



# Geolocalización





## Contenidos

Localización geográfica .....	3
API's .....	3
P_92_Localizacion .....	4
Añadir permisos .....	4
Añadir dependencias.....	4
Layout inicial.....	4
Código inicial de la actividad .....	5
Método seguir().....	6
Método actualizarLayout() .....	6
Trabajando con el emulador .....	7
Emulador GeanyMotion.....	7
Método actualizarUbicacion .....	7
Fijar requerimientos.....	8
Comprobar configuración del usuario coherente .....	9
Método recibeActualizacion() .....	9
Detener actualizaciones .....	10
Obtener dirección .....	10
Método obtenerDirección.....	11
Otros métodos de la clase Geocoder .....	11



## Localización geográfica

Existen varios métodos para obtener la localización de un dispositivo móvil Android: señal móvil, Wi-Fi o GPS.

Para usarlos hay que fijar el permiso "peligroso" (con su debido tratamiento en código para que funcionen con Android 6+) necesario. Puede escogerse entre dos:

- ACCESS\_FINE\_LOCATION. Permiso para acceder a datos de localización con una precisión alta (consume más batería).
- ACCESS\_COARSE\_LOCATION. Permiso para acceder a datos de localización con una precisión baja (sitúa a un edificio en su "manzana").

En la precisión de los datos obtenidos interviene también la configuración del dispositivo que el usuario tenga establecida. Por ejemplo, nuestra aplicación no podría obtener, aunque así lo solicite, una ubicación con máxima precisión si el usuario lleva deshabilitada la Ubicación en el dispositivo, o si el modo que tiene seleccionado en las opciones de ubicación de Android no es el de "Alta precisión". Por tanto, otro paso importante será chequear de alguna forma si las necesidades de nuestra aplicación son coherentes con la configuración actual establecida en el dispositivo y, en caso contrario, solicitar al usuario que la modifique siempre que sea posible.

## API's

La funcionalidad de localización geográfica fue movida hace ya algún tiempo desde el SDK general de Android a las librerías de los Google Play Services. Aunque el sdk de Android incluye clases dentro del paquete android.location, la propia plataforma aconseja utilizar las API's de Google.

Como siempre, en Internet encontrarás mucha documentación sobre la API "antigua" y mucha menos sobre la "nueva".

Usando la [Google API de localización](#) no tenemos que preocuparnos demasiado, al menos no de forma explícita, de qué proveedor de localización queremos utilizar en cada momento ya

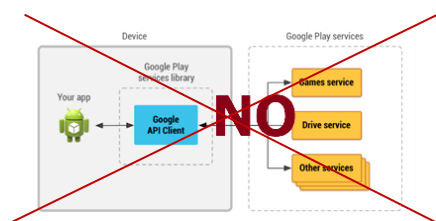
The screenshot shows the documentation for the `FusedLocationProviderClient` class. It includes a sidebar with a navigation menu, a search bar, and tabs for 'SUMMARY', 'GUIDE', 'REFERENCE', and 'DOWNLOADS'. The main content area displays the class name, its inheritance (public class `FusedLocationProviderClient` extends `GoogleApi<Api.ApiOptions.NaOptions>`), and a description: 'The main entry point for interacting with the fused location provider.' Below this, there is a 'Constant Summary' table with one entry: `String KEY_VERTICAL_ACCURACY` with a description. The 'Public Method Summary' section lists several methods: `flushLocations()`, `getLastLocation()`, `getLocationAvailability()`, `removeLocationUpdates(PendingIntent callbackIntent)`, `removeLocationUpdates(LocationCallback callback)`, and `requestLocationUpdates(LocationRequest request, LocationCallback callback, Looper looper)`.

The Google Location Services API, part of Google Play Services, provides a more powerful, high-level framework that automatically handles location providers, user movement, and location accuracy. It also handles location update scheduling based on power consumption parameters you provide. In most cases, you'll get better battery performance, as well as more appropriate accuracy, by using the Location Services API.

To learn more about the Location Services API, see [Google Location Services for Android](#).

que hay disponible un proveedor (Fused Location Provider) que se encargará automáticamente de gestionar todas las fuentes de datos disponibles para obtener la información que nuestra aplicación necesita con poco código. Utiliza la [API de tareas](#). Además permite controlar la exactitud y consumo de energía.

Además a partir de la versión 11 de Google Services, ofrece [mejoras](#) ya que no es necesario hacer uso de la clase `GoogleApiClient` (y como es relativamente nueva, todavía hay muchos manuales en Internet que no lo tienen en cuenta).



La nueva API mejora inmediatamente el código de varias maneras:

- Las llamadas API esperan automáticamente a que se establezca la conexión del servicio, lo que elimina la necesidad de esperar `onConnected` antes de realizar las solicitudes.
- Utiliza la API de [tareas](#) que facilita la composición de las operaciones asíncronas.
- El código es autónomo y podría moverse fácilmente a una clase de utilidad compartida o similar.
- No necesita comprender el proceso de conexión subyacente para comenzar a codificar.



## P\_92\_Localizacion

Creamos un nuevo proyecto con plantilla MainActivity de tipo "Basic" para que agregue las librerías de diseño. Recuerda que el emulador en el que pruebes debe ser de los que cuenten con imagen del sistema que incluya Google APIs

### Añadir permisos

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.pdm.p_102_localizacion">

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

    ...

</manifest>
```

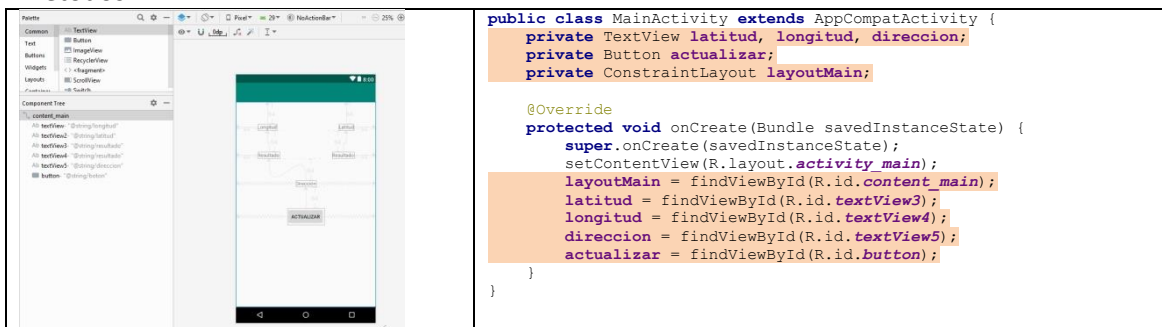
### Añadir dependencias

Es necesario incluir las dependencias a Google API's de localización

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    ...
    implementation 'com.google.android.gms:play-services-location:17.0.0'
}
```

### Layout inicial

TextViews para visualizar Latitud y Longitud, así como la dirección y un botón para actualizar. Instanciamos en el código sus objetos correspondientes para poder utilizarlos en todos los métodos:



The screenshot shows the Android Studio IDE. On the left, the 'Component Tree' and 'Resource Manager' are visible. The 'Component Tree' shows a hierarchy starting with 'android.support.design.widget.CoordinatorLayout' (id: 'content\_main') containing a 'TextView' (id: 'textView1') and a 'Button' (id: 'button1'). The 'Resource Manager' shows the 'layout' folder containing 'activity\_main.xml'. In the center, a preview of the app's main screen is shown, displaying a green header, a white background with a grid of text fields, and a blue button at the bottom. On the right, the 'MainActivity.java' file is open, showing the following code:

```
public class MainActivity extends AppCompatActivity {
    private TextView latitud, longitud, direccion;
    private Button actualizar;
    private ConstraintLayout layoutMain;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        layoutMain = findViewById(R.id.content_main);
        latitud = findViewById(R.id.textView3);
        longitud = findViewById(R.id.textView4);
        direccion = findViewById(R.id.textView5);
        actualizar = findViewById(R.id.button);
    }
}
```



## Código inicial de la actividad

Como toda nuestra actividad necesita usar la Google Api de localización, lo primero que debemos hacer es comprobar que el dispositivo es válido y a continuación solicitar/comprobar que el usuario ha concedido el permiso peligroso (código ya conocido):

```
public class MainActivity extends AppCompatActivity {
    private TextView latitud, longitud, direccion;
    private Button actualizar;
    private ConstraintLayout layoutMain;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        layoutMain = findViewById(R.id.content_main);
        latitud = findViewById(R.id.textView3);
        longitud = findViewById(R.id.textView4);
        direccion = findViewById(R.id.textView5);
        actualizar = findViewById(R.id.button);
        actualizar.setVisibility(View.INVISIBLE);
        boolean valido = comprobarValidezDispositivo();
        if (!valido) {
            finish();
        }
        if (ContextCompat.checkSelfPermission(getApplicationContext(),
Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
            pedirPermiso();
        } else {
            seguir();
        }
    }

    private boolean comprobarValidezDispositivo() {
        GoogleApiAvailability googleApiAvailability = GoogleApiAvailability.getInstance();
        int resultCode = googleApiAvailability.isGooglePlayServicesAvailable(this);
        if (resultCode != ConnectionResult.SUCCESS) {
            if (resultCode == ConnectionResult.SERVICE_INVALID) {
                Toast.makeText(this, "Dispositivo sin Google Play Services",
Toast.LENGTH_LONG).show();
                return false;
            }
            if (googleApiAvailability.isUserResolvableError(resultCode)) {
                googleApiAvailability.getErrorDialog(this, resultCode, 1000).show();
            } else {
                Toast.makeText(this, "Otros problemas ", Toast.LENGTH_LONG).show();
                return false;
            }
        }
        return true;
    }

    private void pedirPermiso() {
        if (ActivityCompat.shouldShowRequestPermissionRationale(this,
Manifest.permission.ACCESS_FINE_LOCATION)) {
            final Activity activity = this;
            Snackbar.make(layoutMain, "Es preciso permiso de ubicación en modo alta precisión ",
Snackbar.LENGTH_INDEFINITE)
                .setAction("OK", new View.OnClickListener() {
                    @Override
                    public void onClick(View view) {
                        ActivityCompat.requestPermissions(activity, new
String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 123);
                    }
                })
                .show();
        } else {
            ActivityCompat.requestPermissions(this, new
String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 123);
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull
int[] grantResults) {
        if (requestCode == 123) {
            if (grantResults.length == 1 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                seguir();
            } else {
                Snackbar.make(layoutMain, "Sin el permiso, no puedo realizar la acción",
Snackbar.LENGTH_LONG).show();
            }
        }
    }

    private void seguir() {
        //Permiso concedido
    }
}
```



## Método seguir()

En nuestro caso lo utilizaremos para obtener una primera posición inicial para inicializar los datos de latitud y longitud de la interfaz. Para ello obtendremos la última posición geográfica conocida (**QUE POSIBLEMENTE NO SEA LA ACTUAL**). ¿Pero por qué "última posición conocida" y no "posición actual"? Porque no existe ningún método del tipo "obtenerPosiciónActual()". Obtener la posición a través de un dispositivo de localización como por ejemplo el GPS no es una tarea inmediata, sino que puede requerir de un cierto tiempo de espera y procesamiento, por lo que no tendría demasiado sentido proporcionar un método de ese tipo. Lo más parecido que encontramos es el **método getLastLocation() que nos devuelve la posición más reciente obtenida por el dispositivo** (no necesariamente solicitada por nuestra aplicación). Y es importante entender esto: este método **NO devuelve la posición actual**, este método **NO solicita una nueva posición al proveedor de localización**, este método se limita a devolver la última posición que se obtuvo a través del proveedor de localización. Y esta posición se pudo obtener hace unos pocos segundos, hace días, hace meses, o incluso nunca (si el dispositivo ha estado apagado, si nunca se ha activado el GPS,...). Por tanto, cuidado cuando se haga uso de la posición devuelta por el método getLastLocation(). En los casos raros en que no existe ninguna posición conocida en el dispositivo este método devuelve null.

Creamos una instancia de la clase FusedLocationProviderClient (LocationServices.getFusedLocationProviderClient(this)), llamamos al método getLastLocation() para obtener la última localización y manejamos la respuesta que es una "Task" que puede usarse para obtener un objeto de tipo Location con las coordenadas de latitud y longitud de una ubicación geográfica.

```
private void seguir() {
    //Permiso concedido
    mFusedLocationClient = LocationServices.getFusedLocationProviderClient(this);
    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) ==
        PackageManager.PERMISSION_GRANTED) {
        mFusedLocationClient.getLastLocation()
            .addOnSuccessListener(this, new OnSuccessListener<Location>() {
                @Override
                public void onSuccess(Location location) {
                    actualizarLayout(location);
                    actualizar.setVisibility(View.VISIBLE);
                }
            });
    }
    else{
        Toast.makeText(getApplicationContext(), "Estáte quieto con los permisos!", Toast.LENGTH_SHORT).show();
        finish();
    }
}
```

## Método actualizarLayout()

El método recibe un objeto de la [clase Location](#) que representa una ubicación geográfica obtenida en la "Task".

```
private void actualizarLayout(Location location) {
    if (location != null) {
        latitud.setText(String.valueOf(location.getLatitude()));
        longitud.setText(String.valueOf(location.getLongitude()));
    }
    else {
        latitud.setText("desconocida");
        longitud.setText("desconocida");
    }
}
```

El objeto Location simplemente encapsula los datos principales de una dirección geográfica, entre otros la latitud y longitud, a los que podemos acceder mediante los métodos getLatitude() y getLongitude() respectivamente.

Recuerda que dicho objeto puede ser null en las siguientes situaciones:

- La ubicación está desactivada en la configuración del dispositivo. El resultado podría ser null aunque existiese una última ubicación porque la inhabilitación ubicación de ubicación también borra la caché.
- El dispositivo nunca registró su ubicación, que podría ser el caso de un dispositivo nuevo o un dispositivo que se haya restaurado a la configuración de fábrica.
- Los servicios de Google Play en el dispositivo se han reiniciado y no hay un cliente activo del Proveedor de ubicación que haya solicitado la ubicación después de reiniciar los servicios. (En nuestra actividad, para evitar esta situación, solicitaremos actualizaciones de ubicación clickando el botón).

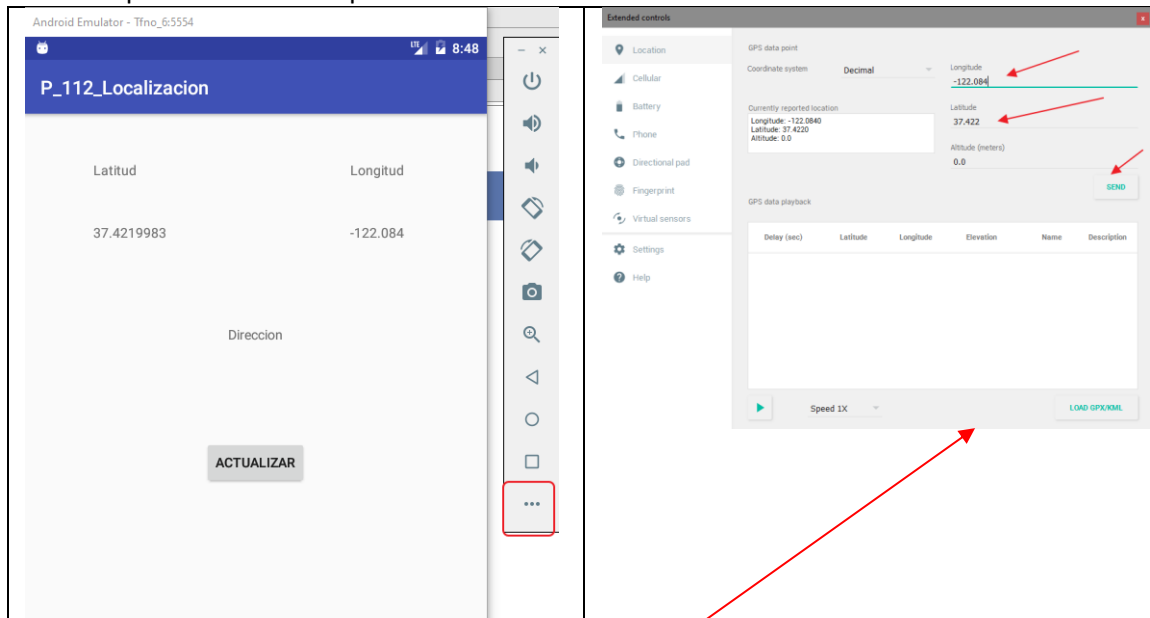


Ejecuta la aplicación. Si lo haces en dispositivo real verás la última ubicación conocida, en emulador seguramente desconocida.

## Trabajando con el emulador

En mis pruebas, la primera vez que ejecuto en emulador no funciona; pero paro y lanzo otra vez y ya funciona (???!!).

Por otra parte recuerda las posibilidades del emulador:

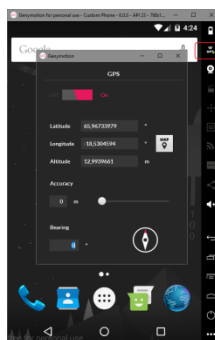


Si además quieres simular un recorrido, podemos enviar una lista de coordenadas en formato GPX o como fichero KML) y fijar su velocidad.

Ambos tipos de fichero son ampliamente utilizados por aplicaciones y dispositivos de localización, como GPS, aplicaciones de cartografía y mapas, etc. Los ficheros KML podemos generarlos por ejemplo a través de la aplicación Google Earth o manualmente con cualquier editor de texto.

Más información de ficheros [KML](#). Tienes un ejemplo de ruta en el fichero ruta.kml en la carpeta Recursos para ejercicios.

## Emulador GeanyMotion



## Método actualizarUbicacion

Tenemos una primera ubicación, que en un dispositivo real puede ser bastante aproximada a menos que llevemos siempre desactivadas todas las opciones de ubicación, pero como ya sabemos no siempre puede corresponderse con la ubicación real actual.

Haremos uso del botón del layout para solicitar al sistema datos actualizados, esta vez sí, de la posición actual del dispositivo, por supuesto siempre cumpliendo con el nivel de permisos y precisión que hayamos solicitado.

```
actualizar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        actualizarUbicacion();
    }
});
```



Ya sabemos que no tenemos ningún método que nos devuelva directamente la posición actual, entre otras cosas porque es impredecible el tiempo que podemos tardar en obtenerla.

Seguiremos por tanto otra estrategia, que consistirá en:

1. indicar al sistema nuestros requerimientos, entre ellos la precisión y periodicidad con que nos gustaría recibir actualizaciones de la posición actual
2. definir un método encargado de procesar los nuevos datos a medida que se vayan recibiendo.
3. comprobar si el usuario tiene configurado los ajustes de ubicación para soportar lo que se le pide (por ejemplo, nuestra aplicación no podría obtener, aunque así lo solicite, una ubicación con máxima precisión si el usuario lleva deshabilitada la Ubicación en el dispositivo, o si el modo que tiene seleccionado en las opciones de ubicación de Android no es el de "Alta precisión") y en caso contrario solicitar al usuario que modifique la configuración.

### Fijar requerimientos

La forma en que nuestra aplicación puede definir sus requerimientos en cuanto a opciones de ubicación será a través de un objeto de tipo [LocationRequest](#). Este objeto almacenará las opciones de ubicación que nuestra aplicación necesita, entre las que destacan:

- Periodicidad de actualizaciones: Se establece mediante el método **setInterval()** y define cada cuanto tiempo (en milisegundos) nos gustaría recibir datos actualizados de la posición. De esta forma, si queremos recibir la nueva posición cada 2 segundos utilizaremos **setInterval(2000)**. Con este método lo único que damos es nuestra preferencia, pero la periodicidad real podría ser mayor o menor dependiendo de muchas circunstancias (conectividad GPS limitada o intermitente, otras aplicaciones han solicitado periodicidades más altas,...).
- Periodicidad máxima de actualizaciones: El proveedor de localización de Android proporciona actualizaciones de la ubicación con la periodicidad más alta que haya solicitado cualquier aplicación ejecutándose en el dispositivo. Por este motivo, es importante indicar al sistema a qué periodicidad máxima (también en milisegundos) nuestra aplicación es capaz de procesar nuevos datos de ubicación de forma que no nos provoque problemas de rendimiento o sobrecarga. Este dato lo proporcionaremos mediante el método **setFastestInterval()**.
- Precisión: La precisión de los datos que queremos recibir se establecerá mediante el método **setPriority()**. Existen varios valores posibles para definir esta información:
  - **PRIORITY\_BALANCED\_POWER\_ACCURACY**: Los datos recibidos tendrán una precisión de unos 100 metros. En este modo el dispositivo tendrá un consumo de energía "comedido" al utilizar normalmente la señal WIFI y de datos móviles para determinar la ubicación.
  - **PRIORITY\_HIGH\_ACCURACY**: Es el modo más preciso para obtener la ubicación, por lo que utilizará normalmente la señal GPS.
  - **PRIORITY\_LOW\_POWER**: Los datos recibidos tendrán una precisión de unos 10 kilómetros, pero se utilizará muy poca energía para obtener la ubicación.
  - **PRIORITY\_NO\_POWER**: En este modo nuestra aplicación solo recibirá datos si éstos están disponibles porque alguna otra aplicación los haya solicitado. Es decir, nuestra aplicación no tendrá un impacto directo en el consumo de energía solicitando nuevas ubicaciones, pero si éstas están disponibles las utilizará.

```
locRequest = new LocationRequest();  
locRequest.setInterval(2000);  
locRequest.setFastestInterval(1000);  
locRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
```





## Comprobar configuración del usuario coherente

Construimos un objeto [LocationSettingsRequest](#) mediante su builder, al que pasaremos el LocationRequest definido en el paso anterior.

```
LocationSettingsRequest.Builder builder = new LocationSettingsRequest.Builder()
    .addLocationRequest(locRequest);
```

Para comparar los requisitos de nuestra aplicación con la configuración actual:

```
SettingsClient client = LocationServices.getSettingsClient(this);
Task<LocationSettingsResponse> task = client.checkLocationSettings(builder.build());
```

Cuando la "Task" finaliza, verificaremos la configuración de ubicación mirando el código de estado del objeto LocationSettingsResponse.

```
task.addOnSuccessListener(this, new OnSuccessListener<LocationSettingsResponse>() {
    @Override
    public void onSuccess(LocationSettingsResponse locationSettingsResponse) {
        recibeActualizacion();
    }
});
task.addOnFailureListener(this, new OnFailureListener() {
    @Override
    public void onFailure(@NonNull Exception e) {
        if (e instanceof ResolvableApiException) {
            // Location settings are not satisfied, but this can be fixed
            // by showing the user a dialog.
            try {
                // Show the dialog by calling startResolutionForResult(),
                // and check the result in onActivityResult().
                ResolvableApiException resolvable = (ResolvableApiException) e;
                resolvable.startResolutionForResult(MainActivity.this, 0x1);
            } catch (IntentSender.SendIntentException sendEx) {
                // Ignore the error.
            }
        }
    }
});
```

Si la configuración es válida (onSuccess), llamaremos a nuestro método recibeActualización(); si no es válida (onFailure) se intenta mostrar un diálogo para que la cambie (startResolutionForResult()).

Comprobamos el resultado de la solicitud realizada al usuario para cambiar la configuración en el caso de RESOLUTION\_REQUIRED sobrescribiendo el método onActivityResult() y centrándonos en el caso de que el requestCode recibido sea igual a la constante que utilizamos en el método startResolutionForResult(). Existen dos posibles resultados:

- RESULT\_OK: Indica que el usuario ha realizado el cambio solicitado. En este caso ya podremos solicitar el inicio de las actualizaciones de ubicación.
- RESULT\_CANCELED: Indica que el usuario no ha realizado ningún cambio. En nuestro caso de ejemplo mostraremos un mensaje y finalizaremos la app.

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode==111) {
        switch (resultCode) {
            case Activity.RESULT_OK:
                recibeActualizacion();
                break;
            case Activity.RESULT_CANCELED:
                Toast.makeText(getApplicationContext(), "No se han realizado los cambios de configuración necesarios", Toast.LENGTH_SHORT).show();
                finish();
                break;
        }
    }
}
```

## Método recibeActualizacion()

Se puede iniciar las actualizaciones regulares llamando al método requestLocationUpdates() pasando la instancia del objeto LocationRequest y de LocationCallback.

```
private void recibeActualizacion() {
    mLocationCallback=new LocationCallback() {
        @Override
        public void onLocationResult(LocationResult locationResult) {
            actualizarLayout(locationResult.getLastLocation());
        }
    };
    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) ==
        PackageManager.PERMISSION_GRANTED ) {
        mFusedLocationClient.requestLocationUpdates(locRequest, mLocationCallback, null);
    } else {
        Toast.makeText(getApplicationContext(), "Estáte quieto con los permisos!", Toast.LENGTH_SHORT).show();
        finish();
    }
}
```



## Detener actualizaciones

Hay que ahorrar batería del dispositivo frenando las actualizaciones de ubicación si no se necesitan.

Un buen lugar para hacerlo es el método `onPause()`:

```
@Override
protected void onPause() {
    super.onPause();
    pararUbicacion();
}

private void pararUbicacion() {
    mFusedLocationClient.removeLocationUpdates(mLocationCallback);
}
```

El ciclo de vida de una actividad frecuentemente es bidireccional. Así que debemos complementar la detención de actualizaciones con una reanudación. En el método `onResume()`.

```
@Override
protected void onResume() {
    super.onResume();
    recibeActualizacion();
}
```

## Obtener dirección

Geocoder es una clase para el manejo de geocodificación y geocodificación inversa.

La geocodificación es el proceso de transformar una dirección u otra descripción de un lugar en las coordenadas (latitud y longitud).

Geocodificación inversa es el proceso de transformación de las coordenadas (latitud, longitud) en una dirección (parcial). La cantidad de detalle en una descripción de ubicación geocodificada inversa puede variar, por ejemplo, uno podría contener la dirección postal completa del edificio más cercano, mientras que otro puede contener solamente un nombre de ciudad y el código postal.

Siempre es conveniente hacer el trabajo en un hilo secundario o tarea asíncrona o desde un servicio ya que la búsqueda de una determinada dirección puede ser costosa, en este ejemplo no lo haremos así para no "liar" aún más el código.

La clase Geocoder requiere un servicio de fondo que no está incluido en el marco básico Android. Los métodos de consulta Geocoder devolverán una lista vacía si no hay servicio de back-end en la plataforma. Hay que usar el método `isPresent()` para determinar si existe una aplicación Geocoder.

```
private void actualizarLayout(Location location) {
    if (location != null) {
        latitud.setText(String.valueOf(location.getLatitude()));
        longitud.setText(String.valueOf(location.getLongitude()));
        if (!Geocoder.isPresent()) {
            Toast.makeText(this, "Dirección imposible de determinar", Toast.LENGTH_LONG).show();
        } else {
            obtenerDireccion(location);
        }
    } else {
        latitud.setText("desconocida");
        longitud.setText("desconocida");
    }
}
```



## Método obtenerDirección

El método **getFromLocation(double latitude, double longitude, int maxResults)** de la clase Geocoder devuelve una matriz de direcciones conocidas ([Address](#)) para describir el área que rodea la latitud y longitud.

Los valores devueltos se obtienen por medio de una búsqueda de red (por eso la conveniencia de que no sean en el hilo principal) y son la mejor respuesta encontrada sin que esté garantizado que sean correctos.

```
private void obtenerDireccion(Location location) {
    Geocoder geocoder = new Geocoder(this, Locale.getDefault());
    List<Address> listaDirecciones = null;
    try {
        listaDirecciones = geocoder.getFromLocation(
            location.getLatitude(),
            location.getLongitude(),
            1); // N° de direcciones que se desean
    } catch (IOException ioException) {
        textoDireccion = "Imposible obtener dirección";
    } catch (IllegalArgumentException illegalArgumentException) {
        // Catch invalid latitude or longitude values.
        textoDireccion = "Valores no válidos";
    }

    if (listaDirecciones == null || listaDirecciones.size() == 0) {
        if (textoDireccion.isEmpty()) {
            textoDireccion = "No encontrada dirección";
        }
    } else {
        Address direccion = listaDirecciones.get(0);
        /* "Formato feo"
        textoDireccion=direccion.getAddressLine(0);
        */
        textoDireccion=direccion.getThoroughfare()+" "+direccion.getSubThoroughfare()+"\n"+direccion.getPostalCode();
        textoDireccion=textoDireccion+" "+direccion.getLocality()+" - "+direccion.getCountryName();
    }
    direccion.setText(textoDireccion);
    direccion.setTextAlignment(TEXT_ALIGNMENT_CENTER);
}
```

## Otros métodos de la clase Geocoder

No los usaremos en este proyecto, pero el método **getFromLocationName(String locationName, int maxResults)** obtiene una lista de direcciones a partir del nombre de la localización.

Ejemplo:

```
try {
    List<Address> list = gc.getFromLocationName("Calle Batalla de Lepanto, 30 , 50002 Zaragoza", 1);
    if(list!= null){
        Address address = list.get(0);
        double lat = address.getLatitude();
        double lng = address.getLongitude();
        // trabajar con lat y lng
    }
} catch (IOException e) { // TODO Auto-generated catch block
    e.printStackTrace();
}
```