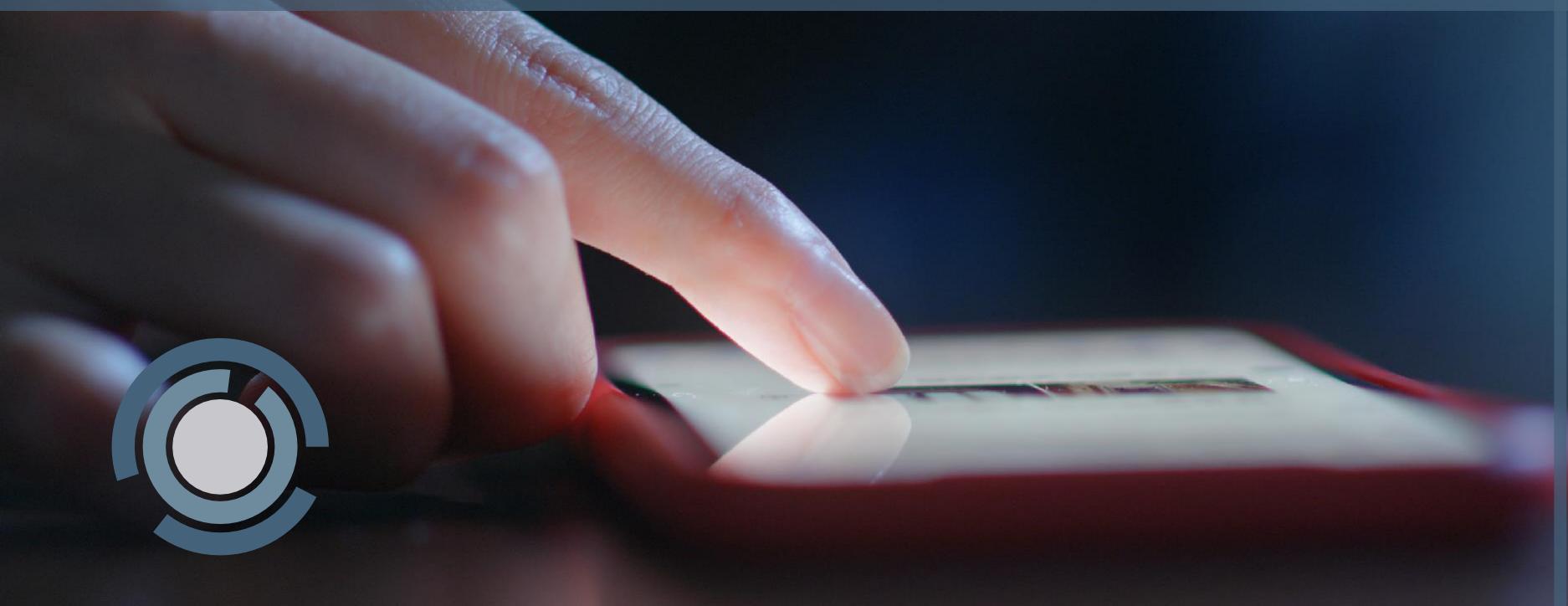


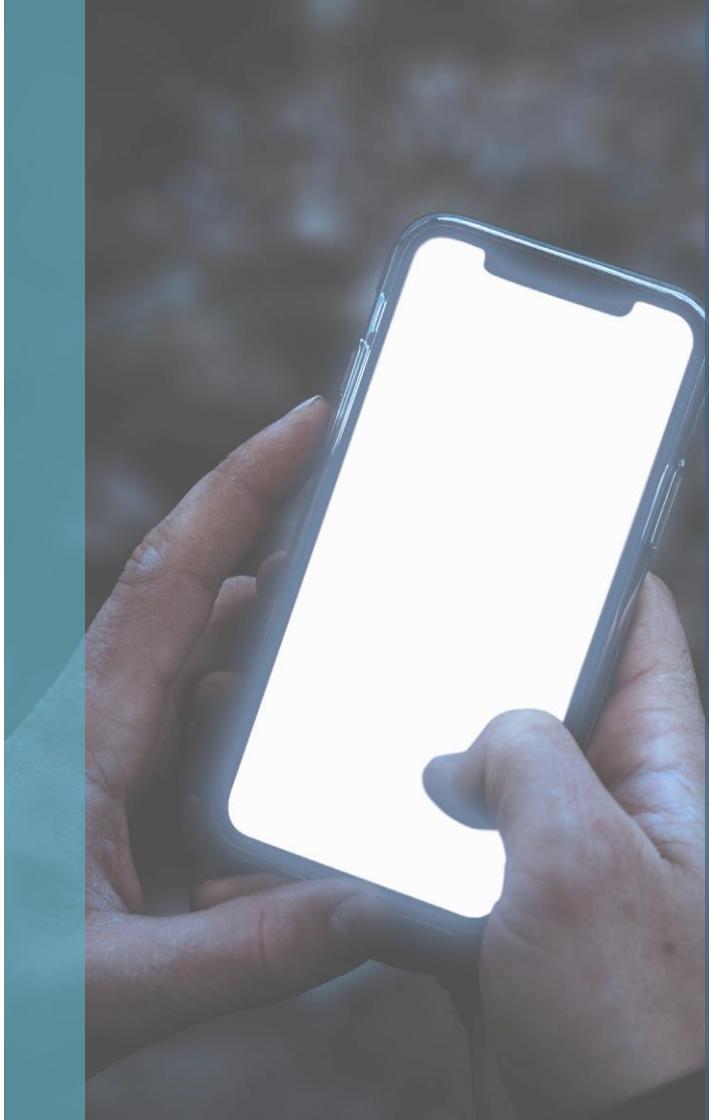
# DESARROLLO DE APLICACIONES MÓVILES II



# Bienvenida

Bienvenido(a) a la asignatura *Desarrollo de Aplicaciones Móviles II*, con la cual comprenderás la importancia del desarrollo de apps, así como el lugar que estas ocupan en el mundo actual. Además, conocerás el proceso para desarrollarlas, desde su diseño hasta la experiencia del usuario final.

Como sabemos, el acceso móvil es la forma como la gente interactúa hoy en día con el mundo; por ello, los usuarios tienen expectativas muy altas en cuanto a las aplicaciones móviles, como tener a su alcance las compras de productos y servicios, o el entretenimiento.



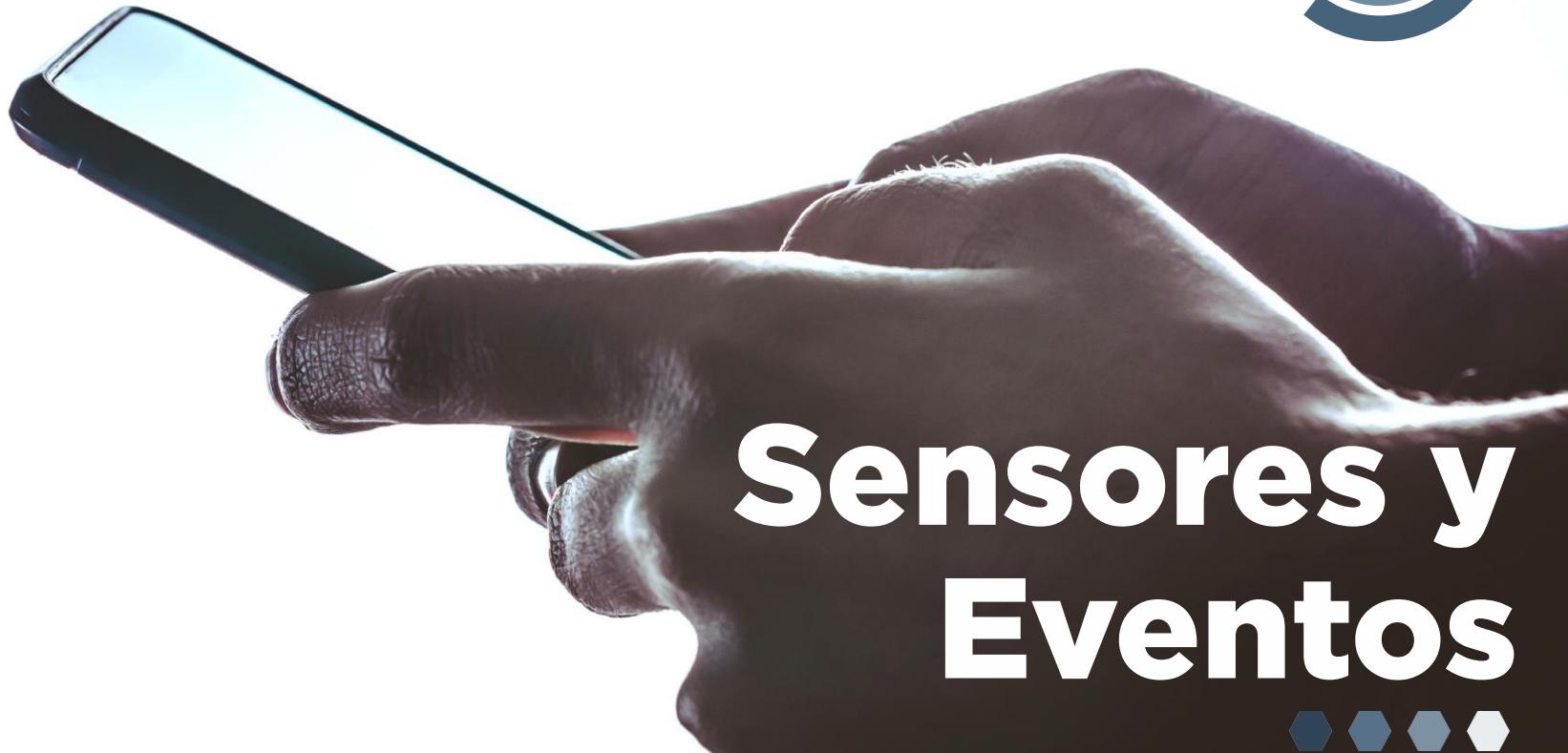
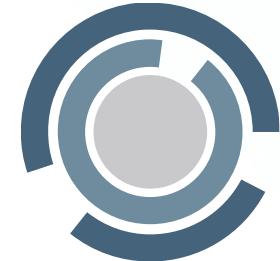


# Libros Recomendados

- Calvo, A. (2015). *Beginning Android Wearables*. Appress.
- Westfall, J., Augusto, R. & Allen, G. (2016) *Beginning Android Web Apps Development: Develop for Android using HTML5, CSS3, and JavaScript*. Appress.



# Unidad 1



# Temario Unidad 1

**1.1**

Tipos de Sensores

**1.2**

Lógica de Eventos





# Introducción



La asignatura *Desarrollo de Aplicaciones Móviles* // tiene como objetivo que el alumno sea capaz de escribir aplicaciones con una GUI simple, el uso de *widgets* integrados y componentes esenciales, así como el trabajo con ficheros para almacenar datos localmente, entre otras características.

En esta primera unidad aprenderás el funcionamiento de los sensores en Android, así como su implementación en la aplicación.

# Competencias a Desarrollar



El alumno será capaz de identificar las características del sensor de pantalla táctil.



El alumno será capaz de conocer el uso de los sensores del acelerómetro y del giroscopio.

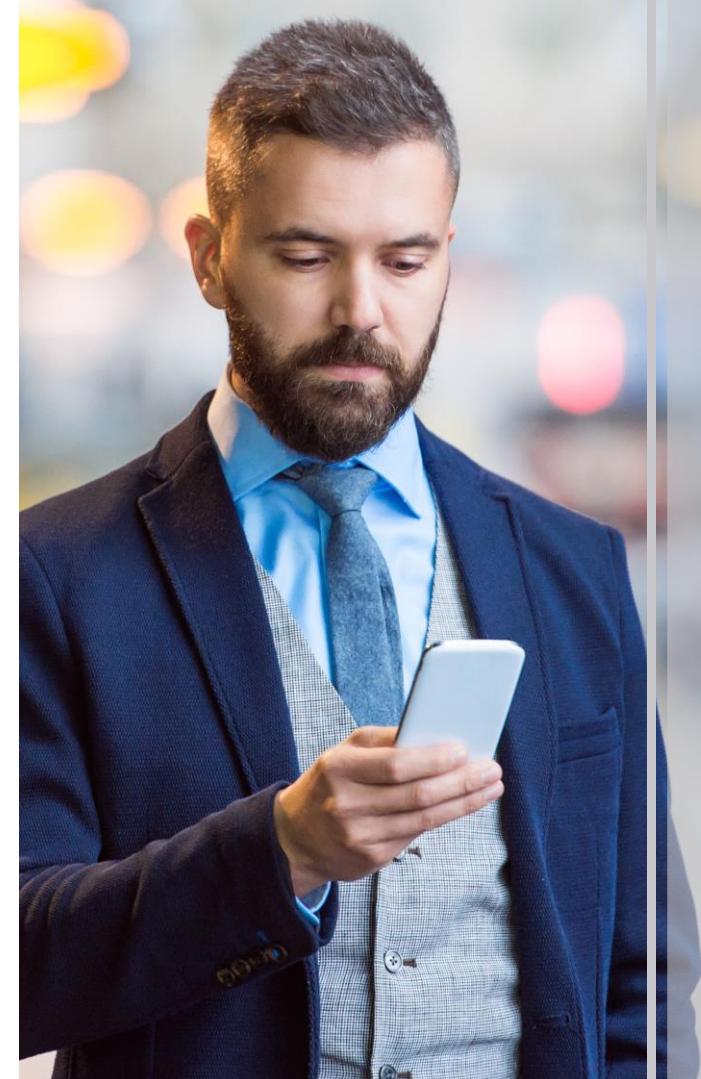


# 1.1. Tipos de sensores

## Pantalla táctil

La mayoría de aplicaciones hacen uso de la pantalla táctil, bien utilizando algún puntero o directamente con los dedos. En ambos casos, deberemos reconocer los eventos de pulsación sobre la pantalla.

Un **gesto** es un movimiento que hace el usuario en la pantalla. El gesto comienza cuando el dedo hace contacto con la pantalla, y finaliza cuando levantamos el dedo de la pantalla.



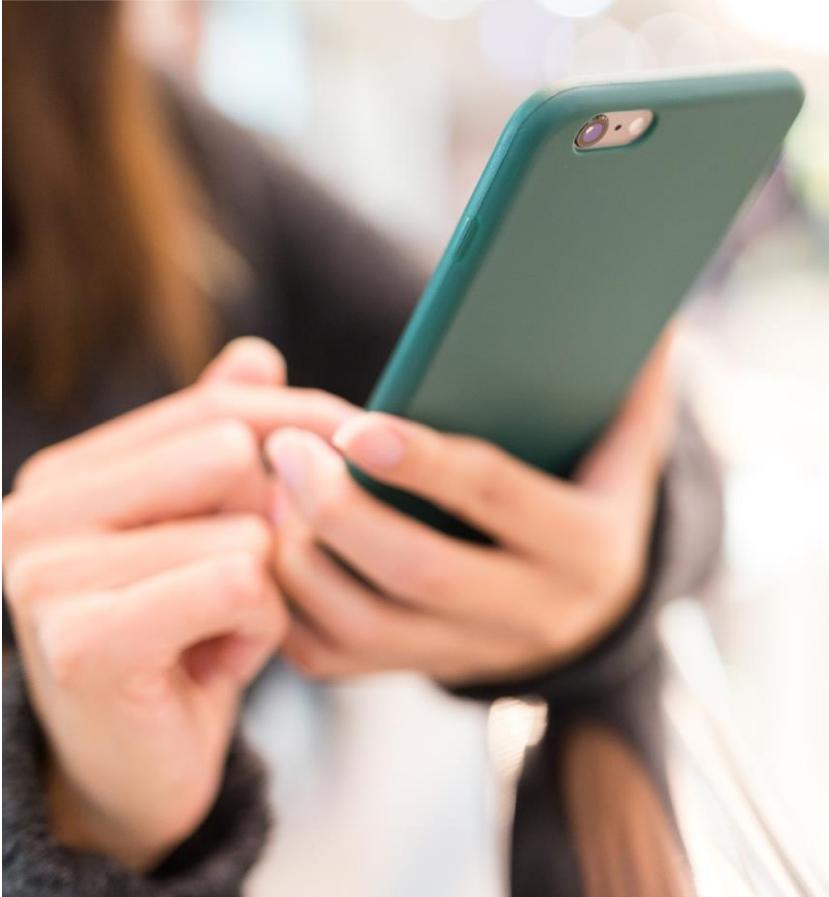
## 1.1. Tipos de sensores

Al crear un componente propio a bajo nivel, deberemos tratar los eventos de la pantalla táctil. Para hacer esto deberemos capturar el evento **OnTouch**, mediante un objeto que implemente la interfaz **OnTouchListener**. De forma alternativa, si estamos creando un componente propio heredando de la clase *View*, podemos capturar el evento simplemente sobrescribiendo el método *onTouchEvent*:

```
public class MiComponente extends View  
...  
@Override  
public boolean onTouchEvent(MotionEvent event) {  
    // Procesar evento return true;  
}  
...
```



## 1.1. Tipos de sensores



El método devolverá *true* si ha consumido el evento y, por lo tanto, no es necesario propagarlo a otros componentes; o *false*, en caso contrario. Del evento de movimiento recibido (***MotionEvent***) destacamos la siguiente información:

- **Acción realizada.** Indica si el evento se ha debido a que el dedo se ha puesto en la pantalla (*ACTION\_DOWN*), se ha movido (*ACTION\_MOVE*), o se ha retirado de la pantalla (*ACTION\_UP*).



## 1.1. Tipos de sensores

También existe la acción *ACTION\_CANCEL*, que se produce cuando se cancela el gesto que está haciendo el usuario.

- **Coordenadas.** Posición del componente en la que se ha tocado o a la que nos hemos desplazado. Podemos obtener esta información con los métodos *getX* y *getY*.



## 1.1. Tipos de sensores

Con esta información, podemos, por ejemplo, mover un objeto a la posición a la que desplacemos el dedo:

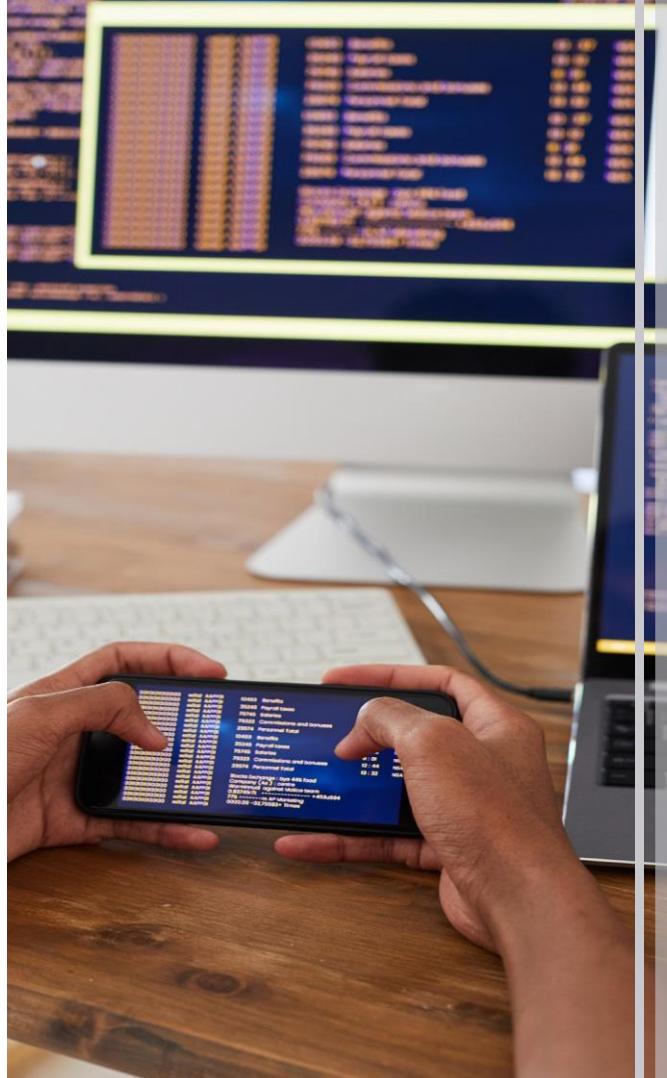
```
@Override  
public boolean onTouchEvent(MotionEvent event) {  
    if(event.getAction() == MotionEvent.ACTION_MOVE) {  
        x = event.getX();  
        y = event.getY();  
        this.invalidate();  
    }  
    return true;  
}
```

**Nota.** Después de cambiar la posición en la que se dibujará un gráfico, es necesario llamar a *invalidate* para indicar que el contenido del componente ya no es válido, y debe ser redibujado (llamando al método *onDraw*).

## 1.1. Tipos de sensores

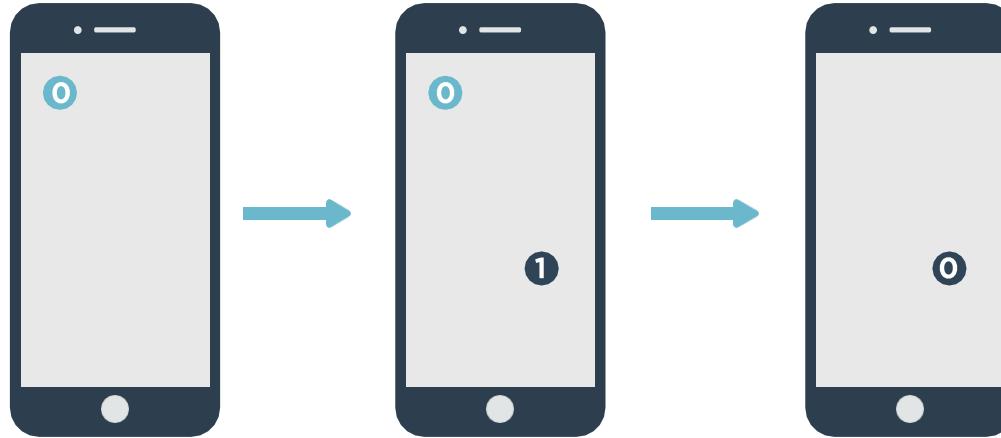
La capacidad **multitouch** se implementa en Android mediante la inclusión de múltiples punteros en la clase ***MotionEvent***. Podemos saber cuántos punteros hay simultáneamente en pantalla llamando al método ***getPointerCount*** de dicha clase.

Cada puntero tiene un índice y un identificador. Si actualmente hay varios puntos de contacto en pantalla, el objeto *MotionEvent* contendrá una lista con varios punteros, y cada uno de ellos estará en un índice de esta lista.



## 1.1. Tipos de sensores

Por ejemplo, si el puntero con índice “0” se levanta de la pantalla, el que tenía índice “1” pasará a tener índice “0”. Esto nos complica el poder realizar el seguimiento de los eventos de un mismo gesto, ya que el índice asociado a su puntero puede cambiar en sucesivas llamadas a `onTouchEvent`. Por este motivo, cada puntero tiene además asignado un identificador que permanecerá invariante y que nos permitirá realizar dicho seguimiento.



## 1.1. Tipos de sensores



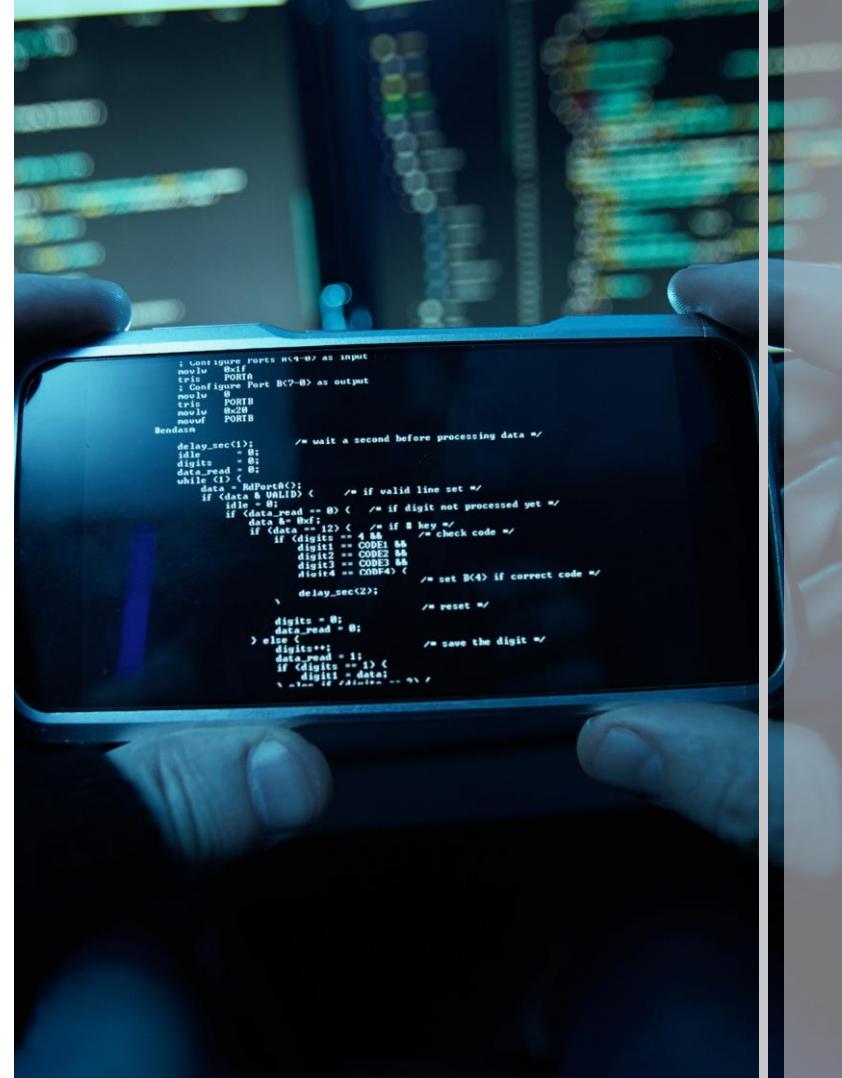
El identificador nos permite hacer el seguimiento, pero para obtener la información del puntero necesitamos conocer el índice en el que está actualmente. Para ello tenemos el método ***findPointerIndex(id)***, que nos devuelve el índice en el que se encuentra un puntero dado su identificador. Para conocer por primera vez el identificador de un determinado puntero, podemos utilizar ***getPointerId(indice)***, que nos devuelve el identificador del puntero que se encuentra en el índice indicado.



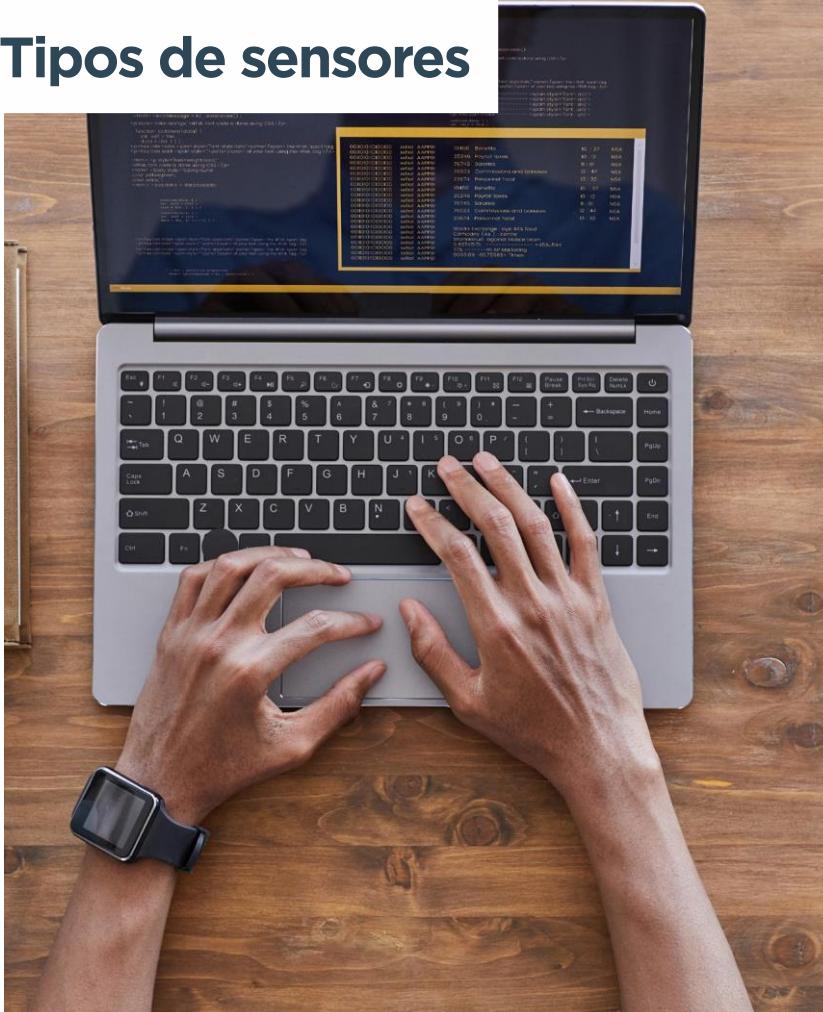
## 1.1. Tipos de sensores

Además, se introducen dos nuevos tipos de acciones:

- **ACTION\_POINTER\_DOWN.** Se produce cuando entra un nuevo puntero en la pantalla, habiendo ya uno previamente pulsado. Cuando entra un puntero sin haber ninguno pulsado se produce la acción *ACTION\_DOWN*.
- **ACTION\_POINTER\_UP.** Se produce cuando se levanta un puntero de la pantalla, pero sigue quedan alguno pulsado. Cuando se levante el último de ellos se producirá la acción *ACTION\_UP*.



## 1.1. Tipos de sensores



Con la clase ***GestureDetector*** podemos detectar varios gestos simples de un solo puntero:

- ***onSingleTapUp***. Se produce al dar un toque a la pantalla; es decir, pulsar y levantar el dedo. El evento se produce tras levantar el dedo.
- ***onDoubleTap***. Se produce cuando se da un doble toque a la pantalla. Se pulsa y se suelta dos veces seguidas.

## 1.1. Tipos de sensores

- **onSingleTapConfirmed.** Se produce después de dar un toque a la pantalla, y cuando se confirma que no le sucede un segundo toque.
- **onLongPress.** Se produce cuando se mantiene pulsado el dedo en la pantalla durante un tiempo largo.
- **onScroll.** Se produce al arrastrar el dedo para realizar *scroll*. Nos proporciona la distancia que hemos arrastrado en cada eje.
- **onFling.** Se produce cuando ocurre un lanzamiento. Esto consiste en pulsar, arrastrar y soltar.



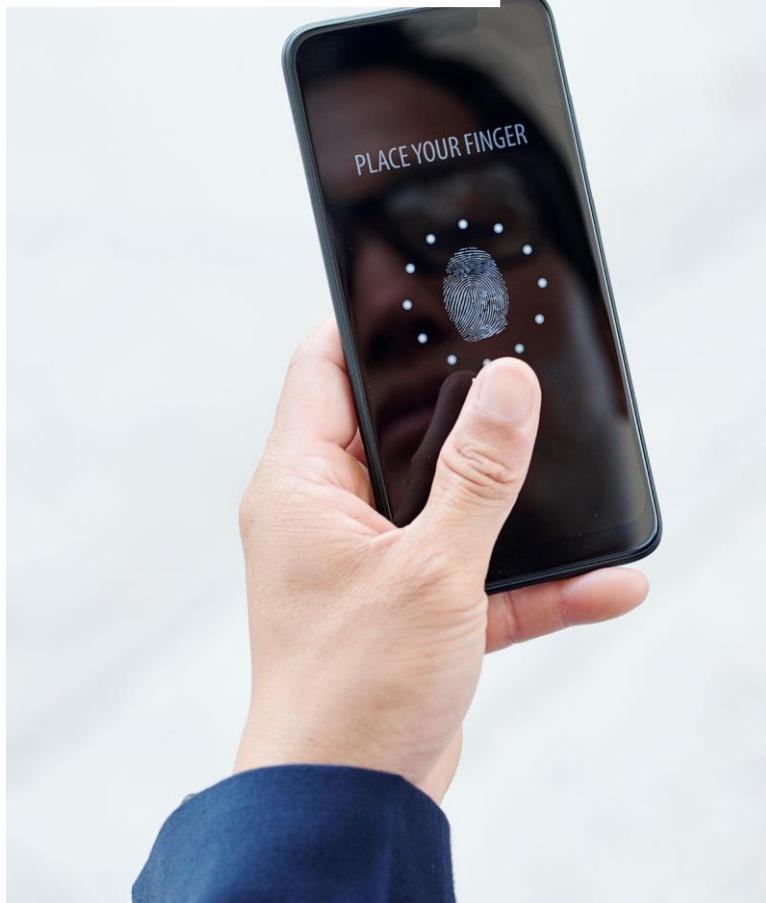
## 1.1. Tipos de sensores

Es importante definir el evento *onDown* en el detector, ya que si no devolvemos *true* en dicho método se cancelará el procesamiento del gesto. Por ejemplo, que se arrastre una caja al pulsar sobre ella y hacer *scroll*.

```
GestureDetector detectorGestos;
public ViewGestos(Context context) {
    super(context);
    ListenerGestos lg = new ListenerGestos();
    detectorGestos = new GestureDetector(lg); detectorGestos.
    setOnDoubleTapListener(lg); }
@Override public boolean onTouchEvent(MotionEvent event) {
    return detectorGestos.onTouchEvent(event); }
class ListenerGestos extends
    GestureDetector.SimpleOnGestureListener {
@Override public boolean onDown(MotionEvent e) {
    return true; }
@Override public boolean onDoubleTap(MotionEvent e) {
    // Tratar el evento
    return true; } }
```



## 1.1. Tipos de sensores



Los **sensores de orientación y movimiento** se han popularizado mucho en los dispositivos móviles, por ejemplo:

- Detección de la orientación del dispositivo para visualizar correctamente documentos o imágenes.
- Implementación de aplicaciones de realidad aumentada, combinando orientación y cámara.
- Aplicaciones de navegación con la brújula.



## 1.1. Tipos de sensores

Para acceder a estos sensores utilizaremos la clase **SensorManager**. Para obtener un gestor de sensores utilizaremos el siguiente código:

```
String servicio = Context.SENSOR_SERVICE;  
SensorManager sensorManager =  
    (SensorManager)  
getSystemService(servicio);
```

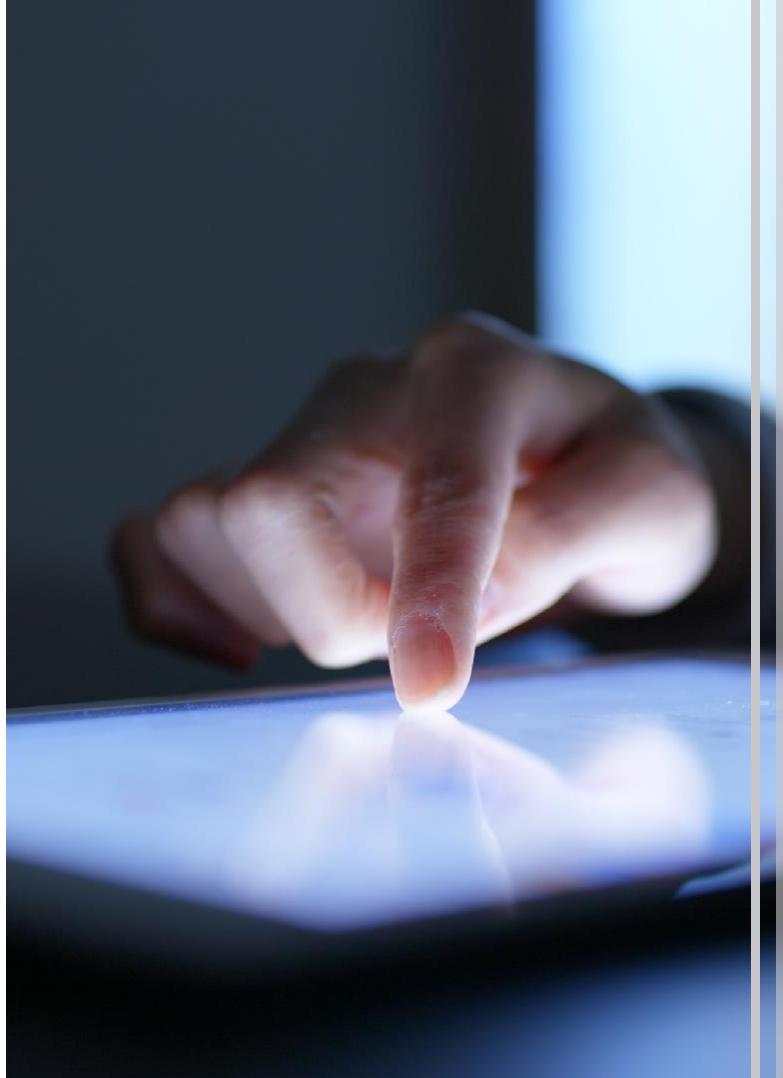
Con esta clase, podemos obtener acceso a un determinado **tipo de sensor**. Los tipos de sensor que podemos solicitar son las siguientes constantes de la clase **Sensor**:

- **TYPE\_ACCELEROMETER**. Acelerómetro de tres ejes que proporciona la aceleración en los ejes “x”, “y”, “z” en m/s<sup>2</sup>.
- **TYPE\_GYROSCOPE**. Giroscopio que proporciona la orientación del dispositivo en los tres ejes.



## 1.1. Tipos de sensores

- **TYPE\_MAGNETIC\_FIELD.** Sensor tipo brújula, que proporciona la orientación del campo magnético de los tres ejes en microteslas.
- **TYPE\_LIGHT.** Detecta la iluminación ambiental, para así poder modificar de forma automática el brillo de la pantalla.



## 1.1. Tipos de sensores



- **TYPE\_PROXIMITY.** Detecta la proximidad del dispositivo a otros objetos, utilizado habitualmente para apagar la pantalla cuando situamos el móvil cerca de nuestra oreja para hablar.
- **TYPE\_TEMPERATURE.** Termómetro que mide la temperatura en grados Celsius.
- **TYPE\_PRESSURE.** Nos proporciona la presión a la que está sometido el dispositivo en kilopascales.



## 1.1. Tipos de sensores

Para solicitar acceso a un sensor de un determinado tipo, utilizamos el siguiente método:

```
Sensor sensor = sensorManager  
        .getDefaultSensor(Sensor.TYPE_  
ACCELEROMETER);
```

Una vez que tenemos el sensor, deberemos definir un ***listener*** para recibir los cambios en las lecturas del sensor. Este será una clase que implemente la interfaz ***SensorEventListener***, que obliga a definir los siguientes métodos:

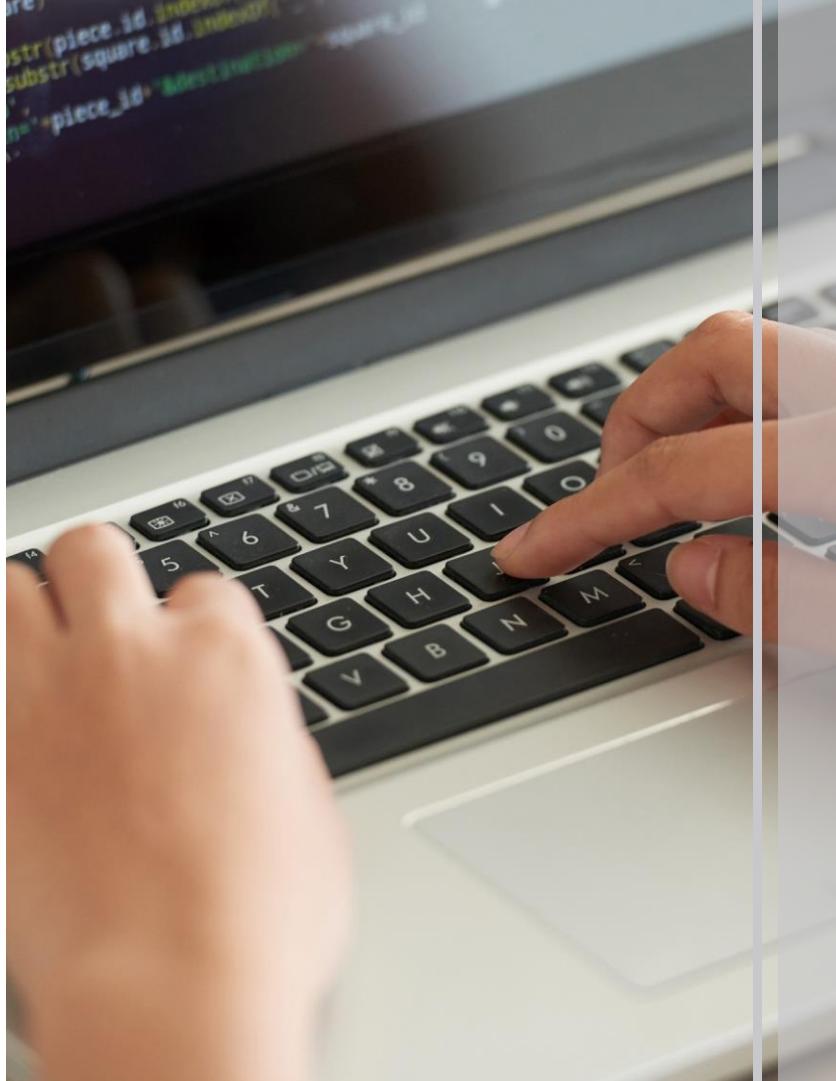
```
class ListenerSensor implements SensorEventListener {  
    public void onSensorChanged(SensorEvent sensorEvent) {  
        // La lectura del sensor ha cambiado  
    }  
    public void onAccuracyChanged(Sensor sensor, int  
accuracy) {  
        // La precisión del sensor ha cambiado  
    } }
```



## 1.1. Tipos de sensores

Una vez definido el *listener*, se registra para que reciba los eventos del sensor solicitado. Para registrarlo, además de indicar el sensor y el *listener*, deberemos especificar la periodicidad de actualización de los datos del sensor. Podemos utilizar como frecuencia de actualización las siguientes constantes de la clase *SensorManager*, ordenadas de mayor o menor frecuencia:

- **SENSOR\_DELAY\_FASTER.** Los datos se actualizan tan rápido como pueda el dispositivo.



## 1.1. Tipos de sensores



- **SENSOR\_DELAY\_GAME.** Los datos se actualizan a una velocidad suficiente para ser utilizados en videojuegos.
- **SENSOR\_DELAY\_NORMAL.** Esta es la tasa de actualización utilizada por defecto.
- **SENSOR\_DELAY\_UI.** Los datos se actualizan a una velocidad suficiente para mostrarlo en la interfaz de usuario.

## 1.1. Tipos de sensores

Una vez tenemos el sensor que queremos utilizar, así como el *listener* al que queremos que le proporcione las lecturas, y la tasa de actualización de estas últimas, podemos registrar el *listener* para empezar a obtener lecturas, de la siguiente forma:

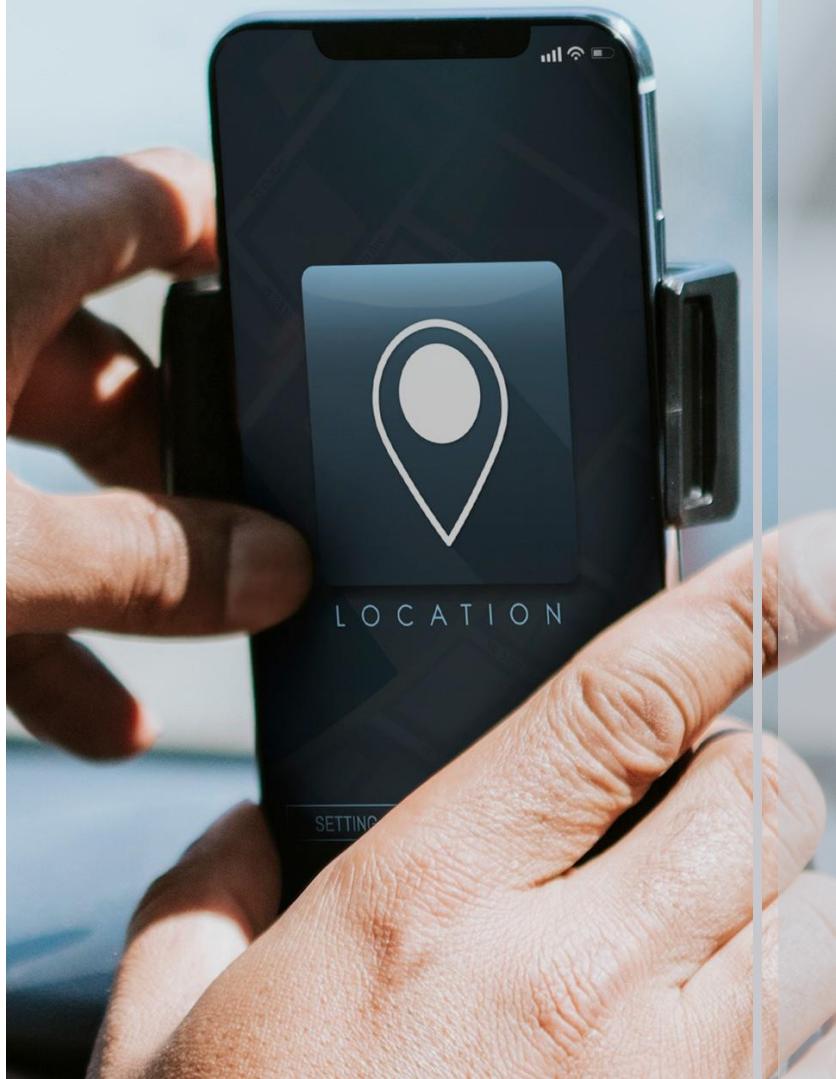
```
ListenerSensor listener = new ListenerSensor();
sensorManager.registerListener(listener, sensor,
SensorManager.SENSOR_DELAY_NORMAL);
```





## 1.1. Tipos de sensores

Los dispositivos móviles son capaces de obtener su **posición geográfica** por diferentes medios. Muchos dispositivos cuentan un con **GPS** capaz de proporcionarnos nuestra posición con un error de unos pocos metros. El inconveniente del GPS es que solo funciona en entornos abiertos. Cuando estamos en entornos de interior, o bien cuando nuestro dispositivo no cuenta con GPS, una forma alternativa de localizarnos es mediante la red 3G o wifi. En este caso, el error de localización es bastante mayor.



## 1.1. Tipos de sensores

Para poder utilizar los servicios de geolocalización, debemos solicitar permiso en el **manifest** para acceder a estos servicios. Se solicita por separado permiso para el servicio de localización de forma precisa (*fine*) y para localizarnos de forma aproximada (*coarse*):

```
<uses-permission android:name=
    "android.permission.ACCESS_FINE_
LOCATION"/>
<uses-permission android:name=
    "android.permission.ACCESS_COARSE_
LOCATION"/>
```

Al tener el permiso de localización precisa, tendremos automáticamente concedido el de localización aproximada. El dispositivo GPS necesita tener permiso para localizarnos de forma precisa, mientras que para la localización mediante la red es suficiente con tener permiso de localización aproximada.



## 1.1. Tipos de sensores

Para acceder a los servicios de geolocalización en Android tenemos la clase *LocationManager*. Esta clase no se debe instanciar directamente, sino que obtendremos una instancia como un servicio del sistema de la siguiente forma:

```
LocationManager manager = (LocationManager) this.  
getSystemService(Context.LOCATION_SERVICE);
```

Para obtener una localización deberemos especificar el proveedor que queramos utilizar. Los principales proveedores disponibles en los dispositivos son el GPS (*LocationManager.GPS\_PROVIDER*) y la red 3G o wifi (*LocationManager.NETWORK\_PROVIDER*). Podemos obtener información sobre estos proveedores con:

```
LocationProvider proveedor = manager  
.getProvider(LocationManager.GPS_PROVIDER);
```



## 1.1. Tipos de sensores



El objeto *Location* obtenido incluye toda la información sobre nuestra posición, entre la que se encuentra la latitud y longitud. Con esta llamada, obtenemos la última posición que se registró, pero no se actualiza dicha posición.

- **Actualización de la posición.** Para poder recibir actualizaciones de nuestra posición deberemos definir un *listener* de clase *LocationListener*:

## 1.1. Tipos de sensores

```
class ListenerPosicion implements LocationListener {  
    public void onLocationChanged(Location location) {  
        // Recibe nueva posición.  
    }  
    public void onProviderDisabled(String provider){  
        // El proveedor ha sido desconectado.  
    }  
    public void onProviderEnabled(String provider){  
        // El proveedor ha sido conectado.  
    }  
    public void onStatusChanged(String provider,  
                               int status, Bundle extras){  
        // Cambio en el estado del proveedor.  
    }  
};
```



## 1.1. Tipos de sensores

Una vez definido el *listener*, podemos solicitar actualizaciones de la siguiente forma:

```
ListenerPosicion listener = new ListenerPosicion();
long tiempo = 5000; // 5 segundos
float distancia = 10; // 10 metros
manager.requestLocationUpdates(
    LocationManager.GPS_PROVIDER,
    tiempo, distancia, listenerPosicion);
```

Podemos observar que cuando pedimos las actualizaciones, además del proveedor y del *listener*, debemos especificar el intervalo mínimo de tiempo (en milisegundos) que debe transcurrir entre dos lecturas consecutivas, y el umbral de distancia mínima que debe variar nuestra posición para considerar que ha habido un cambio de posición y notificar la nueva lectura.



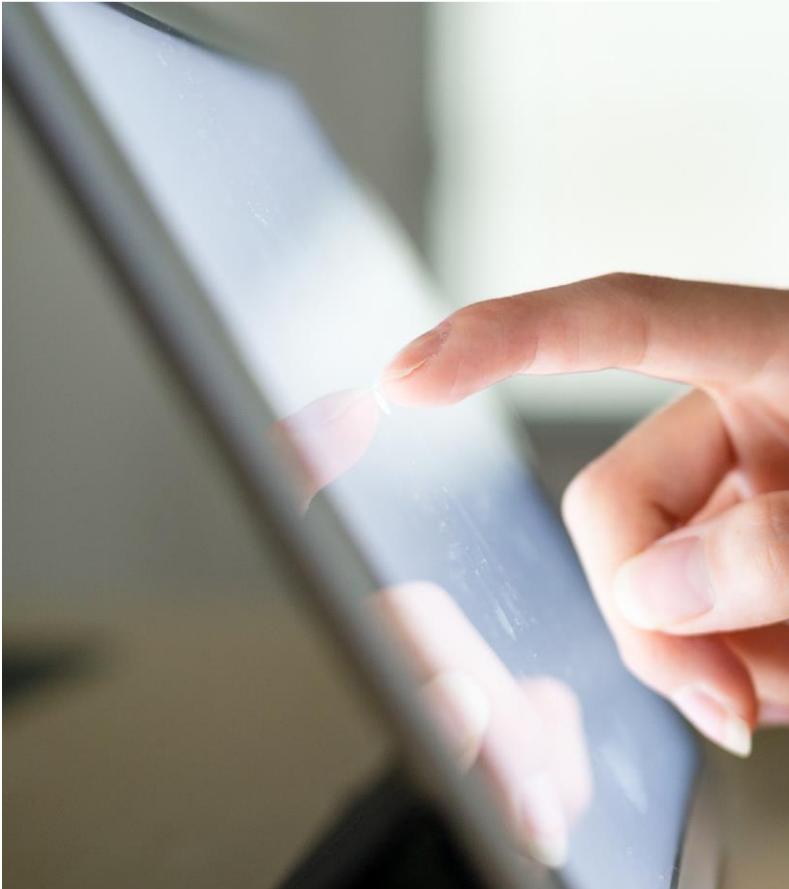
# Video



Te invitamos a ver el siguiente video:



## 1.2. Lógica de eventos



Al acercar los dedos a la pantalla, transmitimos electricidad, por lo que el *smartphone* sabe exactamente dónde tocamos, convirtiendo las pulsaciones en eventos asociados a unas coordenadas concretas. El sistema genera eventos con la posición exacta de cada toque; y las apps reaccionan en consecuencia. Las **pantallas capacitivas** de un *smartphone* poseen una capa conductiva bajo el cristal exterior donde se coloca una matriz de electrodos.



## 1.2. Lógica de eventos

Al poner el dedo, los electrodos verticales y horizontales de la matriz establecen una conexión gracias a la conductividad corporal. Esto no solo sucede con el toque directo sobre el panel, sino que también a escasa distancia del roce. Esto porque existen capas táctiles de alta sensibilidad que crean un campo electrostático, capaz de captar el dedo a menos de un milímetro de tocar la pantalla.



## 1.2. Lógica de eventos

Además, los paneles táctiles de un *smartphone* admiten el toque de varios dedos a la vez (multitoque). La matriz de electrodos puede determinar los cambios de corriente en distintos puntos simultáneos; por lo general hasta 10.



## 1.2. Lógica de eventos



Las **pantallas resistivas** no se aprovechan de la conductividad eléctrica de nuestros dedos, ya que están compuestas por dos capas que buscan el contacto entre sí. Al tener corriente eléctrica cada una de las capas, estas quedan separadas para no tocarse. Basta con que se unan mediante presión para que se cree un cambio en dicha corriente. Después, los sensores situados en el panel táctil pueden determinar la posición del toque calculando el punto exacto de presión.



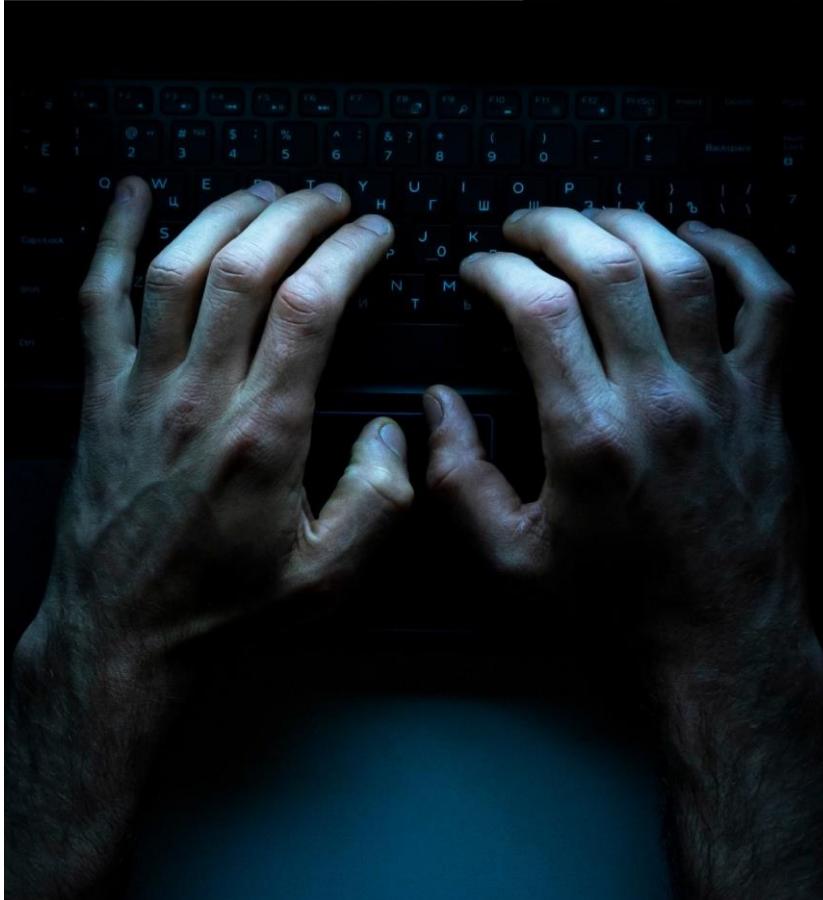
## 1.2. Lógica de eventos

El ciclo de vida de un toque de eventos se inicia cuando el usuario coloca uno o más punteros en la pantalla táctil del dispositivo, y termina cuando se retiran estos puntero(s) de la pantalla.

Mientras que uno o más indicadores se encuentran en contacto con la pantalla, *MotionEvent* recopila información acerca de los eventos de toque. Esta información incluye el toque de movimiento en el evento, en términos de las coordenadas “X” y “Y”; así como la presión y el tamaño de la zona de contacto.



## 1.2. Lógica de eventos

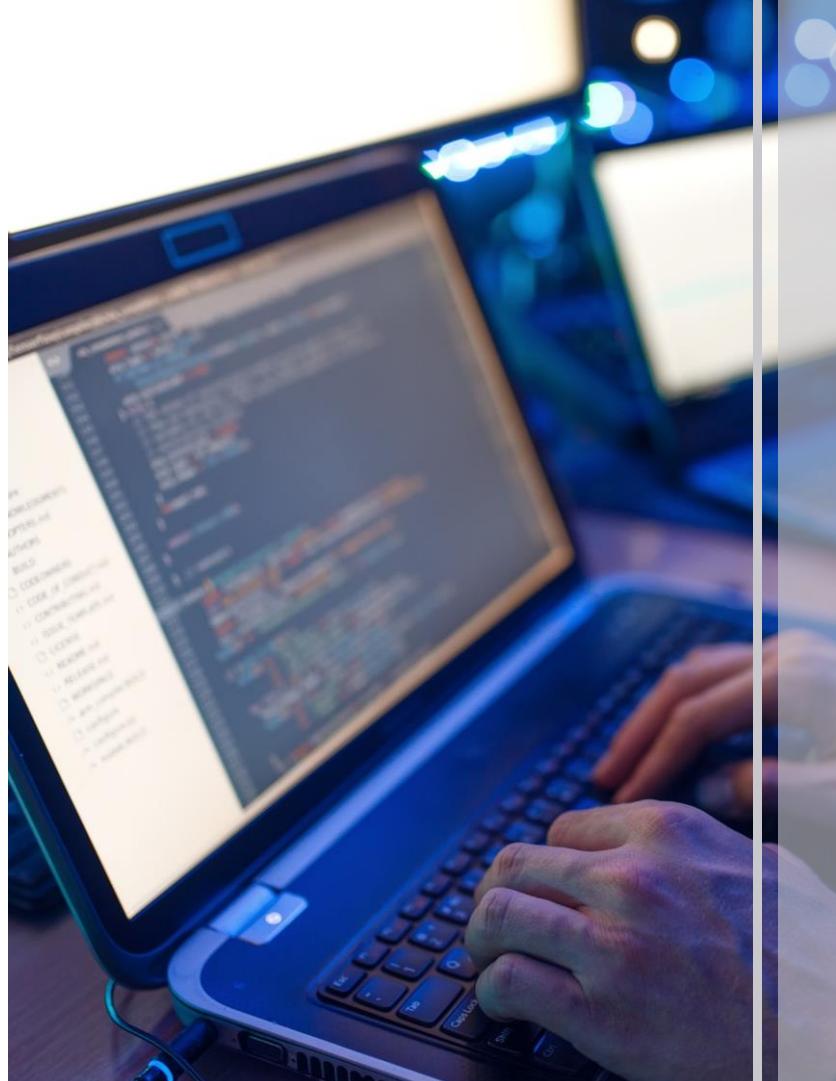


Un **MotionEvent** también describe los eventos de toque del estado, a través de un código de acción. Android soporta una larga lista de códigos de acción, pero algunos de los principales son:

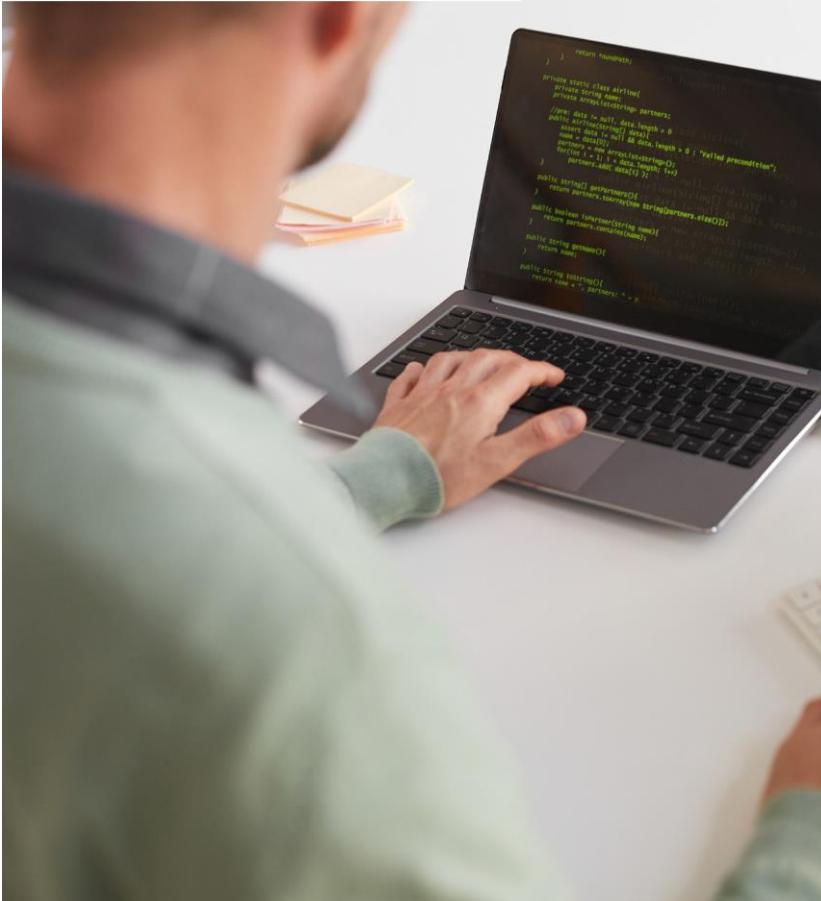
- **ACTION\_DOWN.** Cuando un toque evento haya comenzado. Este es el lugar donde el puntero primero hace contacto con la pantalla.
- **ACTION\_MOVE.** Cuando se ha producido un cambio durante el evento táctil (entre *ACTION\_DOWN* y *ACTION\_UP*).

## 1.2. Lógica de eventos

- **ACTION\_UP.** El toque evento ha terminado. Este contiene la versión final de la ubicación. Suponiendo que el gesto no está cancelada, todos los eventos de toque concluir con *ACTION\_UP*.
- **ACTION\_CANCEL.** El gesto fue cancelado, y Android no puede recibir más información acerca de este. Debes manejar un *ACTION\_CANCEL* exactamente de la misma manera en que manejas un *ACTION\_UP* evento.



## 1.2. Lógica de eventos



El *onTouchEvent()* es un método que se activa cada vez que un puntero se ubica en una posición, ejerce presión o se encuentra en el área de contacto de los cambios. El *MotionEvent* permite a los objetos transmitir el código de la acción y los valores de los ejes para el *onTouchBack()*. Este último es el evento método de devolución de llamada para la vista que recibió este evento táctil.

## 1.2. Lógica de eventos

En el código siguiente también se usa ***getActionMasked()***, para recuperar la acción que se realiza:

```
importación androidx.appcompat.app.AppCompatActivity;
importación androidx.núcleo.vista MotionEventCompat;
import android.os.Paquete;
import android.util.De registro;
import android.vista.MotionEvent;
public class MainActivity extends AppCompatActivity {
    private static final String ETIQUETA = "MyActivity";
    @Override protected void onCreate(Bundle
savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R. layout.activity_main);  }
    @Override public boolean onTouchEvent
(MotionEvent evento){
    int myAction = MotionEventCompat.
getActionMasked(evento);
```



## 1.2. Lógica de eventos

```
switch (myAction) {  
    caso (MotionEvent.ACTION_UP):  
        Registro.yo(TAG, “la acción”);  
        return true;  
    caso (MotionEvent.ACTION_DOWN):  
        Registro.d(TAG, “Abajo”);  
        return true;  
    caso (MotionEvent.ACTION_MOVE):  
        Registro.d(TAG, “acción de Mover”);  
        return true;  
    caso (MotionEvent.ACTION_CANCEL):  
        Registro.d(TAG, “acción de Cancelar”);  
        return true;  
    por defecto:  
        return super.onTouchEvent(evento);  
    } } }
```



# FORO 1

## Entorno de trabajo.

Participa en el foro enviando imágenes que demuestren que ya tienes acceso a las siguientes herramientas en su versión de prueba:

- Kotlin

Presiona el botón para participar en el foro.



# Conclusión



La mayoría de los dispositivos con Android tiene sensores integrados que miden el movimiento, la orientación y diversas condiciones ambientales. Estos sensores son capaces de proporcionar datos sin procesar con alta precisión y exactitud, y son útiles para supervisar el movimiento o posicionamiento tridimensional del dispositivo. O bien, supervisar los cambios en el entorno ambiental cerca de un dispositivo.

La interacción con el usuario, generalmente, se diseña pensando únicamente en el uso mediante la pantalla táctil. Sin embargo, existe una gran cantidad de sensores que puede contribuir a mejorar la experiencia del usuario, o simplemente brindar alternativas de uso.





# ¡Felicitaciones!

Acabas de concluir la primera unidad de tu curso *Desarrollo de Aplicaciones Móviles II*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.



# Unidad 2

---

Multimedia



# Temario Unidad 2

**2.1**

Reproducción de Audio

**2.2**

Reproducción de Video



# Introducción



En esta segunda unidad aprenderás sobre la multimedia móvil, en específico la reproducción de audio y video. De gran importancia en el mundo de las aplicaciones móviles.



# Competencias a Desarrollar



El alumno será capaz de identificar los diferentes tipos de reproductores multimedia.

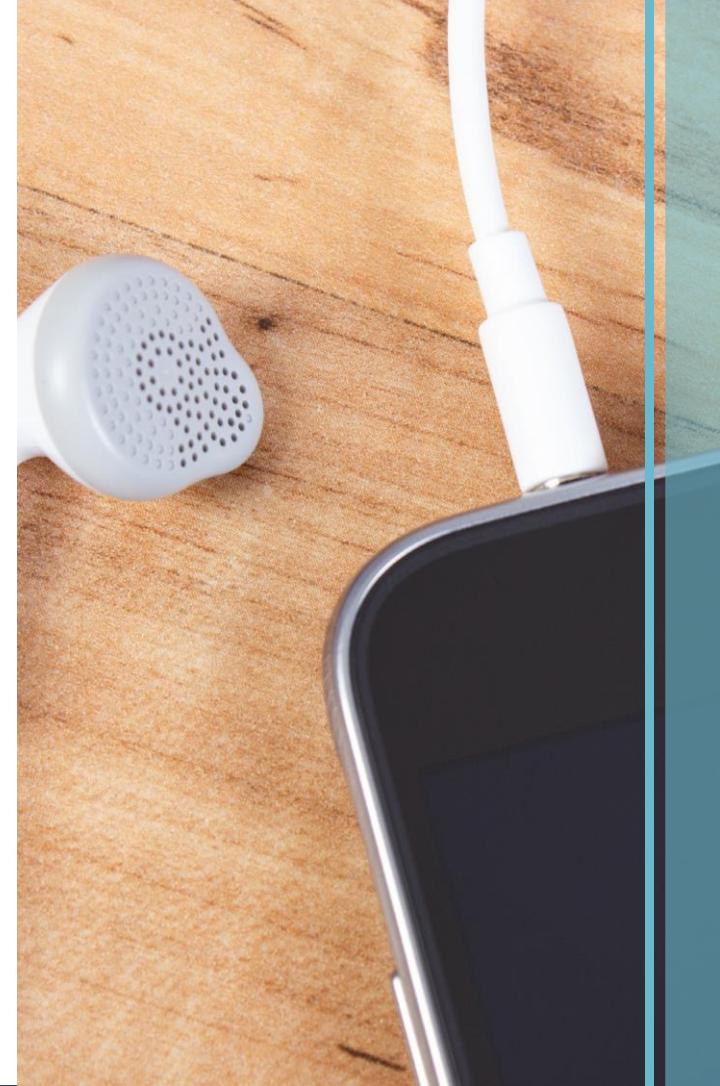


El alumno manipulará el entorno de multimedia en el desarrollo de aplicaciones.



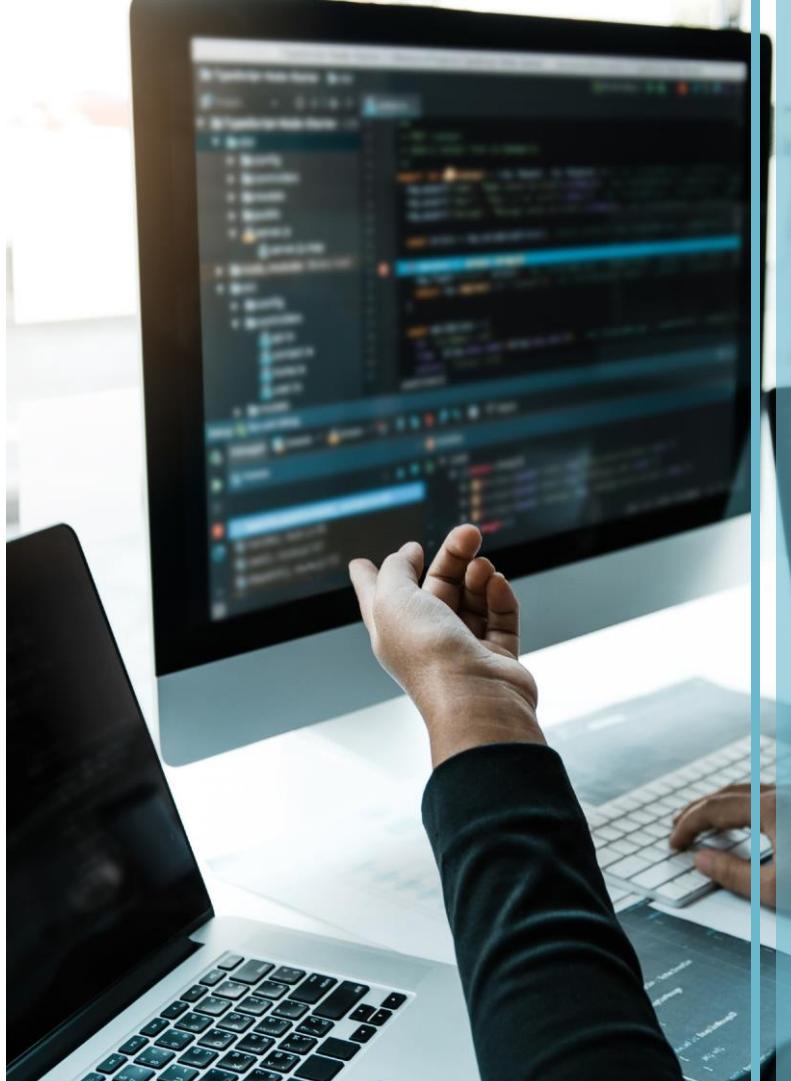
## 2.1. Reproducción de audio

La reproducción de contenido multimedia se lleva a cabo por medio de la clase **MediaPlayer**. Dicha clase permite reproducir archivos multimedia almacenados como recursos de la aplicación, en ficheros locales, en proveedores de contenido, o bien servidos por medio de *streaming* a partir de una URL.

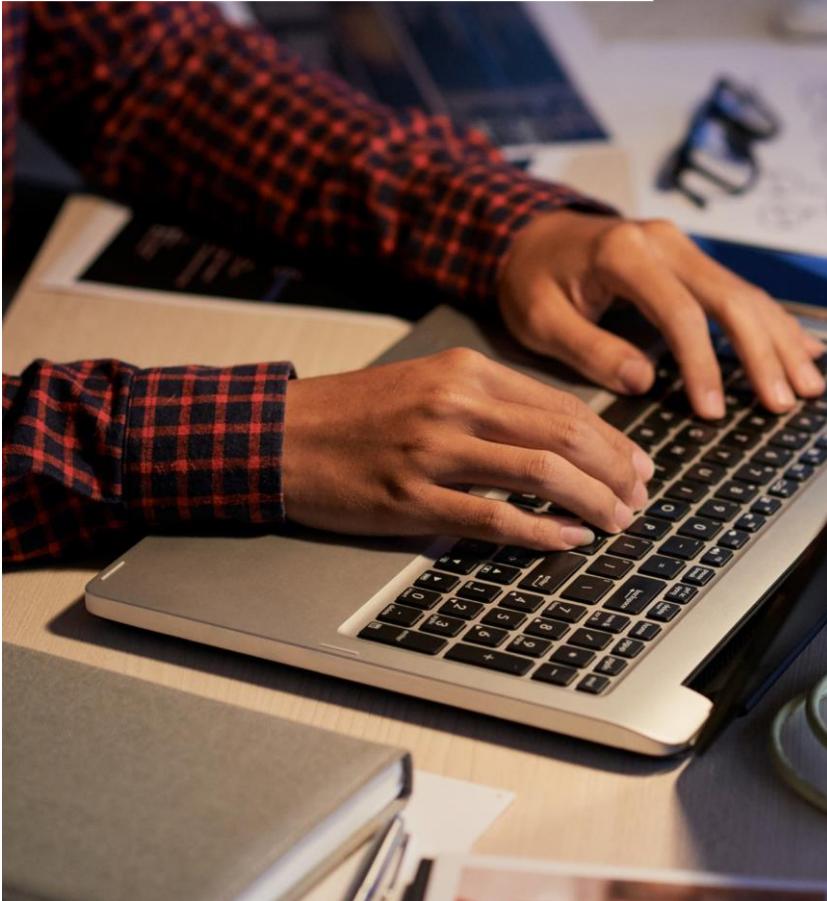


## 2.1. Reproducción de audio

En todos los casos, como desarrolladores, la clase *MediaPlayer* nos facilita abstraernos del formato, así como del origen del fichero a reproducir, a fin de incluir un fichero de audio en los recursos de la aplicación que pueda ser reproducido durante su ejecución.



## 2.1. Reproducción de audio



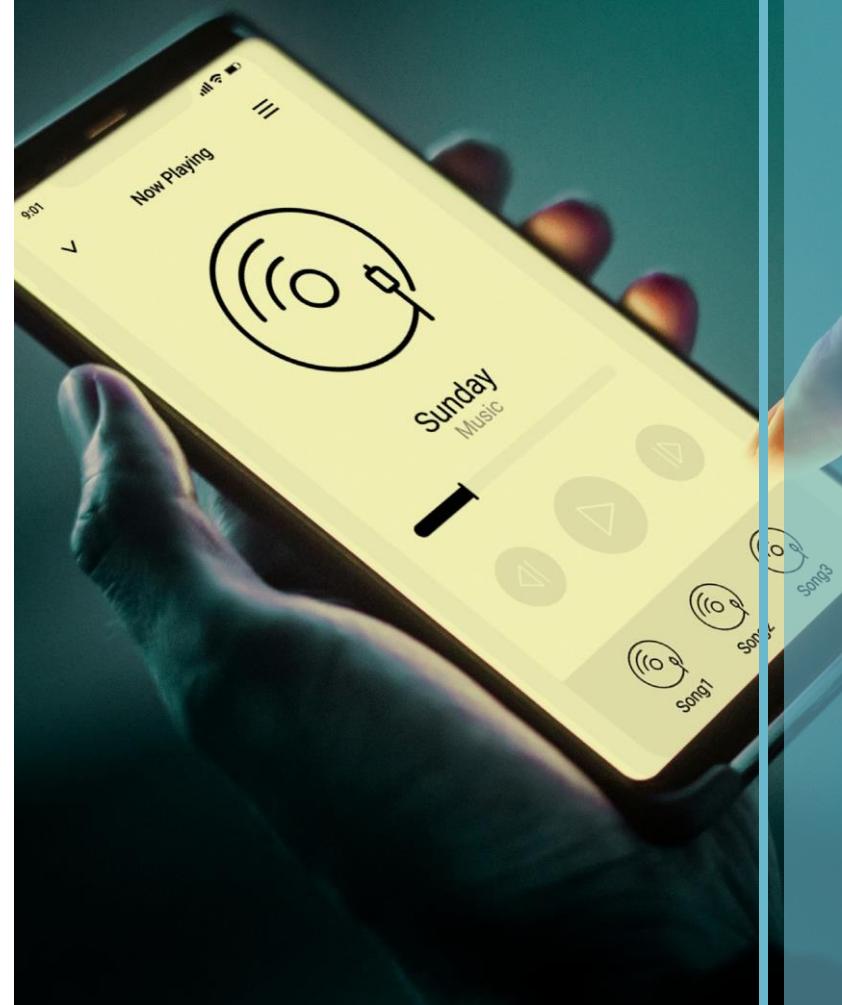
Para insertar un fichero de audio, primero tenemos que crear una carpeta *raw* dentro de la carpeta *res*, y almacenar en ella el fichero o ficheros sin comprimir que deseamos reproducir. A partir de ese momento el fichero se identificará dentro del código como ***R.raw.nombre\_fichero***.



## 2.1. Reproducción de audio

Para **reproducir un fichero de audio** seguimos los siguientes pasos:

1. Crear una instancia de la clase *MediaPlayer*.
2. Indicar qué fichero será el que se reproducirá.
3. Reproducir el contenido multimedia.



## 2.1. Reproducción de audio



Veamos primero cómo inicializar la reproducción. Tenemos dos opciones:

1. Crear una **instancia de la clase MediaPlayer** por medio del método ***create()***. En este caso, debemos pasar como parámetro, además del contexto de la aplicación, el identificador del recurso, tal como se aprecia en el siguiente ejemplo:



## 2.1. Reproducción de audio

```
Context applicationContext = getApplicationContext();
// Recurso de la aplicación
MediaPlayer resourcePlayer = MediaPlayer.create(applicationContext,
R.raw.my_audio);

// Fichero local (en la tarjeta de memoria)
MediaPlayer filePlayer = MediaPlayer.create(applicationContext,
Uri.parse("file:///sdcard/localfile.mp3"));

// URL
MediaPlayer urlPlayer = MediaPlayer.create(applicationContext,
Uri.parse("http://site.com/audio/audio.mp3"));

// Proveedor de contenido
MediaPlayer contentPlayer = MediaPlayer.create(applicationContext,
Settings.System.DEFAULT_RINGTONE_URI);
```



## 2.1. Reproducción de audio

**2.** A través del método ***setDataSource()*** para asignar una fuente multimedia a una instancia ya existente de la clase *MediaPlayer*. En este caso, es muy importante recordar que se debe llamar al método *prepare()* antes de poder reproducir el fichero de audio. Sin embargo, esto último no es necesario si la instancia de *MediaPlayer* se ha creado con el método *create()*.

```
MediaPlayer mediaPlayer = new MediaPlayer();
mediaPlayer.setDataSource("/sdcard/test.mp3");
mediaPlayer.prepare();
```



## 2.1. Reproducción de audio



Una vez que la instancia de la clase *MediaPlayer* ha sido iniciada, podemos comenzar la reproducción mediante el método *start()*. También es posible utilizar los métodos *stop()* y *pause()* para detener y pausar la reproducción. En este último caso, la reproducción continuará tras hacer una llamada al método *play()*.

## 2.1. Reproducción de audio

Otros métodos de la clase **MediaPlayer** que nos pueden ser de utilidad son los siguientes:

- **setLooping()**. Nos permite especificar si el clip de audio debe reproducirse nuevamente cada vez que finalice.

```
if (!mediaPlayer.isLooping())
    mediaPlayer.setLooping(true);
```



## 2.1. Reproducción de audio

- ***ScreenOnWhilePlaying()***. Con él, conseguimos que la pantalla se encuentre activada durante la reproducción. Tiene más sentido en el caso de la reproducción de video.

```
mediaPlayer.setScreenOnWhilePlaying(true);
```

- ***setVolume()***. Modifica el volumen. Recibe dos parámetros que deberán ser dos números reales entre 0 y 1, indicando el volumen del canal izquierdo y del canal derecho, respectivamente. El valor 0 indica silencio total mientras que el valor 1 indica máximo volumen.

```
mediaPlayer.setVolume(1f, 0.5f);
```



## 2.1. Reproducción de audio

- **seekTo()**. Permite avanzar o retroceder a un determinado punto del archivo de audio. Con él podemos obtener la duración total del clip de audio con el método *getDuration()*, mientras que *getCurrentPosition()* nos da la posición actual. En el siguiente código se puede ver un ejemplo de uso de estos tres últimos métodos.

```
mediaPlayer.start();
int pos = mediaPlayer.getCurrentPosition();
int duration = mediaPlayer.getDuration();
mediaPlayer.seekTo(pos + (duration-pos)/10);
```



## 2.1. Reproducción de audio

Una acción muy importante que deberemos llevar a cabo luego de que haya finalizado definitivamente la reproducción (porque se vaya a salir de la actividad, por ejemplo) es destruir la instancia de la clase *MediaPlayer* y liberar su memoria. Para ello, debemos hacer uso del método ***release()***.

```
MediaPlayer.release();
```



## 2.1. Reproducción de audio



Podemos **reproducir audio proveniente de diferentes fuentes:**

- Un recurso de la aplicación.
- Un fichero en el dispositivo local.
- Una URL remota.



## 2.1. Reproducción de audio

### Inicialización asíncrona

La llamada a ***prepare()*** puede resultar bastante costosa y producir un retardo considerable, especialmente cuando accedemos a medios externos a la aplicación.

Podemos crear un hilo secundario y realizar la preparación del medio desde él, pero también contamos con una variante del método anterior que nos facilitará realizar la preparación de forma asíncrona, fuera del hilo de eventos. Esta variante es ***prepareAsync()***.



## 2.1. Reproducción de audio



Es posible utilizar un listener de tipo ***MediaPlayer.OnPreparedListener*** para que, con el método ***onPrepared***, nos sea notificado cuando esté preparado el reproductor, y así poder comenzar la reproducción.



## 2.1. Reproducción de audio

Este **listener** se deberá registrar en el *MediaPlayer* con el método *setOnPreparedListener()*:

```
public class MiActividad extends Activity
    implements MediaPlayer.OnPreparedListener {
    MediaPlayer mMediaPlayer = null;
    public void reproducir() {
        mMediaPlayer = new MediaPlayer();
        mMediaPlayer.setOnPreparedListener(this);
        mMediaPlayer.prepareAsync();
    }
    public void onPrepared(MediaPlayer player) {
        mMediaPlayer.start();
    }
}
```



# Video



Te invitamos a ver el siguiente video:



## 2.2. Reproducción de video

**Principales diferencias entre reproducción de audio y de video:**

- 1.** No se puede reproducir un clip de video almacenado como parte de los recursos de la aplicación. En este caso, debemos utilizar cualquiera de los otros tres medios (ficheros locales, *streaming* o proveedores de contenidos).
- 2.** El video necesita de una superficie para poder reproducirse. Esta superficie se corresponderá con una vista dentro del *layout* de la actividad.



## 2.2. Reproducción de video



Según lo anterior, existen varias alternativas para la reproducción de video. La más sencilla es hacer uso de un control de tipo **Video View**, el cual encapsula la creación de una superficie en la que reproducir el video y el control de este con ayuda de una instancia de la clase *Media Player*.

El primer paso de este método consiste en añadir el control *Video View* a la interfaz gráfica de la aplicación.



## 2.2. Reproducción de video

En el fichero XML correspondiente debemos **agregar un elemento de tipo Video View:**

```
<VideoView android:id="@+id/superficie"  
          android:layout_height="fill_parent"  
          android:layout_width="fill_parent">  
</VideoView>
```

Dentro del código Java podemos acceder a este elemento de la manera habitual, es decir, mediante el método ***findViewById()***.



## 2.2. Reproducción de video

Posteriormente, asignamos una fuente que se corresponderá con el contenido multimedia a reproducir. El control *Video View* se encargará de la inicialización del *MediaPlayer*.

Para asignar un video a reproducir es posible utilizar cualquiera de estos dos métodos:

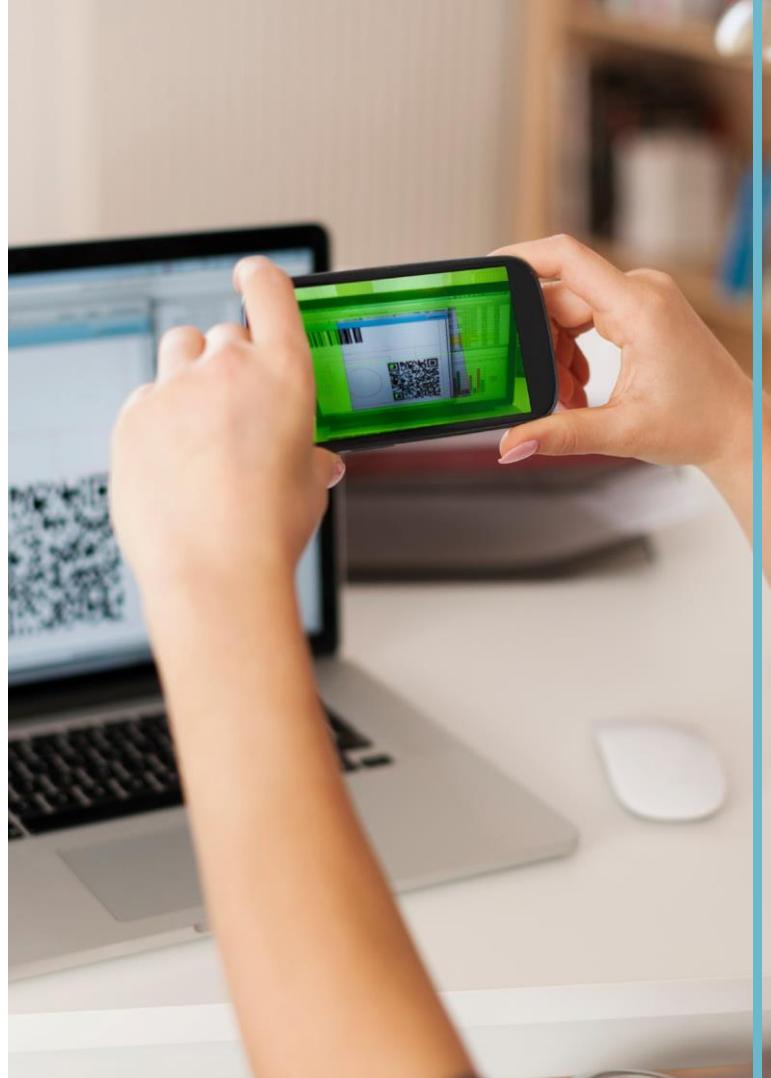
```
videoView1.setVideoUri("http://www.mysite.com/  
videos/myvideo.3gp"); lvideoView2.setVideoPath("/  
sdcard/test2.3gp");
```



## 2.2. Reproducción de video

Una vez iniciado el control, podemos controlar la reproducción con los métodos *start()*, *stopPlayback()*, *pause()* y *seekTo()*.

La clase *Video View* también incorpora el método ***setKeepScreenOn(boolean)***, con la que se puede controlar el comportamiento de la iluminación de la pantalla durante la reproducción del clip de video. Si se pasa como parámetro el valor *true*, la pantalla permanecerá iluminada.



## 2.2. Reproducción de video

El siguiente código muestra un ejemplo de **asignación de un video a un control Video View** y de su posterior reproducción. Dicho código puede ser utilizado como un tipo esqueleto en nuestra propia aplicación.

Enseguida mostramos un ejemplo del uso de `seekTo()`; en este caso, se usa para avanzar hasta la posición intermedia del video:

```
VideoView videoView = (VideoView) findViewById(R.  
    id.superficie);  
videoView.setKeepScreenOn(true);  
videoView.setVideoPath("/sdcard/ejemplo.3gp");  
if (videoView.canSeekForward())  
    videoView.seekTo(videoView.  
        getDuration()/2);  
videoView.start();  
// Hacer algo durante la reproducción  
videoView.stopPlayback();
```



## 2.2. Reproducción de video

La reproducción de video es muy similar a la reproducción de audio, excepto por dos cosas. En primer lugar, no es posible reproducir un clip de video almacenado como parte de los recursos de la aplicación. En este caso, no existe ningún método para reproducir un video a partir de un *id* de recurso. Sin embargo, lo que sí que podemos hacer es **representar un recurso raw de tipo video mediante una URL**. Esta URL tendrá la siguiente forma:

```
Uri.parse("android.resource://es.ua.jtech/" + R.raw.video)
```

Hay que especificar como *host* el paquete de nuestra aplicación, y como ruta el *id* del recurso.



## 2.2. Reproducción de video



La segunda característica es que el video necesita de una superficie para poder reproducirse. Esta superficie se corresponderá con una vista dentro del *layout* de la actividad.

Existen varias alternativas para la reproducción de video. La más sencilla es hacer uso de una vista de tipo *VideoView*, que encapsula tanto la creación de una superficie donde se pueda reproducir el video como el control del mismo mediante una instancia de la clase *MediaPlayer*.



## 2.2. Reproducción de video

Veamos este método. El primer paso consiste en añadir la vista *VideoView* a la interfaz gráfica de la aplicación. Para ello, añadimos el elemento en el archivo de *layout* correspondiente:

```
<VideoView android:id="@+id/superficie"  
    android:layout_height="fill_parent"  
    android:layout_width="fill_parent">  
</VideoView>
```



## 2.2. Reproducción de video

Dentro del código Java podremos acceder a dicho elemento de la forma más común, es decir, mediante el método ***findViewById***. Una vez hecho esto, designamos una fuente, la cual se corresponderá con el contenido multimedia a reproducir. El *VideoView* se encargará de la inicialización del objeto *MediaPlayer*. Para asignar un video a reproducir es posible emplear cualquiera de ambos métodos:

```
videoView1.setVideoUri("http://www.mysite.com/  
videos/myvideo.3gp");  
videoView2.setVideoPath("/sdcard/test2.3gp");
```

# ACTIVIDAD 1

Te invitamos a realizar la siguiente actividad:

Presiona el botón para descargar la actividad:



Presiona el botón para entregar la actividad:



# Conclusión



La multimedia móvil corresponde a varios tipos de contenido a los que se accede a través de dispositivos portátiles o se crean con ellos. Una de las áreas en las que la multimedia móvil se ha vuelto omnipresente es en los teléfonos inteligentes, los cuales suelen incorporar capacidad de reproducción de video y música, cámaras y transmisión inalámbrica de contenido.

El uso de los dispositivos móviles es cada vez más común debido a la poca infraestructura que se necesita para su uso y la portabilidad que ofrecen, pues facilitan el empleo de tabletas o teléfonos inteligentes. Asimismo, la cantidad de aplicaciones a las que se pueden acceder en Internet ayuda a enriquecer todas las actividades





# ¡Felicitaciones!

Acabas de concluir la segunda unidad de tu curso *Desarrollo de Aplicaciones Móviles II*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.





## Unidad 3

# Ficheros y Acceso de Datos



# Temario Unidad 3

**3.1**

Archivos  
de Texto y SQLite

**3.2**

Proveedores  
de Contenidos



# Introducción



En esta tercera unidad aprenderás cómo desarrollar en entornos de computación móvil para que las aplicaciones sean capaces de acceder a datos e información en cualquier red.



# Competencias a Desarrollar



El alumno será capaz de identificar el uso de los objetos del SQLite para la manipulación de datos.



El alumno conocerá distintas opciones de almacenamiento de datos disponibles en Android.

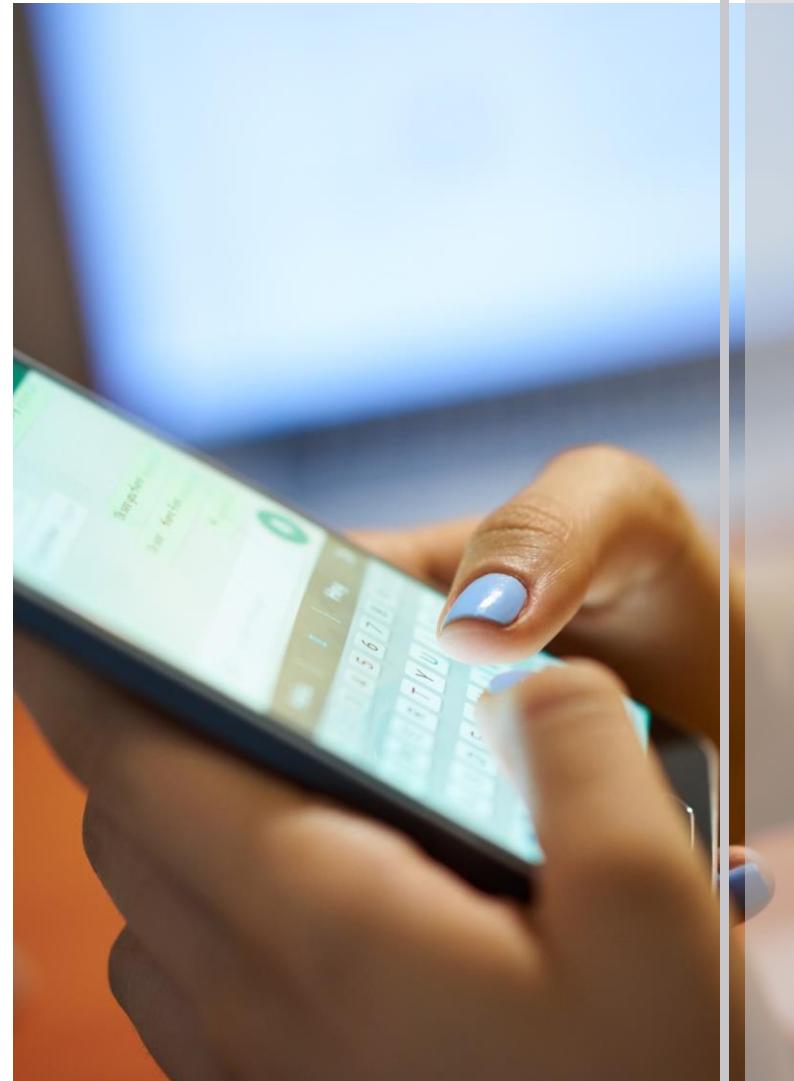


## 3.1. Archivos de texto y SQLITE

### Leer y escribir en un archivo de texto

Para escribir un archivo de texto, este se añade al final del archivo; mientras que la lectura se hace leyendo línea a línea.

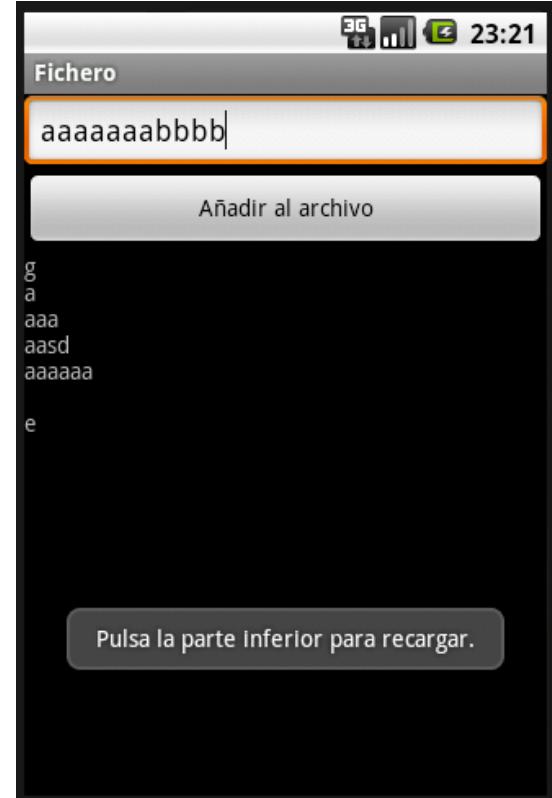
Primero hay que crear una nueva aplicación llamada *Fichero*, cuya única actividad, la principal, mostrará un *EditText*, cuyo texto será recogido cada vez que se pulse un *Button* que habrá debajo. Dicho texto será añadido a un archivo de texto llamado *mytextfield.txt*.



## 3.1. Archivos de texto y SQLITE

Para añadir texto a un archivo primero lo abrimos con ***openFileOutput(FILENAME,Context.MODE\_APPEND)*** y después utilizamos el método *append(...)* de *OutputStreamWriter*:

- La actividad puede pasar a inactiva en cualquier momento, y puede no volver a recuperarse.
- Debajo del campo de texto y el botón, se añade un *TextEdit* que ocupe el resto de la pantalla. Lo hacemos pulsable con el método *setClickable(true)*. En consecuencia, al dar clic sobre este, se leerá el archivo línea a línea, y se mostrará entero.



## 3.1. Archivos de texto y SQLITE



**SQLLite.** Es un gestor de bases de datos relacional de código abierto. Cumple con los estándares, y es extremadamente ligero. Además, guarda toda la base de datos en un único fichero. Es útil en aplicaciones pequeñas para no requerir la instalación adicional de un gestor de bases de datos, así como para dispositivos embebidos con recursos limitados. Android incluye soporte a SQLLite.

## 3.1. Archivos de texto y SQLITE



Para ilustrar el uso de SQLite en Android vamos a ver una **clase adaptador**. En esta clase (*DataHelper*) abriremos la base de datos de una manera estándar mediante una clase abstracta *SQLiteOpenHelper*. Esta nos obliga a implementar nuestro propio *Helper* de apertura de la base de datos, de una manera estándar.

## 3.1. Archivos de texto y SQLITE



Básicamente sirve para que nuestro código esté obligado a saber qué hacer en caso de que la base de datos no exista (normalmente crearla). En el siguiente código se muestra cómo borrar todos los contenidos y crear una nueva base de datos vacía, perdiendo todo lo anterior. Para recorrer y manipular la base de datos el objeto cursor devuelve el resultado de las filas afectadas. El cursor es aleatorio. Además, podemos solicitar el número de registros que contiene.

## 3.1. Archivos de texto y SQLITE

```
package es.ua.jtech.daa.db;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.util.Log;
import java.util.ArrayList;
import java.util.List;
public class DataHelper {
    private static final String DATABASE_NAME = "mibasededatos.db";
    private static final int DATABASE_VERSION = 1;
    private static final String TABLE_NAME = "ciudades";
```

## 3.1. Archivos de texto y SQLITE

```
private static final String[] COLUMNAS =  
{"_id", "nombre", "habitantes"};  
private static final String INSERT = "insert into " + TABLE_  
NAME +  
    ("("+COLUMNAS[1]+"," +COLUMNAS[2]+") values (?,?)";  
private static final String CREATE_DB = "CREATE TABLE " +  
TABLE_NAME +  
    ("("+COLUMNAS[0]+") INTEGER PRIMARY KEY, "  
     +COLUMNAS[1]+" TEXT, "  
     +COLUMNAS[2]+" NUMBER)";  
private Context context;  
private SQLiteDatabase db;  
private SQLiteStatement insertStatement;  
public DataHelper(Context context) {  
    this.context = context;
```



## 3.1. Archivos de texto y SQLITE

```
MiOpenHelper openHelper = new MiOpenHelper(this.context);
this.db = openHelper.getWritableDatabase();
this.insertStatement = this.db.compileStatement(INSERT);
}
public long insert(String name, long number) {
this.insertStatement.bindString(1, name);
this.insertStatement.bindLong(1, number);
return this.insertStatement.executeInsert();
}
public int deleteAll() {
    return db.delete(TABLE_NAME, null, null);
}
public List<String> selectAllNombres() {
    List<String> list = new ArrayList<String>();
```

## 3.1. Archivos de texto y SQLITE

```
Cursor cursor = db.query(TABLE_NAME, COLUMNAS,
    null, null, null, null, null);
if (cursor.moveToFirst()) {
    do {
        list.add(cursor.getString(1));
    } while (cursor.moveToNext());
} if (cursor != null && !cursor.isClosed()) {
    cursor.close();
} return list;
}
private static class MiOpenHelper extends SQLiteOpenHelper {
    MiOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    } @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_DB);    }
}
```

## 3.1. Archivos de texto y SQLITE

```
@Override    public void onUpgrade(SQLiteDatabase db, int oldVersion,  
        int newVersion) {    Log.w("SQL","onUpgrade: eliminando tabla si  
        ésta existe, y creándola de nuevo");  
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);  
        onCreate(db);  
    } } }
```

Podemos ver el uso del cursor y cómo este se recorre para obtener los resultados. En este caso solo se han recogido los nombres de las ciudades. En muchas ocasiones, los adaptadores no devuelven una lista tras hacer un *select*, sino directamente el cursor; después, el código que hace uso de los métodos del adaptador tiene que recorrer el cursor.

# Video



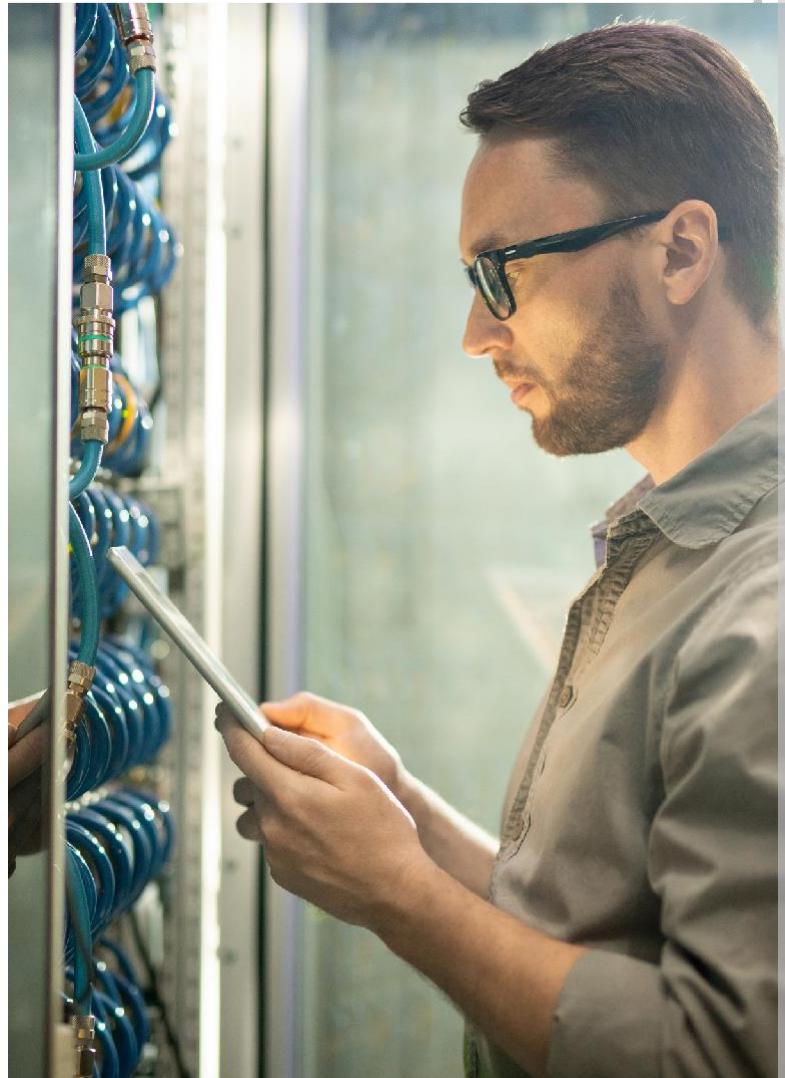
Te invitamos a ver el siguiente video:



## 3.2. Proveedores de contenido

Los proveedores de contenido o **ContentProvider** proporcionan una interfaz para publicar y consumir datos, identificando su fuente con una dirección URI que empieza por **content://**. Representan una forma más estándar que los *adapters* de desacoplar la capa de aplicación de la capa de datos.

Un adaptador es un objeto de una clase que implementa la interfaz *Adapter*. Este actúa como un enlace entre un conjunto de datos y un adaptador vista, un objeto de una clase que extiende a la clase abstracta *AdapterView*.



## 3.2. Proveedores de contenido

El conjunto de datos puede ser cualquier cosa que presente datos en una manera estructurada. Arreglos, objetos *List* y objetos *cursor* usados, por lo general, con conjuntos de datos.

Por otro lado, existe una serie de proveedores de contenidos que Android nos proporciona. Entre estos **proveedores de contenidos nativos** nos encontramos el *Browser*, *CallLog*, *ContactsContract*, *MediaStore*, *Settings*, *UserDictionary*. Para ellos, además, hay que añadir los permisos en el *AndroidManifest.xml*. Por ejemplo:

```
...
<uses-sdk android:minSdkVersion="8" />
<uses-permission android:name="android.permission.READ_
CONTACTS"/>
</manifest>
```

## 3.2. Proveedores de contenido

Accederemos al cursor de los contactos con el método:

```
ContentResolver.query(  
    Uri uri,   String[] projection,   String selection, String[]  
selectionArgs,   String sortOrder)
```

Por ejemplo, para **acceder a un cursor con la lista completa de contactos**:

```
ContentResolver cr = getContentResolver();  
Cursor cursor = cr.query(ContactContract.Contacts.CONTENT_URI,  
null, null, null, null);
```

## 3.2. Proveedores de contenido

Podemos mapear campos de un cursor con componentes GUI. En este caso, podemos hacer que cualquier cambio que ocurra en el cursor se refleje de manera automática en el componente gráfico. Esto se consigue con el método:

```
cursor.setNotificationUri(cr,  
    ContactsContract.Contacts.CONTENT_URI);
```

Para **asignar un *adapter* a una lista del GUI**, concretamente una *ListView*, utilizamos el método *setAdapter(Adapter)*:

```
ListView lv = new (ListView)findViewById(R.id.ListView01);  
SimpleCursorAdapter adapter = new SimpleCursorAdapter(  
    getApplicationContext(),  
    R.layout.textviewlayout,  
    cursor, new String[]{  
        ContactsContract.Contacts._ID,
```

## 3.2. Proveedores de contenido

```
        ContactsContract.Contacts.DISPLAY_NAME},  
        new int[]{ R.id.TextView1,  
                  R.id.TextView2});  
    lv.setAdapter(adapter);
```

En este ejemplo, los identificadores **R.id.TextView1** y **R.id.TextView2** se corresponden con *views* del *layout* que define cada fila de la *ListView*, como se puede ver en el archivo *textviewlayout.xml* que tendríamos que crear:

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/  
    android"  
        android:orientation="horizontal"  
        android:layout_width="fill_parent"  
        android:layout_height="wrap_content">  
    <TextView android:id="@+id/TextView1"
```

## 3.2. Proveedores de contenido

```
        android:textStyle="bold"  
        android:ems="2"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content">  
    </TextView>  
    <TextView android:id="@+id/TextView2"  
        android:textStyle="bold"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content">  
    </TextView>  
  </LinearLayout>
```

La lista en sí (identificada por *R.id.ListView01* en el presente ejemplo) estaría en otro *XML layout*; por ejemplo en *main.xml*.

## 3.2. Proveedores de contenido

Por otro lado, para acceder a nuestras fuentes de datos, de manera estándar, nos interesa implementar *ContentProvider* propios. Para ello, heredaremos de esta clase y sobrecargaremos una serie de métodos, como *onCreate()*, *delete(...)*, *insert(...)*, *query(...)*, *update(...)*. También, debemos sobrecargar el método *getType(Uri)*, que nos devolverá el tipo MIME de los contenidos, dependiendo de la *URI*.

La URI base la declararemos en una constante pública de nuestro proveedor y será de tipo Uri. Por ejemplo, esta podría ser:

```
public static final Uri CONTENT_URI = Uri.parse(  
    "content://es.ua.jtech.daa.proveedores/ciudades");
```

## 3.2. Proveedores de contenido

Para diferenciar entre el acceso a una única fila de datos o a múltiples filas, se pueden emplear números, por ejemplo, “1” si se accede a todas las filas; y “2” si se busca una concreta. En este último caso, también habría que especificar el *ID* de la fila que se busca.

Todas las filas: `content://es.ua.jtech.daa.proveedores/ciudades/1`

Una fila: `content://es.ua.jtech.daa.proveedores/ciudades/2/23`

Para distinguir entre una URI y otra declaramos un *UriMatcher* constante que inicializamos de forma estática en nuestra clase proveedora de contenidos:

```
private static final UriMatcher uriMatcher;
static{
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    case SINGLE_ROW: return
        uriMatcher.addURI("es.ua.jtech.daa.proveedores",
                          "ciudades", ALLROWS);
```

## 3.2. Proveedores de contenido

```
        uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades/#",
SINGLE_ROW);
    }
```

Así, dentro de cada función que sobrecarguemos primero, comprobaremos qué resultado debemos devolver, usando una estructura *switch*:

```
@Override public String getType(Uri uri) {
    switch(uriMatcher.match(uri)){
        case ALLROWS: return "vnd.ua.cursor.dir/
ciudadesprovidercontent";
        "vnd.ua.cursor.item/ciudadesprovidercontent";
        default: throw new IllegalArgumentException("URI no soportada:
"+uri); } }
```

El anterior ejemplo, del método ***getType(Uri)***, muestra una *String* en formato MIME. Esta describe el tipo de datos que muestra el argumento del URI de contenido.

## 3.2. Proveedores de contenido



El argumento *Uri* puede ser un patrón en lugar de un URI específico. La primera parte de la cadena, antes de la barra, debe terminar en *.dir*, si se trata de muchas filas; y en *.item*, si se trata de una sola.

La segunda parte de la cadena debe comenzar por *.vnd* y, a continuación, debe ir el nombre base del dominio, sin extensión. En el caso de *www.jtech.ua.es*, por ejemplo, sería *ua*. Este identificador debe ir seguido de *.cursor*, ya que los datos se devolverán en un cursor.

## 3.2. Proveedores de contenido

Por último, después de la barra, se suele indicar el nombre de la clase seguido de *content*. En este caso es: *ciudadesprovidercontent*, porque la clase se llama *CiudadesProvider*. Para poder usar el proveedor de contenidos propios, este se debe declarar en el *AndroidManifest.xml*, dentro de *application*:

```
<provider  
    android:name="CiudadesProvider"  
    android:authorities="es.ua.jtech.daa.proveedores" />
```

A continuación, la clase *CiudadesProvider* completa en el siguiente listado de código. Se observan muchas similitudes con el *adapter* para la base de datos, ya que al fin y al cabo se trata de otra interfaz diferente (pero más estándar) para acceder a los mismos datos. También, volvemos a utilizar el mismo *Helper* para abrir la base de datos:

```
package es.ua.jtech.daa.proveedores;
```

## 3.2. Proveedores de contenido

```
import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.net.Uri; import android.text.TextUtils;
import android.util.Log;
public class CiudadesProvider extends ContentProvider {
    //Campos típicos de un ContentProvider:
    public static final Uri CONTENT_URI = Uri.parse(
        "content://es.ua.jtech.daa.proveedores/ciudades");
    private static final int ALLROWS = 1;
```

## 3.2. Proveedores de contenido

```
private static final int SINGLE_ROW = 2;
private static final UriMatcher uriMatcher;
static{
    uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades", ALLROWS);
    uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades/#", SINGLE_ROW);
}
public static final String DATABASE_NAME = "mibasededatos.db";
public static final int DATABASE_VERSION = 2;
public static final String TABLE_NAME = "ciudades";
private static final String[] COLUMNAS = {"_id", "nombre", "habitantes"};
private static final String CREATE_DB = "CREATE TABLE " +
    TABLE_NAME +
```

## 3.2. Proveedores de contenido

```
“(“+COLUMNAS[0]+” INTEGER PRIMARY KEY, “  
    +COLUMNAS[1]+” TEXT, “  
    +COLUMNAS[2]+” NUMBER);  
private Context context;  
private SQLiteDatabase db;  
public class CiudadesProvider extends ContentProvider {  
    //Campos típicos de un ContentProvider:  
    public static final Uri CONTENT_URI = Uri.parse(  
        “content://es.ua.jtech.daa.proveedores/ciudades”);  
    private static final int ALLROWS = 1;  
    private static final int SINGLE_ROW = 2;  
    private static final UriMatcher uriMatcher;  
    static{  
        uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);  
    }
```

## 3.2. Proveedores de contenido

```
uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades", ALLROWS);
uriMatcher.addURI("es.ua.jtech.daa.proveedores", "ciudades/#", SINGLE_
ROW);
}
public static final String DATABASE_NAME = "mibasededatos.
db";
public static final int DATABASE_VERSION = 2;
public static final String TABLE_NAME = "ciudades";
private static final String[] COLUMNAS = {"_
id","nombre","habitantes"};
private static final String CREATE_DB = "CREATE TABLE " +
TABLE_NAME +
    "(" +COLUMNAS[0]+") INTEGER PRIMARY KEY, " +
    +COLUMNAS[1]+") TEXT, " +
    +COLUMNAS[2]+") NUMBER");
```



## 3.2. Proveedores de contenido

```
private Context context;
private SQLiteDatabase db;
@Override
public boolean onCreate() {
    this.context = getContext(); MiOpenHelper openHelper = new
MiOpenHelper(this.context);
    this.db = openHelper.getWritableDatabase();
    return true;
}
@Override
public String getType(Uri uri) {
    switch(uriMatcher.match(uri)){
        case ALLROWS: return "vnd.ua.cursor.dir/
ciudadesprovidercontent";
```

## 3.2. Proveedores de contenido

```
        case SINGLE_ROW: return "vnd.ua.cursor.item/  
ciudadesprovidercontent";  
        default: throw new IllegalArgumentException("URI no  
soportada: "+uri);  
    }  
}  
  
@Override  
public int delete(Uri uri, String selection, String[]  
selectionArgs) {  
    int changes = 0;  
    switch(uriMatcher.match(uri)){  
    case ALLROWS:  
        changes = db.delete(TABLE_NAME, selection, selectionArgs);  
        break;  
    case SINGLE_ROW:  
        String id = uri.getPathSegments().get(1);
```

## 3.2. Proveedores de contenido

```
        Log.i("SQL", "delete the id "+id);
        changes = db.delete(TABLE_NAME,
            "_id = " + id +
            (!TextUtils.isEmpty(selection) ?
            " AND (" + selection + ')' :
            ""), selectionArgs);
    break;
    default: throw new IllegalArgumentException("URI no soportada:
"+uri); }
    context.getContentResolver().notifyChange(uri, null); return
changes;
} @Override public Uri insert(Uri uri, ContentValues values) {
    long id = db.insert(TABLE_NAME, COLUMNAS[1], values);
    if(id > 0){
        Uri uriInsertado = ContentUris.withAppendedId(CONTENT_URI,
id);
```

## 3.2. Proveedores de contenido

```
        context.getContentResolver().notifyChange(uriInsertado, null);
        return uriInsertado;
    } throw new android.database.SQLException(
        "No se ha podido insertar en "+uri);    }

@Override
    public Cursor query(
        Uri uri, String[] projection, String selection,
        String[] selectionArgs, String sortOrder) {
        return db.query(TABLE_NAME, COLUMNAS, selection,
selectionArgs,
        null, null, null);
    } @Override
    public int update(Uri uri, ContentValues values, String
selection,
        String[] selectionArgs) {
```

## 3.2. Proveedores de contenido

```
    int changes = 0;
switch(uriMatcher.match(uri)){
    case ALLROWS:
        changes =db.update(TABLE_NAME, values, selection,
selectionArgs);
    break;
    case SINGLE_ROW:
        String id = uri.getPathSegments().get(1);
        Log.i("SQL", "delete the id "+id);
        changes = db.update(TABLE_NAME, values,
"_id = " + id +
(!TextUtils.isEmpty(selection) ?
    " AND (" + selection + ')'
    : ""),
selectionArgs);
```

## 3.2. Proveedores de contenido

```
        break;
default: throw new IllegalArgumentException("URI no soportada: "+uri);
    }
    context.getContentResolver().notifyChange(uri, null);
    return changes;
}
private static class MiOpenHelper extends SQLiteOpenHelper {
    MiOpenHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(CREATE_DB);
    }    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int
newVersion){
```

## 3.2. Proveedores de contenido

```
Log.w("SQL", "onUpgrade: eliminando tabla si ésta existe,"+
        " y creándola de nuevo");
db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
onCreate(db);    }    }    }
```

**Es importante también tomar en cuenta esta línea:**

```
context.getContentResolver().notifyChange(uri, null);
```

El contexto de una aplicación incluye siempre una instancia de la clase *ContentResolver*, a la cual se puede acceder mediante el método *getContentResolver*. Es necesario notificar los cambios a la clase *ContentResolver*, para así poder actualizar los cursores que lo hayan pedido a través del método *Cursor.setNotificationUri(...)*.

## ACTIVIDAD 2 Te invitamos a realizar la siguiente actividad:

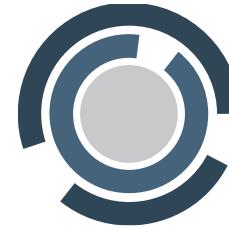
Presiona el botón para descargar la actividad:



Presiona el botón para entregar la actividad:

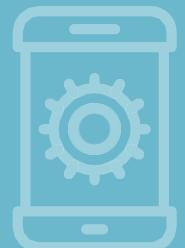


# Conclusión



El incremento en el uso de dispositivos móviles ha traído un cambio significativo en cuanto a la dinámica en la que se desenvuelven las comunicaciones en nuestra sociedad. Tan solo en unos cuantos años, un teléfono móvil ha llegado a tener una funcionalidad muy similar a una computadora de escritorio común. Actividades como revisar el correo electrónico o navegar por Internet se han vuelto más sencillas y fáciles de realizar desde un dispositivo móvil.

Los dispositivos móviles permiten acceder a una inmensa, variada y actualizada cantidad de información y conocimiento en forma inmediata. Por lo cual, ya no hay que preocuparse por la disponibilidad de la herramienta como una limitación para su aprovechamiento.



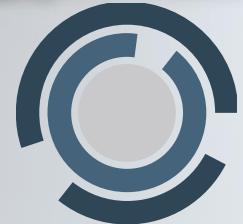


# ¡Felicitaciones!

Acabas de concluir la tercera unidad de tu curso *Desarrollo de Aplicaciones Móviles II*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.



# Unidad 4



## Servicios Avanzados



# Temario Unidad 4

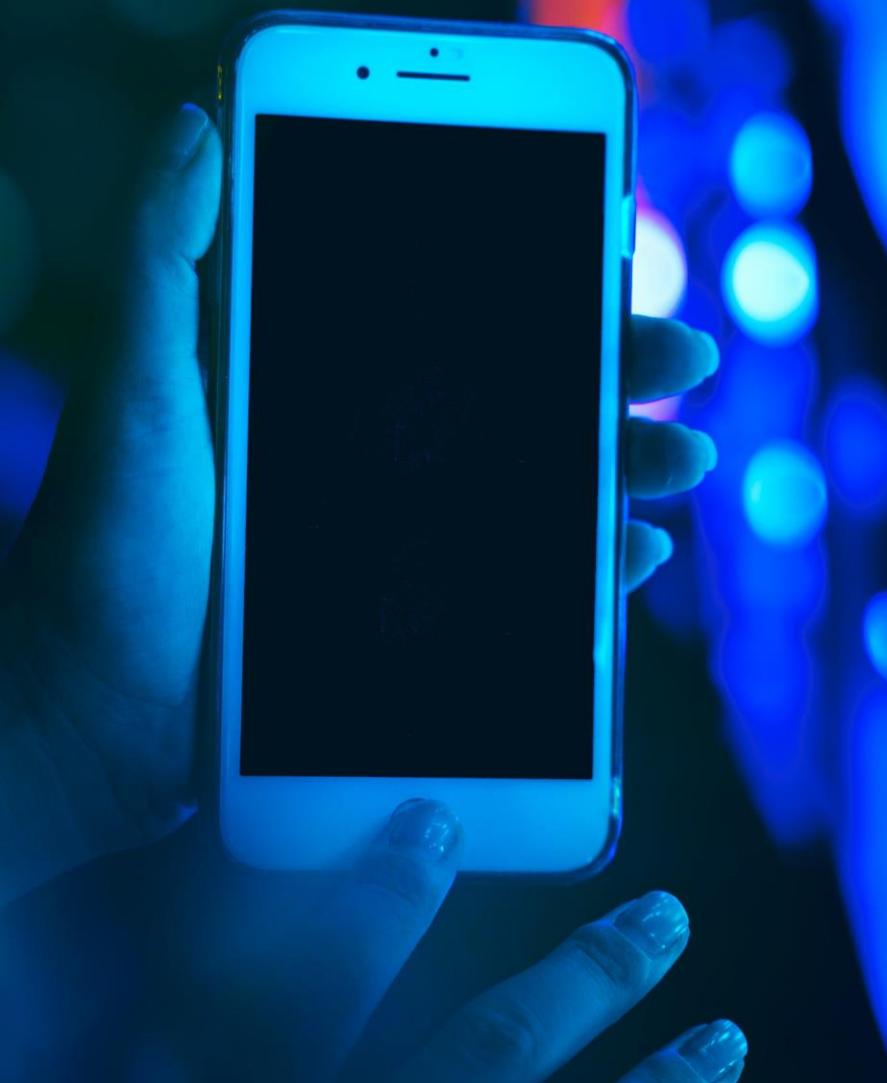
4.1

*Parsing y servicios  
de segundo plano*

4.2

*Notificaciones y widgets*



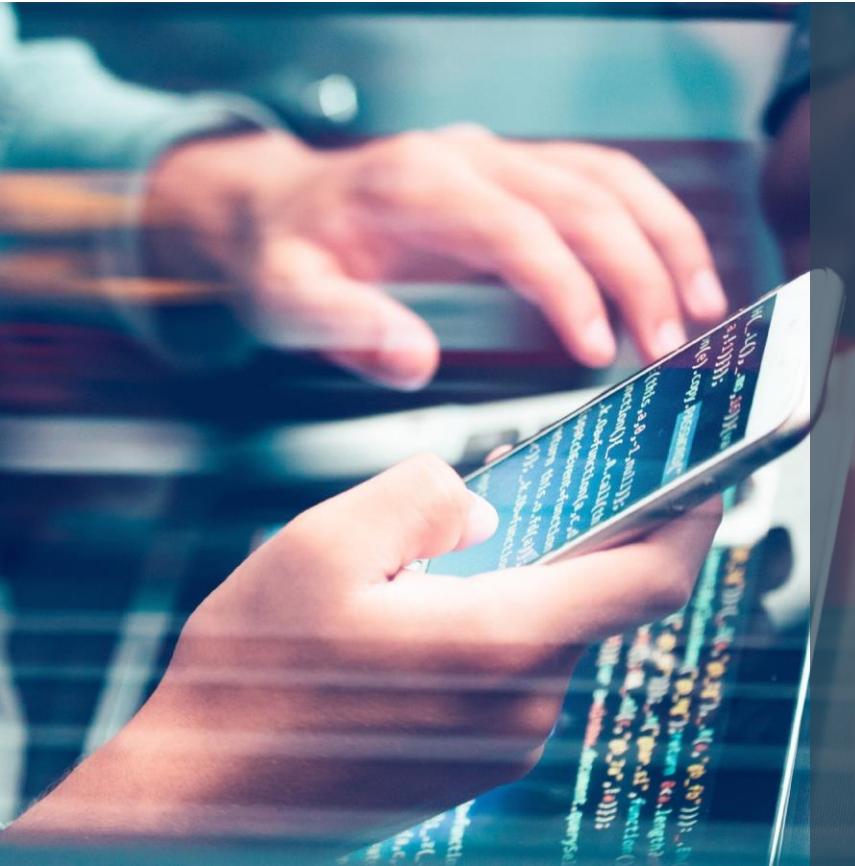


# Introducción



En esta cuarta unidad aprenderás a desarrollar aplicaciones de servicios avanzados para dispositivos móviles.

# Competencias a Desarrollar



El alumno será capaz de implementar *parsing* tanto en XML como en JSON.



El alumno será capaz de implementar notificaciones y *widgets*.



## 4.1. Parsing y servicios de segundo plano

El *parsing* sirve para transmitir información en formato XML. El ejemplo más conocido, después del HTML, son las **noticias RSS**. En este último caso, al delimitar cada campo de la noticia por tags XML permite a los diferentes clientes lectores de RSS obtener solo aquellos campos que les interese mostrar. Android nos ofrece dos maneras de *trocear* o *parsear XML*:

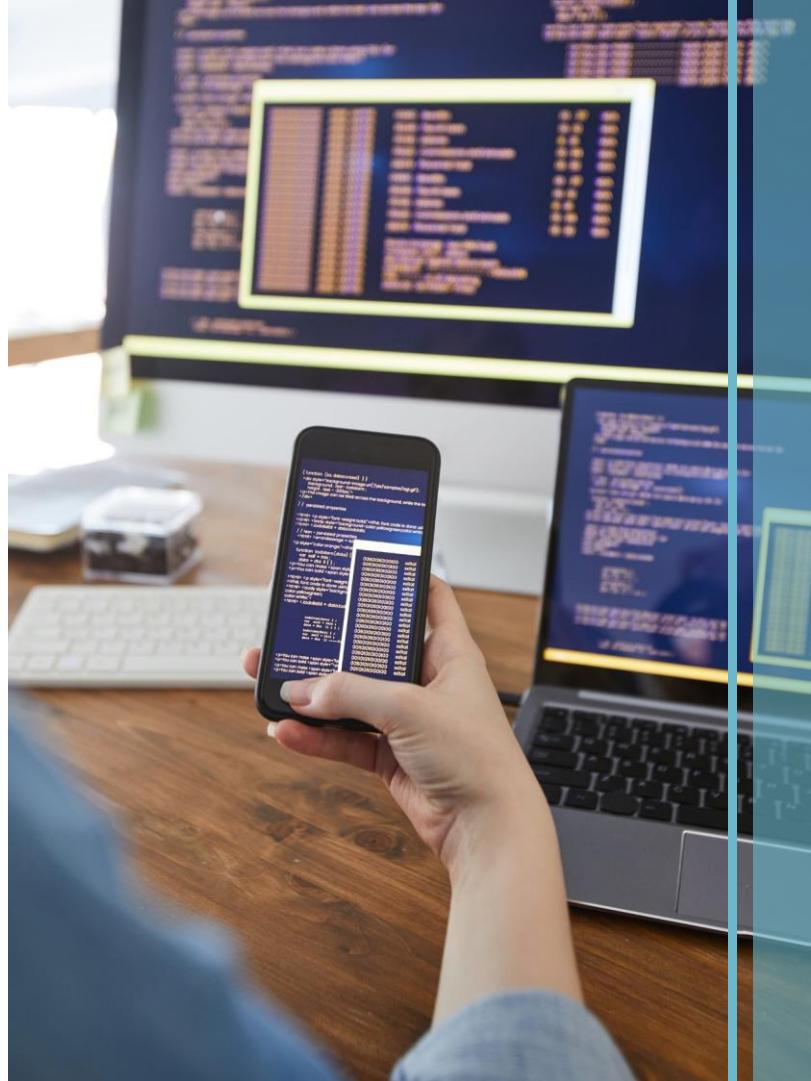
- **SAXParser.** Implementa manejadores que reaccionan a eventos tales como encontrar la apertura o cierre de una etiqueta, o encontrar atributos.



## 4.1. Parsing y servicios de segundo plano

- **XmlPullParser.** Requiere menos implementación. Este consiste en iterar sobre el árbol de XML (sin tenerlo completo en memoria) conforme el código lo va requiriendo, indicándole al parser que tome la siguiente etiqueta (método *next()*) o texto (método *nextText()*).

A continuación, se muestra un ejemplo de uso del *XmlPullParser*. Resaltando las sentencias y constantes, para observar cómo se identifican los distintos tipos de etiqueta, y si son de apertura o cierre. También se puede ver cómo encontrar atributos y su valor:



## 4.1. Parsing y servicios de segundo plano

```
try { URL text = new URL("http://www.ua.es");
    XmlPullParserFactory parserCreator =
    XmlPullParserFactory.newInstance();
    XmlPullParser parser = parserCreator.newPullParser();
    parser.setInput(text.openStream(), null);
    int parserEvent = parser.getEventType();
    while (parserEvent != XmlPullParser.END_DOCUMENT) {
        switch (parserEvent) {
        case XmlPullParser.START_DOCUMENT: break;
        case XmlPullParser.END_DOCUMENT: break;
        case XmlPullParser.START_TAG: String tag = parser.
            getName();
            if (tag.equalsIgnoreCase("title")) {
                Log.i("XML","El titulo es: "+ parser.nextText()); }else
        if(tag.equalsIgnoreCase("meta")){
            String name = parser.getAttributeValue( null, "name");
```



## 4.1. Parsing y servicios de segundo plano

```
        if(name.equalsIgnoreCase("description")){
            Log.i("XML","La descripción es:"+ parser.
getAttributeValue(
                    null,"content"));
        }
        break;
    case XmlPullParser.END_TAG: break;
}
parserEvent = parser.next();
}
} catch (Exception e) {
    Log.e("Net", "Error in network call", e);
}
```



## 4.1. Parsing y servicios de segundo plano

El ejemplo anterior serviría para imprimir en el *LogCat* el título del siguiente fragmento de página web, así como para encontrar el meta, cuyo atributo *name* sea *Description*, para mostrar el valor de su atributo *content*:

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="es"
      xml:lang="es">
  <head>
    <title>Universidad de Alicante</title>
    <meta name="Description" content="Informacion Universidad.
      Estudios,
      master, diplomados, ingenierias, facultades,
      escuelas" />
    <meta http-equiv="pragma" content="no-cache" />
    <meta name="Author" content="Universidad" />
    <meta name="Copyright" content="&copy; Universidad" />
    <meta name="robots" content="index, follow" />
```



## 4.1. Parsing y servicios de segundo plano



**JSON.** Es una herramienta potente para las aplicaciones web, ya que facilita el desarrollo y comprensión de intercambio de datos.

### Tipos de datos en JSON

- Cadenas,
- Números,
- Booleanos,
- Arrays,
- Objetos y
- Valores *null*.



## 4.1. Parsing y servicios de segundo plano

Su propósito es crear objetos que contengan varios atributos compuestos, como pares clave-valor, donde la clave es un nombre que identifique el uso del valor que lo acompaña:

```
{  
  "Id": 101  
  "Nombre": "Carlos",  
  "EstaActivo": true,  
  "Notas": [ 2.3, 4.3, 5.0]  
}
```

La anterior estructura es un objeto JSON, compuesta por los datos de un estudiante. Los objetos JSON contienen sus atributos entre llaves «{}» (al igual que un bloque de código en Javascript), donde cada atributo debe ir separado por coma ',' para diferenciar cada par. La sintaxis de los pares debe contener dos puntos ':' para dividir la clave del valor.

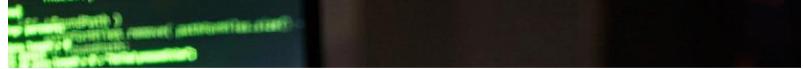


## 4.1. Parsing y servicios de segundo plano

El nombre del par debe tratarse como cadena y añadirle comillas dobles. El *Id* es de tipo entero, ya que contiene un número que representa el código del estudiante. El *Nombre* es un *string*. Este debe usar comillas dobles para ser identificado. *EstaActivo* es un tipo *booleano* que representa si el estudiante se encuentra en la institución educativa o no. Usa las palabras reservadas *true* y *false* para declarar el valor.

*Notas* es un arreglo de números reales. El conjunto de sus elementos debes incluirlos dentro de corchetes «[ ]» y separarlos por coma.





## 4.1. Parsing y servicios de segundo plano



### Usar la clase JsonReader para parsear un arreglo JSON

Debido a la integración que ha tenido la información en la nube, las aplicaciones ya no tienen límites. Si estás creando un servicio web en el cual tu usuario puede tener acceso a sus datos y características desde cualquier dispositivo, entonces el intercambio de datos con tu servidor será de gran importancia en la versión de tu aplicación Android. La idea es crear un mecanismo que permita recibir la información que contiene la base de datos externa en formato JSON hacia la aplicación.



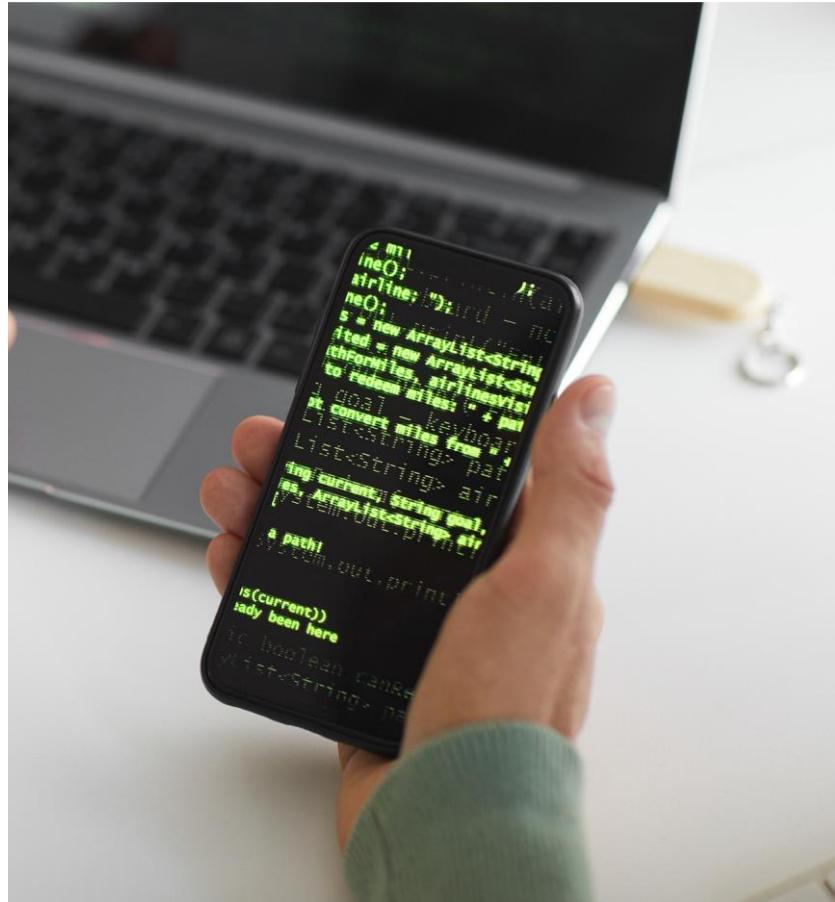
## 4.1. Parsing y servicios de segundo plano

Con ello se *parseará* cada elemento y será interpretado en forma de objeto Java para integrar correctamente el aspecto en la interfaz de usuario.

La clase ***JsonReader*** de Java es ideal para interpretar datos con formato JSON. Provee un sistema poderoso para el *parseo* de arreglos y objetos embebidos en las respuestas de los servidores. No obstante, no se puede considerar como un *parseo*, si no como una clase auxiliar para crear los propios. A continuación se muestra el ***proceso de parseo***:



## 4.1. Parsing y servicios de segundo plano



La aplicación Android a través de un cliente realiza una petición *GET* a la dirección URL del recurso con el fin obtener los datos. Ese flujo entrante debe interpretarse con ayuda de un *parser* personalizado que implementa la clase *JsonReader*.

El resultado final es un conjunto de datos adaptable al API de Android. Dependiendo de tus necesidades, puedes convertirlos en una lista de objetos estructurados que alimenten un adaptador que pueble un *ListView* o simplemente actualizar la base de datos local de tu aplicación en SQLite.



## 4.1. Parsing y servicios de segundo plano

Con ello se parseará cada elemento y será interpretado en forma de objeto Java para integrar correctamente el aspecto en la interfaz de usuario.

La clase **JsonReader** de Java es ideal para interpretar datos con formato JSON. Provee un sistema poderoso para el parseo de arreglos y objetos embebidos en las respuestas de los servidores. No obstante, no se puede considerar como un *parseo*, si no como una clase auxiliar para crear los propios. A continuación se muestra el **proceso de parseo**:



## 4.1. Parsing y servicios de segundo plano

```
[ {  
    "especie": "Leu00f3n",  
    "descripcion": "El leu00f3n (Panthera leo) es un mamu00edfero  
    carnu00edvoro ...",  
    "imagen": "leon"  
}, {  
    "especie": "Elefante",  
    "descripcion": "Los elefantes o elefu00e1ntidos (Elephantidae)  
    son ...",  
    "imagen": "elefante"  
}, {  
    "especie": "Cocodrilo",  
    "descripcion": "Los crocodu00edlidos (Crocodylidae) son una  
    familia de sauru00f3psidos ...",  
    "imagen": "cocodrilo"  
}, {
```



## 4.1. Parsing y servicios de segundo plano

```
        "especie":"Cebra",
        "descripcion":"Se conocen como cebra (o zebra, grafu00eda en
desuso ) a tres ...",
        "imagen":"cebra"
    }, {
        "especie":"u00c1guila",
        "descripcion":"El u00e1guila calva (Haliaeetus leucocephalus),
tambiu00e9n ...",
        "imagen":"aguila"
    }
]
```

Como podemos observar, Zoowak solo tiene una actividad, en la que se encuentra un *ListView*, por lo que se deduce que la salida del proceso de *parsing* debe ser de tipo estructurado. En este caso, es una pieza de código personalizada llamada *Animal*.

## 4.1. Parsing y servicios de segundo plano

Dicho elemento debe converger en el diseño establecido para cada ítem de la lista, así que el adaptador debe inflar elementos con base en dicha configuración.

A continuación, vamos a crear una clase personalizada llamada ***JsonAnimalParser***, la cual interpretará un flujo de datos con formato JSON, y retornará en una lista de objeto de tipo *Animal*. La definición de esta clase es solo un reflejo en Java de los atributos de los objetos JSON que vienen desde el servidor. Veamos:

```
public class Animal {  
    private String especie;  
    private String descripcion;  
    private String imagen;  
    public Animal(String especie,  
    String descripcion, String imagen) {
```



## 4.1. Parsing y servicios de segundo plano

```
        this.especie = especie;
        this.descripcion = descripcion;
        this.imagen = imagen; }
public String getEspecie() {
    return especie; }
public void setEspecie(String especie) {
    this.especie = especie; }
public String getDescripcion() {
    return descripcion; }
public void setDescripcion(String descripcion) {
    this.descripcion = descripcion; }
public String getImagen() {
    return imagen; }
public void setImagen(String imagen) {
    this.imagen = imagen; } }
```



## 4.1. Parsing y servicios de segundo plano

El formato *Json* contiene los campos: *especie*, *descripción* e *imagen* de la base de datos que está en el servidor externo. Sabiendo sus claves, es posible interpretar cada objeto del arreglo. Cabe destacar que el código *Json* viene formateado con el estándar *UTF-8*; por eso se ven tantos códigos en donde van las tildes. Esto nos permite conservar el acento de nuestros datos de texto sin ninguna dificultad.

Para leer el **formato general *Json***, primero se debe implementar un método que coordine el retorno de los objetos de tipo *animal*, y que además cree una instancia del *JsonReader*. Veamos:

```
public List<Animal> readJsonStream(InputStream in) throws IOException {  
    // Nueva instancia  
    JsonReader reader = new JsonReader(new InputStreamReader(in,  
        "UTF-8"));
```



## 4.1. Parsing y servicios de segundo plano

```
try {
    return leerArrayAnimales(reader); // Leer Array
} finally {
    reader.close(); } }
```

El tipo de retorno de la función es una lista de animales, ya que es justo el tipo de estructura que recibirá nuestro adaptador. Además de ello, se crea el lector *Json*, asociando el flujo de entrada y el tipo de encriptado que usarás; en este caso UTF-8. Con ello, los acentos serán interpretados automáticamente. Si todo finalizó con éxito, entonces el lector se cierra con *close()* para limpiar espacios de memoria referenciada.

Por otra parte, para leer el **array de objetos Json**, se creó la función *leerArrayAnimales()*, para recorrer todo el array enviado desde el servidor. Recibe como parámetro el lector *Json* para continuar la labor de lectura.



## 4.1. Parsing y servicios de segundo plano

Luego, se apunta al primer elemento del *array* con el método *beginArray()*, lo que nos permite leer con un bucle *while* el siguiente elemento con el método *hasNext()*:

```
public List leerArrayAnimales(JsonReader reader) throws IOException {  
    ArrayList animales = new ArrayList(); // Lista temporal  
    reader.beginArray();  
    while (reader.hasNext()) { // Leer objeto  
        animales.add(leerAnimal(reader)); }  
        reader.endArray();  
    return animales; }
```

**hasNext()** pregunta al *array* si existen más objetos; si la respuesta es positiva, se añaden los elementos a una lista temporal llamada *animales*, a través del método *leerAnimal()*. Cuando se acaben los objetos, entonces se debe liberar la memoria, cerrando el *array* con el método *endArray()*. Finalmente, se retorna la lista temporal con todos los animales.



## 4.1. Parsing y servicios de segundo plano

Por su parte, para leer los **atributos de cada objeto Json**, el método *leerAnimal()* accede a todos pares clave-valor que convirtió el lector *JsonReader* en datos interpretables. Así que, aprovechando esa estructura definida, es posible recorrer cada objeto particular en busca de los valores que se refieran a una clave:

```
public Animal leerAnimal(JsonReader reader) throws IOException {  
    String especie = null;  
    String descripcion = null;  
    String imagen = null;  
    reader.beginObject();  
    while (reader.hasNext()) {  
        String name = reader.nextName();  
        switch (name) {  
            case "especie":
```

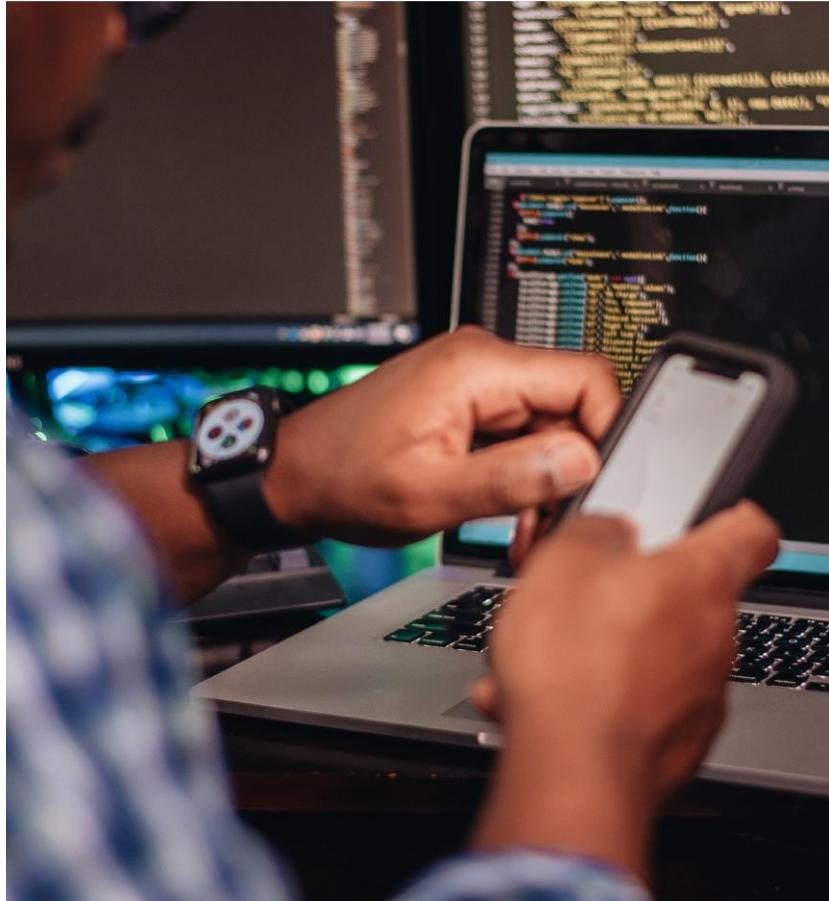


## 4.1. Parsing y servicios de segundo plano

```
        especie = reader.nextString();
                break;
        case "descripcion":
                descripcion = reader.nextString();
                break;
        case "imagen":
                imagen = reader.nextString();
                break;
        default: reader.skipValue();
                break;
    }
}
reader.endObject();
return new Animal(especie, descripcion, imagen); }
```



## 4.1. Parsing y servicios de segundo plano



Similar a la situación del `array`, debemos iniciar el objeto con `beginObject()`, para apuntar al primer par. Luego, comenzamos un bucle para la lectura de todos los pares con el método análogo `hasNext()`. Con este, se obtiene la clave del par con el método `nextName()`, y se asigna a una variable local. Ya con ese valor, es posible abrir una estructura `switch` que relacione todos los valores de las claves posibles que tiene tu objeto. En cada comparación debes usar los métodos `next()` para obtener el valor, (dependiendo del tipo) y asignarlo a una variable temporal.



## 4.1. Parsing y servicios de segundo plano

Si el valor es *String*, entonces se usará *nextString()*; si es de tipo *int*, entonces usas *nextInt()*, y así sucesivamente.

En el caso default puedes usar el método *skipValue()*, el cual permite saltar valores que no te interesan. Si de pronto hubiésemos consultado el *ID* del animal, entonces se usaría este método para evitar su lectura, ya que no interesa en el proceso actual.

Por último, retorna en un nuevo objeto *Animal* que alimente su constructor de todas las variables temporales obtenidas.





## 4.1. Parsing y servicios de segundo plano



Los servicios en **segundo plano** son similares a los *demonios* de los sistemas GNU/Linux. No necesitan una aplicación abierta para seguir ejecutándose. Sin embargo, para el control de un servicio (iniciarlo, detenerlo, configurarlo) sí que es necesario programar aplicaciones con sus actividades con interfaz gráfica.

Los servicios heredan de la clase *Service* e implementan obligatoriamente el **método *onBind(Intent)***.

## 4.1. Parsing y servicios de segundo plano

Este método sirve para comunicación entre servicios y necesita que se defina una interfaz AIDL (*Android Interface Definition Language*). Devolviendo *null*, estamos indicando que no implementamos tal comunicación:

```
public class MiServicio extends Service {  
    @Override  
    public void onCreate() {  
        super.onCreate();  
        //Inicalizaciones necesarias  
    }  
    @Override    public int  
onStartCommand(Intent intent, int flags, int  
startId) {  
        //Comenzar la tarea de segundo plano  
        return Service.START_STICKY;    }  
}
```



## 4.1. Parsing y servicios de segundo plano

```
@Override    public void onDestroy() {  
    super.onDestroy();  
    //Terminar la tarea y liberar recursos  
}  
@Override  
public IBinder onBind(Intent arg0) {  
    return null;  
} }
```

Si el servicio se encontrara declarado dentro de otra clase, el *android:name* contendría: **.MiOtraClase\$MiServicio**.

El ciclo de vida del servicio empieza con la ejecución del método *onCreate()*; después se invoca al método *onStartCommand(...)*, y finalmente al detener el servicio se invoca al método *onDestroy()*.



# Video



Te invitamos a ver el siguiente video:



## 4.2. Notificaciones y widgets

El típico mecanismo de comunicación con el usuario que los Servicios utilizan son las **Notifications**. Se trata de un mecanismo mínimamente intrusivo que no roba el foco a la aplicación actual, y que permanece en una lista de notificaciones en la parte superior de la pantalla, que el usuario puede desplegar cuando le convenga.

Para trabajar mostrar y ocultar notificaciones hay que obtener de los servicios del sistema el *NotificationManager*. Su método **notify(int, Notification)** muestra la notificación asociada a determinado identificador.



## 4.2. Notificaciones y widgets

```
Notification notification;
NotificationManager notificationManager;
notificationManager = (NotificationManager)
getSystemService(
Context.NOTIFICATION_SERVICE);
notification = new Notification(R.drawable.icon,
        “Mensaje evento”, System.
currentTimeMillis());
notificationManager.notify(1, notification);
```



## 4.2. Notificaciones y widgets

El **identificador** sirve para actualizar la notificación en un futuro (con un nuevo aviso de notificación al usuario). Si se necesita añadir una notificación más, manteniendo la anterior, hay que indicar un nuevo *ID*. Esto para actualizar la información de un objeto *Notification* ya creado.

```
notification.setLatestEventInfo(getApplicationContext(),
    "Texto", contentIntent);
```

Donde *contentIntent* es un *Intent* para abrir la actividad a la cuál se desea acceder al pulsar la notificación. Es típico usar las notificaciones para abrir la actividad que nos permita reconfigurar o parar el servicio. También es típico que al pulsar sobre la notificación y abrirse una actividad, esta desaparezca. Dicha acción la podemos implementar en el método *onResume()* de la actividad:

```
@Override protected void onResume() {
    super.onResume();
    notificationManager.cancel(MiTarea.NOTIF_ID); }
```



## 4.2. Notificaciones y widgets

A continuación, se muestra un ejemplo completo de notificaciones usadas por una tarea **AsyncTask**, que sería fácilmente integrable con un Service. Solo haría falta crear una nueva *MiTarea* en *Service.onCreate()*, y arrancarla con *miTarea.execute()* desde *Service.onStartCommand(...)*, para después detenerla con *miTarea.cancel()* desde *Service.onDestroy()*:

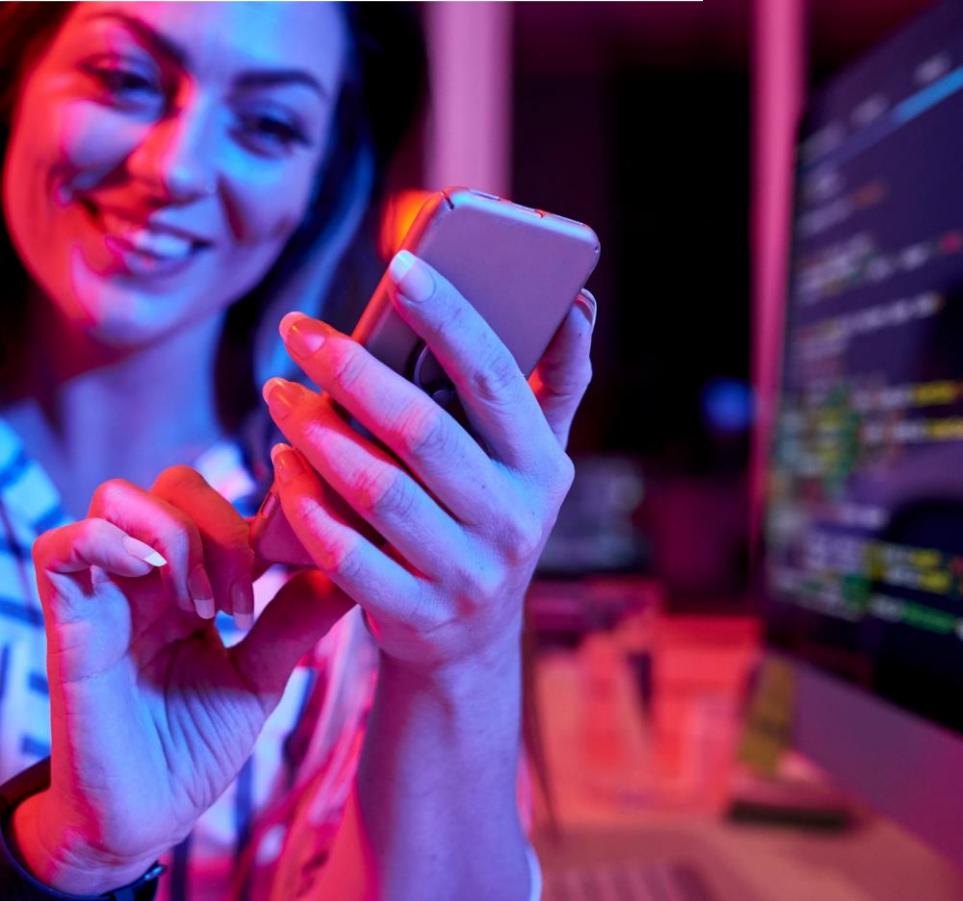
```
private class MiTarea extends AsyncTask<String, String, String>{  
    public static final int NOTIF1_ID = 1;  
    Notification notification;  
    NotificationManager notificationManager;  
    @Override protected void onPreExecute() {  
        super.onPreExecute();  
        notificationManager = (NotificationManager) getSystemService(  
            Context.NOTIFICATION_SERVICE);  
        notification = new Notification(R.drawable.icon, “Mensaje  
        evento”, System.currentTimeMillis());    }  
}
```



## 4.2. Notificaciones y widgets

```
@Override protected String doInBackground(String... params) {  
    while(condicionSeguirEjecutando){  
        if(condicionEvento){  
            publishProgress("Información del evento");  
        }  
        return null;    }  
  
    @Override protected void onProgressUpdate(String... values) {  
        Intent notificationIntent = new Intent(  
            getApplicationContext(), MiActividadPrincipal.class);  
        PendingIntent contentIntent = PendingIntent.getActivity(  
            getApplicationContext(), 0, notificationIntent, 0);  
        notification.setLatestEventInfo(getApplicationContext(),  
            values[0], contentIntent);  
        notificationManager.notify(NOTIF_ID, notification);    }  
}
```

## 4.2. Notificaciones y widgets



Los **widgets**, que desde el punto de vista del programador son *AppWidgets*, son pequeñas interfaces de programas Android que permanecen en el escritorio del dispositivo móvil.

Los *AppWidgets* ocupan determinado tamaño y se refrescan con determinada frecuencia, datos que hay que declarar en el *XML* que define el *widget*.



## 4.2. Notificaciones y widgets

Se puede añadir como nuevo recurso *XML* de Android, y seleccionar el tipo *Widget*. Lo coloca en la carpeta *res/xml/*. Por ejemplo, este es el *res/xml/miwidget.xml*:

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider xmlns:android=http://schemas.android.
com/apk/res/android
    android:minWidth="146dip"
        android:minHeight="72dip"
        android:updatePeriodMillis="600000"
        android:initialLayout="@layout/miwidget_layout"
    />
```

Este *XML* declara que el *Layout* del *widget* se encuentra en *res/layout/miwidget\_layout.xml*.



## 4.2. Notificaciones y widgets

Los AppWidgets no necesitan ninguna actividad, sino una clase que herede de *AppWidgetProvider*. Por tanto en el *AndroidManifest.xml* ya no necesitamos declarar una actividad principal. Lo que tenemos que declarar es el *widget*:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=http://schemas.android.com/apk/res/android
package="es.ua.jtech.ajdm.appwidget"
        android:versionCode="1" android:versionName="1.0">
<application android:icon="@drawable/icon" android:label="@string/
app_name">
        <receiver android:name=".MiWidget" android:label="Mi
Widget">
                <intent-filter>    <action android:name="android.
appwidget.action.APPWIDGET_UPDATE" />
                </intent-filter>
```

## 4.2. Notificaciones y widgets

```
<meta-data android:name="android.appwidget.provider"  
    android:resource="@xml/miwidget" />  
</receiver>  
<service android:name=".MiWidget$UpdateService" />  
</application> <uses-sdk android:minSdkVersion="8" /> </br>  
<manifest>
```

También puede ser necesario declarar, además de los permisos que se requieran, el uso del servicio que actualice el *widget* o que realice alguna tarea en *background*. En el anterior *manifest* se declara el servicio *UpdateService* dentro de la clase *MiWidget*. A continuación, se muestra un ejemplo de cómo clase *MiWidget* que implementa un *widget*:

```
public class MiWidget extends AppWidgetProvider {  
    @Override public void onUpdate(Context context,  
        AppWidgetManager appWidgetManager,  
        int[] appWidgetIds) { // Inicio de nuestro servicio de  
        actualización:
```



## 4.2. Notificaciones y widgets

```
        context.startService(new Intent(context,
UpdateService.class));    }
public class MiWidget extends AppWidgetProvider {
@Override public void onUpdate(Context context, AppWidgetManager
appWidgetManager,
        int[] appWidgetIds) {
// Inicio de nuestro servicio de actualización:
context.startService(new Intent(context, UpdateService.
class));    }
public static class UpdateService extends Service {
@Override public int onStartCommand(Intent intent, int flags, int
startId) {
        RemoteViews updateViews = new RemoteViews(
getPackageName(), R.layout.miwidget_layout);
//Aqui se actualizarían todos los tipos de Views que
hubiera:
        updateViews.setTextViewText(R.id.TextView01,
“Valor con el que refrescamos”); // ...
}
}
```

## 4.2. Notificaciones y widgets

```
//Además, ¿qué hacer si lo pulsamos? Lanzar alguna actividad:  
    Intent defineIntent = new Intent(...);  
    PendingIntent pendingIntent = PendingIntent.getActivity(  
        getApplicationContext(), 0, defineIntent, 0);  
updateViews.setOnClickPendingIntent(R.id.miwidget,pendingIntent);  
    //Y la actualización del widget con el updateViews  
creado:  
    ComponentName thisWidget = new ComponentName(  
        this, MiWidget.class);  
    AppWidgetManager.getInstance(this).updateAppWidget(  
        thisWidget, updateViews);  
    return Service.START_STICKY;  
} @Override  
public IBinder onBind(Intent intent) {  
    return null;  
} } }
```



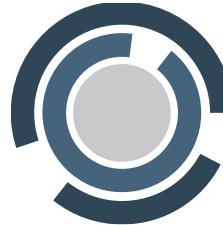
## 4.2. Notificaciones y widgets



La actualización se realiza por medio de la clase *RemoteViews*, con métodos como **.setTextViewText(String)**, **.setImageviewBitmap(Bitmap)**, etc., con el propósito de actualizar los valores del *layout* indicado en la creación del *RemoteViews*, *R.layout.miwidget\_layout* en este caso.

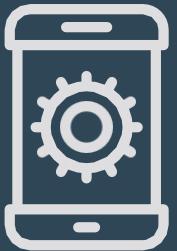


# Conclusión



En la actualidad, los sistemas celulares han ido evolucionando rápidamente. Tan solo en una década se han convertido en el sistema de comunicación más importante a nivel mundial. Esta evolución ha surgido para satisfacer todas las necesidades demandadas por los usuarios, los cuales exigen cada vez mayores servicios.

En esta unidad se revisó cómo implementar el *parsing*, los servicios de segundo plano, notificaciones y *widgets*, funciones y servicios indispensables en toda aplicación móvil de alta calidad.





# ¡Felicitaciones!

Acabas de concluir la *cuarta unidad* de tu curso *Desarrollo de Aplicaciones Móviles II*. Te invitamos a finalizar este esfuerzo realizando el examen parcial correspondiente. Para ello, debes regresar a la pantalla principal y dar clic en *Presentar examen*.





## Bibliografía

- J. Tomás Girones. (2012). *El Gran Libro de Android*. 2da edición. Alfaomega.
- J. E. Amaro Soriano. (2013). *El Gran Libro de Programación Avanzada de Android*. Primera edición. Alfaomega.
- J. Tomas, V. Carbonell, C. Vogt, M. García Pineda, J. Bataller M., y D. Ferri. (2013). *El Gran Libro Android Avanzado*. Primera edición. Marcombo.
- J. D. Luján Castillo. (2017). *Android Studio Aprende a Desarrollar Aplicaciones*. Primera edición. Alfaomega.
- S. Gómez Oliver. (2011). *Curso Programación Android*. Primera edición. [www.sgoliver.net](http://www.sgoliver.net):
- A. Catalán. (2011). *Curso Android: Desarrollo de Aplicaciones Móviles*. Primera edición. Maestros del web.

# PROYECTO FINAL



# Proyecto final



Te invitamos a realizar el siguiente proyecto final:

Presiona el botón para descargar el proyecto final:



Presiona el botón para entregar el proyecto final:

