# CURSO DE TÉCNICO EN DESARROLLO NATIVO SOBRE PLATAFORMAS ANDROID

EOI – Escuela de Organización Industrial

TEMA 7. Desarrollos Avanzados

# CONTENIDOS

1. Recycler View
2. Multimedia
3. Asincronicidad
4. RxAndroid
5. Unit Tests: CI + TDD
6. Ofuscación y seguridad

# RECYCLER VIEW

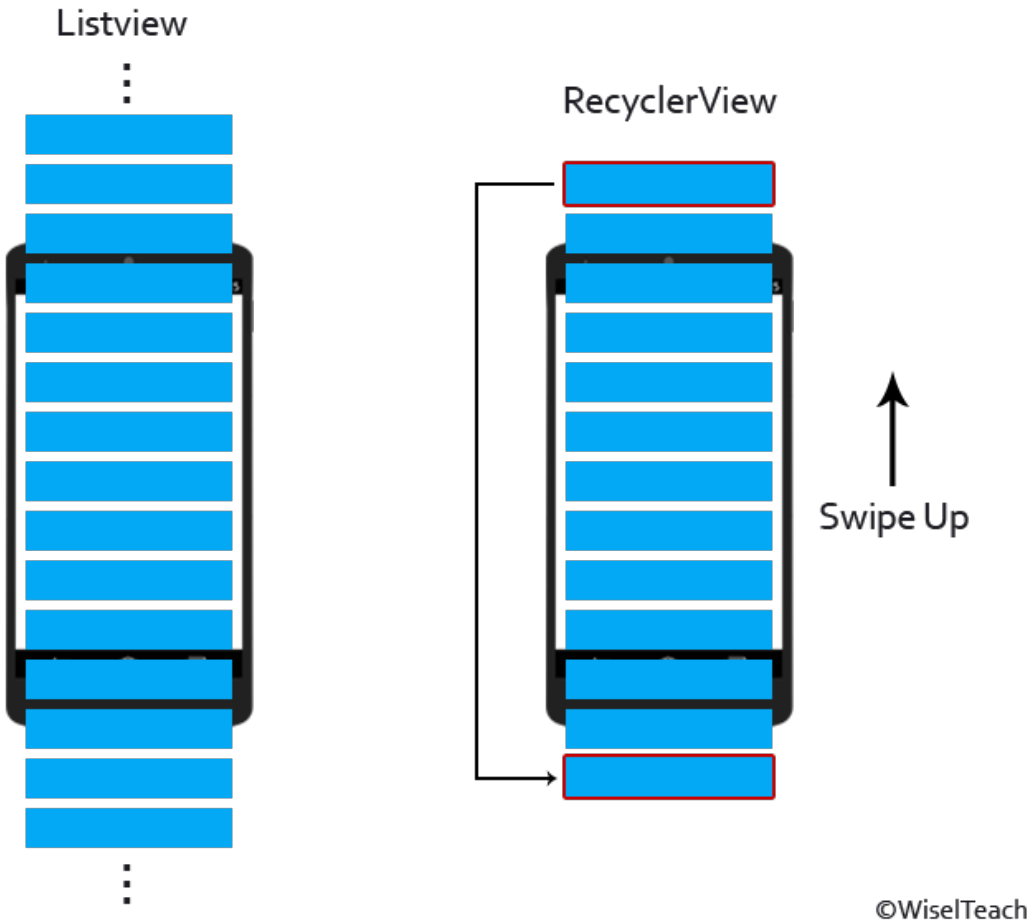
- Para mostrar un listado de datos en Android, ya sea horizontal o vertical, podemos usar la vista **ListView** u **HorizontalListView**
- Dichas vistas necesitan un **Adapter** para instanciar **todas las vistas que representarán cada elemento de la lista**
- Esto supone un problema de *performance* cuando existen grandes cantidades de datos

doc: https://developer.android.com/guide/topics/ui/layout/recyclerview#java

# RECYCLER VIEW

- Para lidiar con éste problema de *performance* existe la vista **RecyclerView** y el patrón **ViewHolder**
  - **RecyclerView:** vista más avanzada y flexible de ListView
  - **ViewHolder:** patrón diseñado para reciclar y reutilizar las vistas creadas

# RECYCLER VIEW



# RECYCLER VIEW

- La vista **RecyclerView** no es nativa Android, sino que se encuentra en la librería **support-v7**
  - Para utilizarla, debemos añadir en **app/build.gradle**

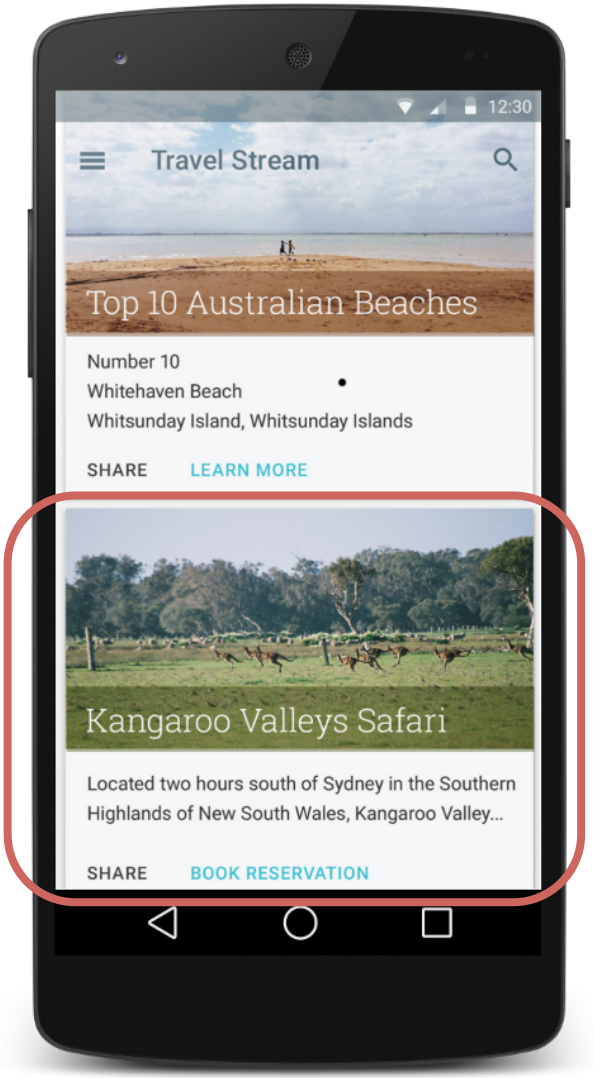
```
dependencies {
    . . .
    implementation 'com.android.support:recyclerview-v7:27.1.1'
    . . .
}
```

# RECYCLER VIEW

- Componentes necesarios para la creación de un **RecyclerView:**
  - **RecyclerView:** contenedor de la lista (vista XML)
  - **LayoutManager:** clase que define cómo mostrar los elementos de la lista (*LinearLayoutManager* o *GridLayoutManager*)
  - **ViewHolder**: clase que representa la vista de cada uno de los elementos de la lista
  - **Adapter:** clase que maneja los ViewHolders y su posición en la lista

# RecyclerView.ViewHolder

- El componente básico es una clase que hereda de **RecyclerView.ViewHolder**
  - Representa la interfaz gráfica de un elemento de la lista



# RecyclerView.ViewHolder

```
public class MyViewHolder
                    extends RecyclerView.ViewHolder {
    private TextView myTextView;

    public MyViewHolder(View itemView) {
        super(itemView);
        myTextView = itemView
            .findViewById(R.id.text_view_view_holder);
     }

     public void bind(String value) {
        myTextView.setText(value);
        myTextView
            .setBackgroundColor(Color.parseColor(value));
     }
}
```

# RecyclerView.ViewHolder

- **view_holder_item.xml:** ubicado en la carpeta *res/layout* del proyecto

```
<TextView xmlns:android="http://schemas.android.com/
apk/res/android"
    android:id="@+id/text_view_view_holder"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:textColor="#FFFFFF"
    android:textSize="18sp"
    android:textStyle="bold"
    android:layout_margin="4dp">
</TextView>
```

# RecyclerView.Adapter<>

- Funciona del mismo modo que **BaseAdapter** para un **ListView,** aunque los métodos son diferentes:
  - **onCreateViewHolder(ViewGroup parent, int viewType)** : Crea las vistas para rellenar la lista
  - **onBindViewHolder(ViewHolder holder, int position)**: Reemplaza el contenido de las vistas con los nuevos datos cuando se necesita
  - **getItemCount()**: indica el número de elementos de la lista

# RecyclerView.Adapter<>

- El **Adapter** debe tener acceso al conjunto de datos a mostrar, por lo que se lo podemos pasar en el constructor:

```
public class MyRecyclerViewAdapter
                    extends RecyclerView.Adapter {
    private String[] dataSet;

    public MyRecyclerViewAdapter(String[] dataSet) {
        this.dataSet = dataSet;
    }
    . . .
}
```

# RecyclerView.Adapter<>

- **getItemCount():** devolverá el número de elementos de la lista

```
public class MyRecyclerViewAdapter
                    extends RecyclerView.Adapter {
    private String[] dataSet;
    . . .
    @Override
    public int getItemCount() {
        return dataSet != null ? dataSet.length : 0;
    }
    . . .
}
```

# RecyclerView.Adapter<>

- **onCreateViewHolder():** infla la interfaz (fichero XML) y la devuelve como un **ViewHolder**
  - Es necesario crear un fichero de interfaz para representar cada elemento de la lista
  - Dicho fichero XML se infla a través de *LayoutInflater* y se construye un objeto **ViewHolder** con él

# RecyclerView.Adapter<>

```
public class MyRecyclerViewAdapter
                    extends RecyclerView.Adapter {

    @NonNull
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(
            @NonNull ViewGroup parent, int viewType) {
        View view = LayoutInflater
          .from(parent.getContext())
          .inflate(R.layout.view_holder_item, parent, false);
        return new MyViewHolder(view);
    }
    . . .
}
```

# RecyclerView.Adapter<>

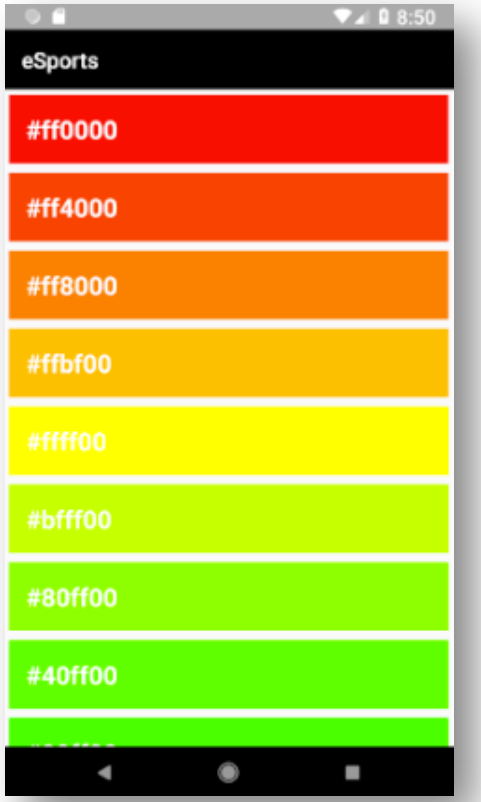
- **onBindViewHolder():** se llama cuando un **ViewHolder** necesita rellenar la vista con datos

```
public class MyRecyclerViewAdapter
                    extends RecyclerView.Adapter {
    private String[] dataSet;
    . . .
    @Override
    public void onBindViewHolder(
            @NonNull  RecyclerView.ViewHolder holder,
            int position) {
        ((MyViewHolde) holder).bind(dataSet[position]);
    }
    . . .
}
```

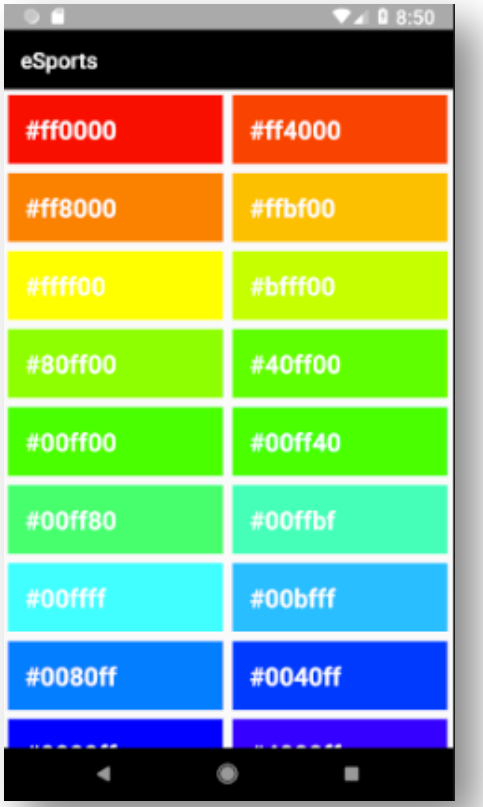
# LayoutManager

- La vista **RecyclerView** puede mostrar los elementos de la lista de diferentes formas, dependiendo del **LayoutManager** que se defina:
  - LinearLayoutManager (horizontal o vertical)
  - GridLayoutManager
  - StaggeredGridLayoutManager

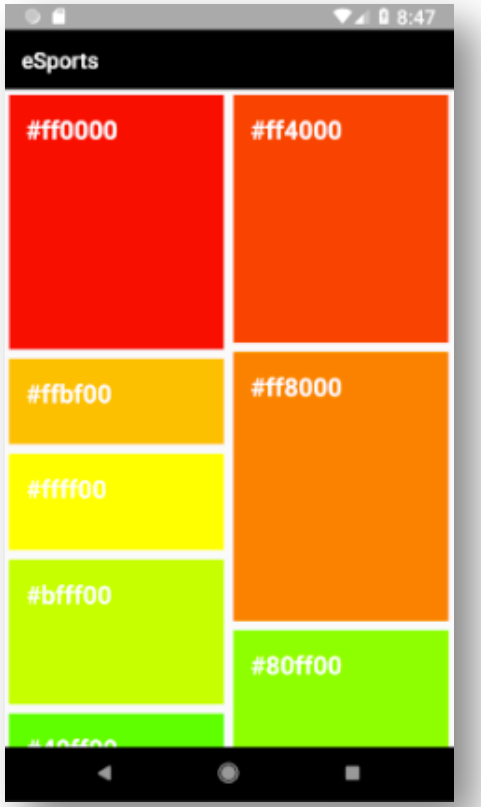
# LayoutManager




**LinearLayoutManager**




**GridLayoutManager**




**StaggeredGridLayoutManager**

# LayoutManager

- Para definir el **LayoutManager**, simplemente hay que indicarlo a través del método **setLayoutManager** de **RecyclerView**

```
private RecyclerView myRecyclerView;

@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    RecyclerView.LayoutManager myLayoutManager =
                new LinearLayoutManager(this);
    myRecyclerView
                .setLayoutManager(myLayoutManager);
}
```

# RecyclerView

- Una vez creados todos los elementos necesarios, sólo queda definir la vista **RecyclerView** en el XML de nuestra Activity y juntar todos los componentes:
  1. Definir <RecyclerView …> en XML
  2. Obtener referencia en el código Java de nuestra Activity
  3. Definir y asignar el **LayoutManager** escogido
  4. Crear y asignar el **Adapter**

# RecyclerView

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
android"
    android:layout_width="match_parent”
    android:layout_height="match_parent”
    android:orientation="vertical”>

     <android.support.v7.widget.RecyclerView
        android:id=“@+id/recycler_view”
        android:layout_width="match_parent”
        android:layout_height="match_parent"/>
</LinearLayout>
```

# RecyclerView

```
private RecyclerView myRecyclerView;
private MyRecyclerViewAdapter myRecyclerViewAdapter;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_recycler_view);
    myRecyclerView = findViewById(R.id.recycler_view);

    RecyclerView.LayoutManager myLayoutManager =
                        new LinearLayoutManager(this);
    myRecyclerView.setLayoutManager(myLayoutManager);

    myRecyclerViewAdapter = new MyRecyclerViewAdapter(
            getResources().getStringArray(R.array.colors)
    );
    myRecyclerView.setAdapter(myRecyclerViewAdapter);
}
```

# PRÁCTICA

1. Crear el fichero de interfaz XML que representará un elemento de nuestra lista que:
   - El nombre del fichero sea **view_holder_item.xml**
   - Esté ubicado en la carpeta **res/layout**
   - Contenga una vista de tipo **TextView** con identificador
   - Personalizar el TextView a nuestro gusto, p.e:

# PRÁCTICA

2. Crear la clase **MyViewHolder** que:
   - Herede de **RecyclerView.ViewHolder**
   - Tenga un constructor que reciba un parámetro de tipo **View**
   - Obtenga la referencia al **TextView** anteriormente creado en el fichero XML
   - Tenga un método llamado **bind(String value)** que modifique el valor de dicho **TextView** y su color de fondo (con el color indicado en *value)*
     - *El string recibido por parámetro será de tipo "#FF00CC"*

# PRÁCTICA

3. Crear la clase **MyRecyclerViewAdapter** que:
   - Herede de **RecyclerView.Adapter**
   - Tenga un constructor donde recibirá por parámetro un conjunto de colores de tipo **String[]**
   - Implemente los tres métodos de *RecyclerView.Adapter:*
     - *onCreateViewHolder*
     - *onBindViewHolder*
     - *getItemCount*

# PRÁCTICA

4. Crear una nueva Activity llamada **RecyclerViewActivity** que:
   - Defina una vista *RecyclerView* en su fichero XML
5. Obtener la referencia al dicho RecyclerView en el método **onCreate**
6. Crear un **LinearLayoutManager** y asignarlo al **RecyclerView**
7. Crear una instancia del **Adapter** y asignarlo al **RecyclerView**
8. Modificar **AndroidManifest.xml** para que sea ésta nueva Activity la primera que se abre al abrir la aplicación

# MULTIMEDIA

- Android incluye un *framework* multimedia que permite la reproducción de audio, vídeo e imágenes de forma sencilla.
- Se pueden reproducir ficheros alojados en la propia aplicación, ficheros disponibles en el dispositivo e incluso ficheros que se encuentren en Internet, vía *streaming*.

# MULTIMEDIA

- La clase principal para la reproducción de ficheros multimedia (audio y vídeo) es **MediaPlayer**
- Permite obtener, codificar y reproducir formatos multimedia con muy pocas líneas de código

doc: https://developer.android.com/guide/topics/media/mediaplayer

# PRÁCTICA

1. Descargar el fichero **cancion.mp3** de:
   - https://github.com/AndroidTeaching/Android-Application/blob/master/app/src/main/res/raw/cancion.mp3
2. Guardar el fichero en la carpeta **res/raw** del proyecto. Crear la carpeta si no existe.

# PRÁCTICA

3. Crear una nueva Activity llamada **MediaPlayerActivity** y escribir el siguiente código en el método **onCreate()**

```
MediaPlayer player = MediaPlayer.create(this, R.raw.cancion);
player.start();
```

4. Ejecutar la aplicación

# MULTIMEDIA

- Para una óptima implementación de **MediaPlayer,** hay que tener en cuenta:
  - Los formatos multimedia y codecs soportados en Android
  - La preparación del objeto **MediaPlayer** puede tardar varios segundos, por lo que hay que realizarla de forma asíncrona
  - **MediaPlayer** es una máquina de estados que conviene saber utilizar
  - Siempre hay que liberar el objeto **MediaPlayer** cuando se deje de utilizar

# PRÁCTICA

1. Guardar una referencia al objeto **MediaPlayer** de **MediaPlayerActivity**
   - Para ello crear la variable **private MediaPlayer myMediaPlayer;** en la clase.
2. En el método **onStop,** liberar el objeto MediaPlayer y asignarle el valor **null**

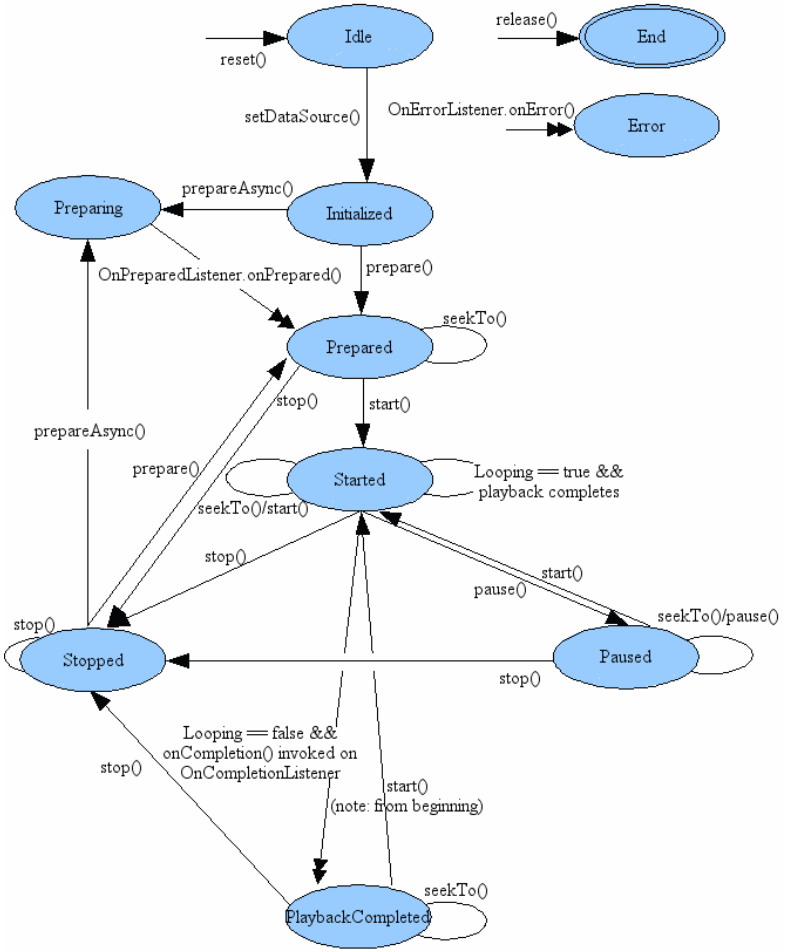
```
myMediaPlayer.release();
myMediaPlayer = null;
```

# MULTIMEDIA

- Android soporta una serie de formatos y códecs de audio y vídeo
  - **.webm**
  - **.mp3**
  - **.mp4**
  - **.ogg**
- La documentación sobre los formatos soportados se puede encontrar en:
  - https://developer.android.com/guide/topics/media/media-formats

# MULTIMEDIA

- Máquina de estados
  - https://developer.android.com/images/mediaplayer_state_diagram.gif



# MULTIMEDIA

- Preparación asíncrona
  - El objeto **MediaPlayer** necesita obtener el contenido multimedia de alguna fuente de datos
  - La preparación consiste en la obtención de dichos datos y la decodificación de los mismos, quedando preparado para la reproducción
  - **MediaPlayer** permite registrar un *listener* para se notificados cuando ya está preparado y listo para reproducir

# PRÁCTICA

- Preparación asíncrona

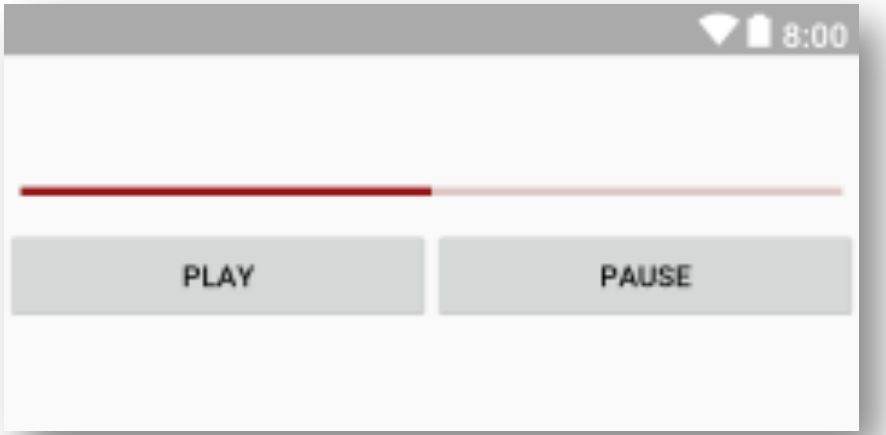
```
String url = "http://........"; // your URL here
final MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setAudioStreamType(AudioManager.STREAM_MUSIC);
mediaPlayer.setDataSource(url);
mediaPlayer.setOnPreparedListener(new
mediaPlayer.setOnPreparedListener() {
    @Override
    public void onPrepared(MediaPlayer mp) {
        mediaPlayer.start();
    }
});
mediaPlayer.prepareAsync();
```

# MULTIMEDIA

- **MediaPlayer** permite controlar la reproducción del fichero multimedia con los métodos básicos:
  - **start()**: función de *play.* Comienza a reproducir.
  - **pause()**: función de *pause.* Pausa la reproducción.
  - **stop()**: function de *stop*. Para la reproducción.
  - **seekTo(int milliseconds):** mueve el momento de reproducción actual a los milisegundos específicos
  - **reset()**: reinicia el reproductor.

# PRÁCTICA

1. Crear en **MediaPlayerActivity** dos botones:
   - **Pause** -> para pausar la reproducción de MediaPlayer
   - **Play** -> para comenzar o reanudar la reproducción de MediaPlayer



# MULTIMEDIA

- Si queremos que la clase **MediaPlayer** se ejecute en segundo plano, aún sin tener ninguna Activity visible, debemos utilizar un **Service:**
  - El **MediaPlayer** se instanciará dentro de la clase **Service**
  - Los botones de **play/pause** arrancarán o pararán el **Service**

# MULTIMEDIA

- Un **Service** es un componente Android (similar a Activity) que no dispone de interfaz gráfica
- Los métodos más importantes de un **Service** son:
  - **onStartCommand()**: se llama cuando se arranca un servicio
  - **onBind():** cuando un componente Android quiere enlazarse con dicho servicio
  - **onDestroy()**: cuando el servicio se destruye. Momento de liberar cualquier recurso.

# MULTIMEDIA

- Ejemplo de **Service**

```
public class MediaPlayerService extends Service {

    @Override
    public int onStartCommand(Intent intent,
                              int flags, int startId) {
        return super.onStartCommand(intent,flags, startId);
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

# MULTIMEDIA

- Para arrancar un servicio, se debe hacer a través de un **Intent:**

```
Intent myIntent = new Intent(this, MediaPlayerService.class);
startService(myIntent);
```

- Para parar un servicio, también:

```
Intent myIntent = new Intent(this, MediaPlayerService.class);
stopService(myIntent);
```

# PRÁCTICA

1. Crear un **Service** llamado **MediaPlayerService** que:
   - Cree, prepare y comience a reproducir un **MediaPlayer** en su método **onStartCommand()**
   - Pare y libere el **MediaPlayer** en el método **onDestroy()** del Service
2. Arrancar **MediaPlayerService** cuando el usuario pulse el botón “Play” de **MediaPlayerActivity**
3. Parar **MediaPlayerService** cuando el usuario pulse el botón “Stop” de **MediaPlayerActivity**

# MULTIMEDIA

- Al arrancar un servicio con **startService()**, éste sigue funcionando hasta que se para explícitamente:
  - Se puede parar con **stopService()**
  - O bien con el método **stopSelf()** dentro del propio servicio

```
stopSelf();
```

# MULTIMEDIA

- Si el usuario no para el servicio explícitamente, podemos pararlo cuando el **MediaPlayer** termine de reproducir el contenido
- Esto se logra implementando la interfaz **MediaPlayer.OnCompletionListener**, que se llamará una vez la reproducción haya finalizado

# MULTIMEDIA

- **MediaPlayer.OnCompletionListener**:

```
myMediaPlayer.setOnCompletionListener(new MediaPlayer
                                .OnCompletionListener() {
    @Override
    public void onCompletion(MediaPlayer mp) {
        stopSelf();
    }
});
```

# PRÁCTICA

1. Parar el servicio con **stopSelf()** cuando el **MediaPlayer** termine la reproducción del fichero

# MULTIMEDIA

- Para la reproducción de vídeo, se puede utilizar también **MediaPlayer**, pero requiere algo más de configuración y su complejidad aumenta.
- En cambio, exista la vista **VideoView** que integra un **MediaPlayer**, y nos permite reproducir vídeo de forma muy sencilla.

# MULTIMEDIA

- **VideoView** es una vista de Android que permite definir un fichero multimedia (ya se encuentre en el proyecto, en el sistema de ficheros o en Internet) y reproducirlo

```
<VideoView
    android:id="@+id/video_view"
    android:layout_width="match_parent"
    android:layout_height="300dp" />
```

# MULTIMEDIA

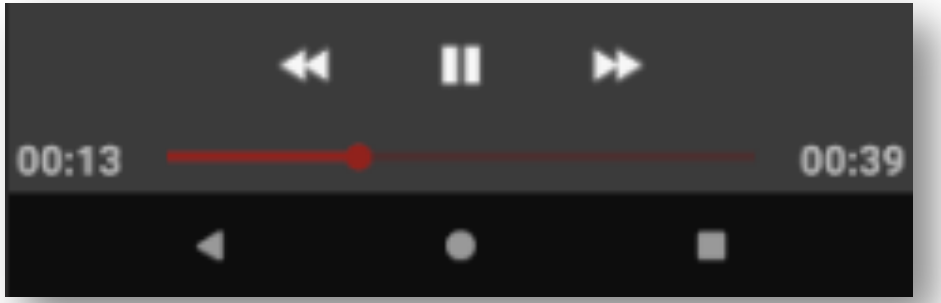
- Algunos métodos de **VideoView** son:
  - **setVideoURI(Uri uri):** define la ruta del fichero multimedia
  - **start()**: comienza la reproducción
  - **pause():** pausa la reproducción
  - **resume():** reanuda la reproducción
  - **seekTo(int millis):** mueve la reproducción actual a los milisegundos indicados
  - **setMediaController(MediaController controller):** define el objeto *MediaController* con los botones multimedia

# PRÁCTICA

1. Crear una vista **VideoView** en **MediaPlayerActivity**
    1. Obtener la referencia en el método **onCreate** de la Activity
    2. Indicar la URI del video a reproducir:
        - https://img-9gag-fun.9cache.com/photo/aBxGoNN_460sv.mp4
    3. Invocar el método **start()** para comenzar a reproducir el vídeo

# MULTIMEDIA

- Para controlar la reproducción de vídeo, podemos:
  - O bien crear nuestra propia interfaz y llamar a los métodos correspondientes de VideoView
  - O bien utilizar la clase **MediaController** que creará unos componentes genéricos



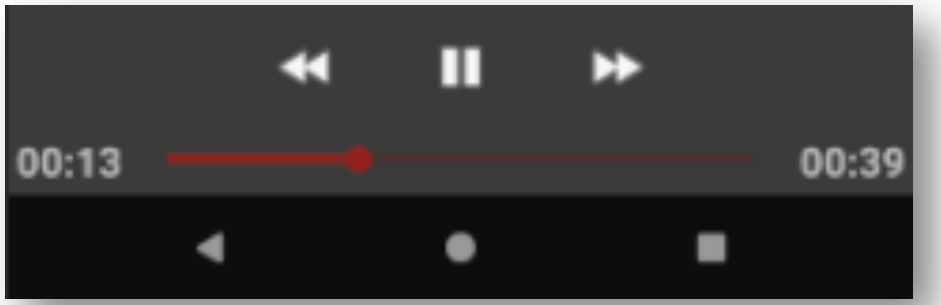
# MULTIMEDIA

- Para crear los controles, únicamente necesitamos:
  - Instanciar un objeto **MediaController**
  - Asignarlo al **ViewView**

```
MediaController mediaController = new MediaController(this);
mediaController.setAnchorView(videoView);
videoView.setMediaController(mediaController);
```

# PRÁCTICA

1. Crear el objeto **MediaController** y asignarlo a la vista **VideoView** que existe en **MediaPlayerActivity**



# ASINCRONICIDAD

- Una aplicación Android se ejecuta en un thread o hilo principal comunmente llamado **UI Thread** o **Main Thread**
- Si dicho hilo se bloquea con una operación que tarda varios segundos en completarse, la aplicación se cerrará mostrando un ANR (Activity Not Responding)

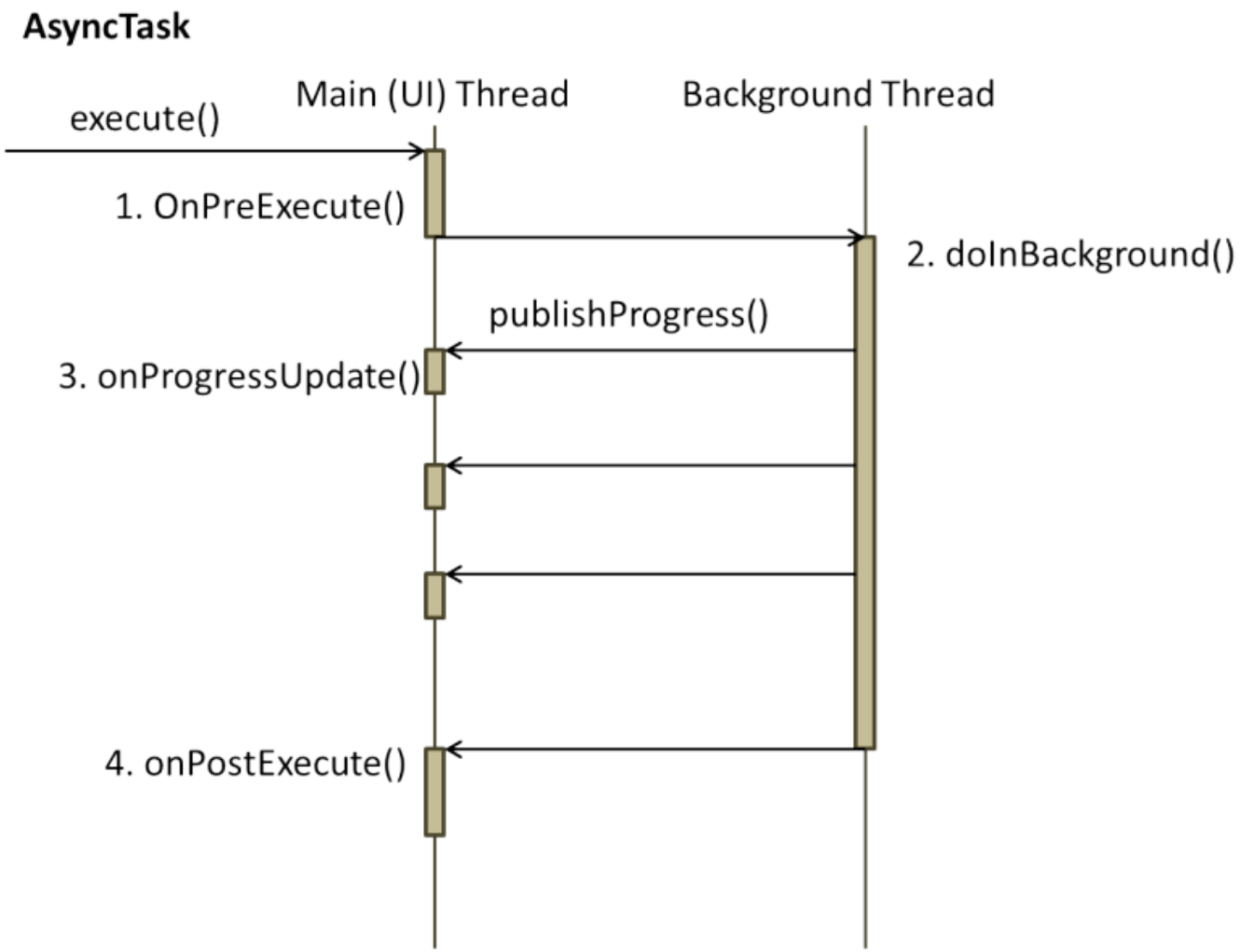
# ASINCRONICIDAD

- Para evitar dicho bloqueo, Android proporciona la clase **AsyncTask**, que permite ejecutar código en un hilo secundario sin interrumpir el **Main Thread**
- Nos evita tener que manipular threads y handlers de forma manual

# ASINCRONICIDAD

- **AsyncTask** permite ejecutar código en un hilo secundario y, a su vez, *publicar* resultados en el **Main Thread**
- El hecho de publicar dichos resultados en el **Main Thread** responde a la necesidad de cambiar vistas en éste hilo (ninguna vista se puede alterar fuera del hilo principal)

# ASINCRONICIDAD



# ASINCRONICIDAD

- Los métodos más importantes de un **AsyncTask** son:
  - onPreExecute(): sirve para inicializar variables o estados antes de ejecutar cualquier operación en segundo plano
  - **doInBackground()**: método que se ejecuta en el hilo secundario
  - onProgressUpdate(): publica un progreso intermedio
  - onPostExecute(): publica el resultado final

# ASINCRONICIDAD

- Para usar una **AsyncTask**, es necesario crear una clase hija (que herede de Async Task) e implementar los métodos arribar mencionados

```
private class ContadorAsyncTask
                    extends AsyncTask<Void, Integer, Integer> {
    . . .
}
```

# ASINCRONICIDAD

- **AsyncTask<Void, Integer, Integer>**
  - Los tres tipos de dato indicados entre **< >** corresponden a los tipos de datos que recibirán los métodos *doInBackground, onProgressUpdate y onPostExecute,* respectivamente
  - Si no interesa definir tipos, podemos usar **Void**

```
private class ContadorAsyncTask
                    extends AsyncTask<Void, Integer, Integer> {
    . . .
}
```

# ASINCRONICIDAD

- Ejemplo:

```
private class ContadorAsyncTask
                extends AsyncTask<URL, Integer, Integer> {
    protected Integer doInBackground(Void... voids) {
        // hilo secundario
    }

    protected void onProgressUpdate(Integer... progress) {
        Log.d("AsyncTask","Valor: " + progress[0].toString());
    }

    protected void onPostExecute(Integer result) {
        Log.d("AsyncTask","Valor final: " + result);
    }
}
```

# PRÁCTICA

1. Crear una nueva clase llamada **ContadorAsyncTask** que herede de **AsyncTask** e implemente los métodos mencionados anteriormente.
2. El método **doInBackground** deberá contar, segundo a segundo, hasta 100. Cada segundo, debemos parar el hilo con (utilizar *while*)**:**

```
try {
    Thread.sleep(1000);
} catch (Exception e) {
  e.printStackTrace();
}
```

# PRÁCTICA

3. Instanciar nuestra AsyncTask e invocar el método **execute()** en el **onCreate()** de **MediaPlayerActivity**
4. El video deberá reproducirse independientemente del contador. Esto demostrará que las operaciones se ejecutan en hilos separados.

# REACTIVE X

- **ReactiveX** es una librería que facilita el manejo de **flujos de datos** y eventos en tiempo real, utilizando el patrón **Observer** e **Iterator**

# REACTIVE X

- **ReactiveX** contiene dos clases principales sobre las que podemos operar:
  - **Observable:** es una clase que emite un flujo de datos o eventos
  - **Observer:** es una clase que actúa sobre los elementos emitidos por un Observable

# REACTIVE X

- Cómo funciona el flujo de datos:
  1. El objeto **Observable** emite, a través del método **onNext()**, un dato a todos los **Observer** que se hayan suscrito
  2. Los **Observer** utilizan el flujo de datos
  3. El **Observable,** una vez terminado de emitir los datos, notifica a los **Observer** a través del método **onComplete()**

# REACTIVE X

- *Ejemplo*: **Observable** que emite 3 enteros (1, 2 y 3), y luego se completa

```
Observable integerObservable = Observable.create(
                        new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter)
                                                throws Exception {
        emitter.onNext(1);
        emitter.onNext(2);
        emitter.onNext(3);
        emitter.onComplete();
    }
});
```

# REACTIVE X

- *Ejemplo*: **Observer** que recibe dichos enteros

```
Observer<Integer> subscriber = new Observer<Integer>()
    . . .
    @Override
    public void onNext(Integer value) {
        Log.d("RxAndroid","Subscriber: onNext("
                                    +value.toString() + ")");
    }

    @Override
    public void onError(Throwable error) {
        Log.d("RxAndroid","Subscriber: onError");
    }

    @Override
    public void onComplete() {
        Log.d("RxAndroid","Subscriber: onComplete");
    }
};
```

# REACTIVE X

- Un **Observable** y un **Subscriber**, por sí solos, no realizan ninguna acción
- Por norma general, el **Observable** comienza a emitir un flujo de datos cuando algún **Observer** se suscribe a su flujo de datos

```
integerObservable.subscribe(subscriber);
```

# REACTIVE X

- La creación de **Observable** y **Observer** se puede simplificar concatenando todas las llamadas y utilizando métodos como:
  - **Observable.just()**

```
Observable.just(1, 2, 3);
```

# REACTIVE X

- Observando **Observable.just()**

```
Observable.just(1, 2, 3).subscribe(new Observer<Integer>() {
    . . .
    @Override
    public void onNext(Integer value) {
        Log.d("RxAndroid","Subscriber: onNext("
                                    + value.toString() + ")");
    }

    @Override
    public void onError(Throwable error) {
        Log.d("RxAndroid","Subscriber: onError");
    }

    @Override
    public void onComplete() {
        Log.d("RxAndroid","Subscriber: onComplete");
    }
});
```

# RxANDROID

- **RxAndroid** es la versión específica de **ReactiveX** para Android
- Para utilizar la librería, necesitamos importarla en el fichero **app/build.gradle**

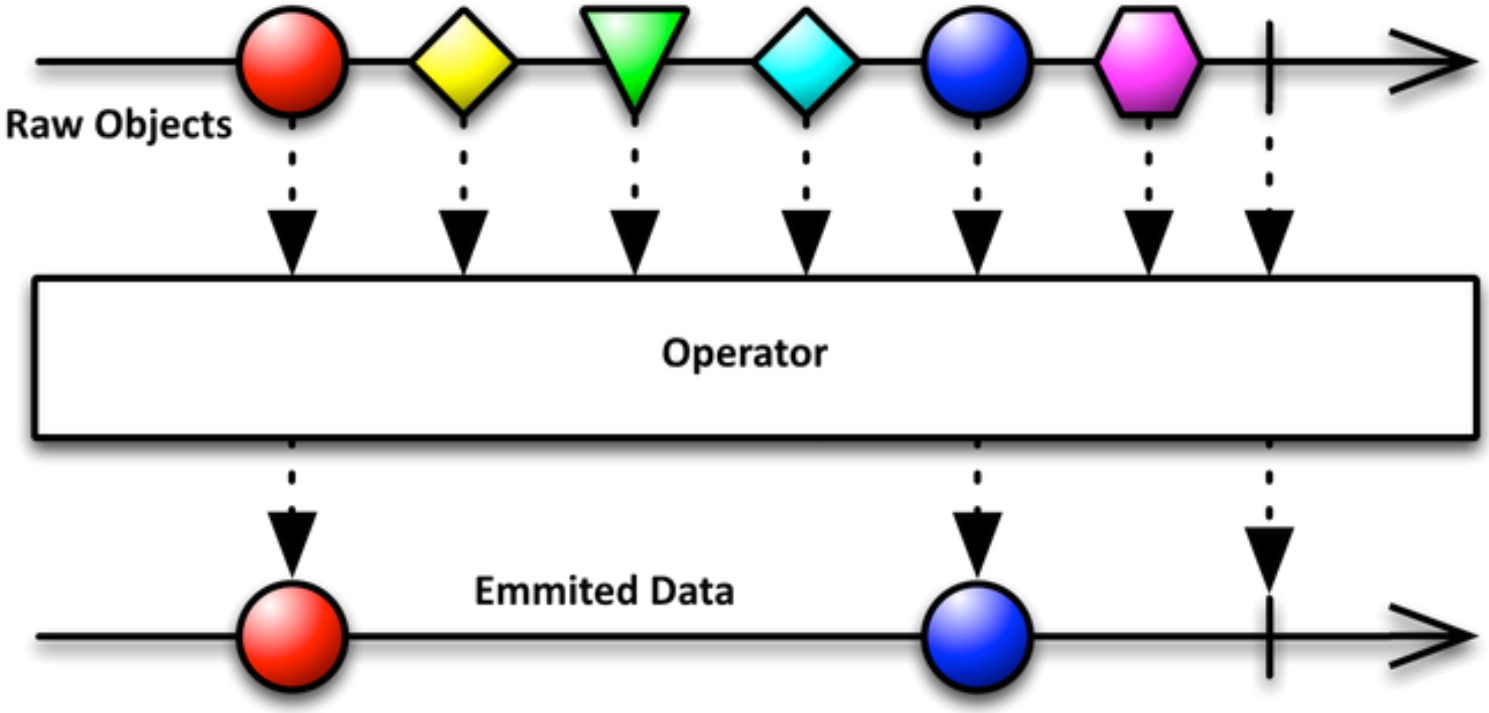
```
dependencies {
    . . .
    implementation 'io.reactivex.rxjava2:rxandroid:2.0.2'
    . . .
}
```

# PRÁCTICA

1. Crear una nueva Activity llamada **ReactiveXActivity** donde realizaremos las prácticas de RxAndroid
2. Importar la librería **RxAndroid** en **app/ build.gradle**
3. Crear el ejemplo de contador (de 0 a 10) con **Observable** y **Observer** en el método **onCreate** de la Activity
4. Imprimir los valores y logs en Logcat

# RxANDROID

- **RxAndroid** nos ofrece **Operators**
  - Funciones que sirven para transformar los datos emitidos por un **Observable**



# RxANDROID

- **RxAndroid** nos ofrece **Operators**
  - *Ejemplo*: filtar el flujo de datos para que únicamente emita valores impares

```
Observable.just(1, 2, 3, 4, 5).filter(new Predicate<Integer>() {
    @Override
    public boolean test(Integer integer) throws Exception {
        return integer % 2 == 1;
    }
});
```

# RxANDROID

- **RxAndroid** nos ofrece **Operators**
  - Los valores emitidos son **transformados** antes de ser enviados al **Observer**
  - Los **Operators** devuelven un objeto **Observable**, al que nos podemos subscribir inmediatamente

```
Observable.just(1, 2, 3, 4, 5).filter(new Predicate<Integer>() {
    @Override
    public boolean test(Integer integer) throws Exception {
        return integer % 2 == 1;
    }
}).subscribe(...);
```

# RxANDROID

- **RxAndroid** nos ofrece **Operators**
  - Existen un gran número de operadores que podemos aplicar:
    - **filter**
    - **map**
    - **concat**
    - …
  - Aquí la lista completa:
    - https://github.com/ReactiveX/RxJava/wiki/Alphabetical-List-of-Observable-Operators

# RxANDROID

- **RxAndroid** nos ofrece **Operators**
  - Otro operador muy utilizado es **map**
  - Sirve para *mapear* los datos, transformándolos de algún modo

```
Observable.just(1, 2, 3, 4, 5)
    .filter(...)
    .map(new Function<Integer, Object>() {
        @Override
        public Object apply(Integer integer)
                                            throws Exception {
            return Math.sqrt(integer);
        }
    });
```

# PRÁCTICA

1. Crear un nuevo **Observable** que emita los **10 primeros enteros: 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10**
2. Añadir un operador que filtre los datos y emita únicamente los números pares
3. Añadir un operador que *mapee* los números pares y los **leve al cuadrado**
   - *Math.pow(value, 2)*
4. Suscribirse al **Observable** e imprimir todos los valores en Logcat

# RxANDROID

- **RxAndroid** también cuenta con unos objetos llamados **Schedulers**
  - Estos objetos nos permiten especificar **en qué hilos se ejecutarán, tanto el Observable como el Observer**
  - Algunos Schedulers disponibles son:
    - Schedulers.io() − para operaciones de entrada/salida
    - Schedulers.computation() − para operaciones computacionales
    - Schedulers.newThread() − crea un nuevo hilo

# RxANDROID

- Si queremos observar o suscribir nuestros objetos en el **Main Thread** de nuestra aplicación Android, debemos utilizar
  - **AndroidSchedulers.mainThread**

```
integerObservable
        .observeOn(Schedulers.io())
        .subscribeOn(AndroidSchedulers.mainThread())
        .subscribe(subcriber);
```

# RxANDROID

- **observeOn()** – define el hilo en el que se ejecutará el **Observable**
- **suscribeOn()** – define el hilo en el que se ejecutará el **Observer**
  - El Observer suele ejecutarse en el hilo principal de Android, ya que suele modificar vistas en sus métodos *onNext, onComplete, onError, etc…*

```
integerObservable
        .observeOn(Schedulers.io())
        .subscribeOn(AndroidSchedulers.mainThread())
        .subscribe(subcriber);
```

# PRÁCTICA

1. Modificar el **Observable** del contador para que observe en el hilo **Schedulers.computation()**
2. Modificar el **Observer** del contador para que se suscriba en el hilo **AndroidSchedulers.mainThread()**

# RxANDROID

- Uno de los principales usos de **RxAndroid** en aplicaciones es la realización de tareas en segundo plano que resultan en algún valor o terminan con un error
- Existe una clase que simplifica mucho este tipo de operaciones: **Single**

# RxANDROID

- **Single** ejecuta un código en segundo plano y finaliza notificando al Observer con:
  - **onSuccess(Object o)** – notifica al Observer que todo ha ido bien y le envía un objeto para su uso
  - **onError(Throwable t)** – notifica al Observer que ha habido un error y le envía un *Throwable*

# RxANDROID

- Ejemplo de **Single**

```
Single.create(new SingleOnSubscribe<Object>() {
    @Override
    public void subscribe(SingleEmitter<Object> emitter)
                                            throws Exception {
        emitter.onSuccess("Operation success!");
    }
})
.observeOn(Schedulers.io())
.subscribeOn(AndroidSchedulers.mainThread())
.subscribe(new SingleObserver<Object>() {
    @Override
    public void onSuccess(Object o) { }

    @Override
    public void onError(Throwable e) { }
});
```

# RxANDROID

- De forma general, cualquier **suscripción** permanece en memoria hasta que es *liberada*.
- Uno de los métodos de **Observer** es **onSubscribe(Disposable d)**
  - El parámetro recibido, de tipo **Disposable**, representa la suscripción
  - Para liberar dicha suscripción, hay que llamar al método **dispose().** Este método parará la suscripción y liberará la memoria utilizada

# RxANDROID

- Ejemplo

```
private Disposable subscription;

@Override
protected void onDestroy() {
    super.onDestroy();
    if (subscription != null && !subscription.isDisposed()) {
        subscription.dispose();
    }
}
```

# PRÁCTICA

1. En **LoginActivity**, utilizar RxAndroid para realizar la acción de **login** del usuario:
   - Crear un **Observable** que acceda a la base de datos y realice todas las comprobaciones necesarias. Llamará al método **onNext(true)** del **Observer** si el login es correcto. Si no es correcto, llamará al método **onError()**
   - Crear el **Observer** que abra el intent de ProfileActivity si ha hecho login correcto, o muestre un toast si existe algún error