

Geolocalización y mapas

Índice

1 Geolocalización.....	2
1.1 Actualización de la posición.....	3
1.2 Alertas de proximidad.....	4
1.3 Geocoding.....	4
2 Mapas.....	5
2.1 Obtención de la clave de acceso.....	6
2.2 Configuración del mapa.....	7
2.3 Controlador del mapa.....	8
2.4 Marcadores.....	8
3 Reconocimiento del habla.....	13

1. Geolocalización

Los dispositivos móviles son capaces de obtener su posición geográfica por diferentes medios. Muchos dispositivos cuentan con un GPS capaz de proporcionarnos nuestra posición con un error de unos pocos metros. El inconveniente del GPS es que sólo 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.

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"/>
```

Si se nos concede 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.

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);
```

La clase `LocationProvider` nos proporciona información sobre las características del proveedor, como su precisión, consumo, o datos que nos proporciona.

Es también posible obtener la lista de todos los proveedores disponibles en nuestro móvil con `getProviders`, u obtener un proveedor basándonos en ciertos criterios como la precisión que necesitamos, el consumo de energía, o si es capaz de obtener datos como la altitud a la que estamos o la velocidad a la que nos movemos. Estos criterios se especifican en un objeto de la clase `Criteria` que se le pasa como parámetro al método `getProviders`.

Para obtener la última localización registrada por un proveedor llamaremos al siguiente método:

```
Location posicion = manager
    .getLastKnownLocation(LocationManager.GPS_PROVIDER);
```

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. A continuación veremos cómo solicitar que se realice una nueva lectura de la posición en la que estamos.

1.1. Actualización de la posición

Para poder recibir actualizaciones de nuestra posición deberemos definir un *listener* de clase `LocationListener`:

```
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.
    }
};
```

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.

Nota

Debemos tener en cuenta que esta forma de obtener la posición puede tardar un tiempo en proporcionarnos un valor. Si necesitamos obtener un valor de posición de forma inmediata utilizaremos `getLastKnownLocation`, aunque puede darnos un valor sin actualizar.

Una vez hayamos terminado de utilizar el servicio de geolocalización, deberemos detener las actualizaciones para reducir el consumo de batería. Para ello eliminamos el *listener* de la siguiente forma:

```
manager.removeUpdates(listener);
```

1.2. Alertas de proximidad

En Android podemos definir una serie de alertas que se disparan cuando nos acercamos a una determinada posición. Recibiremos los avisos de proximidad mediante *intents*. Por ello, primero debemos crearnos un *Intent* propio:

```
Intent intent = new Intent(codigo);
PendingIntent pi = PendingIntent.getBroadcast(this, -1, intent, 0);
```

Para programar las alertas de proximidad deberemos especificar la latitud y longitud, y el radio de la zona de proximidad en metros. Además podemos poner una caducidad a las alertas (si ponemos -1 no habrá caducidad):

```
double latitud = 128.342353;
double longitud = 0.4887897;
float radio = 500f;
long caducidad = -1;

manager.addProximityAlert(latitud, longitud, radio,
    caducidad, pi);
```

Necesitaremos también un receptor de *intents* de tipo *broadcast* para recibir los avisos:

```
public class ReceptorProximidad extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Comprobamos si estamos entrando o saliendo de la proximidad
        String key = LocationManager.KEY_PROXIMITY_ENTERING;
        Boolean entra = intent.getBooleanExtra(key, false);
        ...
    }
}
```

Finalmente, para recibir los *intents* debemos registrar el receptor que acabamos de crear de la siguiente forma:

```
IntentFilter filtro = new IntentFilter(codigo);
registerReceiver(new ReceptorProximidad(), filtro);
```

1.3. Geocoding

El *geocoder* nos permite realizar transformaciones entre una dirección y las coordenadas en las que está. Podemos obtener el objeto *Geocoder* con el que realizar estas transformaciones de la siguiente forma:

```
Geocoder geocoder = new Geocoder(this, Locale.getDefault());
```

Podemos obtener la dirección a partir de unas coordenadas (latitud y longitud):

```
List<Address> direcciones = geocoder
    .getFromLocation(latitud, longitud, maxResults);
```

También podemos obtener las coordenadas correspondientes a una determinada dirección:

```
List<Address> coordenadas = geocoder
    .getFromLocationName(direccion, maxResults);
```

2. Mapas

La forma más habitual de presentar la información obtenida por el GPS es mediante un mapa. Vamos a ver cómo podemos integrar los mapas de Google en nuestra aplicación, y mostrar en ellos nuestra posición y una serie de puntos de interés.

La API de mapas no está incluida en el SDK básico de Android, sino que se encuentra entre las librerías de Google. Por este motivo para poder utilizar dicha librería deberemos declararla en el `AndroidManifest.xml`, dentro de la etiqueta `application`:

```
<application android:icon="@drawable/icon"
    android:label="@string/app_name">

    <uses-library android:name="com.google.android.maps" />

    ...

    <activity android:name=".MapasActivity"
        android:label="@string/app_name">
        <intent-filter></intent-filter>
    </activity>

</application>
```

Importante

Para poder utilizar la API de mapas de Google, deberemos utilizar un emulador configurado con dichas APIs. Es decir, no deberemos crear el emulador con las APIs básicas de Android (por ejemplo *Android 2.2 - API Level 8*), sino que deberemos crearlo con las APIs de Google (por ejemplo *Google APIs - API Level 8*).

Para mostrar un mapa, deberemos crear una actividad que herede de `MapActivity`, en lugar de `Activity`, y dentro de ella introduciremos una vista de tipo `MapView`. Podemos introducir esta vista en nuestro *layout* de la siguiente forma:

```
<?xml version="1.0" encoding="utf-8"?>
<com.google.android.maps.MapView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/mvMapa"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="vzqSGwNXealF0KDyclHvvV6c0JfRQW-mYelyt6Q"
    android:clickable="true"
```

```
/>
```

El mapa no se puede añadir de forma visual, ya que no es un elemento de la API básica de Android, sino que pertenece a las APIs de Google. Por lo tanto, deberemos añadirlo directamente al XML del *layout* mediante un elemento de tipo `com.google.android.maps.MapView`. Otro atributo destacable del mapa es `android:apiKey`. Para poder utilizar los mapas debemos obtener una clave que nos dé acceso a la API de mapas de Google.

2.1. Obtención de la clave de acceso

La clave de acceso a la API de Google Maps estará vinculada al certificado que vayamos a utilizar para distribuir la aplicación en Android Market. Durante el desarrollo la aplicación vendrá firmada por el certificado de depuración, por lo que en ese caso deberemos obtener una clave asociada a este certificado de depuración, y la cambiaremos cuando vayamos a distribuir la aplicación. Para poder obtener la clave de acceso deberemos utilizar una huella MD5 de nuestro certificado (de depuración o distribución). Podemos obtener esta huella con el siguiente comando:

```
keytool -list -alias miclave -keystore mialmacen.keystore
```

Donde `mialmacen.keystore` es el almacén de claves donde tenemos nuestro certificado de distribución, y `miclave` es el *alias* con el que referenciamos la clave que vamos a utilizar dentro de dicho almacén.

Si en lugar de la clave de distribución queremos obtener la clave de desarrollo, podemos acceder al almacén `$HOME/.android/debug.keystore`, y dentro de él la clave con *alias* `androiddebugkey`:

```
keytool -list -alias androiddebugkey
        -keystore $HOME/.android/debug.keystore
        -storepass android -keypass android
```

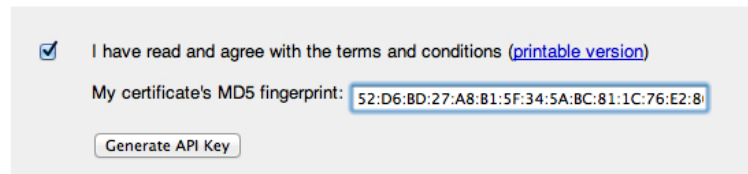
Como podemos ver, tanto el almacén como la clave de depuración están protegidas por la contraseña `android`. Tras hacer esto, veremos una huella digital del certificado:

```
Huella digital de certificado (MD5):
52:D6:BD:27:A8:B1:5F:34:5A:BC:81:1C:76:E2:86:9F
```

Una vez contemos con la huella digital, podemos solicitar una clave de acceso a Google Maps asociada a ella. Esto lo haremos en la siguiente dirección:

```
http://code.google.com/android/maps-api-signup.html
```

Encontramos un formulario donde deberemos introducir nuestra huella digital MD5:



Registro en Google Maps

Tras enviar esta información, nos proporcionará nuestra clave de acceso, junto a un ejemplo de cómo incluirla en nuestra aplicación Android:

Gracias por suscribirte a la clave del API de Android Maps.

Tu clave es:

0KDyeaelyRQfUxN8g3UJrPgXZCdh7CZKfBriu1A

Esta clave es válida para todas las aplicaciones firmadas con el certificado cuya huella dactilar sea:

52:D6:BD:27:A8:B1:5F:34:5A:BC:81:1C:76:E2:86:9F

Incluimos un diseño xml de ejemplo para que puedas iniciarte por los senderos de la creación de mapas:

```
<com.google.android.maps.MapView
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="0KDyeaelyRQfUxN8g3UJrPgXZCdh7CZKfBriu1A"
/>
```

Consulta la [documentación del API](#) para obtener más información.

Clave de la API de Google Maps

2.2. Configuración del mapa

Una vez obtenida la clave de desarrollador y añadida la vista del mapa a nuestro *layout*, desde la actividad del mapa podremos configurar la forma en la que éste se muestra.

Al mostrar esta vista (por ejemplo en el método `onCreate`) podemos configurar la vista del mapa:

```
MapView mapView = (MapView) findViewById(R.id.mvMapa);
mapView.setBuiltInZoomControls(true);
```

Con el código anterior habilitamos el control de *zoom* en el mapa. También podemos establecer si debe mostrar el mapa de tipo satélite o plano:

```
mapView.setSatellite(true);
```



Aspecto del mapa de Google

2.3. Controlador del mapa

Podemos obtener un controlador del mapa que nos permitirá movernos a distintas posiciones o modificar el *zoom*. Podemos obtener el controlador de nuestro mapa con:

```
MapController mc = mapView.getController();
```

Con el controlador podemos establecer el nivel de *zoom* del mapa:

```
mc.setZoom(17);
```

También podemos mover el mapa a una determinada localización. La localización se representa mediante la clase *GeoPoint*, que se inicializará a partir de sus coordenadas:

```
GeoPoint p = new GeoPoint(LATITUD_1E6, LONGITUD_1E6);
```

Una vez definida la localización, podemos mover el mapa a dicha posición mediante una animación con:

```
mc.animateTo(p);
```

Si queremos centrar el mapa en una determinada posición, sin realizar ninguna animación, podemos utilizar:

```
mc.setCenter(p);
```

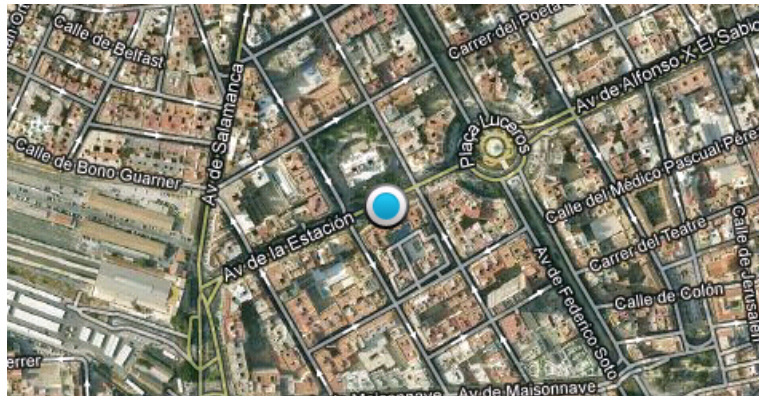
2.4. Marcadores

Es habitual mostrar sobre el mapa una serie de marcadores que indiquen la posición de

determinados puntos de interés, o nuestra propia localización. Estos marcadores se crean mediante objetos de la clase `Overlay`.

Un marcador bastante utilizado es el que indica nuestra posición en el mapa. Lo podemos crear de forma sencilla mediante la clase `MyLocationOverlay`. Añadiremos el *overlay* al mapa de la siguiente forma:

```
MyLocationOverlay myLocation = new MyLocationOverlay(this, mapView);  
mapView.getOverlays().add(myLocation);
```



Marcador de localización actual

Con esto veremos nuestra posición en el mapa mediante un marcador circular azul. Para que este marcador actualice su posición en el mapa conforme nos movemos deberemos habilitar la geolocalización:

```
myLocation.enableMyLocation();
```

Es muy importante que cuando nuestra actividad se cierre, o pase a segundo plano, la geolocalización se desactive, para evitar el consumo de batería que produce el funcionamiento del GPS. Un buen lugar para hacer esto es el método `onPause` de la actividad:

```
@Override  
protected void onPause() {  
    super.onPause();  
  
    // Deshabilita geolocalización  
    location.disableMyLocation();  
}
```

Si queremos que el mapa se centre en nuestra localización de forma automática, podemos solicitar que la próxima vez que se obtengan lecturas de geolocalización se realice dicha animación:

```
myLocation.runOnFirstFix(new Runnable() {  
    public void run() {  
        mc.animateTo(myLocation.getMyLocation());  
    }  
});
```

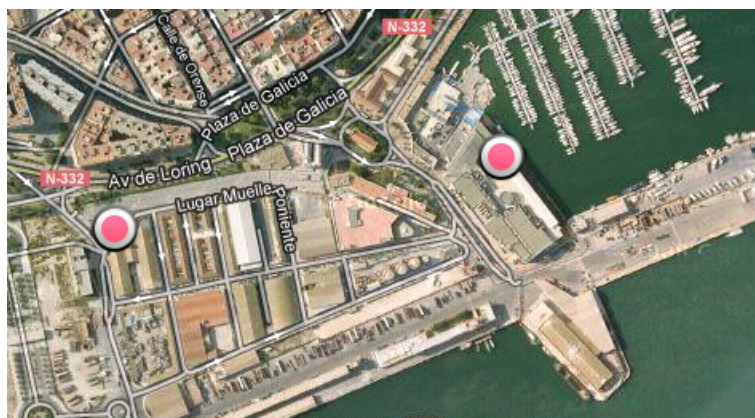
Además de nuestra posición, nos puede interesar mostrar las localizaciones de una serie de puntos de interés mediante marcadores. Para hacer esto el *overlay* más apropiado es el conocido como *ItemizedOverlay*. Se trata de un *overlay* que podemos añadir al mapa como el anterior, ya que hereda de *Overlay*, pero en este caso contiene una lista de *items* a mostrar en el mapa. La clase *ItemizedOverlay* es abstracta, por lo que siempre deberemos definir una subclase que por lo menos defina los métodos *createItem* y *size*. En el primero de ellos deberemos devolver un *item* dado su índice (el *item* será un objeto de tipo *OverlayItem*), y en el segundo deberemos devolver el número total de *items* que tenemos. Estos métodos son los que *ItemizedOverlay* utilizará para poblar el mapa de marcadores. Para que esto ocurra deberemos llamar al método *populate*. No se actualizará el conjunto de marcadores hasta que no llamemos a este método. Deberemos llamar a este método cada vez que cambiemos nuestro conjunto de *items* y queramos mostrarlos en el mapa.

```
class RestaurantesItemizedOverlay extends ItemizedOverlay<OverlayItem> {
    private List<OverlayItem> mRestaurantes =
        new ArrayList<OverlayItem>();

    public RestaurantesItemizedOverlay(Drawable defaultMarker) {
        super(defaultMarker);
        this.mRestaurantes = cargarItemsRestaurantes();
        this.populate();
    }

    @Override
    protected OverlayItem createItem(int i) {
        return mRestaurantes.get(i);
    }

    @Override
    public int size() {
        return mRestaurantes.size();
    }
}
```



Marcadores en el mapa

El *ItemizedOverlay* se creará a partir del *drawable* que haga de marcador en el mapa:

```
Drawable marker = this.getResources().getDrawable(R.drawable.marker);
ItemizedOverlay itemizedOverlay =
    new RestaurantesItemizedOverlay(marker);
```

Si queremos que el *drawable* aparezca centrado exactamente en la localización indicada, podemos hacer una transformación del *drawable* para que su punto de anclaje sea la posición central. La forma más sencilla de hacer esto será utilizar el método estático de `ItemizedOverlay` `boundCenter`. Dado que este método es protegido, deberemos invocarlo dentro del constructor de nuestra subclase de `ItemizedOverlay`:

```
public RestaurantesItemizedOverlay(Drawable defaultMarker) {
    super(boundCenter(defaultMarker));
}
```

Esto es útil por ejemplo si como marcador tenemos un círculo. Si en su lugar tenemos un marcador de tipo "chincheta", podemos ajustar los límites con `boundCenterBottom` para que así su punto de anclaje sea el punto central inferior, que coincidiría con la punta de la chincheta.

Falta por ver cómo crear cada *item* de la lista. Estos *items*, como hemos comentado, son objetos de la clase `OverlayItem`, y se crearán de la siguiente forma:

```
GeoPoint point = new GeoPoint((int) (restaurante.getLatitud() * 1E6),
                               (int) (restaurante.getLongitud() * 1E6));
OverlayItem overlay = new OverlayItem(point, restaurante.getNombre(),
                                       restaurante.getDescripcion());
listaRestaurantes.addOverlay(overlay);
```

Podemos ver que para crear el *item* debemos proporcionar sus coordenadas como un objeto `GeoPoint`. Además, junto a estas coordenadas se guarda un título y una descripción. Desafortunadamente este título y descripción sólo se guarda en el *item* como información relativa a él, pero no nos proporciona ningún mecanismo para mostrar esta información de forma estándar. Tendremos que definir nosotros la forma en la que mostramos esta información en el mapa. Podemos encontrar librerías que se encargan de esto como **mapviewballoons** (<https://github.com/jgilfelt/android-mapviewballoons>), que podemos incluir en nuestro proyecto de forma sencilla, y nos mostrará automáticamente un globo al pulsar sobre cada *item* con su información.

Para utilizar esta librería simplemente deberemos incluir en nuestro proyecto las clases `BalloonItemizedOverlay` y `BalloonOverlayView`, y los *drawables* necesarios para mostrar los marcadores (`marker.png`) y los globos (`balloon_overlay_close.png`, `balloon_overlay_focused.9.png`, `balloon_overlay_unfocused.9.png`, `balloon_overlay_bg_selector.xml`). Una vez añadidos estos componentes, simplemente tendremos que hacer que nuestro *itemized overlay* herede de `BalloonItemizedOverlay` en lugar de `ItemizedOverlay`.

El constructor de `BalloonItemizedOverlay` necesita que se le proporcione la vista del mapa como parámetro, por lo que además de heredar de esta clase deberemos añadir dicho parámetro al constructor:

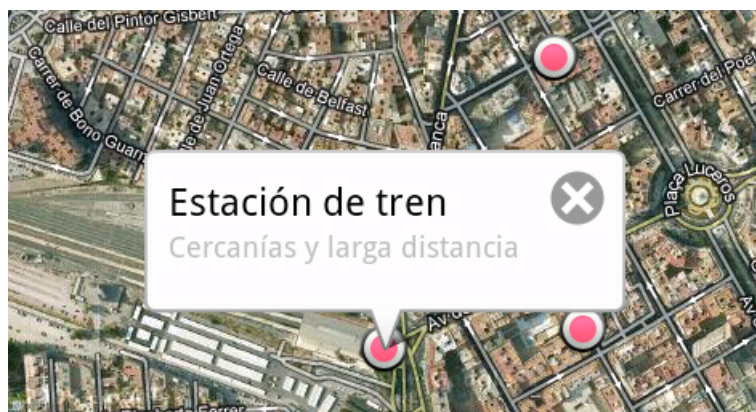
```
class RestaurantesItemizedOverlay
```

```

    extends BalloonItemizedOverlay<OverlayItem> {
    public RestaurantesItemizedOverlay(Drawable defaultMarker,
                                       MapView mapView) {
        super(boundCenter(defaultMarker), mapView);
        ...
    }
    ...
}

```

Con esto se mostrará la información de los puntos del mapa en globos como se muestra a continuación:



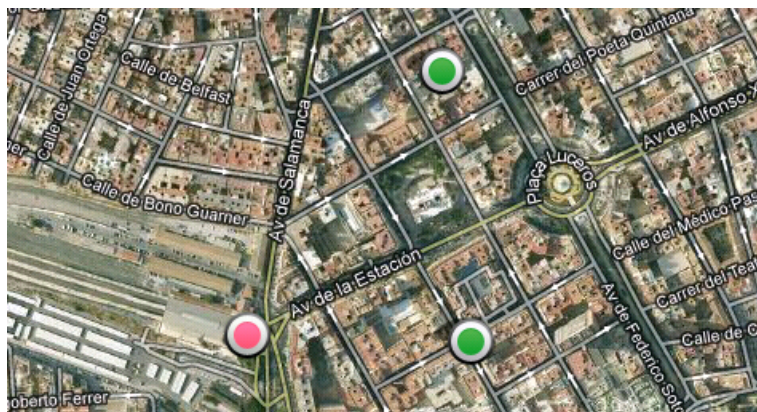
Marcadores con globo informativo

Para cada *item*, también podemos cambiar el tipo de marcador a mostrar, para así no mostrar siempre el mismo marcador por defecto. Esto lo haremos con el método `setMarker` de `OverlayItem`, pero debemos tener en cuenta que siempre deberemos haber definido de forma explícita los límites (*bounds*) del *drawable*:

```

Drawable marcador = this.getResources().getDrawable(R.drawable.marker2);
marcador.setBounds(0, 0, marcador.getIntrinsicWidth(),
                  marcador.getIntrinsicHeight());
overlay.setMarker(marcador);

```



Marcadores de distintos tipos

3. Reconocimiento del habla

Otro sensor que podemos utilizar para introducir información en nuestras aplicaciones es el micrófono que incorpora el dispositivo. Tanto el micrófono como la cámara se pueden utilizar para capturar audio y video. Una característica altamente interesante de los dispositivos Android es que nos permiten realizar reconocimiento del habla de forma sencilla para introducir texto en nuestras aplicaciones.

Para realizar este reconocimiento deberemos utilizar *intents*. Concretamente, crearemos un `Intent` mediante las constantes definidas en la clase `RecognizerIntent`, que es la clase principal que deberemos utilizar para utilizar esta característica.

Lo primer que deberemos hacer es crear un `Intent` para inicial el reconocimiento:

```
Intent intent = new Intent(
    RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
```

Una vez creado, podemos añadir una serie de parámetros para especificar la forma en la que se realizará el reconocimiento. Estos parámetros se introducen llamando a:

```
intent.putExtra(parametro, valor);
```

Los parámetros se definen como constantes de la clase `RecognizerIntent`, todas ellas tienen el prefijo `EXTRA_`. Algunos de estos parámetros son:

Parámetro	Valor
<code>EXTRA_LANGUAGE_MODEL</code>	Obligatorio. Debemos especificar el tipo de lenguaje utilizado. Puede ser lenguaje orientado a realizar una búsqueda web (<code>LANGUAGE_MODEL_WEB_SEARCH</code>), o lenguaje de tipo general (<code>LANGUAGE_MODEL_FREE_FORM</code>).
<code>EXTRA_LANGUAGE</code>	Opcional. Se especifica para hacer el reconocimiento en un idioma diferente al idioma por defecto del dispositivo. Indicaremos el idioma mediante la etiqueta IETF correspondiente, como por ejemplo "es-ES" o "en-US".
<code>EXTRA_PROMPT</code>	Opcional. Nos permite indicar el texto a mostrar en la pantalla mientras se realiza el reconocimiento. Se especifica mediante una cadena de texto.
<code>EXTRA_MAX_RESULTS</code>	Opcional. Nos permite especificar el número máximo de posibles resultados que queremos que nos devuelva. Se especifica mediante un número entero.

Una vez creado el *intent* y especificados los parámetros, podemos lanzar el reconocimiento llamando, desde nuestra actividad, a:

```
startActivityForResult(intent, codigo);
```

Como código deberemos especificar un entero que nos permita identificar la petición que estamos realizando. En la actividad deberemos definir el *callback* `onActivityResult`, que será llamado cuando el reconocimiento haya finalizado. Aquí deberemos comprobar en primer lugar que el código de petición al que corresponde el *callback* es el que pusimos al lanzar la actividad. Una vez comprobado esto, obtendremos una lista con los resultados obtenidos de la siguiente forma:

```
@Override
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
    if (requestCode == codigo && resultCode == RESULT_OK) {

        ArrayList<String> resultados =
            data.getStringArrayListExtra(
                RecognizerIntent.EXTRA_RESULTS);

        // Utilizar los resultados obtenidos
        ...
    }
    super.onActivityResult(requestCode, resultCode, data);
}
```

