Tema 4. Fundamentos de aplicaciones

Prof. Manuel Castillo

Programación de Dispositivos Móviles

Escuela Profesional de Ciencias de la Computación

Facultad de Ciencias

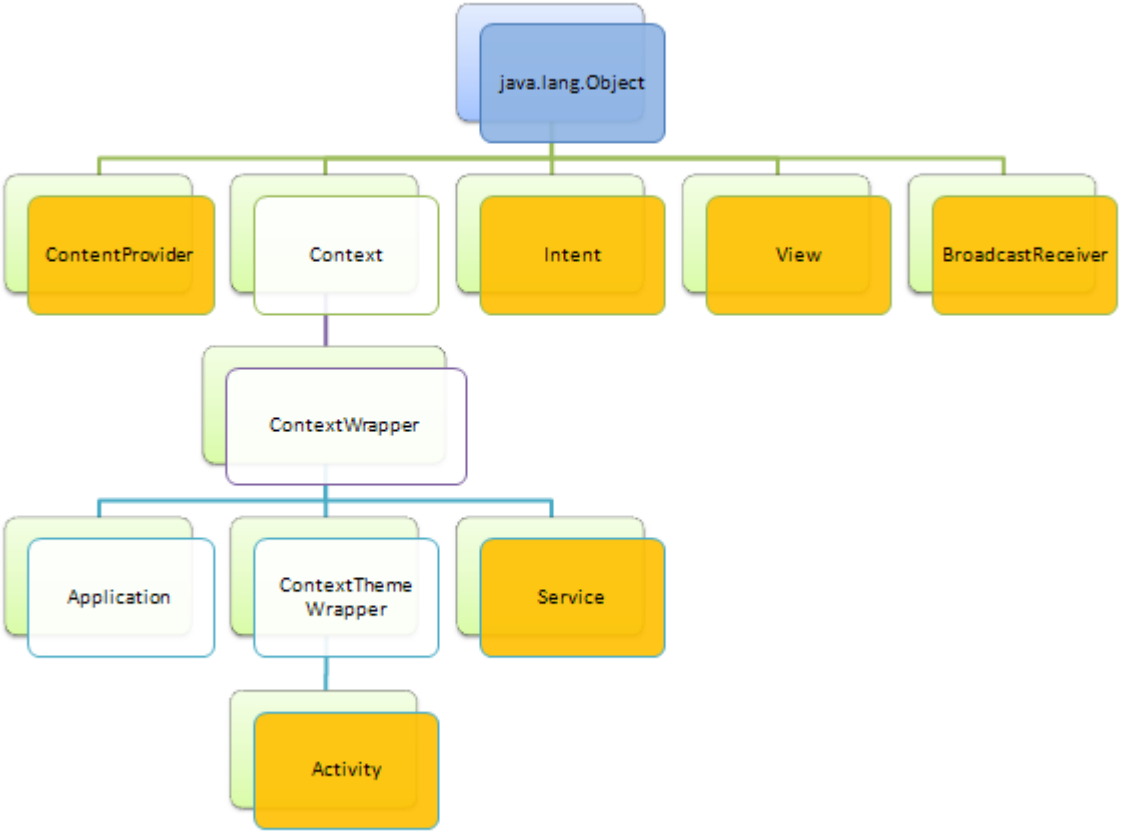
Universidad Nacional de Ingeniería

Objetivos

- Conocer los distintos componentes disponibles en Android.
- Saber para qué utilizar los distintos tipos de componentes.

Índice de contenido

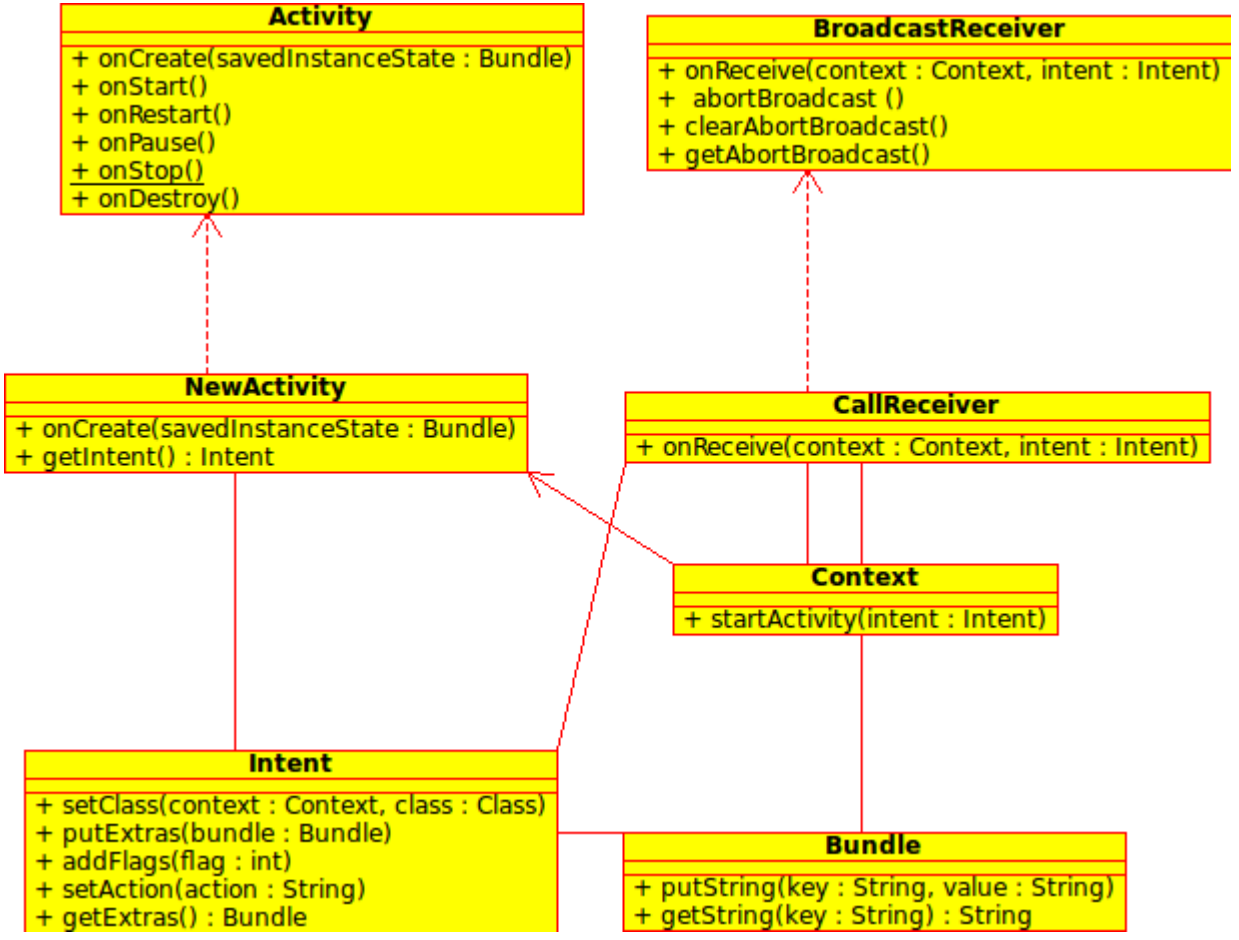
- Introducción
- AndroidManifest
- Activity
- Application
- Intent
- Broadcast Receiver
- Content Provider
- Notification
- Manager



Introducción

- Android maneja una estructura de componentes que permite la generación de aplicaciones muy al estilo de Java.
- Se basa en un modelo de programación muy parecido al MVC.
- Tiene separadas las capas de presentación y la lógica de negocio.

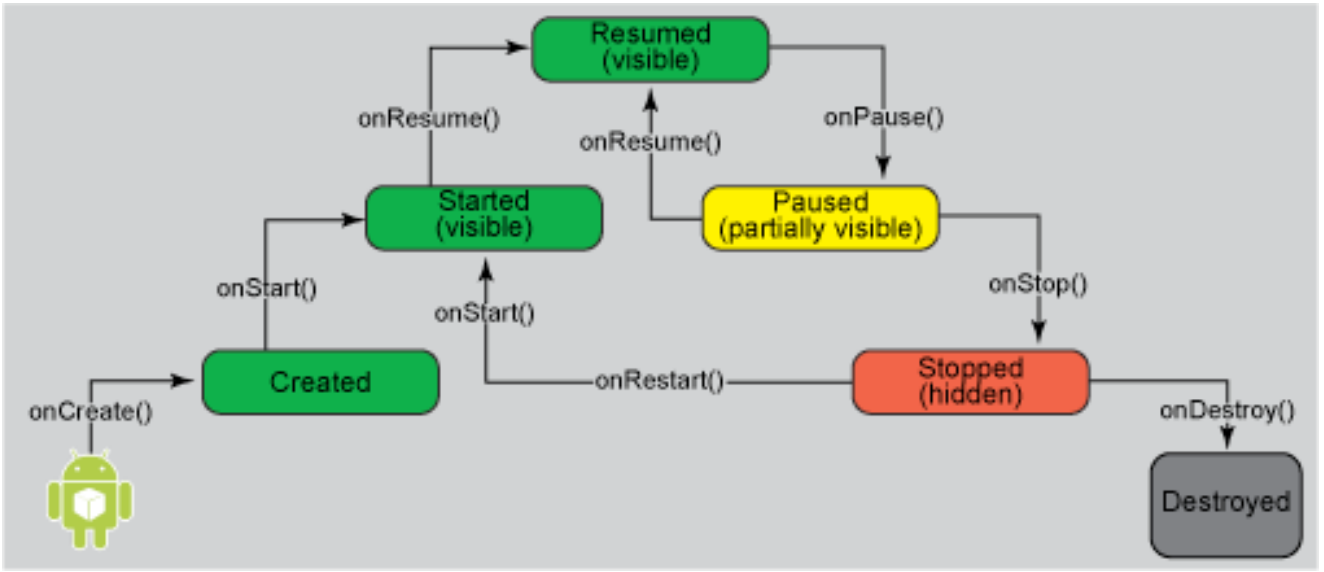
1. AndroidManifest.xml



1.1. Definición

- Permite controlar las aplicaciones distribuidas.
- Establece los permisos de usuario.
- Define las bibliotecas a utilizar.
- Especifica todos los componentes del paquete.
- Registra los componentes que se ejecutan.

2. Activity

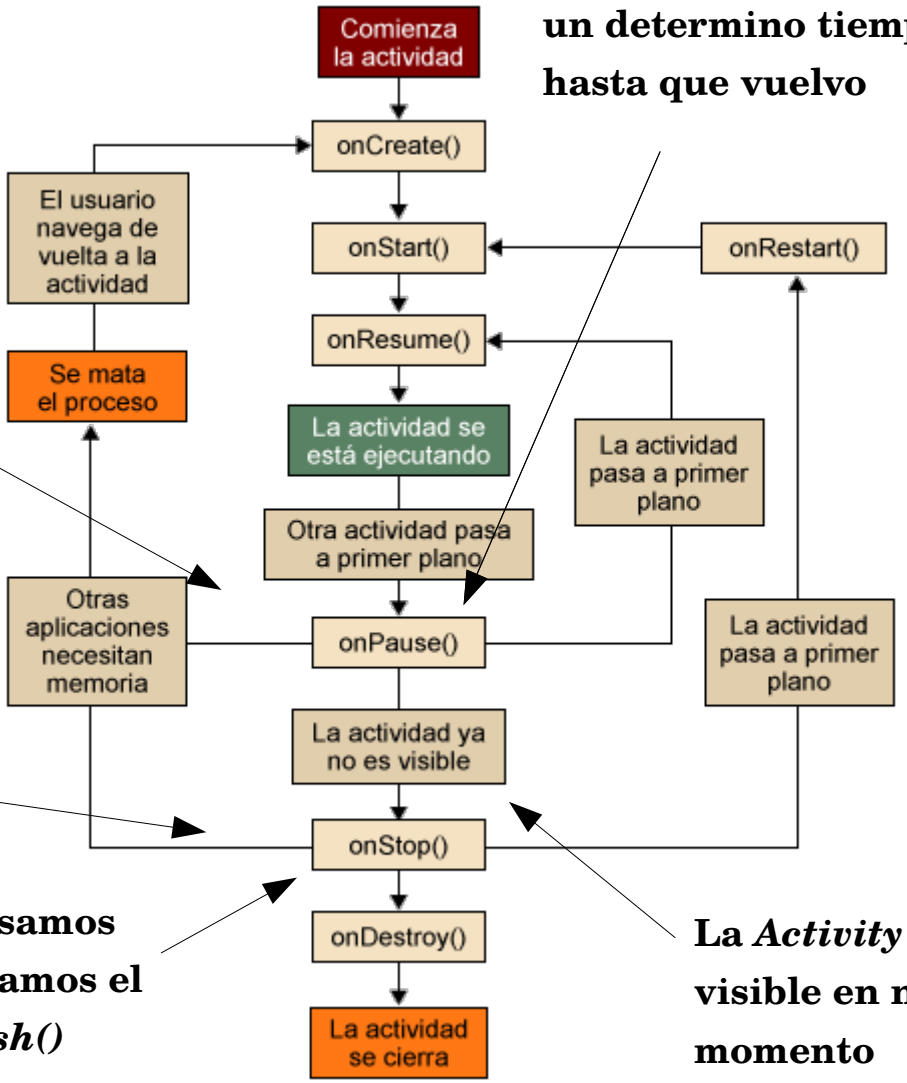

Resumed
(visible)
onResume()
onResume()
onPause()
Started
(visible)
Paused
(partially visible)
onStart()
onStart()
onStop()
Created
onRestart()
Stopped
(hidden)
onCreate()
onDestroy()
Destroyed

2.1. Definición

- Es el controlador principal de la ejecución.
- Permite controlar lo que se va a visualizar en la pantalla.
- Hay distintos tipos de controladores específicos.
- Todos heredan de *Activity* y está relacionados con los distintos tipos de pantallas que maneja Android.

```
import android.support.v7.app.AppCompatActivity;

public class primeraAplicacion extends AppCompatActivity {
    //código omitido
}
```

2.2. Ciclo de vida



Actividad congelada un determino tiempo hasta que vuelvo

Proceso automát. de liberación de memoria de Android, NO perdemos datos

Proceso automát. de liberación de memoria de Android, perdemos datos

Cuando pulsamos atrás o lanzamos el método *finish()*

La *Activity* ya no está visible en ningún momento

2.3. Métodos principales

- *setContentView*: permite cargar una vista.
  - *setContentView(R.layout.activity_main);*
- *findViewById*: permite capturar un objeto de la vista.
  - *private EditText et1 = (EditText)findViewById(R.id.et1);*
- *startActivity*: permite arrancar otra *activity*.
  - *startActivity(intent);*
- *onCreateOptionMenu*: carga un menú de opciones.
- *onOptionsItemSelected*: permite realizar acciones dependiendo del elemento del menú seleccionado.

3. Application

3.1. Definición

- Permite arrancar una clase aplicación cuando arranca la aplicación.
- Permite gestionar las conexiones con las fuentes de datos de manera independiente a las *activities*.
- Luego se puede acceder a la instancia del objeto de la aplicación desde las *activities*.
- Maneja el enlace a la información de nuestra aplicación, por ejemplo, las BBDD.

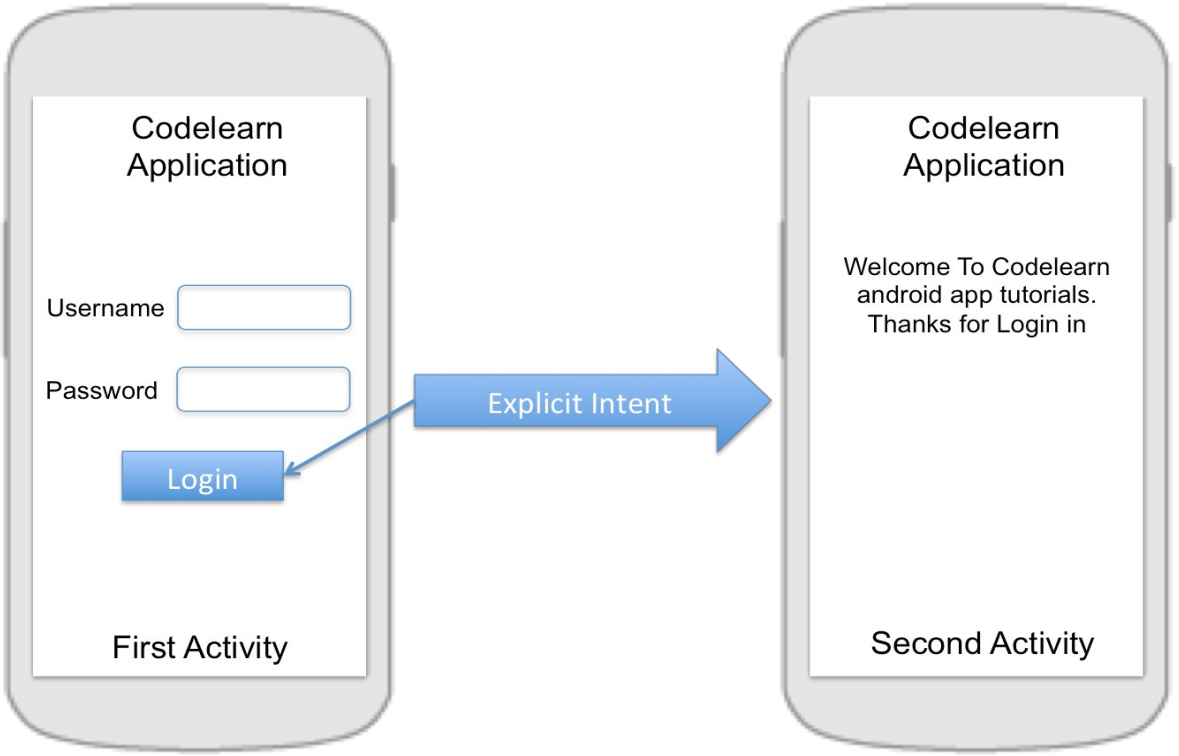
3.2. Estructura


```
public class MyApplication extends Application {
    @Override
    public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
    }
    @Override
    public void onCreate() {
        super.onCreate();
    }
    @Override
    public void onLowMemory() {
        super.onLowMemory();
    }
    @Override
    public void onTerminate() {
        super.onTerminate();
    }
}
```

3.3. Métodos

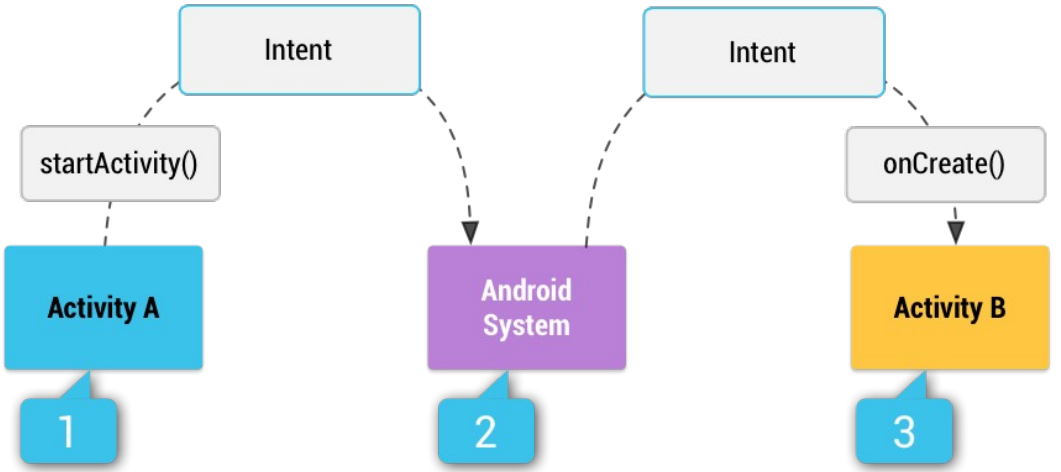
- *onCreate*: método inicial.
- *onLowMemory*: Si nos quedamos sin memoria.
- onTerminate: método final.
- onConfigurationChange: si se ha cambiado la configuración de la aplicación.

4. Intent


Codelearn Application
Username
Password
Login
First Activity
Explicit Intent
Codelearn Application
Welcome To Codelearn android app tutorials. Thanks for Login in
Second Activity

4.1. Definición

- Es el componente que nos permite realizar saltos entre actividades (de una pantalla a otra).
- Permite la comunicación entre aplicaciones de Android.
- El Sistema puede generar *Intents* para notificar acciones que hayan pasado a las aplicaciones.
  - Por ejemplo cuando tenemos una llamada entrante.
- El arranque de una aplicación se realiza a través de una *Intent* que genera el sistema.



4.2. Tipos de Intent

- Implícita:
  - Realiza un salto conforme a la configuración del dispositivos.
  - Permite realizar saltos a *Activities* que desconocemos o que son de sistema.
    - Ejemplo: se especifica la funcionalidad requerida a través de una acción (por ejemplo marcar) y un dato (por exemplo el número que se desea marcar), y Android debe determinar el mejor componente para su utilización.
- Explícita:
  - Nosotros le indicamos el destino de la *Intent* porque conocemos la clase destino declarado previamente en *AndroidManifest.xml* con *intent-filters*.
  - Indicamos qué es lo que queremos y será el sistema el responsable de identificar la *Activity* correspondiente para manejar esa acción.
    - Ejemplo: Al hacer el una fotografía el sistema nos lleva automáticamente a la *Intend* que captura una foto.

4.3. Campos de Intent

- ACTION: un *String* que representa o nombra la operación que va a ser realizada.
  - ACTION_DIAL: queremos que alguien marque un número de teléfono.
  - ACTION_EDIT: queremos que se muestren algunos datos para editar por el usuario.
  - ACTION_SYNC: nos gustaría sincronizar algunos datos en nuestro dispositivo con los datos de un servidor.
  - ACTION_MAIN: queremos iniciar una actividad como la actividad inicial de una aplicación.
- DATA: representa los datos que son asociados con el *Intent*. El formato de los datos es una *URI* (Uniform Resource Identifier).
- CATEGORY: representa información adicional sobre los tipos de componentes que puede manipular o debe manipular este *Intent*.
  - CATEGORY_BROWSABLE: la actividad de destino permite ser iniciado por un navegador web para mostrar datos referenciados por un enlace *URI*, como una imagen o un mensaje de correo electrónico.
  - CATEGORY_LAUNCHER: la actividad es la actividad inicial de una tarea y aparece en la aplicación de inicio del sistema.
- TYPE: especifica los tipos *MIME* (Multipurpose Internet Mail Extensions) de los *Intents* de datos.
- COMPONENT: especifica la Actividad de destino del *Intent*.
- EXTRAS: añaden información adicional al *Intent*.

4.4. Iniciar Actividades

***startActivity*(*Inten intent*)**: éste método es utilizado para empezar una nueva actividad.

```
Intent iniciarNuevaActividadIntent = new Intent(
      ActividadActual.this, SiguienteActividad.class);
startActivity(iniciarNuevaActividadIntent);
```

startActivityForResult(Intent intent, Int int): nos devuelve un *Intent* que contiene cualquier dato adicional que deseemos.

```
private final static int NOMBRE = 0;
...
Intent iniciarNuevaActividadIntent   = new Intent (
      ActividadActual.this, SiguienteActividad.class);
startActivityForResult(iniciarNuevaActividadIntent , NOMBRE);
...
protected void onActivityResult(int requestCode, int resultCode, Intent data){
   super.onActivityResult(requestCode, resultCode, data);
}
```

4.5. Intent-Filter

- Etiqueta de *AndroidManifest.xml*.
- Especifica características de *Intents* que recibe el componente.
  - Si no se indica solamente se podrá iniciar otra Actividad con *explicit Intent*.

Acción a realizar por el Intent

```
<activity android:name="ShareActivity">
   <intent-filter>
      <action android:name="android.intent.action.SEND"/>
      <category android:name="android.intent.category.DEFAULT"/>
      <data android:mimeType="text/plain"/>
   </intent-filter>
```

Datos aceptados por el Intent

Categorías aceptadas por el Intent

4.6. Ejemplos

```
// Abrir página web
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse("https://www.uni.edu.pe"));
        startActivity(intent);
//Marcar número de teléfono
        Intent intent = new Intent(Intent.ACTION_DIAL);
        startActivity(intent);
//Llamada teléfonica
        Intent intent = new Intent(Intent.ACTION_CALL);
        intent.setData(Uri.parse("tel:"+getResources().getString(R.string.tfno)));
        startActivity(intent);
//Mostrar coordenada GPS en mapa
        Intent intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse(getResources().getString(R.string.coord)));
        startActivity(intent);
```

4.7. Envío de información

- Permiten adjuntar información que pasamos con la *Intent*.
- Funciona de manera muy parecida a un *Map* de Java
  - tiene un id y un valor.
- Hay que pasar los **mínimos datos** posibles mediante la *Intent*, es un proceso muy costoso para el sistema
  - *Intent.putExtra*: método para pasar el conjunto de datos.
  - Clase *Bundle*
    - *getExtras*: método para recoger los datos enviado por *putExtra*.

4.8. Ejemplo

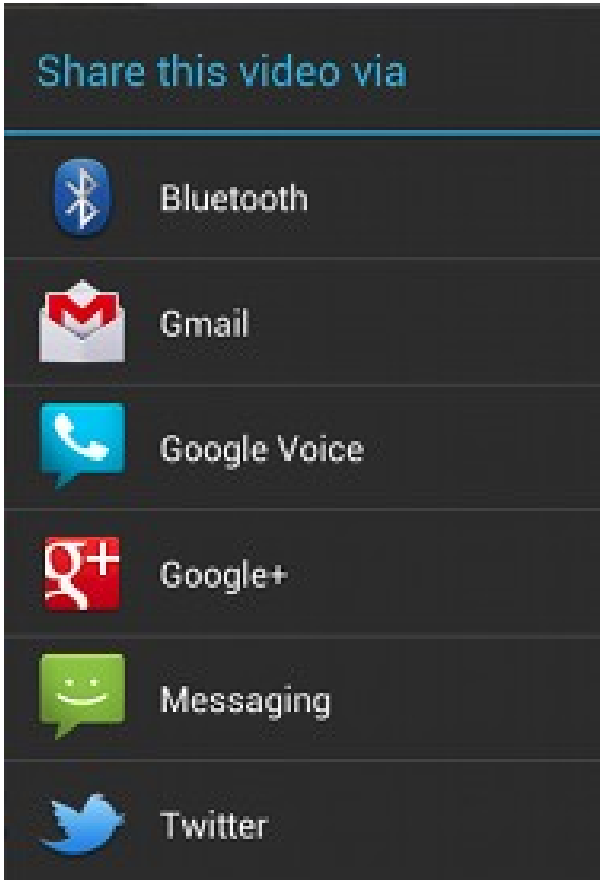

```
//EN LA ACTIVIDAD PRINCIPAL
        Intent intent = new Intent();
        intent.setClass(this, Other_Activity.class);
        intent.putExtra("EXTRA_ID", "SOME DATAS");
        StartActivity(intent);

//EN LA ACTIVIDAD SECUNDARIA (Other_Activity.class)
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                Bundle extras = getIntent().getExtras();
                if (extras != null) {
                        String datas= extras.getString("EXTRA_ID");
                         if (datas!= null) {
                                // do stuff
                        }
                }
        }
```

4.9. PendingIntent

- Es un *Intent* que se manda al sistema, pendiente de una ejecución futura, que desconocemos cuando se producirá.
- Cuando utilizar:
  - Declarar un *Intent* para ser ejecutado cuando el usuario realice una acción con nuestra notificación (*NotificationManager* del sistema Android ejecuta el *Intent*).
  - Declarar un *Intent* para ser ejecutado cuando el usuario realice una acción con nuestro *Widget*.
  - Declarar un *Intent* para ser ejecutado en un momento determinado en el futuro (*AlarmManager* del sistema Android ejecuta el *Intent*).
- No utiliza *startActivity(),* sino:
  - *PendingIntent.getActivity()* para un *Intent* que inicia una Actividad.
  - *PendingIntent.getService()* para un *Intent* que inicia un Servicio.
  - *PendingIntent.getBroadcast()* para un *Intent* que inicia un *BroadcastReceiver*.

4.10. ChooserIntent

- Cuando muestra el diálogo de selección de aplicación a ejecutar el *Intent* externamente a nuestra aplicación.
  - Por ejemplo, cuando queremos compartir un archivo con la acción *ACTION_SEND*.



4.10. Ejemplo


```
Intent sendIntent = new Intent(Intent.ACTION_SEND);
...
// Always use string resources for UI text.
// This says something like "Share this photo with"
String title = getResources().getString(R.string.chooser_title);

// Create intent to show the chooser dialog
Intent chooser = Intent.createChooser(sendIntent, title);

// Verify the original intent will resolve to at least one activity
if (sendIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(chooser);
}
```

5. Procesos en Android

5.1. Definición y características (I)

- Por defecto, todos los componentes de una App son ejecutados en el mismo hilo.
- Existe una jerarquía de procesos que Android gestiona según memoria.
- Tipos de procesos:
  - **Primer plano:** si es verdadera cualquiera de las siguientes condiciones:
    - Método *onResume()* de la actividad ha sido llamado.
    - Anfitrión de un servicio está destinado a la actividad que interactúa con el usuario.
    - Servicio en primer plano mediante el método *startForeground()*.
    - Anfitrión de un *BroadcastReceiver* está ejecutando el método *onReceive()*.

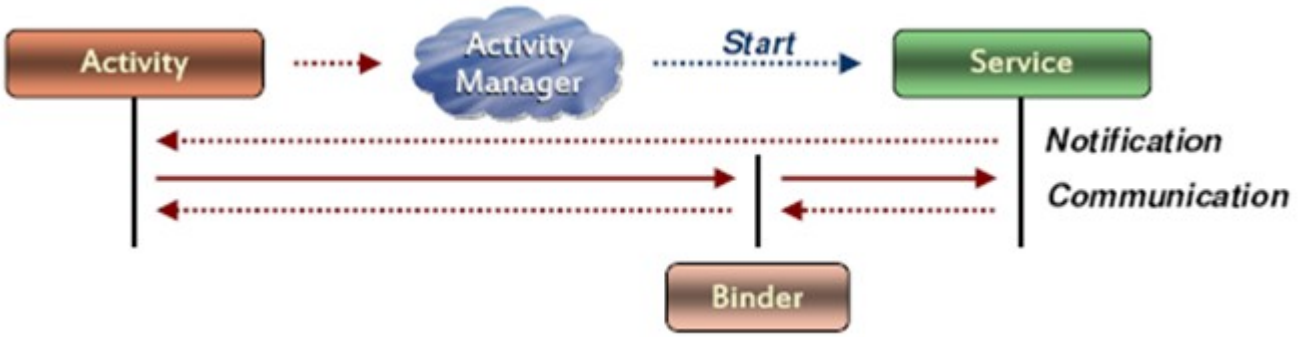
5.1. Definición y características (II)

- **Proceso visible:** no tiene ningún componente en primer plano pero afecta a la visualización del usuario. Condiciones:
  - Es anfitrión de una actividad que no está en primer plano, pero sigue siendo visible para el usuario (su método *onPause()* ha sido llamado).
- **Proceso de servicio:** iniciado con el método *startService()* no cumpliendo ninguna categoría anterior.
  - No están directamente relacionados con lo que el usuario ve.
    - Descarga de datos, copiar elementos…
  - Android mantiene funcionando a menos que no haya memoria.

5.1. Definición y características (II)

- **Proceso en segundo plano:** actividad que no es visible para el usuario, por tanto el método *onStop()* es llamado.
  - No tienen relación directa con el usuario.
  - Pueden ser destruidos en cualquier momento para trasladar memoria a procesos de primer plano.
- **Proceso vacío:** no pertenece a ningún componente de aplicaciones activas.
  - Fines de almacenamiento en caché para mejorar tiempo de inicio la próxima vez que un componente lo ejecute

6. IntentService



6.1. Definición

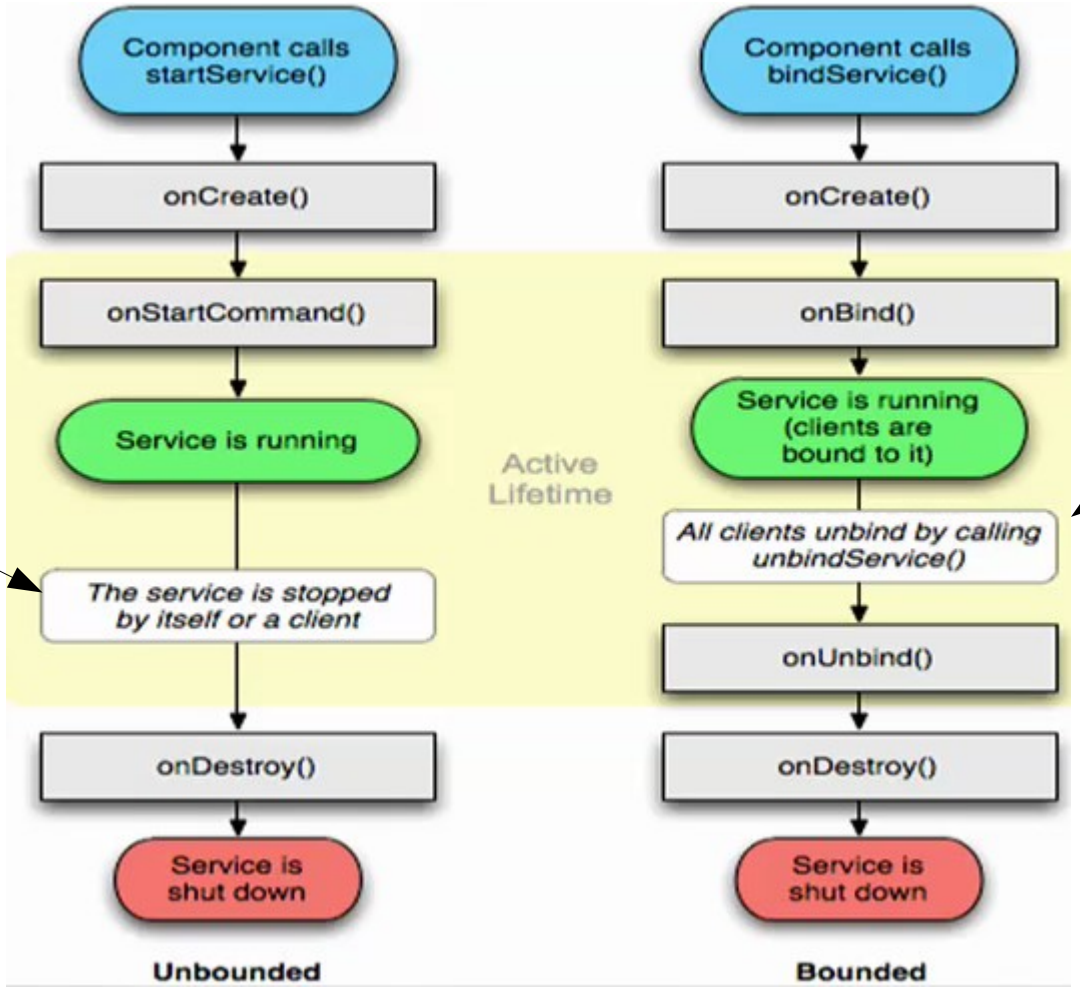
- Componente que funciona sin interactuar con el usuario, en *background*.
  - Puede correr sin que el usuario lo sepa.
- Ejemplo:
  - *Log* de coordenadas *GPS*, Reproductor de música.
- Puede usarse de dos maneras:
  - *Unbound* (Desconectado): Inicio por libre, mediante el método *startService()*.
    - Cuando descargamos una imagen se autodestruye.
  - *Bound*: Unido a una *Activity* mediante *bindService()*: permite a los componentes interactuar con el servicio.
    - Sigue estando operativo mientras el componente que lo inicio siga ejecutándose.
    - El componente que inicia servicios se destruirá cuando estos mueran.

6.2. Aspectos importantes

- El mismo crea y gestiona un hilo de ejecución.
- Se detiene automáticamente una vez concluida la tarea.
- Extiende de la clase *IntentService* y añade su constructor de dicha clase
- Se implementa a través de su método *onHandleIntent()*.
- Utilizado mayormente cuando el segundo hilo no tiene relación con el hilo principal.
- Se suelen utilizar más los Hilos y Tareas.

6.3. Ciclo de vida

Cuando finalice su estado o un cliente lo indique se destruirá



Cuando los clientes se desconectan

6.3.2. Ejemplo (I)

Extiende de la clase padre

```
public class MiIntentService extends IntentService {
    public static final String ACTION_PROGRESO =
        "net.manwest.intent.action.PROGRESO";
    public static final String ACTION_FIN =
        "net.manwest.intent.action.FIN";

    public MiIntentService() {
        super("MiIntentService");
    }
```

Identificadores únicos

Constructor

6.3.2. Ejemplo (I)


```
@Override
   protected void onHandleIntent(Intent intent) {
      int iter = intent.getIntExtra("iteraciones", 0);
       for(int i=1; i<=iter; i++) {
          tareaLarga();
           //Comunicamos el progreso
          Intent bcIntent = new Intent();
          bcIntent.setAction(ACTION_PROGRESO);
          bcIntent.putExtra("progreso", i*10);
          sendBroadcast(bcIntent);
      }
       Intent bcIntent = new Intent();
      bcIntent.setAction(ACTION_FIN);
      sendBroadcast(bcIntent);
   }
```

Método que lanza el hilo secundario

Propiedades de *Intent* para conocer el Nº de iteraciones

Nombre de acción del *Intent*

Añadimos extras de *Intent*

Enviamos en forma de broadcast

Broadcast que indica que la aplicación ha terminado

7. Hilos y Tareas

7.1. Definición

- Permiten la ejecución de tareas en segundo plano.
- Modelo de un solo hilo principal por app (conocido como hilo de interfaz de usuario).
- Se independizan del hilo principal de ejecución (conocidos como hilos de fondo o hilos trabajadores).
- Existen dos tipos de implementación.
  - *ASyncTask:* mediante un sistema de definición de tipo de datos.
  - *Threads*.

7.2. Threads (I)

- Declaración.

```
//Constructor de la Clase Thread
new Thread(new Runnable() {
    public void run() {
        //Aquí ejecutamos nuestras tareas costosas
    }
}).start();
```

Recibe Objeto

Construcción del thread a través del método *run*

7.2. Threads (II)

- ¿Sería correcto?

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            Bitmap mBitmap = loadImageFromNetwork(
                    "http://example.com/image.png");
            mImageView.setImageBitmap(mBitmap);
        }
    }).start();
}
```

7.2. Threads (II)

- No. Dos reglas fundamentales:
  - No bloquee el hilo de interfaz de usuario.
  - No acceder al kit de herramientas de interfaz de usuario desde fuera del hilo de interfaz del usuario.
- ¿Cual regla no cumple?

7.2. Threads (III)

- No acceder al kit de herramientas de interfaz de usuario desde fuera del hilo de interfaz del usuario.
  - Modifica el *ImageView* desde el hilo de trabajo en vez de modificarla desde el hilo de interfaz de usuario.
- Solución: **acceso al hilos desde otros hilos**. Métodos:
  - *Activity.runOnUiThread(Runnable)*: para "enviar" operaciones al hilo principal desde el hilo secundario.
  - *View.post(Runnable)*: para actuar sobre cada control de la interfaz.
  - *View.postDelayed(Runnable, long)*: igual que post pero con un lapso de tiempo.

7.2. Threads (IV)

- ¿Y ahora es correcto?

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap mBitmap = loadImageFromNetwork(
                    "http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(mBitmap);
                }
            });
        }
    }).start();
}
```

7.2. Threads (V)

- Si.
  - el funcionamiento de la red se realiza desde un hilo separado, mientras que el *imageView* se manipula desde el subproceso de la interfaz de usuario.

7.2. Threads (VI)

```
new Thread(new Runnable() {
   public void run() {
      pbarProgreso.post(new Runnable() {
      public void run() {
         pbarProgreso.setProgress(0);
      }
   });

   for(int i=1; i<=10; i++) {
      tareaLarga();
      pbarProgreso.post(new Runnable() {
         public void run() {
            pbarProgreso.incrementProgressBy(10);
            }
         });
   }

   runOnUiThread(new Runnable() {
      public void run() {
         Toast.makeText(MainHilos.this, "Tarea finalizada!",
            Toast.LENGTH_SHORT).show();
      }
   });
   }
}).start();
```

Invocación principal

Método *post* para actuar sobre el control de *ProgressBar*

Método *runOnUiThread()* para mostrar el mensaje TOAST cuando acabe el progreso

7. Handler Class

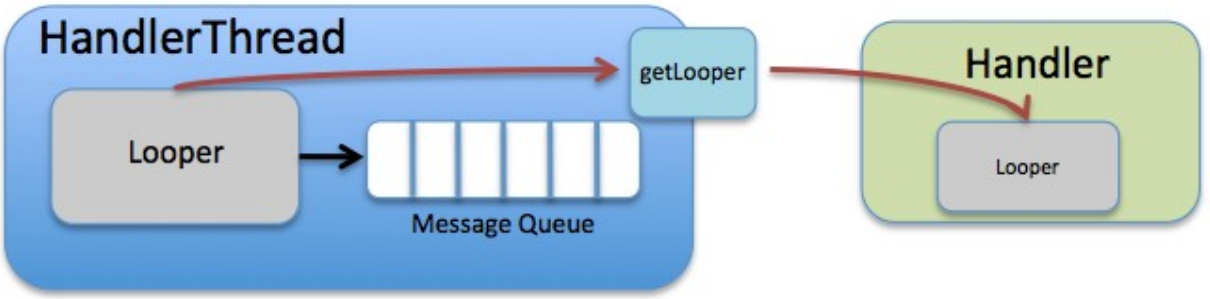


Fig. 4 : Handler Thread Mechanism

8.1. Handler Class (I)

- Clase que nos permite mover datos de un subproceso de fondo al hilo de interfaz de usuario.
- Definir un controlador en el hilo de interfaz de usuario.
- Pasos para la creación
  - Definir el controlador en el hilo de interfaz de usuario.
  - Mover datos de una tarea al hilo de interfaz de usuario.
  - Enviar el estado a la jerarquía de objetos.
  - Mover los datos al hilo interfaz de usuario.

8.2. Definir un controlador en el hilo principal

- Recibe mensajes y ejecuta código para manejarlos.
- Controlador nuevo para cada nuevo hilo.
- Crear una instancia *Handler* en el constructor de la clase que creará sus grupos de subprocesos y almacenará el objeto en una variable global.
- Conectamos al hilo principal con el constructor *Handler(Looper)* que es parte del framework de gestión de hilos en Android.

8.2. Definir un controlador en el hilo principal


```
private PhotoManager() {
   // Definimos un objeto Handler que se une al hilo de interfaz de usuario.
      mHandler = new Handler(Looper.getMainLooper()) {
   ...
/*
 * Dentro del Handler, sobrescribimos el método handleMessage(). El sistema Android invoca
 * a éste método cuando recibe un nuevo mensaje de un hilo; todos los objetos Handler de
 * un hilo reciben el mismo mensaje.
 */
       /*
        * handleMessage() define las operaciones a realizar cuando
        * el Handler recibe un nuevo mensaje para porcesar.
        */
       @Override
       public void handleMessage(Message inputMessage) {
          // Obtenemos la imagen del objeto del mensaje entrante.
          PhotoTask mPhotoTask = (PhotoTask) inputMessage.obj;
          ...
       }
   ...
   }
}
```

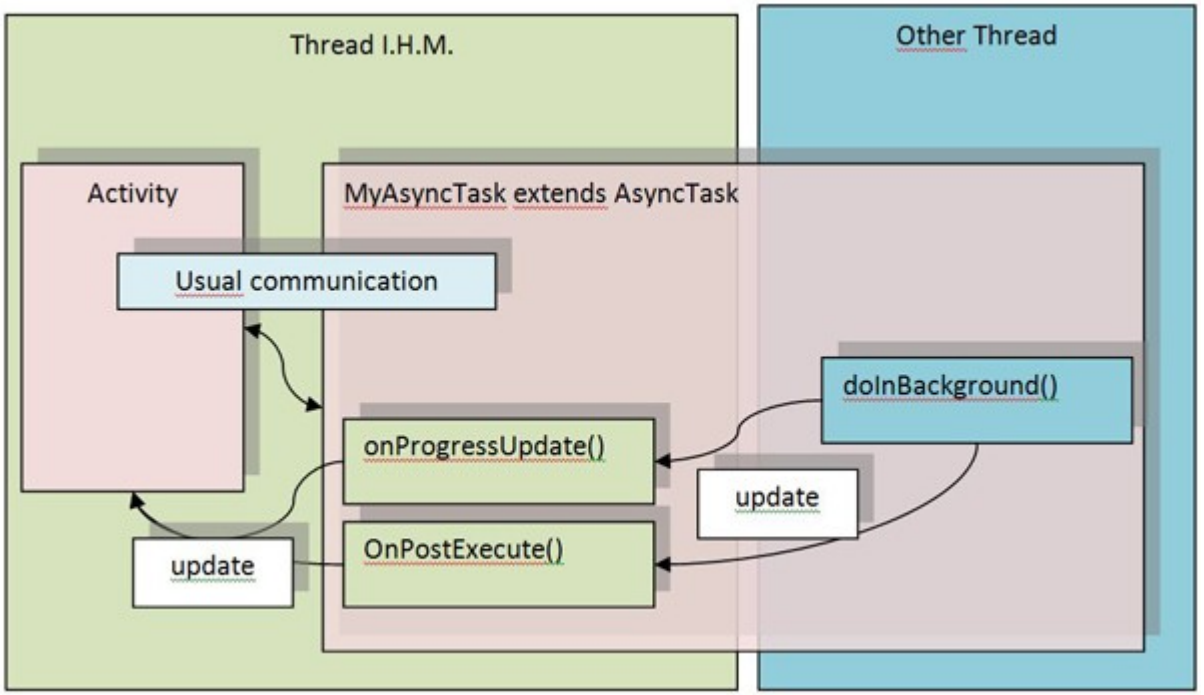
8.3. Mover datos de una tarea al hilo principal

- Comenzar con el almacenamiento a las referencias de los datos y el objeto.
- Luego pasar el objeto de la latera y un código de estado al objeto que crea la instancia del Handler.
- En el ejemplo:
  - Un *Runnable* se ejecuta en un subproceso de fondo y éste decodifica un mapa de bits y lo almacena en el objeto padre *PhotoTask*. El *Runnable* también almacena el código de estado *DECODE_STATE_COMPLETED*.

8.3. Definir un controlador en el hilo principal


```
// Una clase que decodifica archivos de fotos en mapas de bits.
class PhotoDecodeRunnable implements Runnable {
    ...
    PhotoDecodeRunnable(PhotoTask downloadTask) {
        mPhotoTask = downloadTask;
    }
    ...
    // Obtenemos el array de bytes descargados.
    byte[] mImageBuffer = mPhotoTask.getByteBuffer();
    ...
    // Ejecutamos el código para esta tarea.
    public void run() {
        // Intentamos descodificar el búfer de la imágen.
        mReturnBitmap = BitmapFactory.decodeByteArray(
                imageBuffer, 0, imageBuffer.length, bitmapOptions);
        ...
        // Establecemos el Bitmap de la imágen.
        mPhotoTask.setImage(returnBitmap);
        // Informamos del estado "completado".
        mPhotoTask.handleDecodeState(DECODE_STATE_COMPLETED);
        ...
    }
    ...
}
```

9. AsyncTask (I)



9.1. Métodos

- *OnPreExecute*: inicialización de la tarea. Este método se invoca desde el *Thread* principal y puede, por tanto, modificar la interfaz de usuario.
  - *protected void onPreExecute()*
- *doInBackground*: se encarga de realizar el procesamiento de la tarea y se invoca desde el *Thread* secundario. Este método devuelve el resultado del procesamiento.
  - *protected abstract Result doInBackground (Params... params)*
- *publishProgress*: puede ser invocado en cualquier momento por *doInBackgroud* para actualizar la interfaz de usuario según el avance actual del procesamiento. Esta llamada provocará que se invoque el método *onProgressUpdate* desde el *Thread* principal.
  - *protected void onProgressUpdate (Progress... values)*
- *onPostExecute*: método invocado para procesar el resultado. Este método se invoca desde el *Thread* principal y puede, por tanto, modificar la interfaz de usuario.
  - *protected void onPostExecute (Result result)*

9.2. Peculiaridades

- Fijarse que separa la parte trabajo que hace un subproceso (*doInBackground()*) con la parte de interfaz de usuario (*onPostExecute()*).
- Tipos genéricos de *AsyncTask*
  - **Params**: son el tipo de parámetros enviados a la tarea de ejecución.
  - **Progress**: es el progreso de las unidades de progreso publicados durante las tareas realizadas en segundo plano.
  - **Result**: es el resultado de las obtenido por las tareas de que se realizan en segundo plano.

9.2. Ejemplo

```
public void onClick(View v) {
    new DownloadImageTask().execute(
            "http://example.com/image.png");
}
private class DownloadImageTask extends AsyncTask<string, void="" ,="" bitmap=""> {
    /** El sistema llama a éste método para realizar el trabajo
        * en un subproceso de trabajo
      * y realiza la entrega los parámetros dados a AsyncTask.execute() */
    protected Bitmap doInBackground(String... urls) {
        return loadImageFromNetwork(urls[0]);
    }

    /** El sistema llama a éste método para realizar el trabajo en
        * el hilo de interfaz de usuario
      * y entrega el resultado del método doInBackground() */
    protected void onPostExecute(Bitmap result) {
        mImageView.setImageBitmap(result);
    }
}
```

9.3. Ejemplo (I)


```
private class MiTareaAsincrona extends AsyncTask<Void, Integer, Boolean> {

   @Override
   protected Boolean doInBackground(Void... params) {
       for(int i=1; i<=10; i++) {
          tareaLarga();
          publishProgress(i*10);
          if(isCancelled())
             break;
       }
       return true;
   }

   @Override
   protected void onProgressUpdate(Integer... values) {
       int progreso = values[0].intValue();
       pbarProgreso.setProgress(progreso);
   }
```

Es una clase!!!!

Método que lanza el hilo secundario

Notifica al hilo de actualización

9.3. Ejemplo (II)

```
@Override
   protected void onPreExecute() {
      pbarProgreso.setMax(100);
      pbarProgreso.setProgress(0);
   }
    @Override
   protected void onPostExecute(Boolean result) {
      if(result)
         Toast.makeText(MainHilos.this, "Tarea finalizada!",
               Toast.LENGTH_SHORT).show();
   }
    @Override
   protected void onCancelled() {
      Toast.makeText(MainHilos.this, "Tarea cancelada!",
            Toast.LENGTH_SHORT).show();
   }
}
```

Antes de lanzar el hilo secundario

Cuando acaba el hilo secundario

En caso de no finalización de la tarea

10. Notificaciones

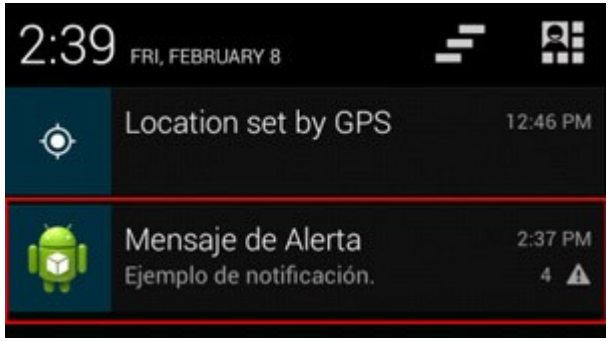
(*Notify*)

10.1. Definición

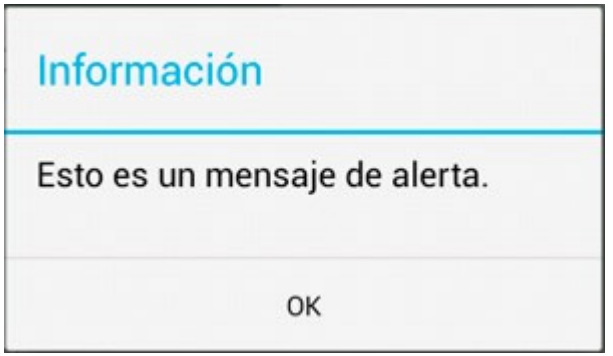
- Permite generar alertas para que el usuario sepa que es lo que está pasando en el sistema o en una aplicación.
- Las notificaciones son visibles desde la barra de notificaciones o desde dispositivos que estén enganchados contra ellas (Android Wear)
- Se pueden generar desde cualquier parte de la aplicación.
- Pueden contener imágenes y ser personalizadas.
- 1. Hay varios tipos de notificaciones: En Toast, Barra de notificaciones, Diálogos y Snack Bar.
- NO ABUSAR.

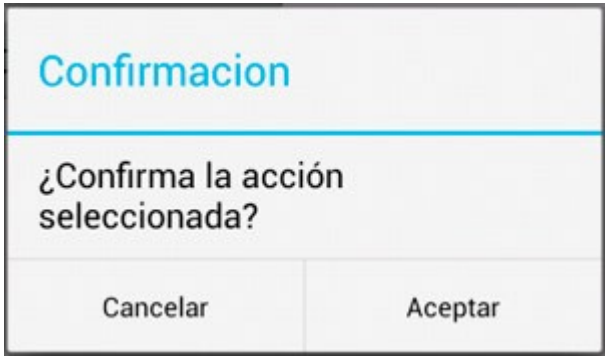
10.2. Tipos

Barra de estado

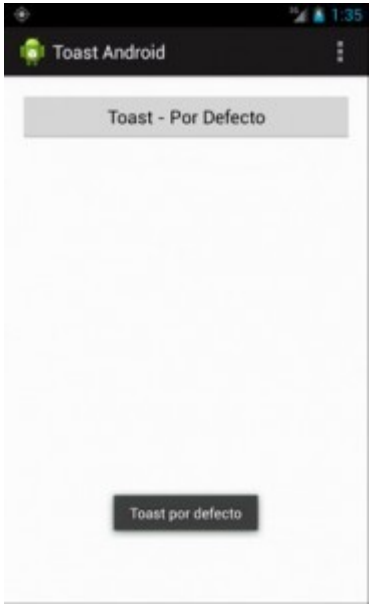


Diálogos

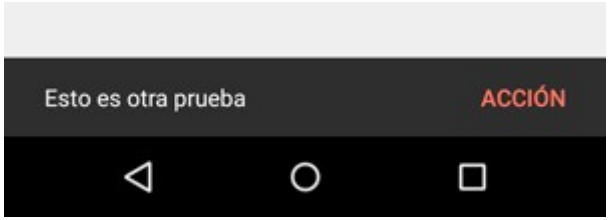




Toast



SnackBar



10.3.1. Toast por defecto

Evento onClick de Button

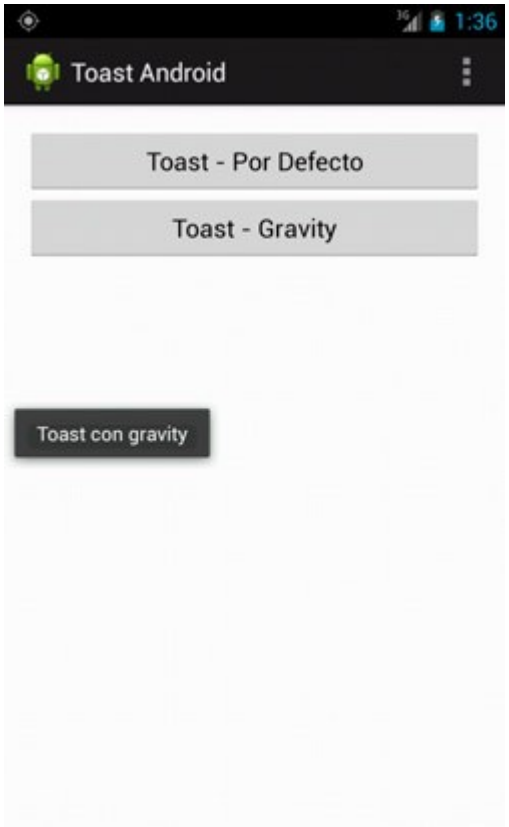
```
btnToast.setOnClickListener(new OnClickListener() {
   @Override
   public void onClick(View arg0) {
      Toast toast =
         Toast.makeText(getApplicationContext(),
               "Toast ejemplo", Toast.LENGTH_SHORT);

      toast.show();
   }
});
```

Lanza notificación *Toast*

10.3.2. Toast con gravedad

```
btnToast.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View arg0) {
        Toast toast =
            Toast.makeText(getApplicationContext(),
                    "Toast con gravedad", Toast.LENGTH_SHORT);

        toast.setGravity(Gravity.CENTER|Gravity.LEFT,0,0);

        toast.show();
    }
});
```

Zona en la que se quiere que aparezca: CENTER/ LEFT/ RIGHT/ BOTTOM



10.4.1. Barra de estado. Declaración


```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(MainActivity.this)
        .setSmallIcon(android.R.drawable.stat_sys_warning)
        .setLargeIcon((((BitmapDrawable)getResources()
            .getDrawable(R.drawable.ic_launcher)).getBitmap()))
        .setContentTitle("Mensaje de Alerta")
        .setContentText("Ejemplo de notificación.")
        .setContentInfo("4")
        .setTicker("Alerta!");
```

Creamos el objeto pasándo el objeto de la aplicación

Icono de la izquierda, derecha y barra de estado

Título y texto de la notificación

Texto que aparece a la izquierda del icono pequeño

Texto que aparece por unos segundos en la barra de estado

10.4.2. Barra de estado. Actividad asociada


```
Intent notIntent =
    new Intent(MainActivity.this, MainActivity.class);

PendingIntent contIntent =  PendingIntent.getActivity(
        MainActivity.this, 0, notIntent, 0);

mBuilder.setContentIntent(contIntent);
```

Objeto *Intent* que indica la clase de la actividad a lanzar. Utilizado para construir el *PendingIntent*

Objeto *PendingIntent* que contiene la información de la actividad asociada a la notificación

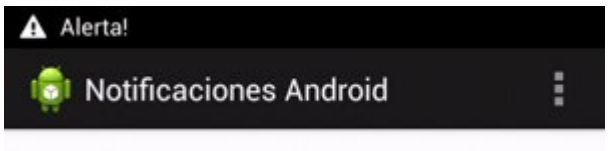
Asocia el objeto a la notificación mediante el método *setContentIntent* de *Builder*

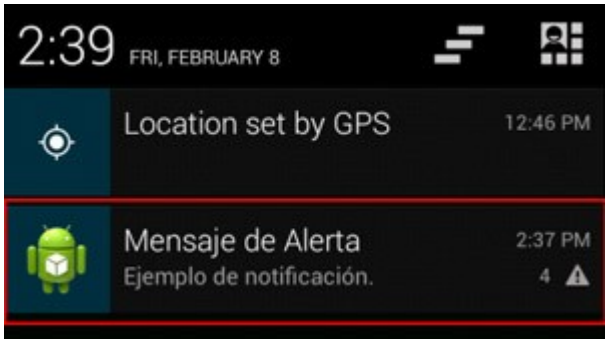
10.4.2. Barra de estado. Llamada

Genera la notificación mediante la clase *NotificationManager* y llamando al método *getSystemService*

```
NotificationManager mNotificationManager = (NotificationManager)
    getSystemService(Context.NOTIFICATION_SERVICE);

mNotificationManager.notify(NOTIF_ALERTA_ID, mBuilder.build());
```

Llamada que lanza la notificación





10.5.1. Diálogo de Alerta. Construcción

Construido a través de *Builder* igual que Notificaciones

```
public class DialogoAlerta extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage("Esto es un mensaje de alerta")
               .setTitle("Alerta")
               .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                   public void onClick(DialogInterface dialog, int id) {
                       dialog.cancel();
                   }
               });
        return builder.create();
    }
}
```

Tipo de diálogo y sus opciones

Evento que al hacer *onClick* en “SI” cierra el diálogo

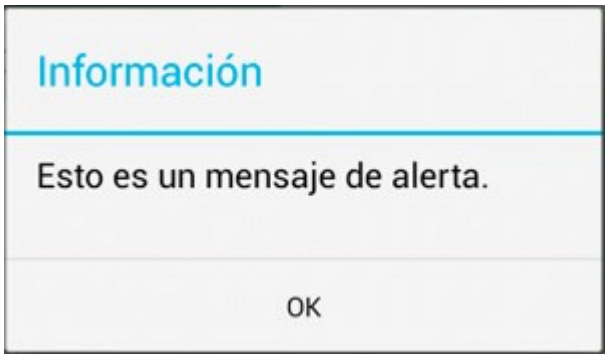
10.5.1. Diálogo de Alerta. Llamada

Llamada al diálogo de alerta en clase principal

```
btnAlerta.setOnClickListener(new View.OnClickListener() {
   public void onClick(View v) {
      FragmentManager fragmentManager = getSupportFragmentManager();
        DialogoAlerta dialogo = new DialogoAlerta();
      dialogo.show(fragmentManager, "tagAlerta");
   }
});
```

Creamos el objeto tipo *DialogAlerta* mediante la clase *FragmentManager*

Método *show* que muestra el diáologo



10.5.2. Diálogo de confirmación. Construcción (I)

Igual que el anterior

Que muestra el Diálogo

```
public class DialogoConfirmacion extends DialogFragment {
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setMessage("¿Confirma la acción seleccionada?")
        .setTitle("Confirmacion")
        .setPositiveButton("Aceptar", new DialogInterface.OnClickListener()  {
            public void onClick(DialogInterface dialog, int id) {
                Log.i("Dialogos", "Confirmacion Aceptada.");
                    dialog.cancel();
                }
            })
```

Evento en el caso de que pulse en el *Button* de Confirmar

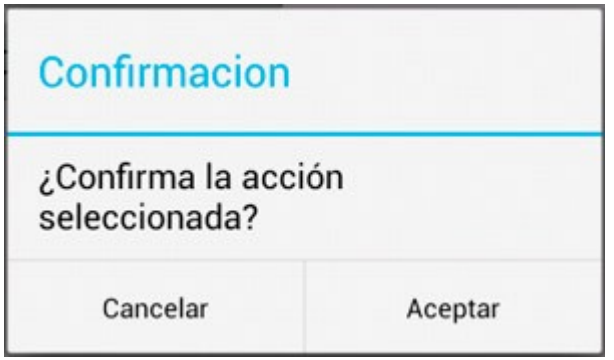
10.5.2. Diálogo de confirmación. Construcción (II)

```
    .setNegativeButton("Cancelar", new DialogInterface.OnClickListener() {
        public void onClick(DialogInterface dialog, int id) {
            Log.i("Dialogos", "Confirmacion Cancelada.");
            dialog.cancel();
        }
    });
    return builder.create();
  }
}
```

En caso de cancelar

La llamada es igual al diálogo anterior

Hay más tipos: Selección, personalizados



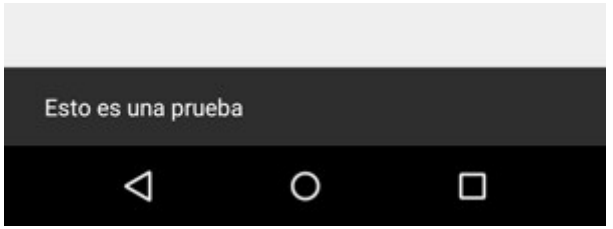
10.6.1. Snackbar Simple

Igual que Toast

```
btnSnackbarSimple.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Snackbar.make(view, "Esto es una prueba", Snackbar.LENGTH_LONG)
                .show();
    }
});
```

Contenedor tipo *view* donde se visualizará

Tiempo en el que está en pantalla

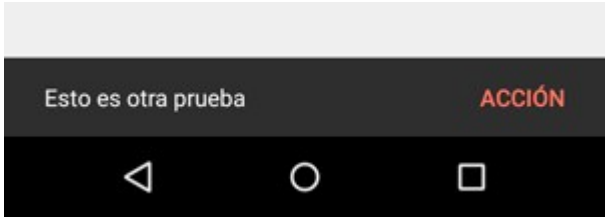


10.6.2. Snackbar con acción

```
Snackbar.make(view, "Esto es otra prueba", Snackbar.LENGTH_LONG)
    .setActionTextColor(getResources().getColor(R.color.snackbar_action))
    .setAction("Acción", new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Log.i("Snackbar", "Pulsada acción snackbar!");
        }
    })
    .show();
```

Al pulsar en "ACCIÓN"

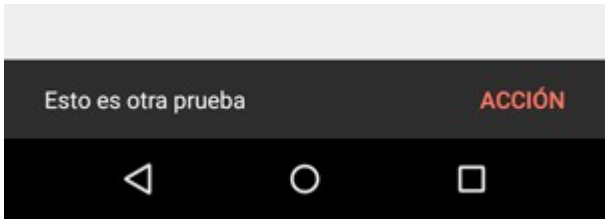
¿Que Significa el método *setActionTextColor*? ¿De dónde obtiene el color y cómo se definiría?



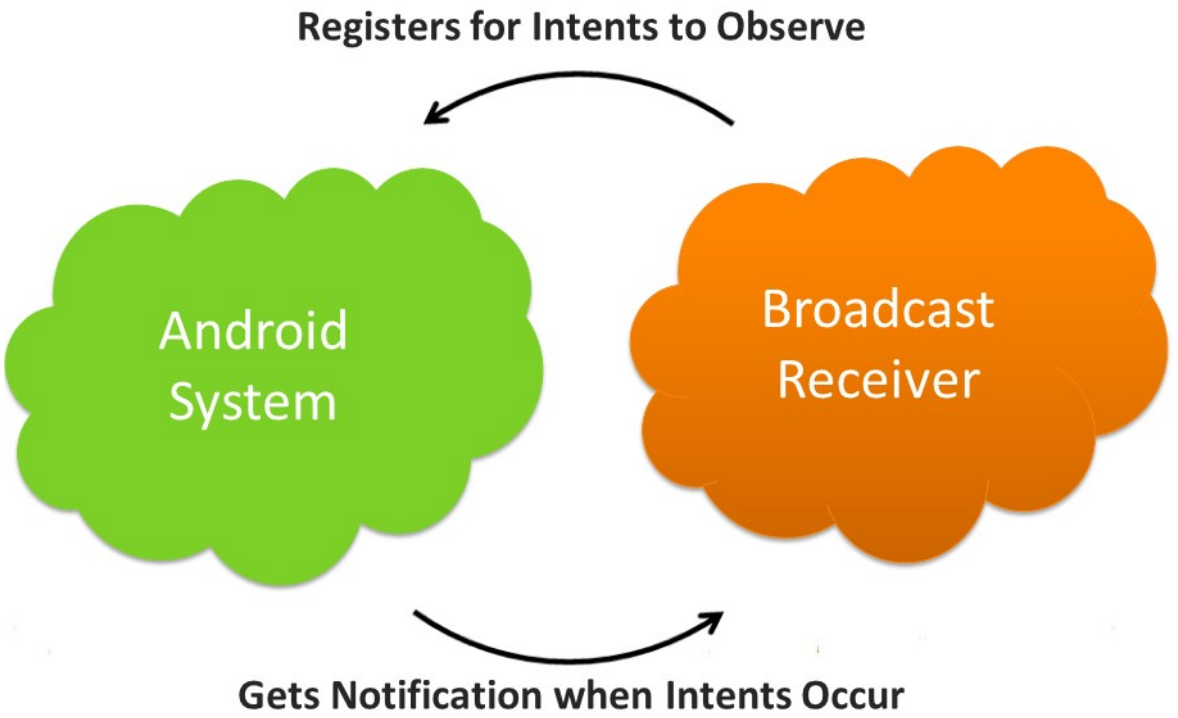
10.6.2. Snackbar con acción

Del fichero */res/values.colors.xml*

```
<resources>
    <color name="snackbar_action">#ffff7663</color>
</resources>
```



11. Receptores de anuncios (BroadcastReceiver)



11.1. Definición

- Permite registrar ante eventos de sistema operativo y de las aplicaciones.
- Se pueden registrar con el Android Manifest o mediante el método *onReceive()*.
- El método principal es *onReceive*, ejecutado por el hilo principal y solo existe durante esta llamada.
- Utilizado principalmente para notificaciones internas de Android:
  - Controlar distintos estados de la Batería baja, llamada entrante, arranque, etc..
- Pueden ser nativos (de sistema) o propios.
- Utilizar para operaciones cortas. Operaciones largas en hilos secundarios.
  - Hay que tener en cuenta que se ejecuta en el hilo principal.

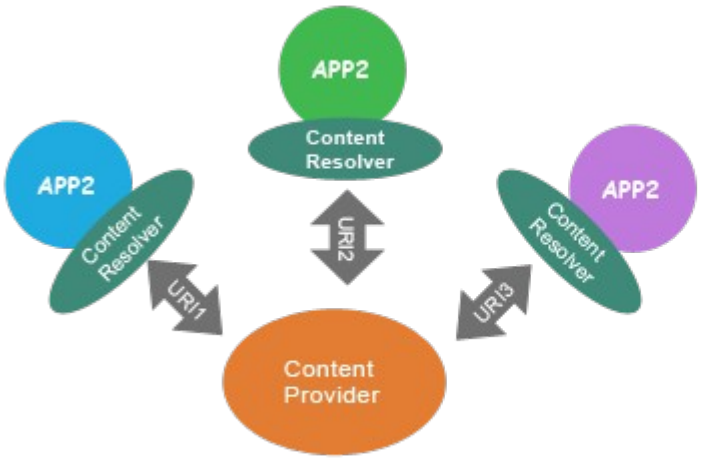
11.2. Formas de registro

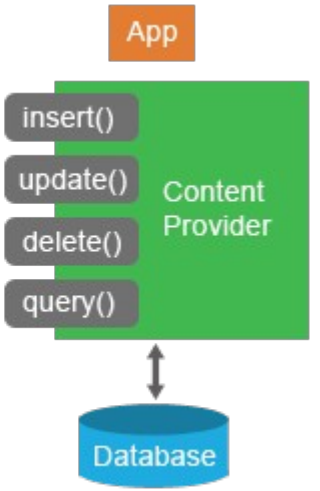
- Estáticamente en *AndoridManifest.xml*. Añadiendo las etiquetas:
  - <*receiver*>*:* indica el nombre del receptor
  - <*intent-filter*>: especifica acciones, datos y categorías.
- Dinámicamente llamando al método *onReceiver()*.
  - Creamos un *IntentFilter*.
  - Creamos un *BroadcastReceiver*.
  - Registramos el *BroadcastReceiver* usando el método *registerReceiver()*.
    - Usamos la clase *LocalBroadcastManager*. Es solo para el uso de la aplicación que estemos desarrollando.
    - Usamos la clase *Context*. Puede ser recibida por otra aplicación de nuestro dispositivo.
    - Llamamos al método un *RegisterReceiver()* para anular el registro.

11.2. Declaración

```
//EN MANIFEST
<receiver android:narne= ".ReceptorAnuncio”>
        <intent-filter>
                <action android:name="android.intent, BATTERY_ LOW " / >
        </intent-f i1ter>
</receiver>

//CLASE JAVA
public class ReceptorAnuncio extends BroadcastReceiver{
@Override
        public void onReceive{Context context, Intent intent)
        {
                //CODIGO OMITIDO
        }
}
```

12. Proveedores de contendio (Content Provider)





12.1. Definición

- Permite la comunicación entre aplicaciones en Android.
- Normalmente funciona como un acceso a base de datos remoto.
- Tenemos dos componentes:
  - La aplicación que tiene los datos.
  - La aplicación que accede a los datos.
    - Un ejemplo es la agenda de contactos.
    - Cualquier aplicación puede acceder a ella si se tienen los permisos oportunos.
- Se debe acceder mediante una *URI* (ver *standard RFC 3296*).
  - *<standard_prefix>://<authority>/<data_path>/<id>*
- Para comenzar, debemos obtener una referencia a un *ContentResolver*, objeto a través del que realizaremos todas las acciones necesarias sobre el content provider. Esto es tan fácil como utilizar el método *getContentResolver*()

12.2. Más importantes

Clase	Información almacenada	Ejemplos de URIs
Browser	Enlaces favoritos, historial de navegación, historial de búsquedas	browser/bookmarks
CallLog	Llamadas entrantes, salientes y pérdidas.	content://call_log/calls
ContactsContract	Lista de contactos del usuario.	content://contacts/people
MediaStore	Ficheros de audio, vídeo e imágenes, almacenados en dispositivos de almacenamiento internos y externos.	content://media/internal/images content://media/external/video content://media/*/audio
Setting	Preferencias del sistema.	content://settings/system/ringtone content://settings/system/notification_sound
UserDictionary (a partir de 1.5)	Palabras definidas por el usuario, utilizadas en los métodos de entrada predictivos.	content://user_dictionary/words
Telephony (a partir de 1.5)	Mensajes SMS y MMS mandados o recibidos desde el teléfono.	content://sms content://sms/inbox content://sms/sent content://mms
Calendar (a partir de 4.0)	Permite consultar y editar los eventos	content://com.android.calendar/time content://com.android.calendar/events

12.3. Introducción a SQLite

- Para comenzar, debemos obtener una referencia a un *Content Resolver*, objeto a través del que realizaremos todas las acciones necesarias sobre el *content provider*.
  - Método *getContentResolver()*
- Para hacer la consulta
  - Cursor cur = cr.query(clientesUri,
    null, //Columnas a devolver
    null, //Condición de la query
    null, //Argumentos variables de la query
    null); //Orden de los resultadosCursor
- Extraer de la BBDD:
  - String id = cur.getString(cur.getColumnIndex(ContactsContract.Contacts._ID));
- Insertar en la BBDD: cr.insert(ClientesProvider.CONTENT_URI, values);
- Eliminar de la BBDD:
  - cr.delete(ClientesProvider.CONTENT_URI,Clientes.COL_NOMBRE + " = 'ClienteN'", null);
- Cerrar nuestro cursos: pCur.close();