUtcDateForToday();

mWeatherDao.deleteOldWeather(today);

}

/\*\*

\* Checks if there are enough days of future weather for the app to display all the needed data.

\*

\* @return Whether a fetch is needed

\*/

private boolean isFetchNeeded() {

Date today = SunshineDateUtils.getNormalizedUtcDateForToday();

int count = mWeatherDao.countAllFutureWeather(today);

return (count < WeatherNetworkDataSource.NUM\_DAYS);

}

/\*\*

\* Network related operation

\*/

private void startFetchWeatherService() {

mWeatherNetworkDataSource.startFetchWeatherService();

}

}

## Recap

Usando Room se puede implementar una base de datos para persistir data.

Con LiveData, se pueden recibir notificaciones cuando ocurre algún cambio en la base de datos. Esto permite también actualizar el UI según el estado de la data sin tener que estar consultando a la base de datos a cada rato.

ViewModel permite guardar en cache los objetos con la data para que sobrevivan a cambios de configuración.

Lifecycle tiene 2 interfaces: un LifecycleOwner y LifecycleObserver que permite a objetos no-lifecycle ser lifecycle aware.

LiveData es un componente lifecycle aware, de hecho un LifecycleObserver.

Los observer se remueven automáticamente cuando la actividad es destruida.

# Lesson 09 – Background Tasks

## Running in the Background

Una de las características importantes de las apps de hoy en día, es poder trabajar en background. Esto implica un buen manejo de recursos, poder enviar notificaciones al usuario y poder tener la data actualizada cuando el usuario utilice el app, para esto se usan los otros dos componentes de Android que no se han visto aun: Servicios y Broadcast.

## Services

Algunas tareas no encajan en el ciclo de vida de una actividad, por ejemplo, realizar un cambio en la actividad de preferencias que requiera sincronizar la base de datos con el server, no tiene sentido hacer esta sincronización en el activity de preferencias, de hecho, no tiene sentido hacer transacciones de red que hablen con una base de datos dentro de una actividad. Para eso están los services.

Los 4 principales componentes de Android (Services, Activity, Content Providers y Broadcast Receivers) están descritos en el Android Manifest y conectados entre si a través de intents.

Los Services están hechos para correr tareas en background que no necesitan un componente visual. Por ejemplo, cargar y actualizar data incluso cuando ninguna actividad este abierta.

## Services vs Loaders

Un Loader se puede usar cuando:

* Cuando se esta cargando información que solo va a ser usada en la actividad, esto incluye algunas transacciones de red.
* Es fácil hacer cambios en el UI y comunicarse con la Actividad.

Un Service puede usarse cuando:

* Cuando la tarea a realizar esta desacoplada del UI.
* Cuando se inicia una tarea, se procesa, se carga o se descarga data que debe continuar incluso cuando no hay UI, por ejemplo, guardar info en la base de datos.

## Iniciando con Servicios

Hay 3 formas de iniciar servicios:

* Iniciarlos de forma manual
* Programar o calendarizar su inicio
* Bind o ligar el servicio

La formas mas simple es llamando el **startService()** desde un contexto como una actividad. Estos se inician pero no tienen una comunicación directa con quien lo inicio.

Si uno quiere que el servicio se inicie según se cumplan ciertas condiciones, se puede usar una clase llamada **JobService**. Con esta clase uno puede definir cuando se inicia el servicio usando un scheduler como **JobScheduler** o Fibase **JobDisptacher**.

En el tercer tipo, el servicio se vuelve como un Server y varios componentes ligados al servicio se vuelven los clientes, usando para esto el **bindService().** Estos si se pueden comunicar de vuelta con los componentes a los que esta ligado. Un ejemplo es el media player que tiene una UI.

Los servicios pueden servir tanto como un BackgroundService (para descargar o subir información) así como un JobService (para bajar información cada cierto tiempo).

Más información sobre servicios vinculados: <https://developer.android.com/guide/components/bound-services.html>

## Corriendo Servicios en Background

Todos los componentes de Android (los 4 principales) inician su vida en el hilo principal de la aplicación.

Los servicios también tienen un ciclo de vida pero este es diferente al de una actividad.

Para un servicio simple, el servicio es creado cuando el contexto (como un activity) llama al **startService()**, pasando un Intent, similar a como se inicia un activity con un intent usando startActivity. Lo que esto hace es disparar el método **onCreate** en el servicio.

El onCreate es el encargado de cualquier setup del servicio.

Luego se llama al **onStartCommand()** y dentro de este método es donde uno programa lo que el servicio hace. El servicio comienza a correr y ahí es donde comienza el AsyncTask o el hilo que corre el código de uno.

Cuando el servicio esta listo, se puede señalar con el **stopSelft()** que puede ser llamado por si mismo o por el cliente, aunque uno como desarrollador es quien suele llamarlo.

Al llamar al onStop, este termina llamando al **onDestroy()** que destruye el servicio.

## Intent Services

Un Intent Service es un servicio que corre en un hilo background totalmente separado, generalmente para procesos que consumen mucho tiempo.

Para usarlo, primero se crea una clase que extienda a IntentService (el servicio que uno creo).

Dentro de esta clase se especifica (mediante una subclase) lo que debe hacer el IntentService en el background al sobre escribir el método **onHandleIntent**.

Luego se crea un Intent y se le pasa ese service class.

Finalmente para iniciarlo, se llama al **startService(intent)** y que por dentro tiene al IntentService.

Ejemplo:

public class MyIntentService extends IntentService{

@Override

protected void onHandleIntent(@Nullable Intent intent) {

// do background work here

}

}

Intent myIntent = new Intent(this, MyIntentService.class);

startService(myIntent);

El intent que se pasa en el startService es el mismo que recibe el onHandleIntent y ese puede llevar data adjunta que se puede usar dentro del onHandleIntent.

Todos los request al IntentService que se hagan se manejan en un único hilo background y en el orden en que se reciben. Cada IntentService toma el tiempo necesario y no bloque el UI, pero solo una llamada al onHandleIntent del IntentService es procesado a la vez.

### Pluralization

Android tiene un mecanismo de “pluralizacion” de strings llamado “Quantity Strings”, por ejemplo:

<plurals name="charge\_notification\_count">

<item quantity="zero">Hydrate while charging reminder sent %d times</item>

<item quantity="one">Hydrate while charging reminder sent %d time</item>

<item quantity="other">Hydrate while charging reminder sent %d times</item>

</plurals>

Cuando se usa un plural uno especifica items quantity para varios números. Estos items especifican que string usar según el numero. En el ejemplo, se usan 3: un item para zero, uno para one y other.

En el MainActivity, se utiliza el string correcto así:

String formattedChargingReminders = getResources().getQuantityString(

R.plurals.*charge\_notification\_count*, chargingReminders, chargingReminders);

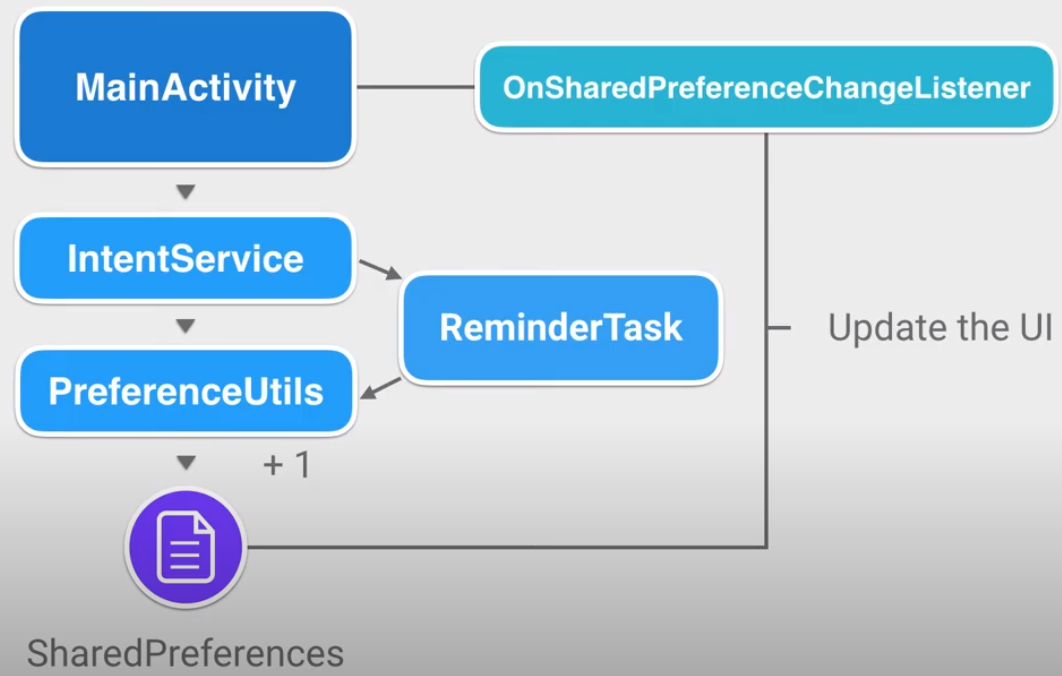
El primer uso de chargingReminders es el numero de cantidad y determina que versión del string plural usar (uno pasa un numero). El segundo uso de chargingReminders ie el numero que se inserta en el string.

Mas información: [https://developer.android.com/guide/topics/resources/string-resource.html#Plurals](https://developer.android.com/guide/topics/resources/string-resource.html" \l "Plurals)

### Ejemplo de IntentService

En el app Hydration-Reminder, en el primer ejercicio se tiene un MainActivity y un PreferenceUtilities. La idea es usar un IntentService para manejar el incremento del contador en el shared preference en el background como practica.

Para eso se crea una clase llamada ReminderTassks que define todas las tareas que van a correr en background para el app.



El MainActivity iniciara el IntentService que ejecutara la tarea incrementWaterCount en el ReminderTask y esta tarea incrementara el contador en los SharedPreferences usando PreferenceUtils.

Los pasos para implementar el IntentService son:

1. Crear una clase que extienda IntentService
2. Sobre escribir el método onHandleIntent
3. En el MainActivity, iniciar el servicio usando startService.

Ejemplo de Clase para las tareas que el IntentService va a ejecutar:

// Create a class called ReminderTasks

public class ReminderTasks {

// Create a public static constant String called ACTION\_INCREMENT\_WATER\_COUNT

public static final String *ACTION\_INCREMENT\_WATER\_COUNT* = "increment-water-count";

// Create a public static void method called executeTask

// Add a Context called context and String parameter called action to the parameter list

public static void executeTask(Context context, String action){

// If the action equals ACTION\_INCREMENT\_WATER\_COUNT, call this class's incrementWaterCount

if(action == *ACTION\_INCREMENT\_WATER\_COUNT*){

*incrementWaterCount*(context);

}

}

// Create a private static void method called incrementWaterCount

// Add a Context called context to the argument list

private static void incrementWaterCount(Context context){

// From incrementWaterCount, call the PreferenceUtility method that will ultimately update the water count

PreferenceUtilities.*incrementWaterCount*(context);

}

}

Ejemplo de IntentService:

// Create WaterReminderIntentService and extend it from IntentService

public class WaterReminderIntentService extends IntentService{

// Create a default constructor that calls super with the name of this class

public WaterReminderIntentService() {

super("WaterReminderIntentService");

}

// Override onHandleIntent

@Override

protected void onHandleIntent(@Nullable Intent intent) {

// Get the action from the Intent that started this Service

String action = intent.getAction();

// Call ReminderTasks.executeTask and pass in the action to be performed

ReminderTasks.*executeTask*(this, action);

}

}

En el IntentService se extiende IntentService, se crea un constructor por default con la llamada a super y enviándole el nombre de la clase y se sobre escribe el onHandleIntent para ejecutar la tarea especificada con el argumento action.

En el MainActivity, en el método donde se incrementa el contador se crea un intent que llamara al service y se pasa el action a través del setAction:

// Create an explicit intent for WaterReminderIntentService

Intent incrementWaterCountIntent = new Intent(this, WaterReminderIntentService.class);

// Set the action of the intent to ACTION\_INCREMENT\_WATER\_COUNT

incrementWaterCountIntent.setAction(ReminderTasks.*ACTION\_INCREMENT\_WATER\_COUNT*);

// Call startService and pass the explicit intent you just created

startService(incrementWaterCountIntent);

Al ejecutar esto, como se tiene registrado el shared preferences para que este escuchando en caso de cambios en el preference, el UI va a desplegar el contador actualizado inmediatamente.

Como cualquier componente, los servicios necesitan ser registrados en el AndroidManifest:

<service android:name=".sync.WaterReminderIntentService"

android:exported="false" />

Este atributo exported trabaja igual que el exported para content providers y controla si otras aplicaciones pueden acceder al servicio o no. Esto es importante si uno quiere que solo el app de uno pueda llamar a dicho servicio.

## Notifications

Las notificaciones no son simples mensajes, también permiten realizar unas cuantas acciones sobre los datos incluidos en la notificación.

Información sobre diseño de notificaciones de Android Material:

[https://material.io/design/platform-guidance/android-notifications.html#usage](https://material.io/design/platform-guidance/android-notifications.html" \l "usage)

Descripción General de Notificaciones:

https://developer.android.com/guide/topics/ui/notifiers/notifications.html

### Notification Channels

Con cada actualización de Android, las notificaciones reciben updates importantes. Desde la versión Oreo, las notificaciones introducen algunos cambios como los Notification Channels, que permiten al usuario controlar las notificaciones con settings de UI consistentes.

Los Notification Channels o también llamados categorías, son colecciones de notificaciones asociadas. Cada notificación en Oreo necesita un canal o no se van a mostrar. Los notification channels también se pueden agrupar en Notification Groups para organizarlos mejor en el UI

Los Notification Channels son importantes porque al hacer una notificación, uno pasa el channel ID para que pueda ser mostrada.

Ademas de los channels, también se agrego Notification Badges o puntos que aparecen en las apps los cuales muestran al usuario que hay notificaciones. Al presionar por un rato sobre el icono de la aplicación que tiene el punto, se muestra la información sobre la notificación.

También se tienen otras características como poder colorear el fondo de las notificaciones con el método setColorized().

Otra es poder ordenar las notificaciones por categoría, prioridad y personas asociadas. Y el MessagingStyle despliega mas contenido en la forma colapsada.

## Pending Intents

Los intents permiten ejecuta varias actividades dentro de la misma app o bien ejecutarlas en otras aplicaciones, de forma explicita o implícita usando URIs. Pero, que pasa si uno quiere que otra aplicación abra un activity en nuestra aplicación, como con una notificación?

Todas las notificaciones son administradas por un servicio del sistema llamado NotificationManager. Para que este servicio pueda abrir un activity en el app, es necesario un PendingIntent. Estos intents son wrappers diseñados para ser usados por otra aplicación. Estos le dan a esa aplicación la habilidad de ejecutar la acción incluida como si fuera parte de su aplicación, con los permisos requeridos, inclusive si nuestra aplicación no esta corriendo, lo cual es útil para las notificaciones.

Para crear una instancia de PendingIntents, hay varios métodos estáticos como **getActivity**, **getService**, **getBroadcast**, dependiendo del intent que uno quiere “envolver”. Todos estos métodos requieren 4 parámetros, por ejemplo para getActivity:

* context: el contexto en el que el el PendingIntent debe comenzar el activity.
* requestCode: un código que se puede usar en caso que uno desee cancelar el activity.
* intent: el intent de la actividad que uno quiere ejecutar.
* flag: uno de múltiples intents que se pueden usar para manejar crear múltiples PendingIntents para el mismo intent.

Para mas información: <https://developer.android.com/reference/android/app/PendingIntent>

Ejemplo de crear un PendingIntent:

// Create a helper method called contentIntent with a single parameter for a Context. It

// should return a PendingIntent. This method will create the pending intent which will trigger when

// the notification is pressed. This pending intent should open up the MainActivity.

private static PendingIntent contentIntent(Context context) {

// Create an intent that opens up the MainActivity

Intent startActivity = new Intent(context, MainActivity.class);

// Create a PendingIntent using getActivity that:

// - Take the context passed in as a parameter

// - Takes an unique integer ID for the pending intent (you can create a constant for

// this integer above

// - Takes the intent to open the MainActivity you just created; this is what is triggered when the notification is triggered

// - Has the flag FLAG\_UPDATE\_CURRENT, so that if the intent is created again, keep the

// intent but update the data

return PendingIntent.*getActivity*(context,

*WATER\_REMINDER\_PENDING\_INTENT\_ID*,

startActivity,

PendingIntent.*FLAG\_UPDATE\_CURRENT*);

}

Un ejemplo mas completo de un ejercicio donde se despliega el notification mediante un botón en el layout del MainActivity:

En el AndroidManifest primero se agrega el permiso para que el dispositivo vibre y al MainActivity se le agrega el launchMode en singleTop para que, si el activity esta abierto en el momento en el que el usuario le da click al notification no reabra el activity:

<!-- Add the VIBRATE permission -->

<uses-permission android:name="android.permission.VIBRATE"/>

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

android:theme="@style/AppTheme">

<!-- Add the launch mode "single top" attribute to the MainActivity so that when you open

MainActivity using a notification, it will take you back to the already opened activity, not

generate a new one -->

<activity

android:name=".MainActivity"

android:screenOrientation="portrait"

android:launchMode="singleTop">

Se creo una clase llamada NotificationUtils que muestra los siguiente:

* Algunas constantes para IDs necesarios (uno para el PendingIntent, Notification Channel y para el Notification que se puede usar para cancelarlo)
* El método remindUserBecauseCharging donde
  + Se obtiene un NotificationManager del getSystemService
  + Se crea un NotificationChannel verificando la versión del android.
  + Se crea un NotificationBuilder para setear el ID y otras propiedades de la notificación
  + Disparar la notificación con el notify() del notificacionManager.
* El método contentIntent permite crear el PendingIntent que abrirá MainActivity.
* El método largeIcon que devuelve un Bitmap para el icono a mostrar en el notification.

*/\*\**

*\* Utility class for creating hydration notifications*

*\*/*

public class NotificationUtils {

// Constant for the Pending Intent ID

private static final int *WATER\_REMINDER\_PENDING\_INTENT\_ID* = 6613;

// Constant for the Notification Channel Id to display the notification

private static final String *WATER\_REMINDER\_NOTIFICATION\_CHANNEL\_ID* = "reminder\_notification\_channel";

// Defines a Notification ID that can be used to cancel it later

private static final int *WATER\_REMINDER\_NOTIFICATION\_ID* = 1023;

// Create a method called remindUserBecauseCharging which takes a Context.

// This method will create a notification for charging. It might be helpful

// to take a look at this guide to see an example of what the code in this method will look like:

// https://developer.android.com/training/notify-user/build-notification.html

public static void remindUserBecauseCharging(Context context) {

// Get the NotificationManager using context.getSystemService

NotificationManager notificationManager = (NotificationManager)context

.getSystemService(Context.*NOTIFICATION\_SERVICE*);

// Create a notification channel for Android O devices

NotificationChannel mChannel = null;

if (android.os.Build.VERSION.*SDK\_INT* >= android.os.Build.VERSION\_CODES.*O*) {

mChannel = new NotificationChannel(

*WATER\_REMINDER\_NOTIFICATION\_CHANNEL\_ID*,

context.getString(R.string.*main\_notification\_channel\_name*),

NotificationManager.*IMPORTANCE\_HIGH*);

notificationManager.createNotificationChannel(mChannel);

}

// In the remindUserBecauseCharging method use NotificationCompat.Builder to create a notification

// that:

// - has a color of R.color.colorPrimary - use ContextCompat.getColor to get a compatible color

// - has ic\_drink\_notification as the small icon

// - uses icon returned by the largeIcon helper method as the large icon

// - sets the title to the charging\_reminder\_notification\_title String resource

// - sets the text to the charging\_reminder\_notification\_body String resource

// - sets the style to NotificationCompat.BigTextStyle().bigText(text)

// - sets the notification defaults to vibrate

// - uses the content intent returned by the contentIntent helper method for the contentIntent

// - automatically cancels the notification when the notification is clicked

NotificationCompat.Builder notificationBuilder =

new NotificationCompat.Builder(context, *WATER\_REMINDER\_NOTIFICATION\_CHANNEL\_ID*)

.setColor(ContextCompat.*getColor*(context, R.color.*colorPrimary*))

.setSmallIcon(R.drawable.*ic\_drink\_notification*)

.setLargeIcon(*largeIcon*(context))

.setContentTitle(context.getString(R.string.*charging\_reminder\_notification\_title*))

.setContentText(context.getString(R.string.*charging\_reminder\_notification\_body*))

.setStyle(new NotificationCompat.BigTextStyle().bigText(

context.getString(R.string.*charging\_reminder\_notification\_body*)

))

.setDefaults(Notification.*DEFAULT\_VIBRATE*)

.setContentIntent(*contentIntent*(context))

.setAutoCancel(true);

// If the build version is greater than or equal to JELLY\_BEAN and less than OREO,

// set the notification's priority to PRIORITY\_HIGH.

if(Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*JELLY\_BEAN* &&

Build.VERSION.*SDK\_INT* < Build.VERSION\_CODES.*O*){

notificationBuilder.setPriority(NotificationCompat.*PRIORITY\_HIGH*);

}

// Trigger the notification by calling notify on the NotificationManager.

// Pass in a unique ID of your choosing for the notification and notificationBuilder.build()

notificationManager.notify(*WATER\_REMINDER\_NOTIFICATION\_ID*, notificationBuilder.build());

}

// Create a helper method called contentIntent with a single parameter for a Context. It

// should return a PendingIntent. This method will create the pending intent which will trigger when

// the notification is pressed. This pending intent should open up the MainActivity.

private static PendingIntent contentIntent(Context context) {

// Create an intent that opens up the MainActivity

Intent startActivity = new Intent(context, MainActivity.class);

// Create a PendingIntent using getActivity that:

// - Take the context passed in as a parameter

// - Takes an unique integer ID for the pending intent (you can create a constant for

// this integer above

// - Takes the intent to open the MainActivity you just created; this is what is triggered

// when the notification is triggered

// - Has the flag FLAG\_UPDATE\_CURRENT, so that if the intent is created again, keep the

// intent but update the data

return PendingIntent.*getActivity*(context,

*WATER\_REMINDER\_PENDING\_INTENT\_ID*,

startActivity,

PendingIntent.*FLAG\_UPDATE\_CURRENT*);

}

// Create a helper method called largeIcon which takes in a Context as a parameter and

// returns a Bitmap. This method is necessary to decode a bitmap needed for the notification.

private static Bitmap largeIcon(Context context) {

// Get a Resources object from the context.

Resources resources = context.getResources();

// Create and return a bitmap using BitmapFactory.decodeResource, passing in the

// resources object and R.drawable.ic\_local\_drink\_black\_24px

Bitmap largeIcon = BitmapFactory.*decodeResource*(resources, R.drawable.*ic\_local\_drink\_black\_24px*);

return largeIcon;

}

}

Otra información para crear notificaciones expandibles:

https://developer.android.com/training/notify-user/expanded.html

Mas información sobre BigPictureStyle:

https://developer.android.com/reference/android/app/Notification.BigPictureStyle.html

### TaskStackBuilder

A veces uno crea un intent que es lo que el Pending Intent va a mostrar, pero también puede querer que el usuario cuando presione el back button pueda navegar a otro intent. Por ejemplo, si se crea un Pending Intent que va a desplegar un DetailsActivity pero uno quiere que al presionar el back button el usuario pueda navergar al MainActivity uno puede usar un TaskStackBuilder.

Este objeto básicamente permite definir una actividad padre para el intent a traves del metodo addParentStack. Y luego, de este objeto se puede obtener el Pending Intent. Por ejemplo:

// (3) Create an Intent with the proper URI to start the DetailActivity

Intent detailActivity = new Intent(context, DetailActivity.class);

detailActivity.setData(todaysWeatherUri);

// (4) Use TaskStackBuilder to create the proper PendingIntent

TaskStackBuilder stackBuilder = TaskStackBuilder.*create*(context);

// Set the addParentStack to specify that we want to navigate to the MainActivity

// from the DetailActivity

stackBuilder.addParentStack(MainActivity.class);

stackBuilder.addNextIntent(detailActivity);

PendingIntent pendingIntent = stackBuilder

.getPendingIntent(0, PendingIntent.*FLAG\_UPDATE\_CURRENT*);

// (5) Set the content Intent of the NotificationBuilder

notificationBuilder.setContentIntent(pendingIntent);

## Notification Actions

En las versiones actuales, las notificaciones pueden desplegar hasta 3 botones para ejecutar diferentes PendingIntents que a su vez trabajen con diferentes acciones en el app.

Para agregar acciones a la notificación se usa el método **addAction()** hasta 3 veces cuando se esta creando el notificationBuilder. Este método recibe un parámetro de tipo NotificationAction.

Dentro de NotificationUtils se crea un método para limpiar las notificaciones:

// Create a method to clear all notifications

public static void clearAllNotifications(Context context){

NotificationManager notificationManager = (NotificationManager)context

.getSystemService(Context.*NOTIFICATION\_SERVICE*);

notificationManager.cancelAll();

}

En otra clase llamada ReminderTasks, donde se tiene el método para ejecutar la tarea según el id o action que se pase, se crea otro action para cuando el usuario presiona el botón “No, thanks” que lo que hace es limpiar las notificaciones. Este método también se llama al incrementar el contador de agua:

public class ReminderTasks {

public static final String *ACTION\_INCREMENT\_WATER\_COUNT* = "increment-water-count";

// Add a public static constant called ACTION\_DISMISS\_NOTIFICATION

public static final String *ACTION\_DISMISS\_NOTIFICATION* = "dismiss-notification";

public static void executeTask(Context context, String action) {

if (*ACTION\_INCREMENT\_WATER\_COUNT*.equals(action)) {

*incrementWaterCount*(context);

}

// If the user ignored the reminder, clear the notification

else if(*ACTION\_DISMISS\_NOTIFICATION*.equals(action)){

NotificationUtils.*clearAllNotifications*(context);

}

}

private static void incrementWaterCount(Context context) {

PreferenceUtilities.*incrementWaterCount*(context);

// If the water count was incremented, clear any notifications

NotificationUtils.*clearAllNotifications*(context);

}

}

Se crean dos métodos estáticos para las acciones, uno para ignorar el reminder:

// Add a static method called ignoreReminderAction

private static NotificationCompat.Action ignoreReminderAction(Context context){

// Create an Intent to launch WaterReminderIntentService

Intent ignoreReminderIntent = new Intent(context, WaterReminderIntentService.class);

// Set the action of the intent to designate you want to dismiss the notification

ignoreReminderIntent.setAction(ReminderTasks.*ACTION\_DISMISS\_NOTIFICATION*);

// Create a PendingIntent from the intent to launch WaterReminderIntentService

PendingIntent ignoreReminderPendingIntent = PendingIntent.*getService*(

context,

*ACTION\_IGNORE\_PENDING\_INTENT\_ID*,

ignoreReminderIntent,

PendingIntent.*FLAG\_UPDATE\_CURRENT* );

// Create an Action for the user to ignore the notification (and dismiss it)

NotificationCompat.Action ignoreReminderAction = new NotificationCompat.Action(

R.drawable.*ic\_cancel\_black\_24px*,

context.getString(R.string.*action\_button\_ignore\_reminder*),

ignoreReminderPendingIntent);

// Return the action

return ignoreReminderAction;

}

// Add a static method called drinkWaterAction

private static NotificationCompat.Action drinkWaterAction(Context context) {

// Create an Intent to launch WaterReminderIntentService

Intent drinkWaterIntent = new Intent(context, WaterReminderIntentService.class);

// Set the action of the intent to designate you want to increment the water count

drinkWaterIntent.setAction(ReminderTasks.*ACTION\_INCREMENT\_WATER\_COUNT*);

// Create a PendingIntent from the intent to launch WaterReminderIntentService

PendingIntent drinkWaterPendingIntent = PendingIntent.*getService*(

context,

*ACTION\_DRINK\_WATER\_INTENT\_ID*,

drinkWaterIntent,

PendingIntent.*FLAG\_UPDATE\_CURRENT*);

// Create an Action for the user to tell us they've had a glass of water

NotificationCompat.Action drinkWaterAction = new NotificationCompat.Action(

R.drawable.*ic\_drink\_notification*,

context.getString(R.string.*action\_button\_drink\_water\_reminder*),

drinkWaterPendingIntent);

// Return the action

return drinkWaterAction;

}

Por ultimo, estas dos acciones se deben agregar al notificationBuilder al crearlo:

NotificationCompat.Builder notificationBuilder = new NotificationCompat.Builder(context,*WATER\_REMINDER\_NOTIFICATION\_CHANNEL\_ID*)

.setColor(ContextCompat.*getColor*(context, R.color.*colorPrimary*))

.setSmallIcon(R.drawable.*ic\_drink\_notification*)

.setLargeIcon(*largeIcon*(context))

.setContentTitle(context.getString(R.string.*charging\_reminder\_notification\_title*))

.setContentText(context.getString(R.string.*charging\_reminder\_notification\_body*))

.setStyle(new NotificationCompat.BigTextStyle().bigText(

context.getString(R.string.*charging\_reminder\_notification\_body*)))

.setDefaults(Notification.*DEFAULT\_VIBRATE*)

.setContentIntent(*contentIntent*(context))

// Add the two new actions using the addAction method and your helper methods

.addAction(*drinkWaterAction*(context))

.addAction(*ignoreReminderAction*(context))

.setAutoCancel(true);

## Foreground Services

Este tipo de servicios son como los del media player o el maps. Son servicios donde el usuario esta consciente de que están corriendo porque Android requiere que el servicio muestre una notificación continua.

Generalmente se usan para mostrar el progreso de alguna operación que toma mucho tiempo en finalizar.

Android le dará prioridad a los foreground services, incluso si el sistema tiene poca memoria.

## Prioridad de la Aplicación

Android maneja 4 categorías para las apps y servicios para determinar cuales detener si ocupa recursos:



Se puede ver como 3 leyes:

* Android mantendrá todas las apps que interactúan con el usuario corriendo
* Android mantendrá todas las apps con actividades visibles seguidas por sus servicios corriendo, a no ser que violen la primera ley.
* Android mantendrá todas las apps corriendo en background, a no ser que violen la primera o segunda ley.

## Scheduling Jobs

Hay varias formas de calendarizar jobs, una es con el FirebaseJobDispatcher y otra es con el WorkManager, esta ultima parece ser la forma preferida. Para ver la información de WorkManager esta esta documentación:

https://developer.android.com/topic/libraries/architecture/workmanager

FireaseJobDispatcher necesita que uno cree un GooglePlayDriver para luego usarlo al crear una nueva instancia del FirebaseJobDispatcher.

Luego uno crea un Job usando el metodo newJobBuilder del dispatcher y especificando todos los constraints.

El tiempo se puede setear con el .setTrigger(Trigger.executionWindow(0,900)) donde el tiempo es dado en segundos. El primero especifica el tiempo mínimo para iniciar, y el segundo el máximo antes de que se ejecute. Al especificar esta ventana uno deja que el sistema escoja el mejor tiempo para calendarizar el job en esa ventana.

Ejemplo del JobDispatcher:

Driver driver = **new** GooglePlayDriver(context);

FirebaseJobDispatcher dispatcher = **new** FirebaseJobDispatcher(driver);

Job myJob = dispatcher.newJobBuilder()

*// the JobService that will be called*

.setService(MyJobService.**class**)

*// uniquely identifies the job*

.setTag("complex-job")

*// one-off job*

.setRecurring(false)

*// don't persist past a device reboot*

.setLifetime(Lifetime.UNTIL\_NEXT\_BOOT)

*// start between 0 and 15 minutes (900 seconds)*

.setTrigger(Trigger.executionWindow(0, 900))

*// overwrite an existing job with the same tag*

.setReplaceCurrent(true)

*// retry with exponential backoff*

.setRetryStrategy(RetryStrategy.DEFAULT\_EXPONENTIAL)

*// constraints that need to be satisfied for the job to run*

.setConstraints(

*// only run on an unmetered network*

Constraint.ON\_UNMETERED\_NETWORK,

*// only run when the device is charging*

Constraint.DEVICE\_CHARGING

)

.build();

## Agregando un JobService con FirebaseJobDisptacher

Para el ejemplo con FirebaseJobDispatcher se ocupa realizar lo siguiente:

1. Agregar la dependencia en Gradle
2. Crear una nueva tarea en el ReminderTasks
3. Crear un nuevo Service que extienda desde JobService. Por defecto, los JobServices se ejecutan en el main thread, por lo que se pueden crear clases abstractas que extiendan al AsyncTask para evitar problemas de recursos. En el método **onStartJob**() se crea la tare asíncrona. La tarea asíncrona va a ocupar el doInBackground que se ocupara de ejecutar la tarea de charging reminder y se sobreescribe el onPostExecute para llamar al jobFinished e indicar que la tarea ha terminado. El método **onJobStop()** es llamado si los requerimientos del job no se cumplen. El jobFinished ocupa 2 argumentos, un JobParameters que es un bundle y un boolean para saber si se necesita recalendarizar el job.
4. Agregar el JobService al manifest
5. Calendarizar con FirebaseJobDispatcher

Dependencia en el Gradle:

// COMPLETED (1) Add gradle dependency for Firebase Job Dispatcher

implementation 'com.firebase:firebase-jobdispatcher:0.8.5'

Nueva tarea en el ReminderTasks:

public static final String *ACTION\_CHARGING\_REMINDER* = "charging\_reminder";

public static void executeTask(Context context, String action) {

if (*ACTION\_INCREMENT\_WATER\_COUNT*.equals(action)) {

*incrementWaterCount*(context);

} else if (*ACTION\_DISMISS\_NOTIFICATION*.equals(action)) {

NotificationUtils.*clearAllNotifications*(context);

} else if(*ACTION\_CHARGING\_REMINDER*.equals(action)){

*issueChargingReminder*(context);

}

}

// COMPLETED (2) Create an additional task for issuing a charging reminder notification.  
// This should be done in a similar way to how you have an action for incrementingWaterCount  
// and dismissing notifications. This task should both create a notification AND  
// increment the charging reminder count (hint: there is a method for this in PreferenceUtilities)  
// When finished, you should be able to call executeTask with the correct parameters to execute  
// this task. Don't forget to add the code to executeTask which actually calls your new task!  
private static void issueChargingReminder(Context context){  
 PreferenceUtilities.*incrementWaterCount*(context);  
 NotificationUtils.*remindUserBecauseCharging*(context);  
}

Ejemplo del Service

// COMPLETED (3) WaterReminderFirebaseJobService should extend from JobService

@RequiresApi(api = Build.VERSION\_CODES.*LOLLIPOP*)

public class WaterReminderFirebaseJobService extends JobService {

private AsyncTask mBackground;

// COMPLETED (4) Override onStartJob

@Override

public boolean onStartJob(final JobParameters jobParameters) {

// COMPLETED (5) By default, jobs are executed on the main thread, so make an anonymous class extending

// AsyncTask called mBackgroundTask.

mBackground = new AsyncTask() {

// COMPLETED (6) Override doInBackground

@Override

protected Object doInBackground(Object[] objects) {

// COMPLETED (7) Use ReminderTasks to execute the new charging reminder task you made, use

// this service as the context (WaterReminderFirebaseJobService.this) and return null

// when finished.

Context context = WaterReminderFirebaseJobService.this;

ReminderTasks.*executeTask*(context, ReminderTasks.*ACTION\_CHARGING\_REMINDER*);

return null;

}

// COMPLETED (8) Override onPostExecute and call jobFinished. Pass the job parameters

// and false to jobFinished. This will inform the JobManager that your job is done

// and that you do not want to reschedule the job.

@Override

protected void onPostExecute(Object o) {

jobFinished(jobParameters, false);

}

};

// COMPLETED (9) Execute the AsyncTask

mBackground.execute();

// COMPLETED (10) Return true to specify the job is still doing some work

return true;

}

// COMPLETED (11) Override onStopJob

@Override

public boolean onStopJob(JobParameters jobParameters) {

// COMPLETED (12) If mBackgroundTask is valid, cancel it

if(mBackground != null) mBackground.cancel(true);

// COMPLETED (13) Return true to signify the job should be retried when the

// conditions of the job are re-met

return true;

}

}

Al manifest también se agrega el servicio para el FirebaseJobService con el exported en false y con un intent-filter para el action ACTION\_EXECUTE

should have an intent filter for the action com.firebase.jobdispatcher.ACTION\_EXECUTE -->

<service android:name=".sync.WaterReminderFirebaseJobService"

android:exported="false">

<intent-filter>

<action android:name="com.firebase.jobdispatcher.ACTION\_EXECUTE"/>

</intent-filter>

</service>

Para calendarizar el job, se creo una clase utilitaria para el método que se encargara de eso. Esta clase es ReminderUtilities:

public class ReminderUtilities {

// COMPLETED (15) Create three constants and one variable:

private static final int *REMINDER\_INTERVAL\_MINUTES* = 1;

// - REMINDER\_INTERVAL\_SECONDS should be an integer constant storing the number of seconds in 15 minutes

private static final int *REMINDER\_INTERVAL\_SECONDS* = (int)(TimeUnit.*MINUTES*.toSeconds(*REMINDER\_INTERVAL\_MINUTES*));

// - SYNC\_FLEXTIME\_SECONDS should also be an integer constant storing the number of seconds in 15 minutes

private static final int *SYNC\_FLEXTIME\_SECONDS* = *REMINDER\_INTERVAL\_SECONDS*;

// - REMINDER\_JOB\_TAG should be a String constant, storing something like "hydration\_reminder\_tag"

private static final String *REMINDER\_JOB\_TAG* = "hydration\_reminder\_tag";

// - sInitialized should be a private static boolean variable which will store whether the job

// has been activated or not

private static boolean *sInitialized*;

// COMPLETED (16) Create a synchronized, public static method called scheduleChargingReminder that takes

// in a context. This method will use FirebaseJobDispatcher to schedule a job that repeats roughly

// every REMINDER\_INTERVAL\_SECONDS when the phone is charging. It will trigger WaterReminderFirebaseJobService

// Checkout https://github.com/firebase/firebase-jobdispatcher-android for an example

synchronized public static void scheduleChargingReminder(Context context) {

// COMPLETED (17) If the job has already been initialized, return

if(*sInitialized*) return;

// COMPLETED (18) Create a new GooglePlayDriver

Driver driver = new GooglePlayDriver(context);

// COMPLETED (19) Create a new FirebaseJobDispatcher with the driver

FirebaseJobDispatcher dispatcher = new FirebaseJobDispatcher(driver);

// COMPLETED (20) Use FirebaseJobDispatcher's newJobBuilder method to build a job which:

Job constraintReminderJob = dispatcher.newJobBuilder()

// - has WaterReminderFirebaseJobService as it's service

.setService(WaterReminderFirebaseJobService.class)

// - has the tag REMINDER\_JOB\_TAG

.setTag(*REMINDER\_JOB\_TAG*)

// - only triggers if the device is charging

.setConstraints(Constraint.*DEVICE\_CHARGING*)

// - has the lifetime of the job as forever

.setLifetime(Lifetime.*FOREVER*)

// - has the job recurring

.setRecurring(true)

// - occurs every 15 minutes with a window of 15 minutes. You can do this using a

// setTrigger, passing in a Trigger.executionWindow

.setTrigger(Trigger.*executionWindow*(

*REMINDER\_INTERVAL\_SECONDS*,

*REMINDER\_INTERVAL\_SECONDS* + *SYNC\_FLEXTIME\_SECONDS*))

// - replaces the current job if it's already running

.setReplaceCurrent(true)

// Finally, you should build the job.

.build();

// COMPLETED (21) Use dispatcher's schedule method to schedule the job

dispatcher.schedule(constraintReminderJob);

// COMPLETED (22) Set sInitialized to true to mark that we're done setting up the job

*sInitialized* = true;

}

}

El método scheduleChargingReminder es **synchronized** porque uno no quiere que el método se ejecute mas de una vez a la vez.

## Migrando desde FirebaseJobDispatcher a WorkManager

Para las nuevas versiones de Android ahora se recomienda usa WorkManager en lugar de FirebaseJobDispatcher. Para migrara a WorkManager desde FirebaseJobDispatcher hay que hacer ciertos pasos:

* Actualizar build.gradle para que use las dependencias de WorkManager
* Actualizar las clases que extienden a JobService por un worker. La unidad fundarmental de WorkManager es el **ListenableWorker** aunque hay otros como Worker, RxWorker y CoroutineWorker.
* Implementar los métodos de ListenableWorker (startWork) y un constructor que reciba un context y un WorkerParameters.
* El **startWork()** es el subproceso principal que devuelve un ListenableFuture que se usa para indicar la finalización del trabaja de forma asíncrona.
* Eventualmente, ListenableFuture muestra un tipo ListenableWorker.Result que puede ser Result.success(), Result.success(Data outputData), Result.retry(), Result.failure() o Result.failure(Data outputData).
* Se llama a **onStopped()** para indicar que ListenableWorker necesita detenerse, ya sea porque ya no se cumplen las restricciones (por ejemplo, porque la red ya no está disponible) o porque se llamó a un método WorkManager.cancel…(). También se puede llamar a onStopped() si el SO decide finalizar el trabajo por algún motivo.

Información completa aquí:

https://developer.android.com/topic/libraries/architecture/workmanager/migrating-fb

Como iniciar con WorkManager:

https://developer.android.com/topic/libraries/architecture/workmanager/basics

## Broadcast Receivers

Los System Broadcast Intent son un intent especial enviado por el sistema cuando ocurre un evento en el dispositivo.

Algunos ejemplos son:

* android.intent.action.SCREEN\_ON
* android.intent.action.HEADSET\_PLUG
* android.intent.action.DOWNLOAD\_COMPLETE

Un broadcast receiver es un componente que permite a las apps recibir intents que son transmitidos por el sistema o por otras apps. Un broadcast receiver puede ser disparado incluso cuando el app no esta corriendo.

Esto es lo que se usa cuando uno requiere saber si un cambio ha ocurrido. Gracias al IntentFilter uno puede especificar el receiver en el que uno esta interesado.

Los IntentFilter no son específicos de los broadcast receivers. Por ejemplo, el main activity suele tener un intent filter que especifica que esta es la actividad principal y la que se desplegara cuando el app se abra.

Hay 2 tipos de Broadcast Receivers:

* **Static**: estos se disparan siempre que la transmision del intent ocurre. El receiver se dispara aun si el app esta offline
* **Dynamic**: este se apega al ciclo de vida del app. Es mejor apegarse a estos o bien usar jobs scheduler.

Para usarlo se ocupa registrar el receiver en el AndroidManifest y especificar un intent-filter.

Para el receiver se crea una clase que extiende **BroadcastReceiver** y sobre escribe un método **onReceive** que recibe un Context y un Intent como argumentos.

Hay que tener cuidado sobre los broadcast intents que se definen ya que algunos podrían ralentizar el dispositivo y hasta sentirse un poco abusivo para el usuario. Algunos intents no van a dejar que uno los defina de forma estática, específicamente los que tienen el flag FLAG\_RECEIVER\_REGISTERED\_ONLY.

Entonces, como recomendación:

* Si el app no esta iniciada:
  + Usar un Job cuando se pueda, ya que estos manejan limitaciones para las situaciones mas comunes y son battery-friendly.
  + En algunos casos se puede ocupar usar receivers estáticos.
* Si el app esta iniciada:
  + Utilizar un broadcast receiver dinámico.

El broadcas receiver dinamico luce parecido al estatico, pero la clas ocupa que uno registre y des-registre el receiver, generalmente en los métodos onResume y on onPause respectivamente.

### NOTA Sobre ADB

Una forma sencilla de emular el plug and unplug del cargador para el emulador es abrir el adb en la linea de comando, por ejemplo, con un cmd como admin, ir a:

<YOUR SDK LOCATION>/platform-tools/adb

Algunos comandos útiles son:

* Simular que el telefono esta desconectado del cargador usb:
  + adb shell dumpsys battery set usb 0
  + adb shell dumpsys battery set ac 0
* Si el device tiene Android 6.0 o mas, tambien se puede usar:
  + adb shell dumpsys battery unplug
* Para simular que esta conectado al cargador:
  + adb shell dumpsys battery reset

Ejemplo:

En el ejemplo siguiente se tiene un activity que declara un IntentFilter al que se inicializa y se le agregan dos acciones en el onCreate, una para detectar cuando el cable de corriente se conecto y otra cuando se desconecto.

Se crea una clase para el BroadcastReceiver el cual tomara el action del intent que recibe y lo compara para saber si el action es el cable de corriente conectado, si es así ejecuta un metodo que cambia una imagen en la pantalla.

Por ultimo, el broadcast receiver se registra en el onResume y se desregistra en el onPause.

public class MainActivity extends AppCompatActivity implements

SharedPreferences.OnSharedPreferenceChangeListener {

private TextView mWaterCountDisplay;

private TextView mChargingCountDisplay;

private ImageView mChargingImageView;

private Toast mToast;

IntentFilter mChargingIntentFilter;

ChargingBroadcastReceiver mChargingReceiver;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.*activity\_main*);

*/\*\* Get the views \*\*/*

mWaterCountDisplay = (TextView) findViewById(R.id.*tv\_water\_count*);

mChargingCountDisplay = (TextView) findViewById(R.id.*tv\_charging\_reminder\_count*);

mChargingImageView = (ImageView) findViewById(R.id.*iv\_power\_increment*);

*/\*\* Set the original values in the UI \*\*/*

updateWaterCount();

updateChargingReminderCount();

ReminderUtilities.*scheduleChargingReminder*(this);

*/\*\* Setup the shared preference listener \*\*/*

SharedPreferences prefs = PreferenceManager.*getDefaultSharedPreferences*(this);

prefs.registerOnSharedPreferenceChangeListener(this);

// Create and instantiate a new instance variable for your ChargingBroadcastReceiver

// and an IntentFilter

mChargingIntentFilter = new IntentFilter();

// Call the addAction method on your intent filter and add Intent.ACTION\_POWER\_CONNECTED

// and Intent.ACTION\_POWER\_DISCONNECTED. This sets up an intent filter which will trigger

// when the charging state changes.

mChargingIntentFilter.addAction(Intent.*ACTION\_POWER\_CONNECTED*);

mChargingIntentFilter.addAction(Intent.*ACTION\_POWER\_DISCONNECTED*);

// Instantiate the receiver using the class created

mChargingReceiver = new ChargingBroadcastReceiver();

}

// Override onResume and setup your broadcast receiver. Do this by calling

// registerReceiver with the ChargingBroadcastReceiver and IntentFilter.

@Override

protected void onResume() {

super.onResume();

registerReceiver(mChargingReceiver, mChargingIntentFilter);

}

// Override onPause and unregister your receiver using the unregisterReceiver method

@Override

protected void onPause() {

super.onPause();

unregisterReceiver(mChargingReceiver);

}

*/\*\**

*\* Updates the TextView to display the new water count from SharedPreferences*

*\*/*

private void updateWaterCount() {

int waterCount = PreferenceUtilities.*getWaterCount*(this);

mWaterCountDisplay.setText(waterCount+"");

}

*/\*\**

*\* Updates the TextView to display the new charging reminder count from SharedPreferences*

*\*/*

private void updateChargingReminderCount() {

int chargingReminders = PreferenceUtilities.*getChargingReminderCount*(this);

String formattedChargingReminders = getResources().getQuantityString(

R.plurals.*charge\_notification\_count*, chargingReminders, chargingReminders);

mChargingCountDisplay.setText(formattedChargingReminders);

}

// Create a new method called showCharging which takes a boolean. This method should

// either change the image of mChargingImageView to ic\_power\_pink\_80px if the boolean is true

// or R.drawable.ic\_power\_grey\_80px it it's not. This method will eventually update the UI

// when our broadcast receiver is triggered when the charging state changes.

private void showCharging(boolean isCharging){

if(isCharging){

mChargingImageView.setImageResource(R.drawable.*ic\_power\_pink\_80px*);

}

else {

mChargingImageView.setImageResource(R.drawable.*ic\_power\_grey\_80px*);

}

}

*/\*\**

*\* Adds one to the water count and shows a toast*

*\*/*

public void incrementWater(View view) {

if (mToast != null) mToast.cancel();

mToast = Toast.*makeText*(this, R.string.*water\_chug\_toast*, Toast.*LENGTH\_SHORT*);

mToast.show();

Intent incrementWaterCountIntent = new Intent(this, WaterReminderIntentService.class);

incrementWaterCountIntent.setAction(ReminderTasks.*ACTION\_INCREMENT\_WATER\_COUNT*);

startService(incrementWaterCountIntent);

}

@Override

protected void onDestroy() {

super.onDestroy();

*/\*\* Cleanup the shared preference listener \*\*/*

SharedPreferences prefs = PreferenceManager.*getDefaultSharedPreferences*(this);

prefs.unregisterOnSharedPreferenceChangeListener(this);

}

*/\*\**

*\* This is a listener that will update the UI when the water count or charging reminder counts*

*\* change*

*\*/*

@Override

public void onSharedPreferenceChanged(SharedPreferences sharedPreferences, String key) {

if (PreferenceUtilities.*KEY\_WATER\_COUNT*.equals(key)) {

updateWaterCount();

} else if (PreferenceUtilities.*KEY\_CHARGING\_REMINDER\_COUNT*.equals(key)) {

updateChargingReminderCount();

}

}

// Create an inner class called ChargingBroadcastReceiver that extends BroadcastReceiver

private class ChargingBroadcastReceiver extends BroadcastReceiver {

// Override onReceive to get the action from the intent and see if it matches the

// Intent.ACTION\_POWER\_CONNECTED. If it matches, it's charging. If it doesn't match, it's not

// charging.

@Override

public void onReceive(Context context, Intent intent) {

String action = intent.getAction();

boolean isCharging = (action.equals(Intent.*ACTION\_POWER\_CONNECTED*));

// Update the UI using the showCharging method you wrote

showCharging(isCharging);

}

}

}

## Obteniendo el Estado Actual de Batería

Hay 2 formas de obtener el estado de la batería:

### Obteniendo el Estado de Carga a partir del API 23

Se puede usar el battery manager system service asi:

BatteryManager batteryManager = (BatteryManager) getSystemService(BATTERY\_SERVICE);

**boolean** isCharging = batteryManager.isCharging();

### Obtener el Estado de Carga con un Sticky Intent

Previo al API 23 se ocupaba un Sticky Intent que es un broadcast intent que se queda alrededor, haciendo que uno lo pueda accesar en cualquier momento. Para un sticky intent no se necesita un broadcast receiver, pero se usa un codigo similar para registrar el receiver:

IntentFilter ifilter = **new** IntentFilter(Intent.ACTION\_BATTERY\_CHANGED);

Intent batteryStatus = context.registerReceiver(**null**, ifilter);

En lugar de pasarle un broadcast receiver se le pasa null. El registerReceiver devuelve un intent que tiene la información de la batería.

**boolean** isCharging = status == BatteryManager.BATTERY\_STATUS\_CHARGING || status == BatteryManager.BATTERY\_STATUS\_FULL;

Para mas información sobre el status de la batería:

https://developer.android.com/training/monitoring-device-state/battery-monitoring.html

Ejemplo de metodo onResume donde se verifica la version del Build, si es M (API 23+) se usa el approach del BatteryManager si no, se usa el StickyIntent:

@Override

protected void onResume() {

super.onResume();

registerReceiver(mChargingReceiver, mChargingIntentFilter);

// The developer documentation shows how to get battery information pre Android M:

// https://developer.android.com/training/monitoring-device-state/battery-monitoring.html

// In Android M and beyond you can simply get a reference to the BatteryManager and call

// isCharging.

boolean isCharging;

// (1) Check if you are on Android M or later, if so...

if(Build.VERSION.*SDK\_INT* >= Build.VERSION\_CODES.*M*) {

// (2) Get a BatteryManager instance using getSystemService()

BatteryManager batteryManager = (BatteryManager) getSystemService(*BATTERY\_SERVICE*);

// (3) Call isCharging on the battery manager and pass the result on to your show

// charging method

isCharging = batteryManager.isCharging();

showCharging(batteryManager.isCharging());

Log.*d*(*TAG*, "onResume: Is Battery Charging?: " + isCharging);

}

else {

// (4) If your user is not on M+, then...

// (5) Create a new intent filter with the action ACTION\_BATTERY\_CHANGED. This is a

// sticky broadcast that contains a lot of information about the battery state.

IntentFilter batteryIntentFilter = new IntentFilter(Intent.*ACTION\_BATTERY\_CHANGED*);

// (6) Set a new Intent object equal to what is returned by registerReceiver, passing in null

// for the receiver. Pass in your intent filter as well. Passing in null means that you're

// getting the current state of a sticky broadcast - the intent returned will contain the

// battery information you need.

Intent intentBatteryStatus = this.registerReceiver(null, batteryIntentFilter);

// (7) Get the integer extra BatteryManager.EXTRA\_STATUS. Check if it matches

// BatteryManager.BATTERY\_STATUS\_CHARGING or BatteryManager.BATTERY\_STATUS\_FULL. This means

// the battery is currently charging.

int batteryStatus = intentBatteryStatus.getIntExtra(BatteryManager.*EXTRA\_STATUS*, -1);

isCharging = batteryStatus == BatteryManager.*BATTERY\_STATUS\_CHARGING* ||

batteryStatus == BatteryManager.*BATTERY\_STATUS\_FULL*;

// (8) Update the UI using your showCharging method

showCharging(isCharging);

}

registerReceiver(mChargingReceiver, mChargingIntentFilter);

}

\* Cabe notar que hay un bug con esto porque el batteryManager.isCharging() siempre retorna false.

# Lesson 10 – Completing the UI

## View and ViewGroups

Los Views son básicamente rectángulos en la pantalla que controlan el despliegue de elementos y el manejo de eventos de los mismos. La documentación de Android tiene una parte especifica para la interfaz de usuario:

https://developer.android.com/guide/topics/ui

Un ViewGroup es un contenedor para Views y es la clase base para todos los layouts de Android.

#### FrameLayout

Un view que se agregue en un FrameLayout se va a ubicar en la parte superior izquierda. Si se agrega otro view, este ultimo se va a ubicar en la misma posición, por lo que generalmente, estos layouts se usan para desplegar solo un item.

#### LinearLayout

Organizan los views vertical u horizontalmente. Se les puede dar un weight para que tomen mas o menos espacio que los otros.

#### RelativeLayout

Layouts donde uno puede alinear los views hijos con respecto al padre, ya sea la izquierda, derecha, arriba, abajo, incluso se puede especificar que un view se alinee con respecto a otro view.

#### GridLayouts

Los views llenan filas en un grid y uno puede tener celdas que se extiendan varios espacios.

Todos los layouts extienden de la clase View.

Uno puede anidar diferentes layouts en otros, pero hay que tener cuidado porque si se tienen muchos, esto puede afectar el performance.

#### ConstraintLayout

Es como el RelativeLayout porque permite poner views hijos relativos al padre o a otros views. Pero, este layout usa constraints para especificar como debe estar posicionado cada view con respecto a los demas.

### Gravity

Permite alinear los elementos o su contenido, por ejemplo, con la propiedad **android:gravity** = “center” se puede posicionar el contenido al centro, es decir, si es un TextView, el contenido del mismo se va a ver centrado.

También esta l apropiedad **layout\_gravity** la cual posiciona el elemento como tal con respecto a su padre. Es decir, con layout\_gravity = “center” se puede posicionar el elemento en el centro del layout.

La propiedad visibility permite mostrar, ocultar o quitar el elemento del layout, si esta invisible el elemento no se va a ver pero siempre se mantiene el placeholder lo que afecta las vistas alrededor de el. Con el valor gone no se mantiene el placeholder, es como si no estuviera ahí.

En la documentación para View se pueden ver todas las propiedades xml que se pueden setear:

https://developer.android.com/reference/android/view/View

## ConstratintLayouts

Es un layout similar al RelativeLayout pero su performance es mejor.

Los ConstraintLayout son ahora el layout por defecto de Android Studio al crear un layout. Pero si se trabaja con alguna app cuya version es vieja, hay que agrega la dependencia

implementation "androidx.constraintlayout:constraintlayout:2.0.4"

Cuando se agrega un view al constraint layout en la vista de diseño se posiciona donde uno lo deja, pero si no se definen 2 o mas constraints para ese view, la posición se resetea a la posición default. Por lo que hay que definir al menos 1 constraint horizontal y uno vertical.

Si uno liga el lado izquierdo de un View al lado izquierdo de otro View, para que este ultimo sea su referencia, si uno mueve el View de referencia, el otro también se moverá.

Como recomendación, uno no debería usar el atributo android:text para setear fake text, si no usar un atributo tools:text, ya que con el tools:text el texto que uno setee no se va a ver al correr el app.

En el siguiente ejemplo se posiciona un ImageView que servira como un conector entre otros dos ImageViews. Este ImageView tiene como id: divider, y para posicionarlo a la mitad de los ImageView que “conecta”, se setean las propiedades

* app:layout\_constraintBottom\_toBottomOf a que apunte al ImageView de la derecha
* app:layout\_constraintTop\_toTopOf a que apunte al ImageView de la derecha también.
* app:layout\_constraintHorizontal\_bias con el valor 0.0

Los Views en el layout serian así:

<!-- (5) Create an ImageView for the left rectangle -->

<!-- (6) Set the background to the shape\_rectangle\_stroke drawable -->

<ImageView

android:id="@+id/leftRectangle"

android:layout\_width="60dp"

android:layout\_height="80dp"

android:layout\_marginStart="16dp"

android:layout\_marginLeft="32dp"

android:layout\_marginTop="16dp"

app:layout\_constraintStart\_toStartOf="parent"

app:layout\_constraintTop\_toBottomOf="@+id/textViewPassengerName"

app:srcCompat="@drawable/shape\_rectangle\_stroke" />

<!-- (7) Create an ImageView for the divider -->

<ImageView

android:id="@+id/divider"

android:layout\_width="0dp"

android:layout\_height="1dp"

android:background="@color/colorPrimary"

app:layout\_constraintTop\_toTopOf="@id/rightRectangle"

app:layout\_constraintBottom\_toBottomOf="@id/rightRectangle"

app:layout\_constraintEnd\_toStartOf="@+id/rightRectangle"

app:layout\_constraintHorizontal\_bias="0.0"

app:layout\_constraintStart\_toEndOf="@+id/leftRectangle" />

<!-- (8) Create an ImageView for the rightRectangle -->

<ImageView

android:id="@+id/rightRectangle"

android:layout\_width="60dp"

android:layout\_height="80dp"

android:layout\_marginTop="16dp"

android:layout\_marginEnd="16dp"

android:layout\_marginRight="32dp"

app:layout\_constraintEnd\_toEndOf="parent"

app:layout\_constraintTop\_toBottomOf="@+id/textViewPassengerName"

app:srcCompat="@drawable/shape\_rectangle\_stroke" />

Y esto se vería asi:



El constraint llamado baseline es para TextViews y permite crear constratints entre diferentes TextViews independientemente de que tan largo es el TextView. Permite alinear el texto dentro.

## Adding Vector Images

Los SVG son los mejores tipos de imagenes para trabajar en Android porque estas se pueden redimensionar facilmente para trabajar con cualquier tamaño de teléfonos sin afectar la calidad de la imagen.

Si uno tiene una imagen svg y quiere importarla al proyecto, uno le da click derecho a res/drawable > Vector Asset.

Luego se selecciona Local file (SVG, PSD), se elige el archivo y se le da un nombre. Se puede seleccionar la opción de auto mirroring para layouts de derecha a izquierda donde se lee de derecha a izquierda y para que la imagen ejemplifique de donde sale y a donde va.

Luego se le da Next y Finish. Se agrega un archivo en formato XML. En este archivo uno puede ver la definición de la imagen incluyendo el fillColor y esta propiedad se puede usar los colores definidos en @color y otros recursos.

Para usarla, se puede usar un ImageView y escoger este archivo como fuente.

## Data Binding

Android ofrece una librería llamada Data Binding Library que ayuda a ligar cualquier UI con datos sin tener que llamar al findViewById para cada item.

Para esto se necesita:

1. Habilitar data binding en build.gradle.

Para habilitar data binding en el build.gradle, dentro de android {} agregar:

dataBinding.enabled = true

1. Agregar <layout> como el tag raiz en el UI

Cuando un layout se envuelve con el tag <layout> y se compila, se generan unas binding clases.

1. Crear una instancia de binding.

Si el layout se llama por ejemplo MainActivity, el binding que se crea llega a tener el nombre ActivityMainBinding.

1. Establecer el content view usando DatabindingUtil

Para esto se usa DataBindingUtil al llamar al **setConventView** y pasando el activity. Esto setea el contexto de la clase binding a los elementos en el layout.

// Set the Content View using DataBindingUtil to the activity\_main layout

mBinding = DataBindingUtil.*setContentView*(this, R.layout.*activity\_main*);

1. Ligar cada atributo en las vistas con su correspondiente dato.

La clase SimpleFormatter es una clase para dar formato y convertir fechas. Permite formatear fechas a texto y convertir texto a fechas y normalización. Mas info: https://developer.android.com/reference/java/text/SimpleDateFormat

Ejemplo:

public class MainActivity extends AppCompatActivity {

// (3) Create a data binding instance called mBinding of type ActivityMainBinding

ActivityMainBinding mBinding;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.*activity\_main*);

// Set the Content View using DataBindingUtil to the activity\_main layout

mBinding = DataBindingUtil.*setContentView*(this, R.layout.*activity\_main*);

// (5) Load a BoardingPassInfo object with fake data using FakeDataUtils

BoardingPassInfo fakeBoardingInfo = FakeDataUtils.*generateFakeBoardingPassInfo*();

// (9) Call displayBoardingPassInfo and pass the fake BoardingInfo instance

displayBoardingPassInfo(fakeBoardingInfo);

}

private void displayBoardingPassInfo(BoardingPassInfo info) {

// Use mBinding to set the Text in all the textViews using the data in info

mBinding.textViewPassengerName.setText(info.passengerName);

mBinding.textViewOriginAirport.setText(info.originCode);

mBinding.textViewFlightCode.setText(info.flightCode);

mBinding.textViewDestinationAirport.setText(info.destCode);

// Use a SimpleDateFormat formatter to set the formatted value in time text views

SimpleDateFormat formatter = new SimpleDateFormat( getString(R.string.*timeFormat*),

Locale.*getDefault*());

String boardingTime = formatter.format(info.boardingTime);

String arrivalTime = formatter.format(info.arrivalTime);

String departureTime = formatter.format(info.departureTime);

mBinding.textViewBoardingTime.setText(boardingTime);

mBinding.textViewArrivalTime.setText(arrivalTime);

mBinding.textViewDepartureTime.setText(departureTime);

// Use TimeUnit methods to format the total minutes until boarding

long totalMinutesUntilBoarding = info.getMinutesUntilBoarding();

long hoursUntilBoarding = TimeUnit.*MINUTES*.toHours(totalMinutesUntilBoarding);

long minutesLessHoursUntilBoarding = totalMinutesUntilBoarding -

TimeUnit.*HOURS*.toMinutes(hoursUntilBoarding);

String timeUntilBoarding = String.*format*( getString(R.string.*countDownFormat*),

hoursUntilBoarding, minutesLessHoursUntilBoarding);

mBinding.textViewBoardingInCountdown.setText(timeUntilBoarding);

mBinding.textViewTerminal.setText(info.departureTerminal);

mBinding.textViewGate.setText(info.departureGate);

mBinding.textViewSeat.setText(info.seatNumber);

}

}

Mas información actualizada sobre esto: https://developer.android.com/topic/libraries/data-binding/expressions

## Accessibility & Internationalization

Accessibility es que el app pueda ser usada por personas que tengan algún problema visual o limitaciones físicas.

Internationally Friendly es que se pueda traducir a varios idiomas sin afectar el diseño del layout, como cuando se usa lenguajes que se leen de derecha a izquierda.

Android provee características de accesibilidad como:

* **TalkBack**: un lector de pantalla pre-instalado de Google. Usa feedback de voz para describir los resultados de acciones como lanzar un app y eventos como notificaciones.
* **Explore by Touch**: sistema que trabaja con TalkBack, permitiendo que uno toque la pantalla y escuche que hay bajo el dedo. Especial para personas con baja vision.
* **Accessibility Settings**: permite modificar settings del display y el sonido del device, como incrementar el texto, cambiar la velocidad con la que se el texto es leído y mas.

Para aprovechar estas características se puede seguir esta guiá:

[https://developer.android.com/guide/topics/ui/accessibility/apps#recommendations](https://developer.android.com/guide/topics/ui/accessibility/apps" \l "recommendations)

Para describir los campos se puede usar la propiedad contentDescription y usar recursos de string.

Se recomienda que no se use en TextView porque los lectores leen el contenido por default.

Los botones basados en texto tampoco suelen necesitar esta propiedad porque es el mismo caso que con los TextView.

Esta descripción no se presenta en pantalla, es para herramientas de accesibilidad como los lectores. Ejemplo:

<ImageView android:contentDescription="@string/origin\_label"/>

Idealmente, lo que uno quiere describir son ImageViews, ImageButtons y todos los Checkboxes.

Otras cosas que se pueden usar para mejorar la accesibilidad son:

* Habilidad la navegación enfocada que asegura que los usuarios puedan navegar por la pantalla usando hardware externo como teclados bluetooth.
* No comentarios solo audio, que garantiza que cualquier comentario puede tener un mecanismo secundario para soportar usuarios sordos o con poca audición.

## Localization

Es lo que se conoce como internacionalización o poder adaptar el producto a varios lenguajes.

1. **Traducción**

Uno debe diseñar el app de forma que sea fácil de traducir. Para eso, todos los textos que uno espera que se puedan traducir, deben estar en un recurso string como res/values/strings.xml

Esto permite que uno cree otras versiones de strings.xml para otros lenguajes. Esto se hace creando un nuevo folder *values* con el patrón **value-xx** donde xx es la abreviacion del lenguaje de ISO 639 (<https://www.loc.gov/standards/iso639-2/php/code_list.php> ), por ejemplo, res/values-fr/strings.xml para la versión de string en francés.

De esta forma, cuando un usuario establezca el francés en su teléfono como idioma por defecto, Android automáticamente cargara la versión de strings en francés.

Algunas veces hay recursos que uno no quiere que se traduzcan, como los identificadores de vistas o nombres de variables o formatos. Para esos casos, al string se le pone un atributo llamado translatable en false:

<string name="timeFormat" translatable="false">hh:mm a</string>

1. Soporte RTL

Si uno trabaja con clientes que usen RTL o right-to-left, uno debe considerar implementar el soporte a RTL para los layouts y para editar y desplegar texto.

Para girar las imágenes en estos casos solo es de agregar el atributo autoMirrored:

<vector android:autoMirrored="true"> </vector>

Otros set de atributos relacionados al RTL son:

android:layout\_marginStart

android:layout\_marginEnd

Que corresponden a:

android:layout\_marginLeft

android:layout\_marginRight

respectivamente cuando se usa LTR como en ingles, pero cuando es RTL es al reves, es decir, Start se mapea a Right y End a Left.

Es importante tomar en cuenta que esto es un poco nuevo, para soportar devices viejos (anteriores a 4.1) se deben reservar los margenes para Start y End con los Left y Right obsoletos con los mismos valores. En dispositivos mas nuevos, los margenes para Left y Right se ignoran y se usan Start y End en su lugar.

En este link se ven mas tareas a realizar en cuanto a la localización:

https://developer.android.com/distribute/best-practices/launch/localization-checklist

## Responsive Design

Para poder brindar un diseño responsive mas optimo, Android ofrece los alternate layouts. Basicamente uno puede crear un layout XML nuevo para el modo landscape. Para eso solo hay que crear un folder llamado l**ayout-land** o layout-landscape en el folder res.

Copiar ahi los layouts a los que se les va a generar un layout para landscape mode. En Android Studio aparece como otro xml con el label land bajo el mismo xml.

Cuando hay partes que se pueden dejar exactamente igual tanto para landscape como para portrait es mejor mover esas vistas a layouts mas pequeños y usar el tag include para combinarlas en el layout default y lanscape. Así se reutiliza ciertas partes y es mas fácil de mantener.

## Diseño Especifico para RecyclerView

El RecyclerView permite que uno infle diferentes layouts para cada uno de sus items. Esto permite un diseño más amistoso para el usuario. Por ejemplo, para la aplicación sobre el clima, se puede tener un diseño diferente para los datos del día de hoy.

Para eso se necesita usar algunas funcionalidades del Adapter del RecyclerView:

1. Crear un layout para el día de hoy

2. Crear valores boleanos para personalizar la experiencia

* Se crean un par de booleans para espeicificar si se va a usar esta vista, uno de ellos en un folder values-port que indica que son valores para portrait.
* También se crea una variale para leer el valor de los booleanos que se crearon.

3. Se sobre escribe el método getItemViewType del RecyclerView

* Si el valor de la variable del boolean que se metió en bools.xml es true y es la primera posición entonces se retorna un id para identificar que es el día de hoy para usar el layout customizado.

4. En el onCreateViewHolder se usa el id retornado por getItemViewType para definir el layout a cargar:

# Lesson 11 – Polishing the UI

## Android Design Principles

Cosas a tener en cuenta al diseñar un app:

* Los usuarios juzgan un app en los primeros 30 segundos.
* La experiencia del usuario es critica.
* El app debe ser sorprender al usuario con animaciones y transiciones suaves que contribuyan al usuario a sentirse en poder con poco esfuerzo.
* Debe permitir al usuario tocar e interactuar con objetos directamente en lugar de usar botones y menus.
* Debe poseer imágenes ricas en lugar de muchas palabras y largas oraciones.
* Debe permitir al usuario personalizar la aplicación.
* Proveer atajos para completar tareas complejas.
* Recordar los settings y la automatización

## Visual Mocks and Keylines

Una parte inicial del diseño de un app, es crear los llamados **mock ups,** que es un modelo de una aplicación usada para la evaluación del diseño.

Estos mock ups tienen lineas que especifican el tamaño y espaciado de los componentes, llamados **Keylines.**

Todo esto se hace basado en el Material Design, que es una guía de principios para crear visuales útiles y bellos.

## Color Guidelines

Android Material Design recomienda tener un color Primary y un Accent.

Primary: es el color principal del app. Generalmente usado por componentes como el action bar y otros.

Accent: un color mas brillante y que se usa para llamar la atención de elementos claves como botones.

Hay varias herramientas para poder trabajar con paletas de colores y probar las que uno quiera para el app. Generalmente se escogen 3 tonos del color primary y uno para el accent.

Algunos links utiles para los colores:

[https://material.io/design/color/the-color-system.html#tools-for-picking-colors](https://material.io/design/color/the-color-system.html" \l "tools-for-picking-colors)

https://www.materialpalette.com/

## Font Guidelines

Otra característica importante en un app son los tipos de letra que se usan. El texto default para Android es uno llamado Roboto, el cual viene en diferentes familias.

**Font-Families** son grupos de de fonts que comparten características de diseño similares, como serif o sans-serif.

Sans Serif es el default en Android y hay varios tipos, como el sans serif thin, medium y condensed.

Android Material Design tiene recomendaciones sobre font-families a usar para legibilidad.

Es recomendado apegarse a un tipo de font-family y cambiar colores, weight o tamaño cuando se necesite resaltar algunos textos. También es recomendado mantener un estandar de estilos para los diferentes componentes de texto que se tenga

Algo importante sobre el tamaño del texto es que este en unidades sp. Scale-Independent Pixels permite mantener el mismo tamaño físico a través de diferentes resoluciones de pantallas.

Los sp también se usan para accesibilidad, ya que Android re dimensiona cualquier texto en sp si el usuario cambia el tamaño de texto en los settings generales de Android.

## Cambiando Colores y Fondos

En las apps, una forma de cambiar los colores es usar el archivo colors.xml, donde uno define o redefine colores, por ejemplo, se pueden cambiar los colores por default para colorPrimary, colorAccent, colorPrimaryDark y otros.

En el archivos de strings.xml uno también puede definir tipos de fonts a usar para asignar a las propiedades fontFamily de los views.

## Styles and Themes

Una forma mas ágil de trabajar los colores y fonts es a través de Styles. Un style es un recurso XML, separado de los layouts, donde uno puede especificar todas las propiedades en un solo lugar. Luego, se puede aplicar ese estilo a cualquier vista.

Un Theme es creado igual al Style, y es un estilo que se aplica por completo a una Actividad o una aplicación y no solo a una vista.

Los estilos se aplican a las vistas por medio de la propiedad **style**.

## Style Inheritance

Una forma de reutilizar estilos ya definidos es por medio de la herencia. Cuando uno define un estilo nuevo que tiene las mismas propiedades que un estilo ya definido, se puede heredar de esta usando la propiedad **parent** del elemento style:

<!-- Create a new style "folderStyle" to group properties for the mail folder text views -->

<!-- This style should group all the property values found in the inbox view, like layout\_width, height, etc, that are meant to repeat -->

<style name="FolderStyle">

<item name="android:layout\_width">match\_parent</item>

<item name="android:layout\_height">wrap\_content</item>

<item name="android:drawablePadding">8dp</item>

<item name="android:paddingTop">4dp</item>

<item name="android:paddingBottom">4dp</item>

<item name="android:textSize">28sp</item>

<item name="android:textColor">@android:color/black</item>

</style>

<!-- Create a new "inboxStyle" that inherits from "folderStyle" and includes a bold textStyle property -->

<style name="InboxStyle" parent="FolderStyle">

<item name="android:textStyle">bold</item>

</style>

Un estilo que herede de otro, puede sobre escribir propiedades que el padre tenga definidos.

Android viene con algunos temas ya definidos como Theme.AppCompact.Light.ActionBar o Theme.AppCompact.Dark.ActionBar

Para cambiar el tema de la aplicación, solo hace falta cambiar la propiedad android:theme en el elemento application en el Android Manifest:

<application

android:allowBackup="true"

android:icon="@mipmap/ic\_launcher"

android:label="@string/app\_name"

android:supportsRtl="true"

**android:theme="@style/AppTheme">**

<activity android:name=".MainActivity">

Un ejemplo donde se crea un estilo para mostrar solo un logo y no el titulo en el action bar es este:

<!-- Base application theme. -->

<style name="AppTheme" parent="@style/Theme.AppCompat.Light.DarkActionBar">

<item name="colorPrimary">@color/colorPrimary</item>

<item name="colorPrimaryDark">@color/colorPrimaryDark</item>

<item name="colorAccent">@color/colorAccent</item>

<!-- Used for settings preference theme -->

<item name="preferenceTheme">@style/PreferenceThemeOverlay</item>

</style>

<!-- (1) Create a style that shows a Sunshine logo instead of a title -->

<style name="ActionBar.Solid.Sunshine.NoTitle" parent="Widget.AppCompat.Light.ActionBar.Solid.Inverse">

<item name="logo">@mipmap/ic\_launcher</item>

<item name="displayOptions">useLogo|showHome</item>

</style>

<!-- (2) Create a style for MainActivity called AppTheme.Forecast that uses our custom ActionBar style -->

<style name="AppTheme.Forecast" parent="AppTheme">

<item name="actionBarStyle">@style/ActionBar.Solid.Sunshine.NoTitle</item>

</style>

Es importante notar que algunas de esas propiedades no llevan el prefijo android:

## Designing for Multiple Screens

El diseño de las apps para diferentes pantallas depende de varios factores:

### Screen Density

Android categoriza las pantallas de dispositivos usando 2 propiedades generales: Tamaño y Densidad de Pixeles. Ambas se deben tomar en cuenta al diseñar los layouts y los recursos de imágenes.

**Density**: el numero de pixeles en el área física de la pantalla, generalmente medida en dots per inch (dpi). Existen 5 rangos o categorías llamadas Density Buckets en los que puede entrar un dispositivo:



**Density-Independent Pixels**: o dips, es el mismo tamaño físico en cada dispositivo.

Por lo tanto, al diseñar tanto los espacios como las imágenes de las apps, se trabaja con dpi y no con pixeles.

En cuanto a las imágenes, se debe proporcionar una imagen para cada density bucket generalizado. Si solo se brinda una imagen, Android va a intentar redimensionarla pero esto toma poder de procesamiento y la imagen puede verse distorsionada. Por lo que tener diferentes imagenes para los diferentes density buckets ayuda no solo a como se ve la imagen si no también al performance del app.

## Resource Folder Qualifiers

En Android uno puede crear diferentes layouts para diferentes tipos de devices. Uno puede crear diferentes recursos para esto usando calificadores.

Estos recursos se agregan en folders a los que se les agrega el calificador usando un guion y luego el calificador. Por ejemplo:

* values-fr/
* values-fr-rCA/
* layout-desk/
* layout-stylus/
* drawable-xhdpi/
* layout-land/
* layout-sw720dp/

En runtime, Android verifica las propiedades del device, como la densidad de pixeles, lenguaje, tamaño de pantalla, etc y carga los recursos según estas propiedades. Estos folder incluso pueden encadenar diferentes calificadores para hacer los recursos aun mas específicos.

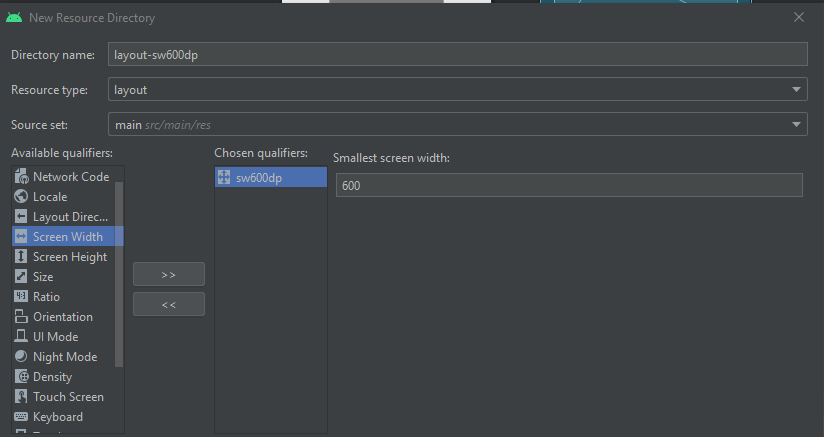
## Smallest Width Qualifier

Uno de los calificadores mas útiles es el smallest width o ancho mas pequeño que viene siendo **-sw<size in dp>**

Por ejemplo, si se tiene un folder **layout-sw600dp/** y **layout-sw720dp/** para los layouts de 600dp o mas y para las de 720dp o mas. Si se tiene un Nexus 7 cuyas dimensiones son 960dp x 600dp, el ancho mas pequeño van a ser 600dp, por lo que los layouts del folder layout-sw600dp se usarían en lugar del default.

Los archivos de valores dimens.xml suelen tener valores para margenes, paddings y otros que afectan a todos los layouts.

Cuando se crean folders de recursos uno puede elegir uno o mas calificadores en el wizard. Por ejemplo:



## Touch Selectors

Estos son efectos visuales que hacen que por ejemplo, una vista o un elemento de la pantalla se muestre de otro color o resalte de alguna forma, dándole un feedback al usuario y mejorando la experiencia del usuario.

Para estos selectores se usan drawables, que son cosas que se pueden dibujar en pantalla. Al agregar un folder de tipo Drawable se puede agregar un elemento Root, para un selector, se escoge el que dice selector.

A los touch selectors, se les agrega items que tienen 2 propiedades:

* android:drawable: que es el color de background que tendrá
* state: un valor que representa el estado en el que se va a cambiar el color.

Ejemplo:

Los primeros 3 estados representan los posibles estados cuando un item es clickeado, mientras que el ultimo, no tiene un estado definido porque representa el estado por defecto.

<selector xmlns:android="http://schemas.android.com/apk/res/android">

<!-- Add the primary light color to three items for: pressed, activated, and selected states -->

<item android:drawable="@color/colorPrimaryLight" android:state\_pressed="true"/>

<item android:drawable="@color/colorPrimaryLight" android:state\_activated="true"/>

<item android:drawable="@color/colorPrimaryLight" android:state\_selected="true"/>

<!-- Add one more item for the default (unselected) state -->

<!-- This should have a default drawable color = "@android:color/background\_light" -->

<item android:drawable="@android:color/background\_light" />

</selector>

Para poder usarlo, en el layout para cada list item, se agrega la propiedad background al linear layout y por valor se le pone el selector. De esta forma, el color de background cambiara según el estado del list item.

<!-- Set the touch selector as the background property for the list item layout -->

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:background="@drawable/list\_item\_selector" >