Localización en Android y Google Maps API v2

Arturo Macías Grande
2º DAM

IES San Andrés del Rabanedo

Índice

1. Localización en Android

La localización en Android se puede obtener a partir de diversos medios. Los más comunes son:

- GPS
- A través de la red de telefonía
- A través de puntos de acceso WIFI

Se conocen como proveedores de localización o **location providers.**

1.1 ¿Cómo conocer los proveedores al alcance?

La clase "estrella" en la localización en Android es la clase **LocationManager**. A través de ella y con el uso de su método ***getAllProviders()***, se podrá conocer todos los proveedores de posicionamiento disponibles.
Un ejemplo de código para ello:

```
LocationManager locMan = (LocationManager) getSystemService(LOCATION_SERVICE);
List<String> providerList = locMan.getAllProviders();
```

Los proveedores de posición tienen características distintas, ya sea a nivel de exactitud, requerimientos de energía, información disponible, etc.
A través de un objeto de la clase **LocationProvider** y de los métodos asociados a ella se puede obtener esta información:

```
LocationProvider provider = locMan.getProvider(providerList.get(0));
int exactitud = provider.getAccuracy();
boolean ofreceAltitud = provider.supportsAltitude();
int recursos = provider.getPowerRequirement();
```

Más info Clase LocationProvider:

http://developer.android.com/reference/android/location/LocationProvider.html

1.2 ¿Cómo saber cuáles cumplen los criterios que se necesitan?

Para ello se utiliza la clase **Criteria,** con la que se establecen los requisitos mínimos que se necesitan para el uso de un proveedor de localización. En el ejemplo, tres requisitos

```
Criteria requirements = new Criteria();
//Se fijan los requerimientos de posicionamiento
requirements.setSpeedRequired(true);//Para que proveea información de velocidad
requirements.setAccuracy(Criteria.ACCURACY_LOW);//Baja exactitud
requirements.setPowerRequirement(Criteria.POWER_LOW);//Bajo requerimiento energético
```

A continuación los métodos ***getProviders()*** *o* ***getBestProvider()*** mostrarán la lista de proveedores que mejor se ajustan a los criterios definidos o el proveedor que mejor se ajusta a dicho criterio, respectivamente. El segundo parámetro de tipo boolean indica si se quiere que sólo devuelvan los proveedores activados en ese momento.

Más info Clase **Criteria:**

http://developer.android.com/reference/android/location/Criteria.html

```
//Mejor proveedor con los requerimientos establecidos
String mejorProviderCrit = locMan.getBestProvider(requirements, false);

//Lista de proveedores con los criterios establecidos
List<String> listaProvidersCrit = locMan.getProviders(requirements, false);
```

1.3 Localización con GPS

El grueso de este trabajo será el uso de las localizaciones mediante GPS, por lo que será en el que se centre a partir de ahora.

A la hora de conocer si el GPS está activo se utiliza el método ***isProviderEnabled(String nombreProveedor);*** aplicable sobre un objeto de la clase **LocationManager**. Este objeto se crea a partir de la siguiente línea de código:

```
//Obtenemos una referencia al LocationManager
locManager = (LocationManager)getSystemService(Context.LOCATION_SERVICE);
```

La siguiente imagen muestra la comprobación de estado del proveedor:

```
//Comprobación inicial. Si el GPS está desactivado lanza un toast
if (!locManager.isProviderEnabled(LocationManager.GPS_PROVIDER)) {
    Toast.makeText(this, "GPS desactivado", Toast.LENGTH_SHORT).show();
}
```

Una vez se active el GPS en el dispositivo, el quid de la cuestión es ¿Cómo se obtiene la posición en la que se encuentra?.
Existen diversos métodos que ayudarán a dar respuesta a esta pregunta.

El primero de ellos es ***getLastKnownLocation(String provider),*** aplicable de nuevo sobre un objeto de la clase **LocationManager,** y cuyo se resultado se guarda en un objeto de la Clase **Location.**

```
//Obtenemos la ltima posici n conocida
Location loc = locManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);
```

Este método devuelve la última posición conocida devuelta por el proveedor que se le pasa como parámetro. Es decir, con él no se conoce la posición actual sino la última que recogió dicho proveedor, en caso de haber estado funcionando en alguna ocasión. En caso contrario no devolverá ninguna posición.

Para obtener la posición actual se opera de la siguiente manera. Se activará el proveedor de localización (GPS en este caso) y se recogerán las notificaciones generadas cuando se cambie de posición.

En términos de Android, el funcionamiento consistiría en suscribirse al evento que el proveedor lanza cada vez que recoge una notificación de cambio de posición.

Habrá que indicar el periodo de tiempo o la distancia que marca el periodo entre recepción y recepción.

El método que se utiliza para ello es ***requestLocationUpdates(provider, minTime, minDistance, listener).*** En la imagen un ejemplo:

```
locManager.requestLocationUpdates(
        LocationManager.GPS_PROVIDER, 30000, 0, locListener);
```

Este método recibe cuatro parámetros:

- **Nombre del proveedor** de localización al que se suscribe.
- **Tiempo mínimo** entre actualizaciones, en milisegundos.
- **Distancia mínima** entre actualizaciones, en metros.
- Instancia de un objeto **LocationListener,** implementado previamente para definir las acciones a realizar al recibir cada nueva actualización de la posición.

Si el **tiempo mínimo y/o la distancia mínima** se establecen con valor cero, estos criterios no se tendrán en cuenta, por lo que las actualizaciones se recibirán tan pronto como estén disponibles.

En cuanto al tiempo mínimo es importante tener en cuenta una cuestión. Ese tiempo, en el ejemplo 30 segundos, no implicaría que tengamos una lectura de posición cada 30 segundos, sino que al menos habrá una lectura nueva cada 30 segundos. Por lo que en ese periodo de tiempo el receptor GPS estará registrando numerosas posiciones.

El objeto **LocationListener**, que se le pasa al método anteriormente mencionado , tendrá una serie de métodos definidos, en los que se indicarán qué acciones llevar a cabo cuando se reciba una nueva actualización. Los métodos son los siguientes:

- ***onLocationChanged(location).*** Lanzado cada vez que se recibe una actualización de la posición.

- ***onProviderDisabled(provider).*** Lanzado cuando el proveedor se deshabilita.
- ***onProviderEnabled(provider).*** Lanzado cuando el proveedor se habilita.
- ***onStatusChanged(provider, status, extras).*** Lanzado cada vez que el proveedor cambia su estado, que puede variar entre OUT_OF_SERVICE, TEMPORARILY_UNAVAILABLE, AVAILABLE.

En la siguiente imagen se ve un ejemplo de implementación de **LocationListener**:

```
//Nos registramos para recibir actualizaciones de la posición
locListener = new LocationListener() {
    public void onLocationChanged(Location location) {
        mostrarPosicion(location);
    }
    public void onProviderDisabled(String provider){
        lblEstado.setText("Provider OFF");
    }
    public void onProviderEnabled(String provider){
        lblEstado.setText("Provider ON ");
    }
    public void onStatusChanged(String provider, int status, Bundle extras){
        Log.i("", "Provider Status: " + status);
        lblEstado.setText("Provider Status: " + status);
    }
};
```

En el caso anteriormente descrito cabe mencionar el método ***onLocationChanged(Location location)***, que será de los más significativos, pues responde a los cambios de posición. En este caso llama a un método propio ***mostrarPosicion(location)***, qué básicamente, asignará valores a los elementos existentes en el layout de la aplicación:

```
private void mostrarPosicion(Location loc) {
    if(loc != null)
    {
        lblLatitud.setText("Latitud: " + String.valueOf(loc.getLatitude()));
        lblLongitud.setText("Longitud: " + String.valueOf(loc.getLongitude()));
        lblExactitud.setText("Exactitud: " + String.valueOf(loc.getAccuracy()));
        Log.i("", String.valueOf(loc.getLatitude() + " - " + String.valueOf(loc.getLongitude())));
    }
    else
    {
        lblLatitud.setText("Latitud: (sin_datos)");
        lblLongitud.setText("Longitud: (sin_datos)");
        lblExactitud.setText("Precision: (sin_datos)");
    }
}
```

Como apunte de este método, destacar la posibilidad de que la localización sea **null**. Esto se puede deber a varias causas, por ejemplo que la última localización conocida no exista, por no haberse activado nunca el GPS. También puede ser que en un primer momento no esté disponible ninguna posición.

La aplicación que acompaña este tutorial, tiene un botón **Activar**, que hará una llamada al método ***comenzarLocalizacion()*** que si bien, ya se ha ido desgranando arriba en alguno de los pasos, se puede ver completo en la siguiente imagen:

```
private void comenzarLocalizacion()
{
    //Obtenemos una referencia al LocationManager
    locManager = (LocationManager)getSystemService(Context.LOCATION_SERVICE);

    //Obtenemos la última posición conocida
    Location loc = locManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);

    //Mostramos la última posición conocida
    mostrarPosicion(loc);

    //Nos registramos para recibir actualizaciones de la posición
    locListener = new LocationListener() {

    locManager.requestLocationUpdates(
            LocationManager.GPS_PROVIDER, 30000, 0, locListener);
}
```

Para desactivar la recepción de nuevas posiciones, que en el ejemplo se lleva a cabo al pulsar el botón **Desactivar**, el cual ejecuta el siguiente método:

```
btnDesactivar.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        locManager.removeUpdates(locListener);
    }
});
```

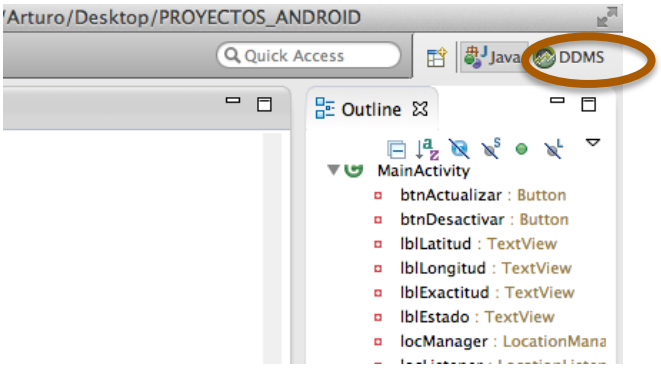
Con el método ***removeUpdates(LocationListener loclistener)*** se consigue paralizar la obtención de nuevas posiciones por parte del proveedor.

1.4 Simulación de recepción de posiciones en el emulador

Como se puede deducir, el emulador del entorno de desarrollo no puede captar una posición real. Por esa razón, muy probablemente, la primera vez que se lance la aplicación en el emulador no ofrecerá ninguna posición.

Para solventar esto, Eclipse tiene una serie de herramientas que permitirán emular los cambios de posición.

Para ello se debe cambiar la perspectiva de uso en Eclipse, pulsando en **DDMS,** en la esquina superior izquierda:

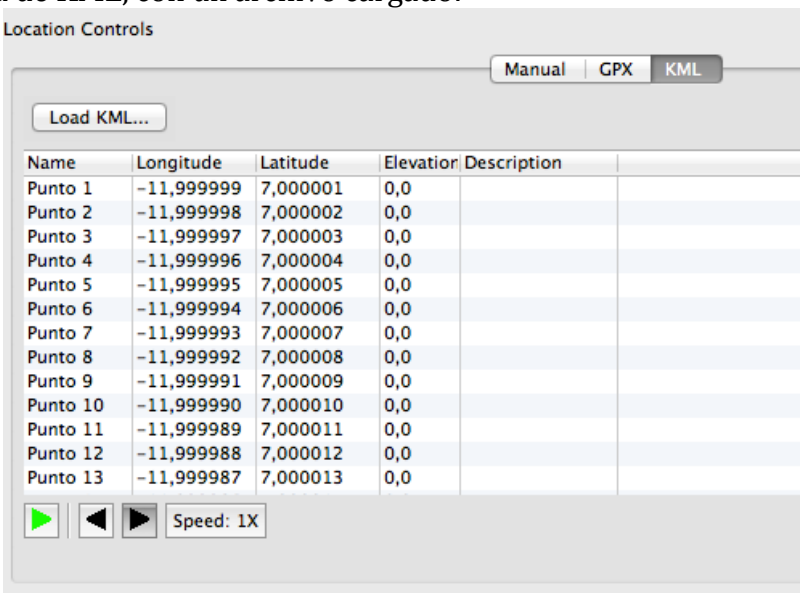


A continuación se pincha en **Emulator Control**, y se trabaja con sección **Location Controls.**

Existen tres posibilidades para enviar localizaciones, como se ve en la imagen:

- **Manual:** una vez fijadas la longitud y latitud, tras pulsar en el botón *Send*, se habrá enviado un cambio de posición al emulador. Al nivel interno esto implica la invocación del método ***onLocationChanged()***, por lo que se mostrarán en la interfaz los nuevos datos de posición.
- **GPX y KML:** se podría considerar "análogo" el funcionamiento de ambos. Consistiría en pasar un archivo que almacena una lista de coordenadas. Los archivos de tipo **.gpx** son aquellos que se generan en la mayoría de dispositivos GPS, mientras que los **.kml** son aquellas que se pueden generar en Google earth tras levantar unos puntos y guardarlos en un archivo.

En este tutorial se ha adjuntado un archivo llamado **localizaciones_android.kml** que contiene una serie de coordenadas para hacer pruebas.
La interfaz de las pestañas **GPX** y **KML** son muy similares. En la siguiente imagen se ve la de **KML**, con un archivo cargado:



Existen cuatro controles distintos a la hora de enviar posiciones al emulador:

- Avanzar automáticamente por la lista de coordenadas..
- Ir a la posición anterior de la lista manualmente.
- Ir a la posición siguiente de la lista manualmente.
- Establecer la velocidad de avance automático.

En la siguiente imagen se puede ver la aplicación una vez lanzada:

El estado del proveedor podrá tener dos valores, dependiendo de la recepción de localizaciones:

- Después de recibir cada grupo de lecturas el proveedor pasa a estado 1 **(TEMPORARILY_UNAVAILABLE).**
- Tras empezar a recibir de nuevo lecturas el proveedor pasa a estado 2 **(AVAILABLE).**

2 Google Maps para Android

2.1 Introducción

La nueva API de Google Maps para Android se presentó en diciembre de 2012, aportando numerosas novedades respecto a su predecesora.

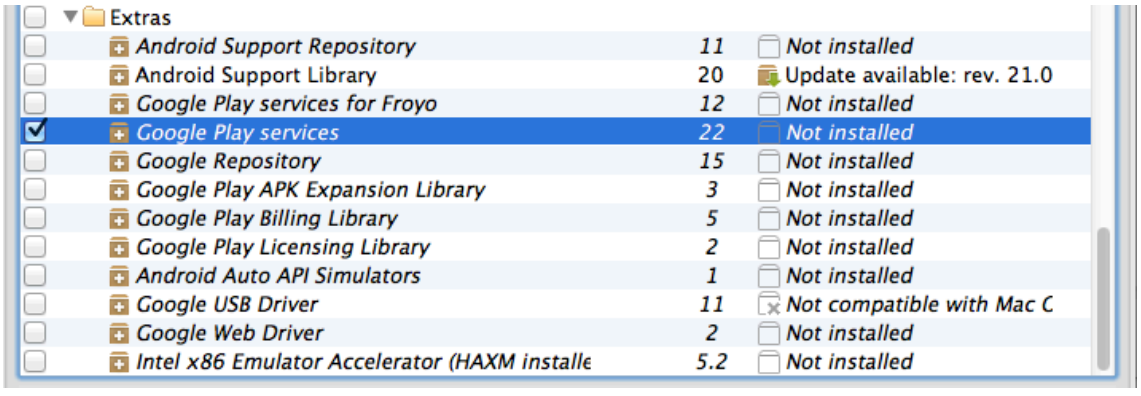
A nivel de desarrollador, una de las más importantes será el tipo de componente a usar a la hora de introducir en una aplicación el uso de esta herramienta de Google. En esta versión se utilizará un tipo específico de fragment llamado **MapFragment.**

La API de Google Maps está integrada con los *Google Play Services* y la *Consola de APIs* de Google, por lo que habrá que realizar algunos pasos previos antes de poder utilizar esta API.

2.2 Instalación de Google Play Services

Google Play Services integra numerosas APIS para desarrolladores de Android. Estos Google Play Services se encuentran dentro de todos los dispositivos Android, por lo que sólo habrá que conectarse a estos servicios cuando se necesiten, encargándose el propio **Google Play Services** de realizar las actualizaciones necesarias en cada momento.

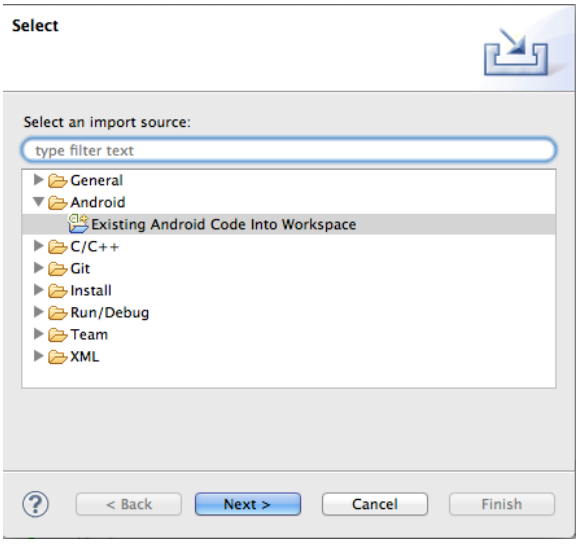
El primer paso para el uso de las APIS contenidas en **Google Play Services** es importar en eclipse la librería que contiene estos servicios. Para ello se accede al SDK Manager de Android, sección Extras y se marca el paquete llamado **Google Play Services.**

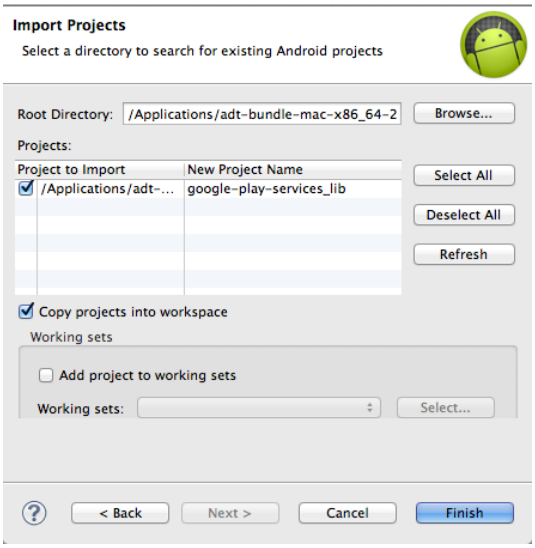


Una vez instalado se localiza en la ruta:

<ruta-sdk>\extras\google\google_play_services\libproject\google-play-services_lib

El siguiente paso será importarlo en Eclipse, en las siguientes imágenes se ven los pasos necesarios:

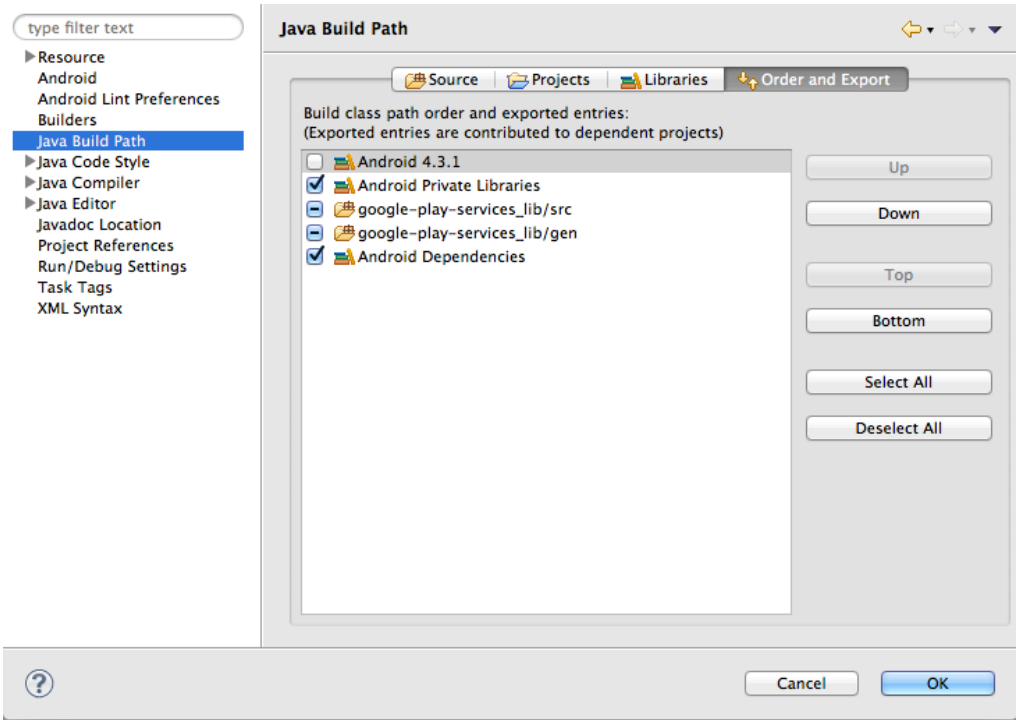




Se selecciona el proyecto google-play-services_lib, de la ruta anteriormente indicada y, a la hora de importar se clica en **Copy projects into workspace**.

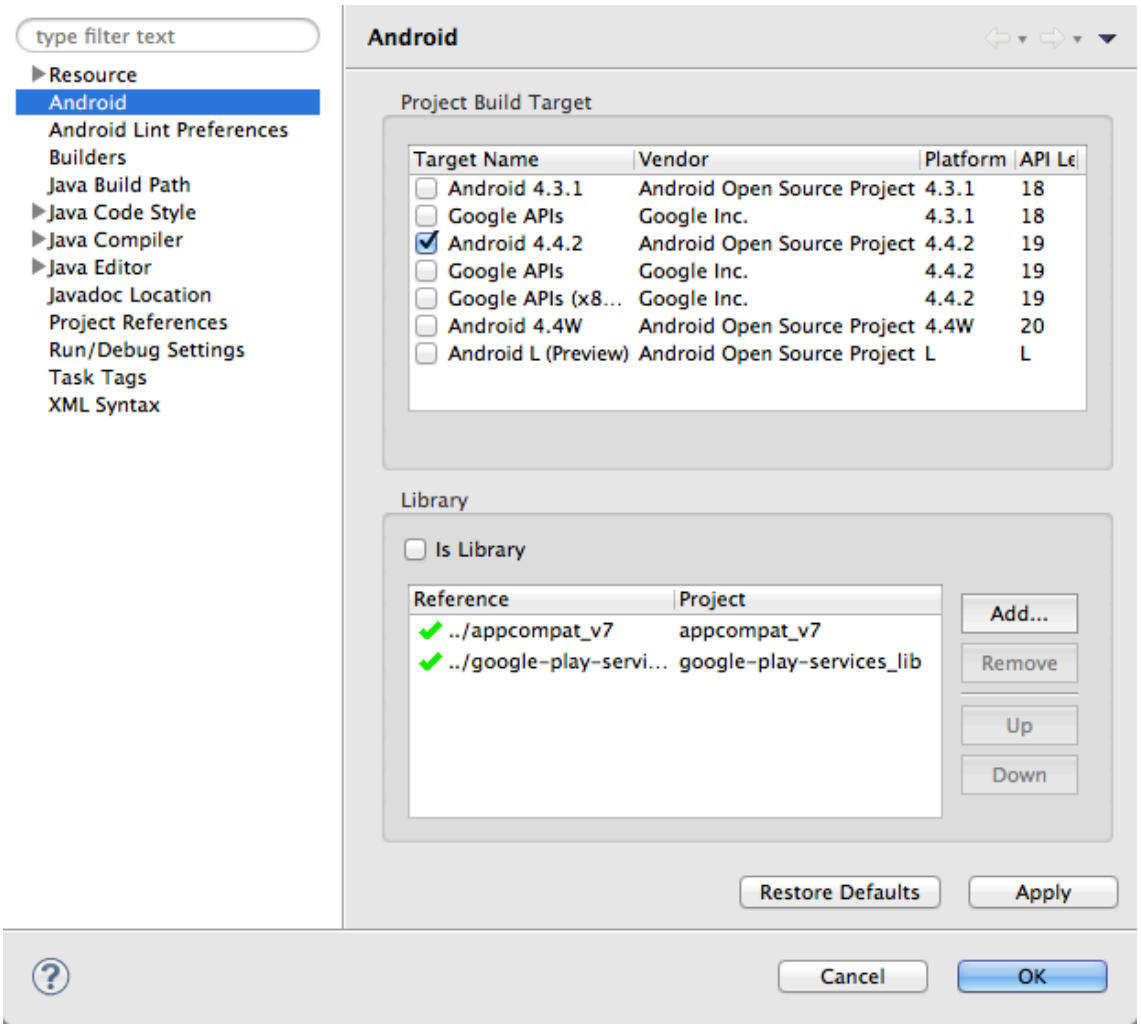
Una vez importado se debe realizar sobre este proyecto una comprobación:

Botón derecho → Properties → sección "Java Build Path" → opción **"Android Private Libraries"** marcada en **"Order and Export"**



El siguiente paso será añadir al proyecto sobre el que se va a trabajar la referencia a la librería de **Google Play Services**:

Botón derecho → Properties →Opción Android→Library→ se añade el proyecto google-play-services_lib



La siguiente modificación necesaria se lleva a cabo en el fichero AndroidManifest.xml.
Dentro del elemento**<application>** se añade lo siguiente:

```
<meta-data android:name="com.google.android.gms.version"
   android:value="@integer/google_play_services_version" />
```

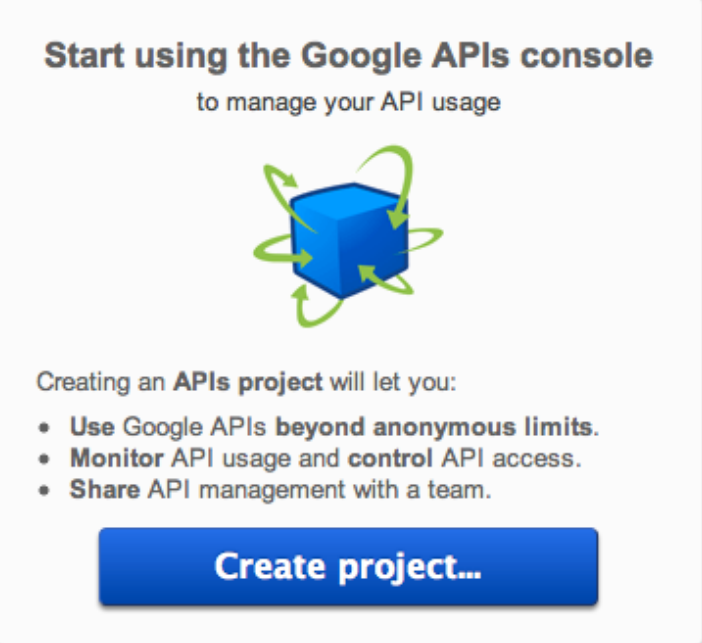
Y en el fichero **proguard-project.txt**, se añadirán las siguientes líneas, que evitarán la eliminación de algunas clases necesarias:

```
-keep class * extends java.util.ListResourceBundle {
    protected Object[][] getContents();
}

-keep public class
com.google.android.gms.common.internal.safeparcel.SafeParcelable {
```

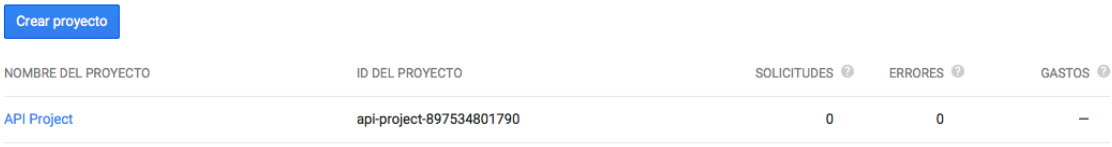
```
    public static final *** NULL;
}

-keepnames @com.google.android.gms.common.annotation.KeepName class *
-keepclassmembernames class * {
    @ccom.google.android.gms.common.annotation.KeepName *;
}

-keepnames class * implements android.os.Parcelable {
    public static final ** CREATOR;
}
```

2.3 Obtención de API Key para usar Google Maps en la aplicación

El primer paso será acceder a la consola de API Keys de google a través del enlace https://code.google.com/apis/console/ .
Una vez se entra con una cuenta de Google, se pulsará en Create Project:

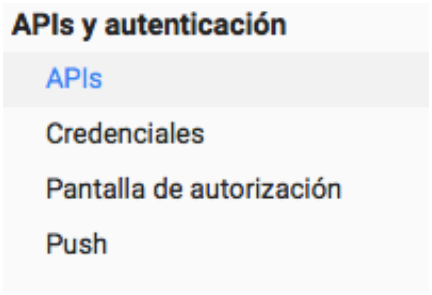


Automáticamente se creará un proyecto, con un nombre y un ID determinado. De todos modos se pueden crear proyectos nuevos con su ID correspondiente.

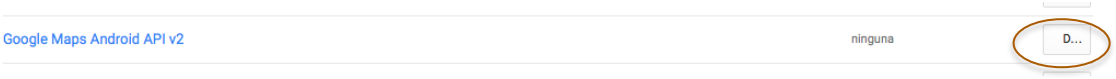


Nombre: API Project
ID: api-project-897534801790

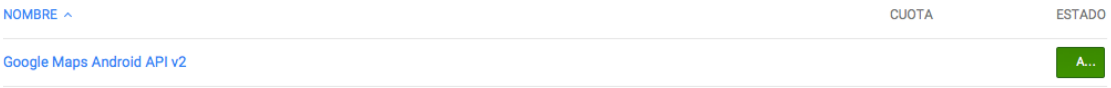
Al pinchar sobre el nombre del proyecto, aparecerá un menú lateral, teniendo que pinchar en **APIs y autenticación:**



Una vez pinchado se desplegarán todas las APIs disponibles. En esa lista se seleccionará la que interesa para este ejemplo, que será **Google Maps API v2**.

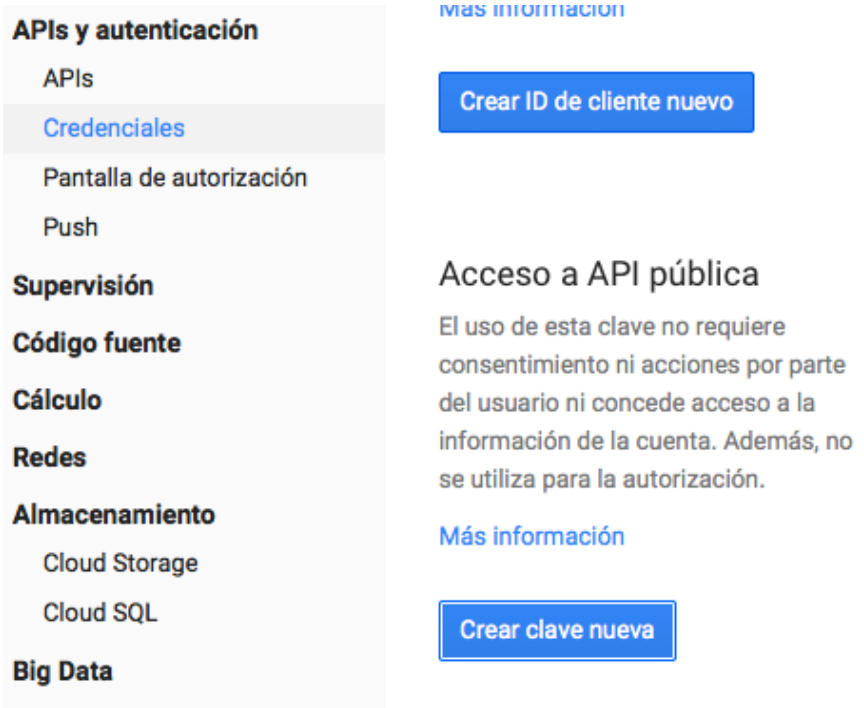


Por defecto aparece desactivada. Se activa, quedando de la siguiente manera:



El siguiente paso será registrar la aplicación con el fin de obtener una API Key que permita utilizar el servicio de mapas desde ella.

Para ello se clica en:
Credenciales → Acceso a API pública → Crear clave nueva

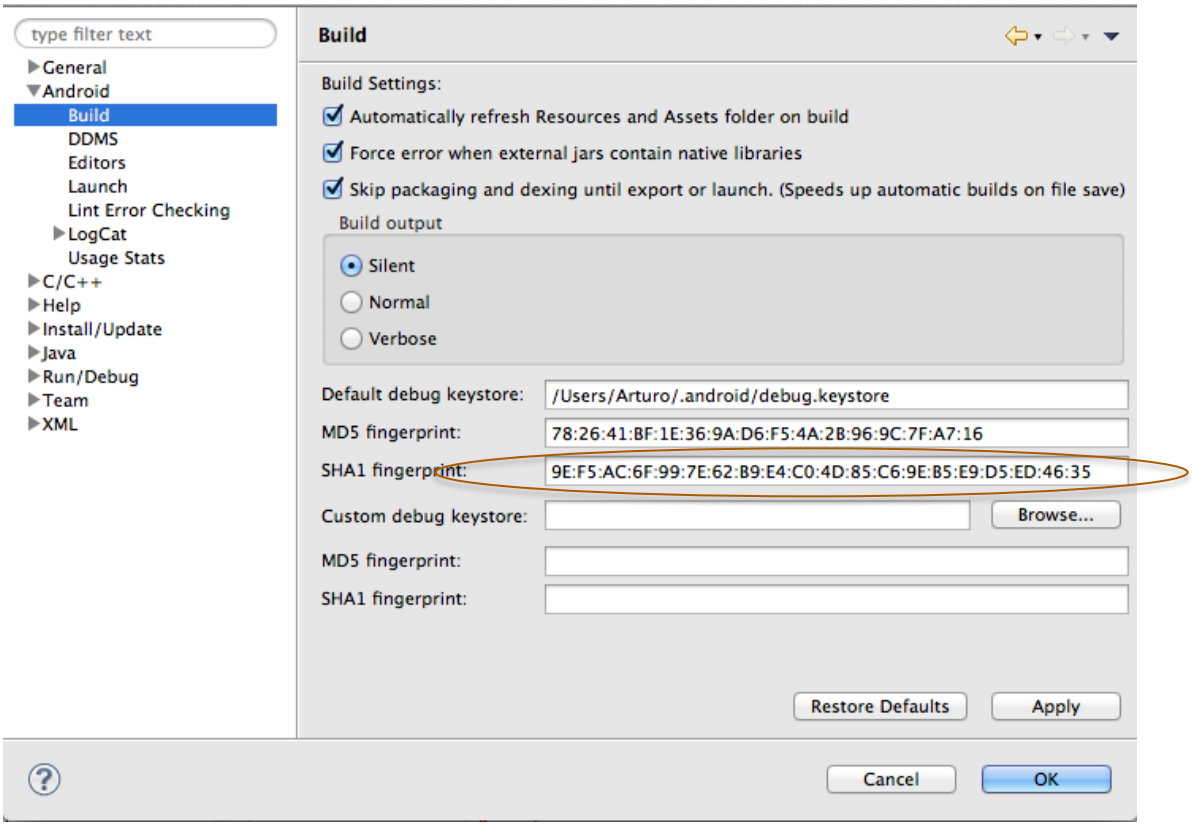


Se indicará que la clave es para Android. La siguiente ventana requerirá la huella digital SHA1 del certificado con el que se firma la aplicación.

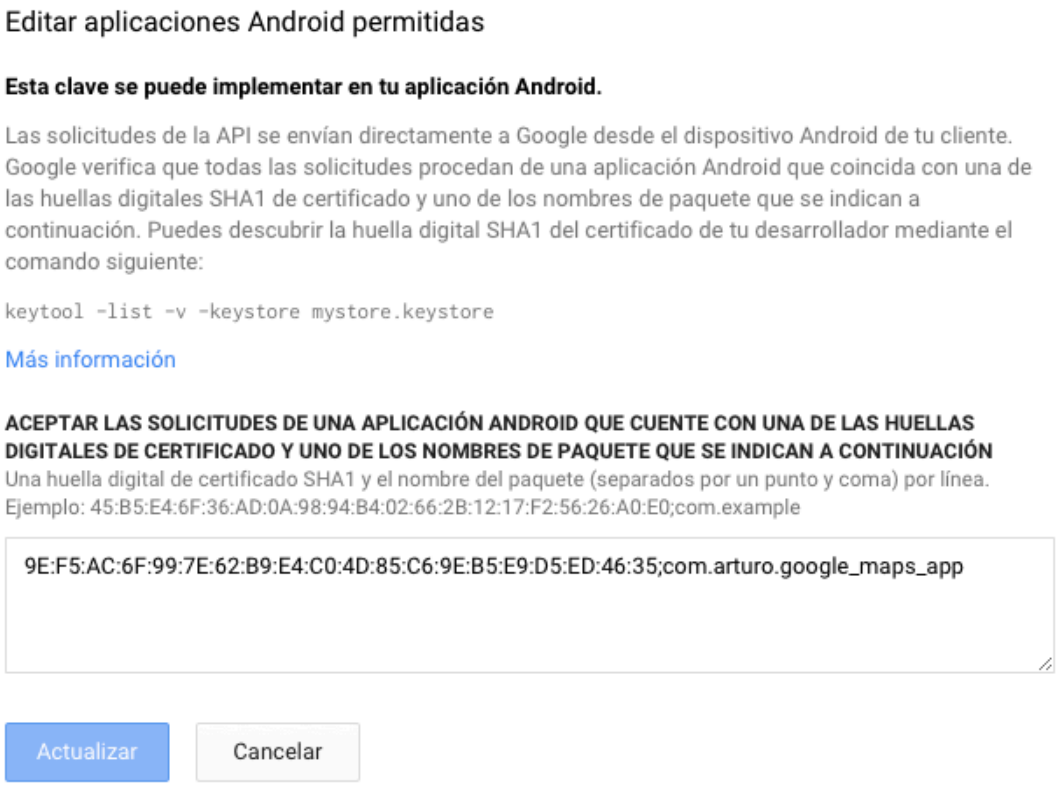
Toda aplicación Android debe ir firmada para poder ejecutarse en un dispositivo, tanto físico como emulado. Este proceso de firma es uno de los pasos que se hace, por ejemplo, antes de distribuir públicamente una aplicación.

Para acceder a él, se debe seguir en Eclipse la ruta:

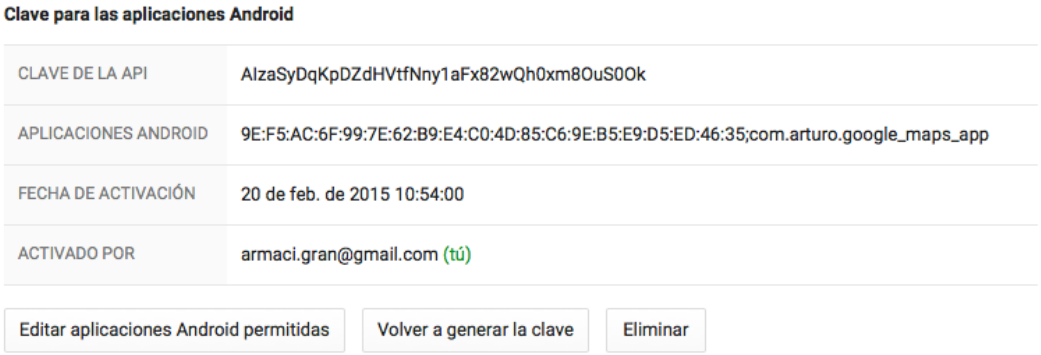
Window→ preferences→ sección **Android Build**



Ese valor es el que se introduce en la ventana emergente que se estaba configurando. Habrá que añadir el SHA1 acompañado de ; y el nombre del paquete, en este caso:
9E:F5:AC:6F:99:7E:62:B9:E4:C0:4D:85:C6:9E:B5:E9:D5:ED:46:35;com.arturo.google_maps_app



Una vez pulsado en **Crear (se encontraría en la posición de Actualizar de la imagen)**, ya estaría este paso completado, siendo la información generada la siguiente:



La clave de la API será la APLI Key que se buscaba en este paso para poder acceder al servicio Google Maps desde la aplicación.
API Key: **AIzaSyDqKpDZdHVtfNny1aFx82wQh0xm8OuS0Ok**

Una vez hecho este paso habrá que realizar ciertas configuraciones en el Proyecto de Android

2.4 Configuraciones del proyecto para uso de Google Maps

El primer paso será añadir al AndroidManifest.xml la **API Key** generada en el anterior paso.
Dentro de la etiqueta **<application>** se añade un nuevo elemento **<meta-data>**, que acompañará al que se añadió anteriormente para el uso de los Google Play Services, con los siguientes datos:

```
<meta-data android:name="com.google.android.maps.v2.API_KEY"
        android:value="AIzaSyDqKpDZdHVtfNny1aFx82wQh0xm8OuS0Ok"/>
```

Donde se puede apreciar que en **android:value** se añade la API Key que se obtuvo antes.

Otras modificaciones que habrá que realizar en este archivo, para dar permisos específicos:

- **Internet**→ para descargar los mapas de los servidores.
- **Access_Network_State**→ para comprobar el estado de la conexión y la posibilidad de descargar nuevos datos.
- **Write_External_Storage**→ modificar/eliminar contenido del almacenamiento externo para la caché de mapas.
- **Read_GServices**→ permitir el acceso a los servicios de Google basados en la web.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES"/>
```

Puesto que la API v2 de Google Maps Android utiliza OpenGL ES versión 2, se especifica ese requisito en el **AndroidManifest.xml**:

```
<uses-feature android:glEsVersion="0x00020000"
              android:required="true"/>
```

2.5 Implementaciones iniciales en la aplicación que usará Google Maps

Una vez llevadas a cabo estas configuraciones iniciales, ya se puede abordar el "desarrollo" como tal de la aplicación.

Como se indicó anteriormente, el control que estará presente en el layout para el trabajo con estas herramientas es de tipo **Fragment.**

Para ello se modifica el layout , editando a nivel de xml, con el siguiente código:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/map"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        class="com.google.android.gms.maps.SupportMapFragment"/>
```

Al utilizar **Fragments** la MainActivity debe de extender de **FragmentActivity**.

```
import android.support.v4.app.FragmentActivity;

public class MainActivity extends FragmentActivity {
```

Una vez realizadas todas estas configuraciones ya se puede ejecutar la aplicación. Hay que tener en cuenta que al lanzar desde un emulador da problemas. Esto se debe a que el emulador debe tener instalado los **Google Play Services** y **Google Play Store**.

En el siguiente enlace se explica una posible solución:

http://maurizziof.blogspot.com.es/2013/08/como-usar-googlemaps-api-v2-en.html

Si se ejecuta desde un móvil externo no habrá ningún tipo de problema, ya que ambos ya están instalados.

2.6 Funcionalidades para una aplicación con Google Maps

La clase "estrella" para este tipo de funcionalidades es la clase **GoogleMap**. Sobre un objeto de esta clase se realizarán todas las acciones de trabajo sobre el mapa que se visualiza. Para obtener una referencia:

```
private GoogleMap mapa;

mapa = ((SupportMapFragment) getSupportFragmentManager()
            .findFragmentById(R.id.map)).getMap();
```

En este caso se obtiene a través del método ***.getMap()***, aplicado al fragment que ocupa el layout.

El mapa se puede ver de distintas maneras, que serían las siguientes:

- **MAP_TYPE_NORMAL**: El formato normal que se puede ver el Google maps.
- **MAP_TYPE_HYBRID:** Formato de satélite y divisiones administrativas.
- **MAP_TYPE_SATELLITE:** Vista satélite sin divisiones administrativas.
- **MAP_TYPE_TERRAIN:** Vista de formato normal pero con relieve.

Estas variables posibles se pasan al método ***setMapType()***, aplicable sobre el objeto Google Maps.

En la aplicación de ejemplo se agrupan dentro del ***método alternarVista()***, que será llamado desde una de las opciones del menú.

```
private void alternarVista()
{
    vista = (vista + 1) % 4;

    switch(vista)
    {
        case 0:
            mapa.setMapType(GoogleMap.MAP_TYPE_NORMAL);
            break;
        case 1:
            mapa.setMapType(GoogleMap.MAP_TYPE_HYBRID);
            break;
        case 2:
            mapa.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
            break;
        case 3:
            mapa.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
            break;
    }
}
```

El movimiento sobre el mapa, o como se llama específicamente en este tipo de servicios, ***"el movimiento de cámara"***, se realiza a través de un objeto de la clase **CameraUpdate**.

Para los movimientos más básicos(latitud, longitud, zoom), se utiliza la clase **CameraUpdateFactory** y sus métodos estáticos.

De manera genérica:

- Para el zoom:
  - ***CameraUpdateFactory.zoomIn().*** Aumenta en 1 el nivel de zoom.
  - ***CameraUpdateFactory.zoomOut().*** Disminuye en 1 el nivel de zoom.
  - ***CameraUpdateFactory.zoomTo(nivel_de_zoom).*** Establece el nivel de zoom.
- Para modificar longitud y latitud:
  - ***CameraUpdateFactory.newLatLng(lat, long).*** Establece la lat-lng expresadas en grados.
- Para modificar de forma conjunta longitud, latitud y zoom:
  - ***CameraUpdateFactory.newLatLngZoom(lat, long, zoom).*** Establece la lat-lng y el zoom.
- Para desplazarse haciendo *panning*(desplazamiento lateral lento):
  - ***CameraUpdateFactory.scrollBy(scrollHorizontal, scrollVertical).*** Scroll expresado en píxeles.

Una vez construido el objeto de la clase **CameraUpdate** con alguno de los métodos definidos en estas líneas anteriores, se debe llamar a los métodos ***moveCamera(CameraUpdate obj)*** o ***animateCamera(CameraUpdate obj)*** del objeto GoogleMap, para que actualice la vista de manera directa o animada.

En las distintas opciones de menú de la aplicación que acompaña a este documento se llamarán a estos métodos.

Por ejemplo para centrar el mapa en España, con un zoom de nivel 5:

```
CameraUpdate camUpd2 =
    CameraUpdateFactory.newLatLngZoom(new LatLng(40.41, -3.69), 5F);
mapa.animateCamera(camUpd2);
```

Para modificar otros parámetros de la cámara se utilizaría el método ***CameraUpdateFactory.newCameraPosition(CameraPosition object).***

Este método recibe un objeto de tipo ***CameraPostion*** que se crea llamando al método ***Builder()*** de la siguiente manera:

```
LatLng madrid = new LatLng(40.417325, -3.683081);
CameraPosition camPos = new CameraPosition.Builder()
        .target(madrid)   //Centramos el mapa en Madrid
        .zoom(19)         //Establecemos el zoom en 19
        .bearing(45)      //Establecemos la orientaci□n con el noreste arriba
        .tilt(70)         //Bajamos el punto de vista de la c□mara 70 grados
        .build();

CameraUpdate camUpd3 =
        CameraUpdateFactory.newCameraPosition(camPos);

mapa.animateCamera(camUpd3);
```

Una vez creado el objeto de tipo **CameraPosition** se le modifican distintos parámetros. En el caso de arriba se puede ver como se consigue una vista en 3D.

Existe también la posibilidad de conocer la posición de la "cámara" en un momento determinado. Para ello se obtiene un objeto de la clase **CameraPosition** a través del método ***getCameraPosition()*** aplicado sobre el objeto **GoogleMaps**. Se ve la implementación en la siguiente imagen:

```
CameraPosition camPos2 = mapa.getCameraPosition();
LatLng pos = camPos2.target;
Toast.makeText(MainActivity.this,
        "Lat: " + pos.latitude + " - Lng: " + pos.longitude,
        Toast.LENGTH_LONG).show();
```

2.7 Eventos de mapa, marcadores, dibujo de líneas y polígonos

Existen escuchadores de eventos, o Listener sobre los que se pueden recoger eventos de acciones llevadas a cabo sobre el mapa. Los que se desarrollaran en este apartado son los siguientes:

- ***setOnMapClickListener()***
- ***setOnMapLongClickListener()***
- ***setOnCameraChangeListener()***

En el primero de los métodos, ***setOnMapClickListener()*** se sobrescribe su método ***onMapClick(LatLng object)***. El objeto **LatLng** contiene la longitud y la latitud del punto sobre el que ha pulsado el usuario. Para recoger estas coordenadas hay que utilizar como intermediario un objeto de la clase **Projection**, que se obtiene a través del método ***getProjection()***. A continuación estas coordenadas se almacenan en un objeto de la clase **Point** a través del método ***toScreenlocation(Projection object)***.

En la imagen, un ejemplo del código implementado en la aplicación:

```
//Para el click corto sobre el mapa
mapa.setOnMapClickListener(new OnMapClickListener() {
    public void onMapClick(LatLng point) {
        Projection proj = mapa.getProjection();//se obtiene la proyeccion del mapa
        Point coord = proj.toScreenLocation(point);//se guardan las coordenadas
        //Toast con información del punto pulsado
        Toast.makeText(
                MainActivity.this,
                "Click\n" +
                "Lat: " + point.latitude + "\n" +
                "Lng: " + point.longitude + "\n" +
                "X: " + coord.x + " - Y: " + coord.y,
                Toast.LENGTH_SHORT).show();
    }
});
```

Para el evento de pulsación larga o continuada, se utiliza el segundo de los escuchadores, ***setOnMapLongClickListener()***, cuyo funcionamiento es análogo al anterior:

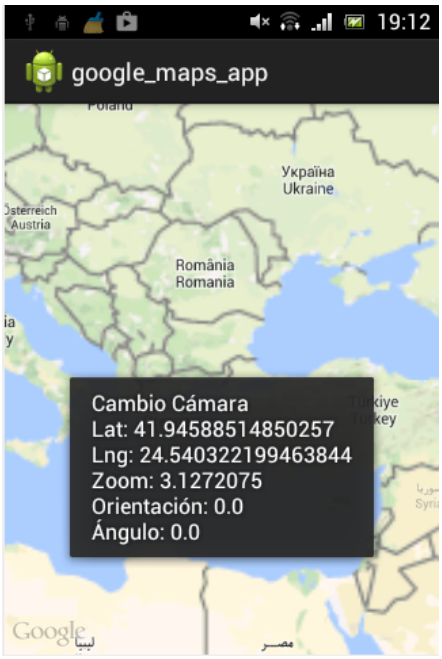
```
//Para el click continuado/largo
mapa.setOnMapLongClickListener(new OnMapLongClickListener() {
    public void onMapLongClick(LatLng point) {
        Projection proj = mapa.getProjection();
        Point coord = proj.toScreenLocation(point);

        Toast.makeText(
                MainActivity.this,
                "Click Largo\n" +
                "Lat: " + point.latitude + "\n" +
                "Lng: " + point.longitude + "\n" +
                "X: " + coord.x + " - Y: " + coord.y,
                Toast.LENGTH_SHORT).show();
    }
});
```

En el dispositivo se vería de la siguiente manera:

El tercero de los escuchadores de eventos mencionados anteriormente es ***setOnCameraChangeListener()***. Con este escuchador se recogerán los movimientos del mapa llevados a cabo por el usuario, ya sean acciones de zoom, modificaciones de orientación, ángulo de visión, etc.
En este caso el método a sobrescribir es ***onCameraChange(CameraPosition object)***, que recibe un objeto de la clase **CameraPosition**, del cuál en este ejemplo se obtiene distinta información a partir de sus diversos métodos, y que se mostrará a través de un **Toast**:

```
//Para los movimientos del mapa
mapa.setOnCameraChangeListener(new OnCameraChangeListener() {
    public void onCameraChange(CameraPosition position) {
        Toast.makeText(
                MainActivity.this,
                "Cambio Cámara\n" +
                "Lat: " + position.target.latitude + "\n" +
                "Lng: " + position.target.longitude + "\n" +
                "Zoom: " + position.zoom + "\n" +
                "Orientación: " + position.bearing + "\n" +
                "Ángulo: " + position.tilt,
                Toast.LENGTH_SHORT).show();
    }
});
```

Y lo que se vería en el dispositivo móvil sería:



Otro elemento importante y recurrente en las aplicaciones que utilizan Google Maps son los **marcadores**.
Para agregar un marcador se utiliza el método ***addMarker(MarkerOptions object)*** que recibe un objeto de la clase **MarkerOptions**, al que se le pasa un objeto **LatLng** con la latitud y la longitud y el título del marcador. En este ejemplo se encuentra dentro de un método que se llamará desde un ítem del menú:

```
private void mostrarMarcador(double lat, double lng)
{
    mapa.addMarker(new MarkerOptions()
    .position(new LatLng(lat, lng))//Posición para el marcador
    .title("Aquí está Sebastopol"));//Texto del marcador

}
```

Y en la aplicación se vería de la siguiente manera:

En este caso al activar un marcador aparecen dos iconos en la parte baja de la pantalla. Servirán para lanzar la aplicación propia de Maps de la aplicación y poder, por ejemplo, navegar hasta ese punto.

Los marcadores también tienen escuchadores de pulsación. Para capturar el evento de una pulsación se utiliza el método ***setOnMarkerClickListener()*** teniendo que sobrescribir su método ***onMarkerClick(Marker object)***, que recibe un objeto de la clase **Marker**, a través del cual se puede acceder a todas sus características con sus diversos métodos asociados. En el ejemplo, al pulsar lanza un **Toast** con información del titulo del marcador.

```
//Para la pulsación sobre un marcador
mapa.setOnMarkerClickListener(new OnMarkerClickListener() {
    public boolean onMarkerClick(Marker marker) {
        Toast.makeText(
                MainActivity.this,
                "Marcador pulsado:\n" +
                marker.getTitle(),
                Toast.LENGTH_SHORT).show();
        return false;
    }
});
```

Los marcadores son elementos del API Google Maps muy versátiles, dando mucho juego a la hora de modificarlos, hacerlos arrastables, cambiar su color, etc.
En la documentación del desarrollador se pueden localizar todos los métodos aplicables y todas las modificaciones que se pueden llevar a cabo. En el siguiente enlace se puede ver la información:

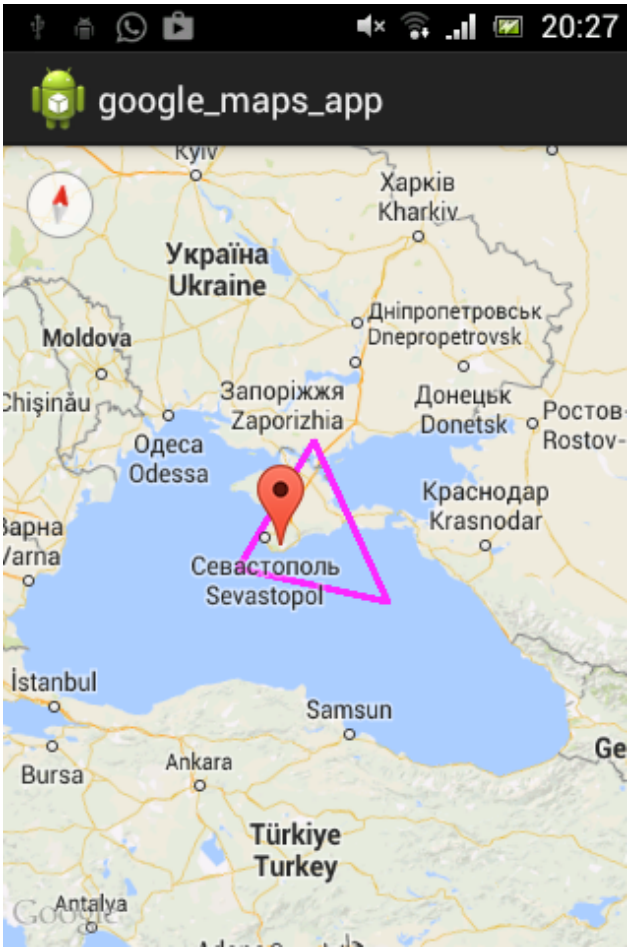
https://developers.google.com/maps/documentation/android/marker

Otros elementos muy útiles y típicos en las aplicaciones con Google Maps son las representaciones de **líneas y polígonos** sobre el mapa.
Para ello se hace uso de dos clases **PolylineOptions** (para dibujar líneas) y **PolygonOptions** (para dibujar polígonos).

Para el dibujo de la línea se declara un objeto **PolylineOptions**, y con su método ***add(LatLng object)***, se le añade las coordenadas de cada uno de los nodos de la línea.
A continuación se establece el grosor y color de la línea llamando a los métodos ***width()*** y ***color()*** respectivamente, y por último añadiremos la línea al mapa mediante su método ***addPolyline(PolylineOptions object)***.

```
//Triángulo con líneas

PolylineOptions lineas = new PolylineOptions()
    .add(new LatLng(46.40, 34.40))
    .add(new LatLng(44.0, 33.0))
    .add(new LatLng(43.80, 36.80))
    .add(new LatLng(46.40, 34.40));

lineas.width(3);
lineas.color(Color.MAGENTA);

mapa.addPolyline(lineas);
```

Y en el dispositivo se verá:



La misma figura se puede representar utilizando, en lugar de líneas, polígonos directamente, el proceso es muy similar al anterior.
Se crearía un objeto de la clase **PolygonOptions**, y se añadirían los nodos en el sentido de las agujas del reloj.
En este caso el ancho y el color de la línea se establecen con los métodos ***strokeWidth()*** y ***strokeColor()***. Finalmente se añade al mapa el polígono con el método ***addPolygon(PolygonOptions object)***.

```
PolygonOptions triangulo = new PolygonOptions()
                .add(new LatLng(46.40, 34.40))
              .add(new LatLng(44.0, 33.0))
              .add(new LatLng(43.80, 36.80))
              .add(new LatLng(46.40, 34.40));

triangulo.strokeWidth(3);
triangulo.strokeColor(Color.MAGENTA);

mapa.addPolygon(triangulo);
```

2.8 BIBLIOGRAFÍA

Localizaciones GPS

http://www.sgoliver.net/blog/localizacion-geografica-en-android-i/

http://www.sgoliver.net/blog/localizacion-geografica-en-android-ii/

https://blog.codecentric.de/en/2014/05/android-gps-positioning-location-strategies/

API Google Maps en Android

http://www.sgoliver.net/blog/mapas-en-android-google-maps-android-api-v2-i/

http://www.sgoliver.net/blog/mapas-en-android-google-maps-android-api-v2-ii/

http://www.sgoliver.net/blog/mapas-en-android-google-maps-android-api-v2-iii/

NOTA: El contenido de este documento y los ejemplos usados son una adaptación del material perteneciente a Salvador Gómez Oliver (www.sgoliver.net). Su única función es la de resumir y aglutinar la información relativa a este tema para la consulta por parte de los alumnos del Ciclo FP DAM des IES "San Andrés del Rabanedo", no existiendo ninguna intención de suplantar su autoría original ni los derechos del propietario.