INDICE

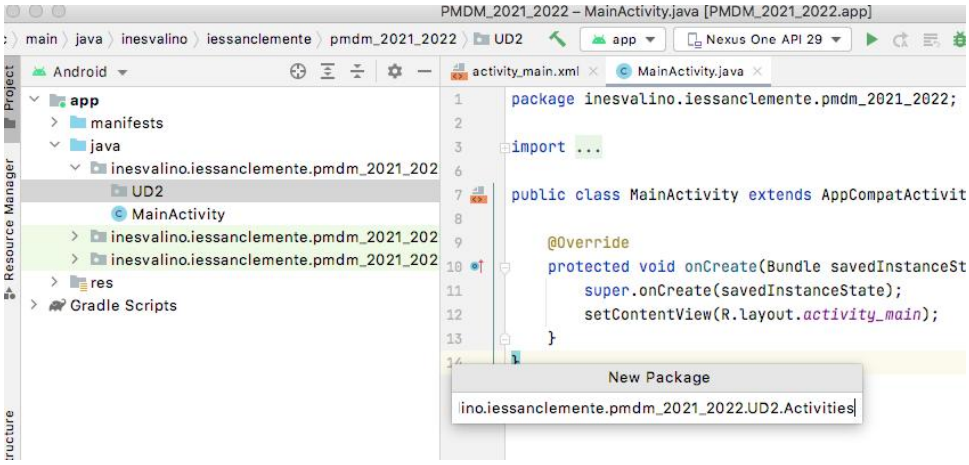
1.1. Introducción

Recordad que podemos identificar una "***Activity***" con cada una de las pantallas que componen una aplicación. Aunque esto no es del todo cierto siempre, ya que podemos tener aplicaciones que no cuentan con una interfaz gráfica, como servicios, programas que se ejecutan en segundo plano y responden a un determinado tipo de evento.

En este tema, explicaremos los diferentes métodos por los que pasa una actividad cuando se crea, se pone en segundo plano o se destruye. Indicaremos en cada uno de ellos cuáles serían las principales funciones que podríamos programar. También explicaremos cómo se define una actividad a nivel de proyecto.

Preparación previa:

- Crearemos un proyecto llamado ´***PMDM_2021_2022***´ (se trata del proyecto base)
- Crearemos un paquete '***UD2***' y dentro de éste otro paquete de nombre '***Activities'***.



1.2. Creando Activities

- Hay muchos tipos de '***Activities***' en Android. Todos los tipos lo que tienen en común son ventanas donde se 'dibujarán' elementos de la interfaz de usuario.
- En este punto nos centraremos en dos tipos de actividad:
  - **Activity**, son actividades clásicas donde el usuario interactuará con componentes gráficos.
  - **AppCompatActivity**, estas son actividades que se derivan de Activity y que se encuentran dentro de la librería de compatibilidad v7. Tienen la misma función que la clase Activity (sirven como pantallas para interactuar con componentes gráficos). Inicialmente se crearon para que AppBar pudiera usarse en versiones de Android anteriores a su aparición.

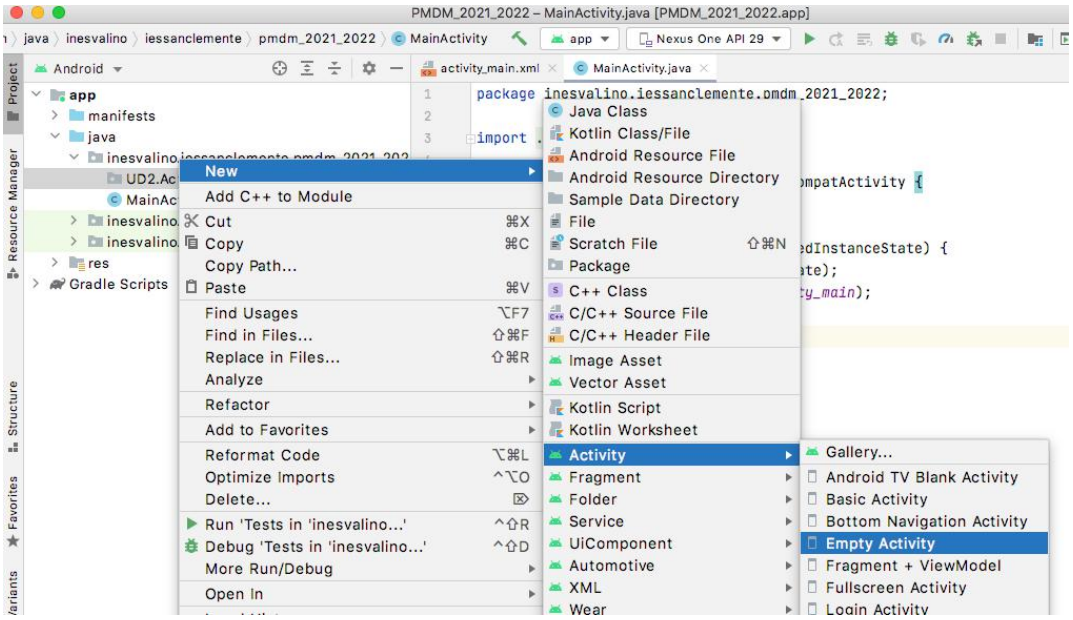
Como vimos en el tema anterior, es posible utilizar nuevas funcionalidades en versiones de Android previas a su aparición gracias al uso de esta clase.

En la versión actual de Android Studio (agosto de 2021) ya se crean por defecto las actividades de esta clase.

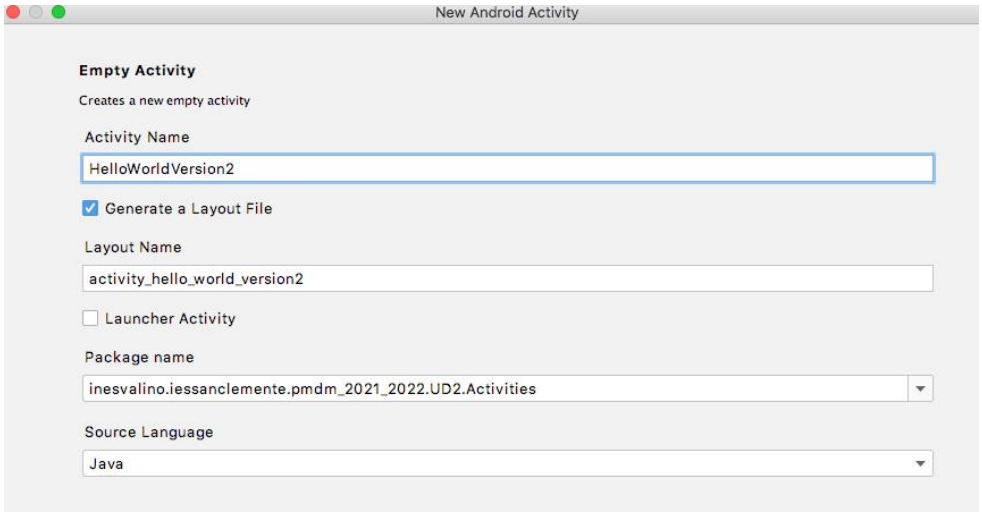
- Cada vez que se crea una Activity, aparece una nueva entrada en el archivo **AndroidManifiest.xml** .

  Veamos un ejemplo y creemos una nueva actividad con el asistente:

Creando una nueva activity en el paquete UD2.Activities

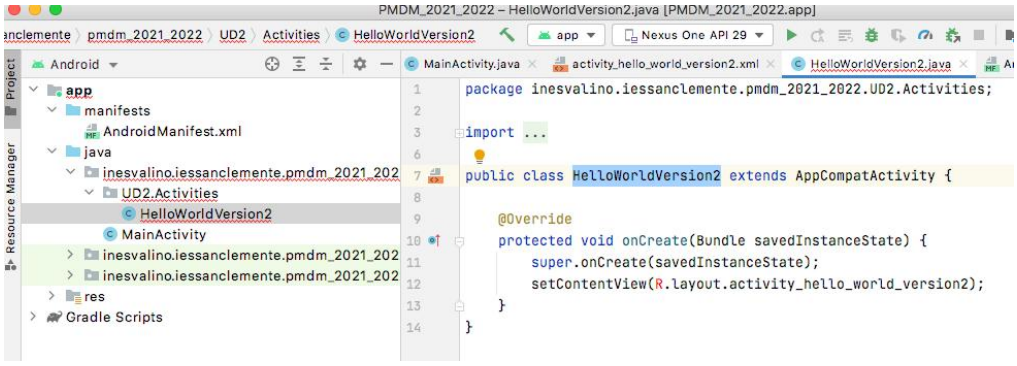


Creamos una "*Empty Activity*". Podemos crear diferentes actividades con una serie de componentes gráficos ya añadidos previamente y así ahorrarnos trabajo.



No marcamos la opción de que se pueda lanzar (*Launcher Activity*).

Como vemos en el archivo *AndroidManifiest.xml* se crea una nueva entrada para la actividad creada. Antepone un punto ya que Android para lanzar la actividad hace uso del identificar completo formado por el nombre del paquete y el nombre de la activity (os aparecerá un paquete diferente al de la imagen, pero es el mismo).



Como sucede en Java, el nombre de la clase debe ser el mismo que el nombre físico del archivo donde se crea. Este nombre debe coincidir con el nombre especificado en *AndroidManifiest.xml*.

Si al crear la actividad da un error con la clase R (aparece en rojo como en la última imagen), puede deberse a que no importó esa clase.

Importación de clases manual y automática

Hacemos un inciso para explicar cómo importar clases de forma automática y manual. El fallo que se resalta en trazo azul en la imagen anterior, se debe a que falta importar la clase R. Lo podemos hacerlo de dos formas como se ve a continuación.

- De forma **manual**, insertando en el código el *import* correspondiente:

```
package inesvalino.iessanclemente.pmdm_2021_2022.UD2.Activities;

import inesvalino.iessanclemente.pmdm_2021_2022.R;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;

public class HelloWorldVersion2 extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_world_version2);
    }
}
```

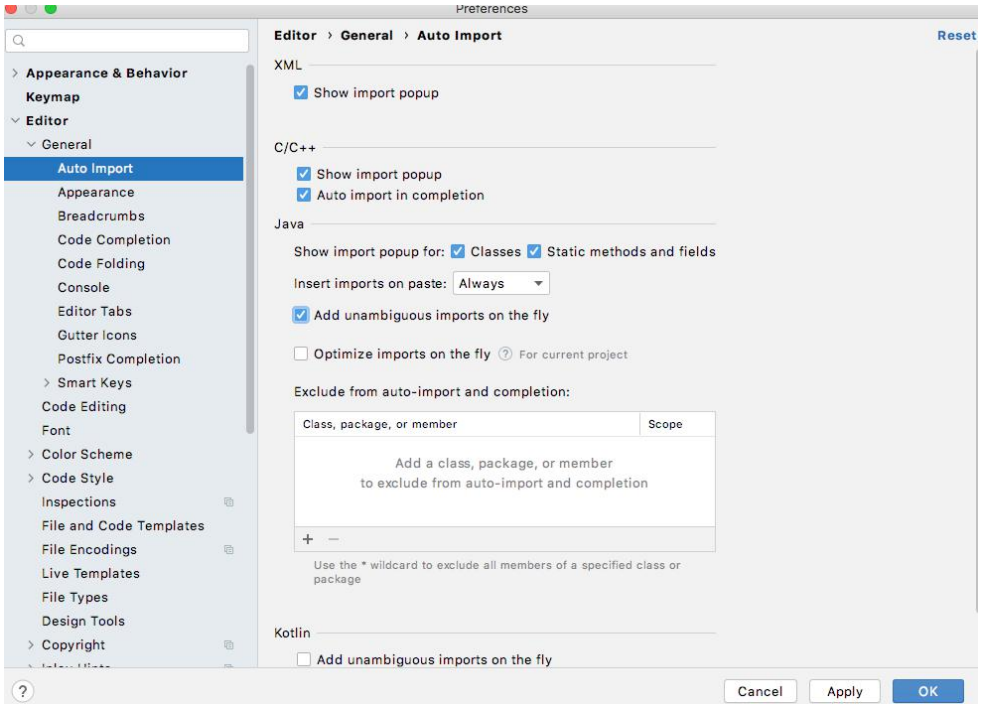
- De forma **automática**:

  File → Settings → Editor → General → Auto Import

  o dependiendo de tu versión

  Preferences-> Editor → General → Auto Import

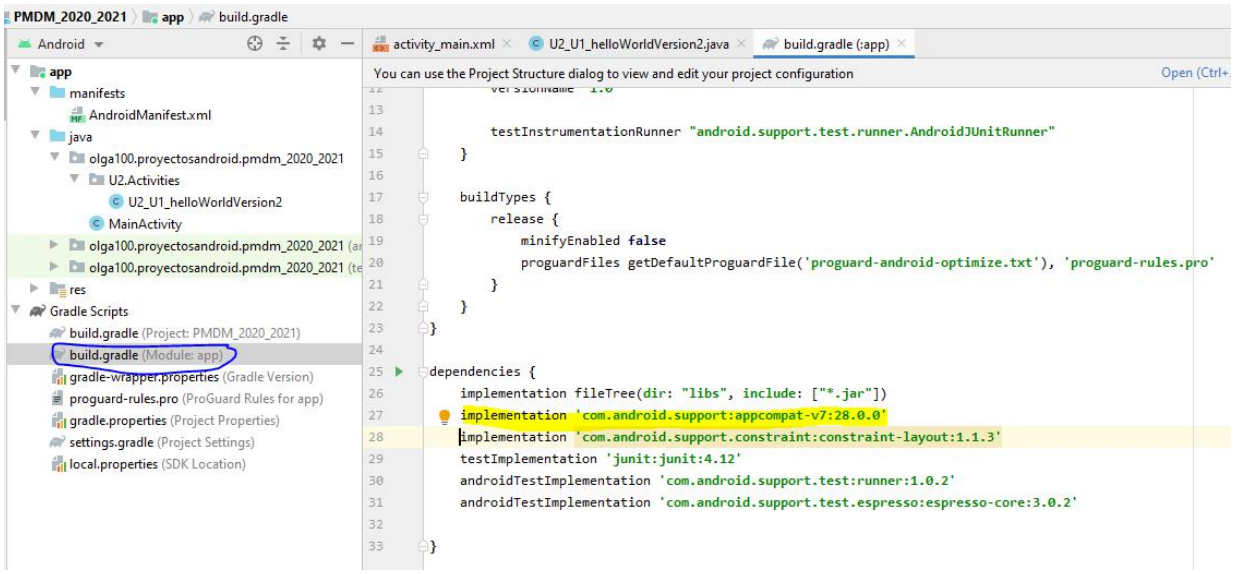
  Seleccionando "Add unambiguous imports on the fly"



- Al cambiar la forma en que Android Studio crea la actividad predeterminada (clase AppCompatActivity), tendremos dos consecuencias:

  - Si usamos la opción de bibliotecas de compatibilidad antiguas:

La actividad creada ya no se deriva de la clase Activity, sino de la clase AppCompatActivity del paquete de soporte v7. Como se ilustra en este otro ejemplo.



La biblioteca de compatibilidad se ha agregado al archivo **build.gradle** ya que la usamos en la actividad creada.

- Si usamos AppCompactActivity con bibliotecas de compatibilidad androidx:

```
MainActivity.java | activity_hello_world_version2.xml | HelloWorldVersion2.java

1  package inesvalino.iessanclemente.pmdm_2021_2022.UD2.Activities;
2
3  import androidx.appcompat.app.AppCompatActivity;
4
5  import android.os.Bundle;
6
7  import inesvalino.iessanclemente.pmdm_2021_2022.R;
8
9  public class HelloWorldVersion2 extends AppCompatActivity {
10
11     @Override
12     protected void onCreate(Bundle savedInstanceState) {
13         super.onCreate(savedInstanceState);
14         setContentView(R.layout.activity_hello_world_version2);
15     }
16 }
```

La actividad creada ya no se deriva de la clase Activity , sino de la clase *AppCompatActivity* del paquete de compatibilidad *androidx.appcompat.app*

La biblioteca de compatibilidad se ha agregado al archivo **build.gradle** ya que la usamos en la actividad creada.

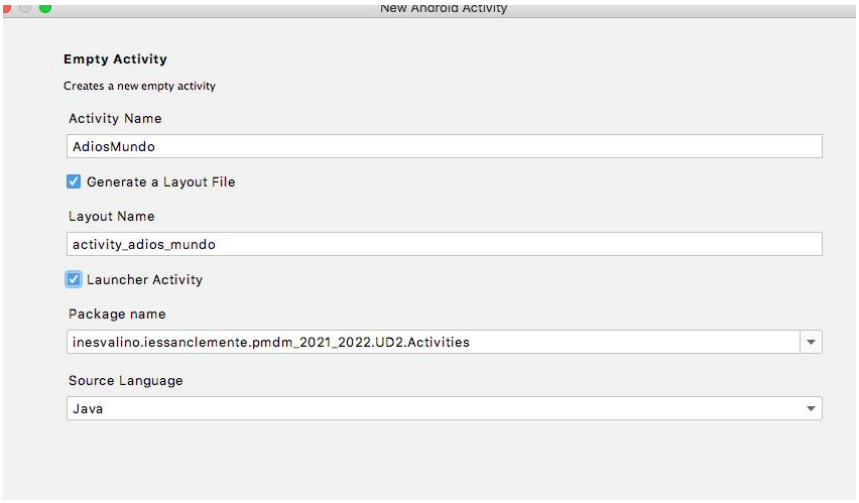
¿Qué pasa si habilitamos la opción "Launcher Activity" al crear una actividad?

Como podemos ver en el asistente, al crear una actividad existe una opción que indica **Launcher Activity**:

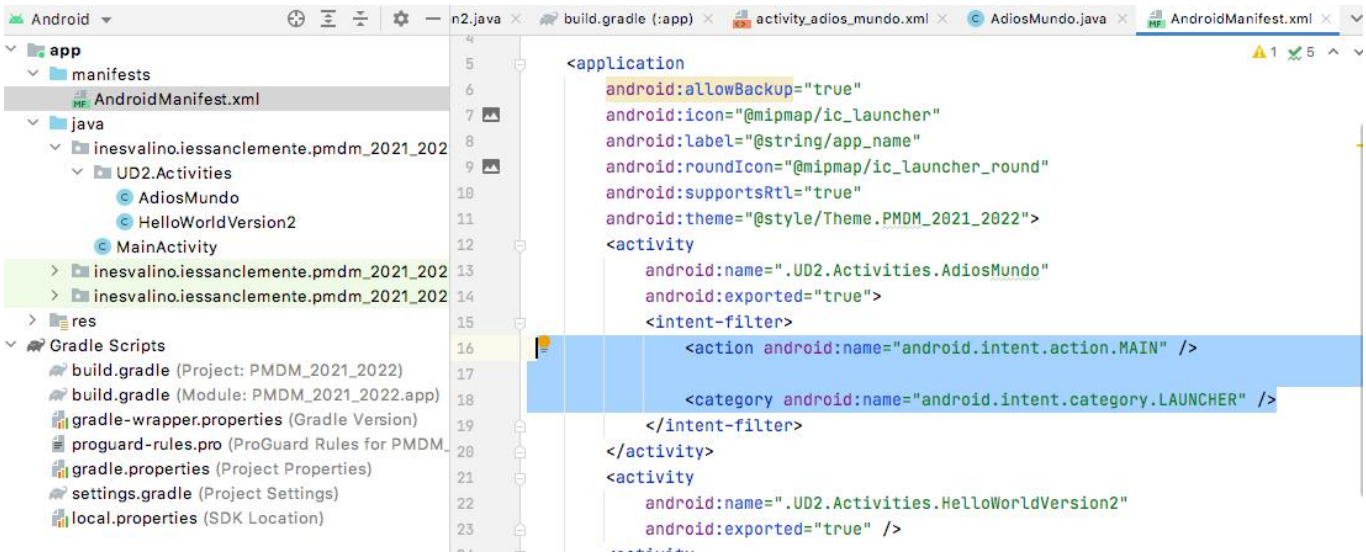
- **Launcher Activity**, hace que esta actividad se pueda iniciar o ejecutar de forma independiente. Esto significa que otras actividades que no tengan esta opción marcada deberán invocarse desde otra actividad.

Veamos qué ocurre al marcar esta opción.

Creamos una segunda actividad en el mismo paquete (UD2.Activities) con la opción *Launcher Activity* activada.



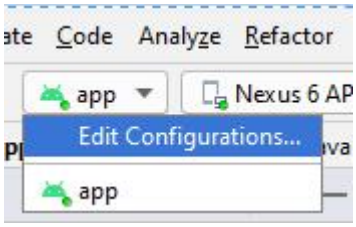
Se agrega una nueva sección a la actividad en el archivo *AndroidManifiest.xml*. Más adelante, veremos para qué sirve, pero esas dos líneas le dicen al sistema operativo Android qué actividad se puede iniciar de forma independiente y que no recibirá datos de otra actividad.



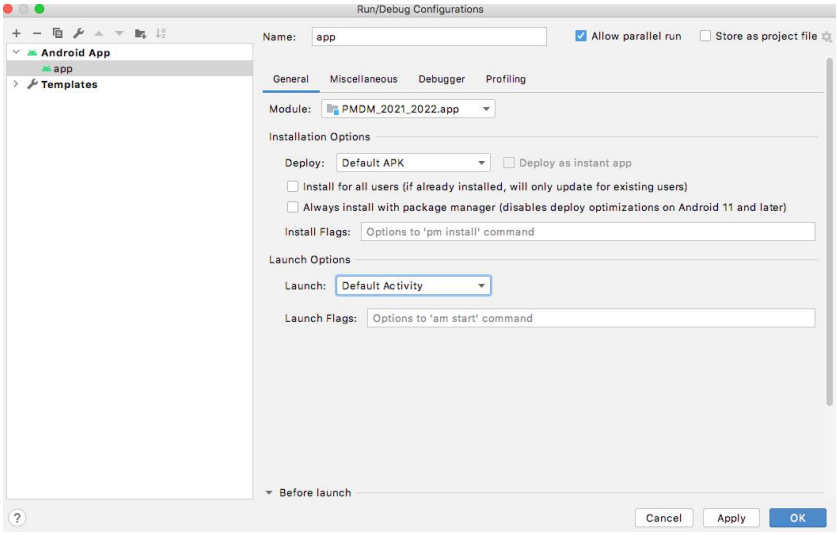
Si lanzas la aplicación y te diriges al dispositivo físico o al emulador puedes comprobar cómo aparecen las dos actividades que se pueden lanzar de forma independiente (notad que aparecen con el mismo nombre)

- Indicar que en las opciones de configuración de ejecución del proyecto podemos indicar cuál es la actividad por defecto a lanzar (si no se especifica será la primera en ser de tipo lanzador) y el emulador por defecto a lanzar:

¿Cómo modificar las opciones de configuración de ejecución?



Editamos la configuración...

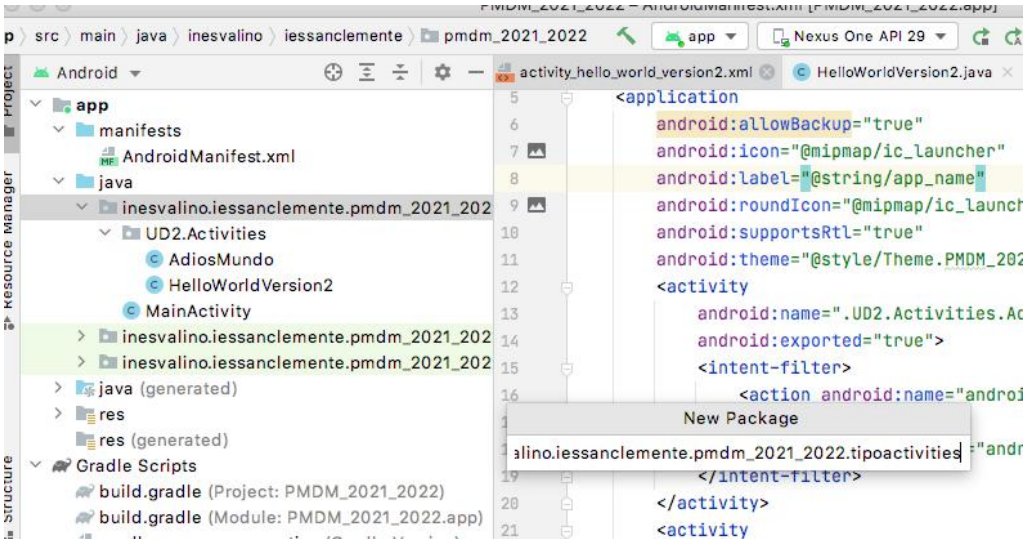


Si abrimos el desplegable se observa como podemos elegir la actividad a lanzar especificándola.

Haciendo más pruebas...

- Crearemos un nuevo paquete de nombre ***'tiposactivities'*** y moveremos las activities creadas anteriormente a este.
- Dejar al menos una actividad en el paquete inicial, porque si no gráficamente, Android Studio la hace desaparecer.

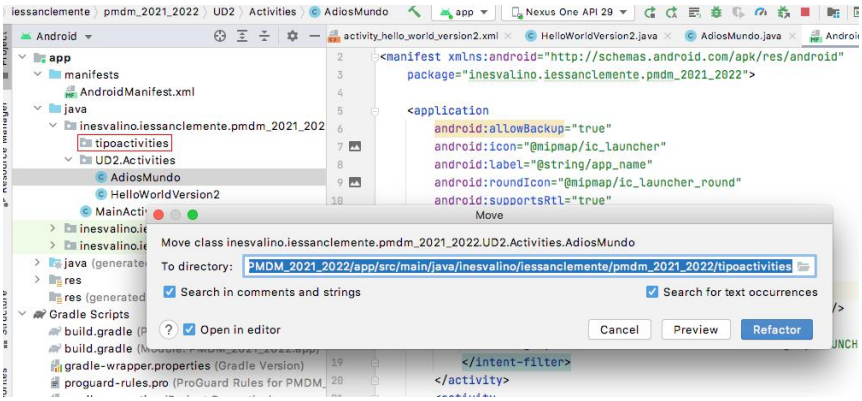
Creando un nuevo paquete



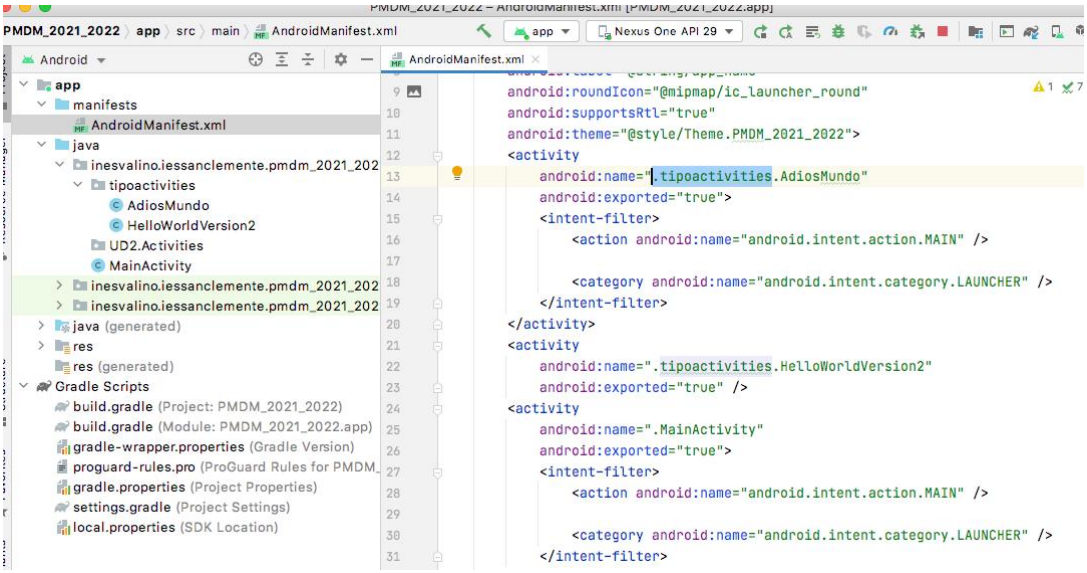
- Hacemos clic derecho sobre el paquete en el que queremos crear un subpaquete.
- Ingresamos el nombre del paquete a crear, en este caso 'tipoactivities' .

Recordar que cada paquete está dentro de una carpeta. Recordad que en Android Studio un paquete que no contiene clases/ficheros no lo mostrará, es decir, un paquete sin contenido, no lo muestra.

Arrastramos las actividades al nuevo paquete. Aparecerá una ventana y presionaremos el botón **Refactorizar**. Si se produce algún error, intentar arrastrarlos uno por uno, limpiar el proyecto y volver a compilar.



Pulsamos el botón **'Refactor'**.



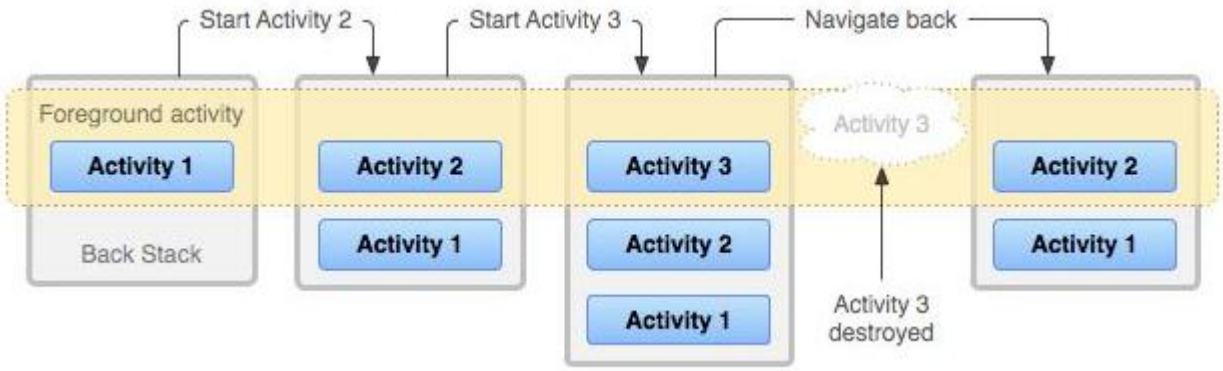
Observar cómo ahora en *AndroidManifiest.xml* aparece el nuevo paquete.

Nota: Recordar, como en Java, cada paquete se traduce a una carpeta de nivel de sistema operativo.

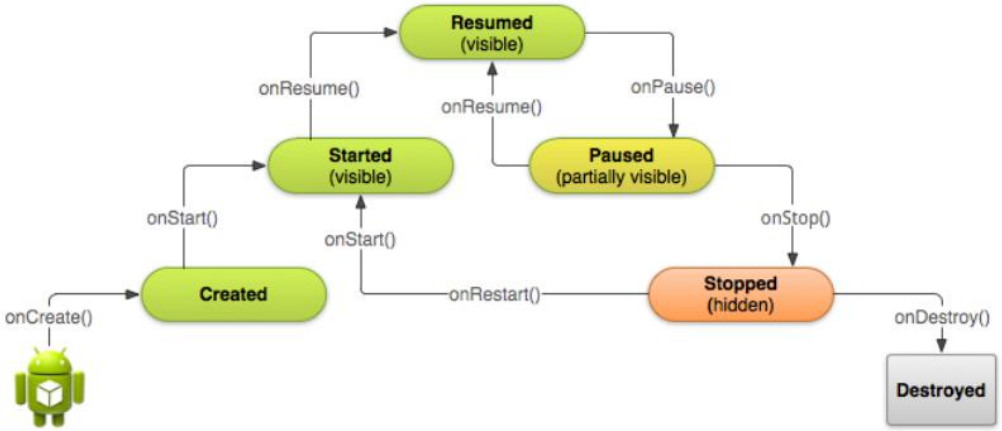
1.3. Ciclo de vida de las Activities

Se puede encontrar información detallada sobre el ciclo de vidas de las activities en este enlace, pero aquí haremos una breve descripción .

- Una aplicación de Android puede estar compuesta por muchas activities (pantallas) e incluso llamar a otras aplicaciones.
- Cuando iniciamos una actividad, esta pasa al primer plano (Visible) y la actividad anterior se detiene y se envía justo por detrás de la actual en la Pila (Back stack).
- Esta pila usa el mecanismo de colas LIFO. Cuando presionamos el botón Back (tecla de retroceso) en el teléfono se destruye la actividad actual, y recarga la actividad que está en la parte más alta de la cola.



- Desde que iniciamos una activity hasta que salimos de ella pasa por una serie de estados: **El ciclo de vida de una actividad.**



1.3.1. Métodos asociados a los estados

Cuando una actividad cambia de **estado** (porque se presiona una tecla del teclado, por se presiona un botón, etc) este cambio es notificado a la actividad a través de los métodos **callback**. Todos estos métodos callback capturan los cambios de estado que se van produciendo en la actividad y pueden ser sobreescritos para que realicen las operaciones que deseamos.

- Si se observa la imagen esquemática de arriba, vemos como tiene forma de pirámide, desde el **estado** en el que se **lanza** una actividad hasta que llega al estado **Resumed** (Running), esta pasa por los estados **Created** y **Started**.
- Cada uno de esos cambios de estado de la actividad lleva asociado un **método** callback que será llamado en el momento de producirse el cambio. Por orden: **onCreate(), onStart(), onResume()**.

- Una actividad estará **Paused** si está semi-visible porque hay otra actividad en primer plano por encima de la primera que no ocupa toda la pantalla. En este cambio se llamará al método **onPause().** De este estado puede pasar a **Running**, llamando al método **onResume()** para este cambio de estado.

- Una actividad pasa a segundo plano pasa al estado **Stopped** (no visible) bien porque se abrió una nueva actividad o porque se presiona el botón **Home** del móvil. Si pasamos del estado **Resumed** a **Stopped** estes cambios son capturados por los métodos **onPause()** y **onStop().**
- Una actividad puede pasar de **Stopped** a **Resumed**, pasando por **Started** porque vuelve a pasar a primer plano la actividad que estaba oculta en la pila de actividades, en este caso los métodos que capturan estos cambios son: **onRestart(),** y de nuevo **onStart()** y **onResume()**.
- Cuando una actividad está en estado **Stopped** se retienen todas sus actividades, información de estado y recursos que está usando.

- Una actividad pasa al estado **Destroyed** en los siguientes casos:
  - A actividad está en primer plano (**Resumed**) y se presiona el botón **Back**, en este caso pasamos de **Resumed** a **Destroyed**, pasando por **Paused** y **Stopped**, llamando a todos los métodos que hay por el camino.
    - En caso de haber una actividad en segundo plano (que ahora está en estado Stopped) antes de destruir la actual, esa será traída a primer plano pasando de Stopped a Resumed.
  - La actividad tiene programado en su código que se destruya explícitamente con el método **finish()**, por ejemplo al tocar el botón de salir.
  - La actividad está en el fondo de la pila de actividades (en estado **Stopped**) y el sistema necesita sus recursos para poder asignarlos a una nueva actividad que el usuario quiere abrir. En este caso el sistema destruye esta actividad y por tanto pierde todo aquello que no fuese guardado antes de pasar al estado Stopped.
  - Desde el **administrador de aplicaciones**.

Fijándonos en los diferentes métodos uno por uno, lo que se debe hacer en cada uno de ellos es lo siguiente:

- **Método onCreate**:
  - Definir / instanciar variables – clases
  - Defina la interfaz de usuario.
  - Vincular datos a listas.

Una vez llamado al método *onCreate*, llama a los métodos *onStart* y *onResume* consecutivamente.

- **Método onDestroy**: Se llama cuando finaliza la aplicación y se libera de la memoria del dispositivo, cuando **cerramos** una actividad o cuando pulsamos el botón ATRÁS. También puede ser causado por el sistema operativo para liberar espacio en la memoria. Si la aplicación utiliza subprocesos en segundo plano u otro recurso que pueda consumir memoria, debe destruirse en este caso. Por lo general, liberaremos recursos que han sido 'instanciados' en el método *onCreate*.

  Nota: Solo hay un caso en el que no pasa por los métodos *onPause* y *onStop*, es cuando destruimos la aplicación llamando al método *finish ()* desde *onCreate*.
- **Método onPause**: cuando una aplicación ocupa parte de la pantalla, pero deja una parte visible de la aplicación en ejecución, pasa a un estado de PAUSA (por ejemplo, un cuadro de diálogo abierto por una aplicación). De lo contrario, si está completamente cubierto, pasa al estado STOP.

Cuando regresa del estado de pausa, se llama al método *onResume*.

Por lo general, cuando el proc. *onPause*, es que la aplicación dejará de ser utilizada por el usuario. En este caso, este estado se usa para:

- Detenga las animaciones u otras actividades que consuman la CPU.
- Guarde la información que el usuario espera encontrar si vuelve a cargar la aplicación (escribiendo un correo electrónico sin terminar,…).
- Libera recursos del sistema, como transferencias, sensores (GPS) o cualquier función que pueda afectar la batería (como la cámara).

Este método no debe sobrecargarse con un "trabajo" pesado de liberación, ya que podemos ralentizar el paso al siguiente estado. Para ello tenemos el método *onStop*.

- **Método onResume**: este método se utiliza para recrear los recursos que publicamos en el método *onPause*.

  Tener en cuenta que ingresamos a este método cuando iniciamos la aplicación por primera vez y cuando la detuvimos, por lo que puede ser necesario verificar si los recursos ya se crearon antes de volver a crearlos.

  La aplicación permanece en este estado hasta que cambia de actividad o veamos un supuesto ejemplo en el que hacemos uso de la cámara. Como vemos comprobamos si el objeto que usará la cámara ya está inicializado. Esto puede suceder en una aplicación de múltiples ventanas en la que movemos el foco a otra ventana y luego regresamos a la ventana inicial.

```
1    @Override
2    protected  void onResume(){
3        super.onResume();
4
5        if (mCamara==null){
6            inicializarCamara();
7        }
8    }
```

- **Método onStop**.

  Hay algunos casos en los que esto sucede:

  - El usuario abre otra aplicación o va a aplicaciones recientes y elige otra. Hasta que regresa a nuestra aplicación, entra en un estado de detención y permanece en segundo plano.
  - El usuario provoca la aparición de otra actividad. Se detiene la primera actividad y comienza la segunda. Si el usuario presiona el botón 'Atrás', se reinicia la primera actividad.
  - El usuario recibe una llamada telefónica.

  Por lo tanto, tenemos los métodos *onStop* y *onRestart* para respaldar estos eventos.

  En el método *onStop* es donde deberíamos realizar las operaciones que requieren la mayor liberación de recursos, como almacenar datos en una base de datos.

Nota: El sistema guarda el estado de las Views, por lo que el contenido de los cuadros de texto y otros recursos gráficos se mantienen cuando se reinicia la aplicación. Solo se pierden si el sistema operativo necesita espacio en la memoria y elimina la aplicación de la pila, destruyendo todo. En ese caso, si el usuario regresa, se inicializarán los gráficos.

- **Método onRestart** y **método onStart** :

  Cuando el sistema llama al método de reinicio, también llama al método *onStart* (cada vez que la aplicación se hace visible).

  Debido a esto, uno debe inicializar los componentes que se lanzaron en el método onStop y también debe aprovechar ese mismo código cuando se crea la aplicación por primera vez.

  Por ejemplo, en el método onStart puedes poner las condiciones necesarias para que la aplicación funcione (como recursos que necesitemos, como GPS, una cámara, ...).

  En el método onRestart, se pueden recuperar los cursores de acceso a datos que se habían cerrado previamente en el método onStop. Más información en este enlace.

  La aplicación no permanece en estos estados, a menos que vaya directamente al estado Resumed y llame al método asociado onResume ().

NOTA IMPORTANTE: Si se sobreescribe cualquiera de estos métodos, primero debe llamar al método de la superclase de la forma: *super.Method()*;