Module: Mobile Application development (Android)

Session 35: Google Maps, MapView, MapActivity

**Google Maps, MapView, MapActivity**

1 The Elements of the Google Maps Android API

2 Getting Ready to use the Google Maps Android API

2.1 Installing the Google APIs

2.2 Downloading the Google Play Services SDK

2.3 Adding the Google Play Services Library Project to the Eclipse Workspace

2.4 Adding the Google Play Services Library to a Project Build Path

2.5 Obtaining Your Developer Signature

2.6 Registering the Project in the Google APIs Console

3 Adding Map Support to the AndroidManifest.xml File

4 Checking for Google Play Services Support

5 Understanding Geocoding and Reverse Geocoding

6 Adding a Map to an Application

7 Displaying the User’s Current Location

8 Changing the Map Type

9 Displaying Map Controls to the User

10 Handling Map Gesture Interaction

10.1 Map Zooming Gestures

10.2 Map Scrolling/Panning Gestures

10.3 Map Tilt Gestures

10.4 Map Rotation Gestures

11 Creating Map Markers

12 Controlling the Map Camera

When Google decided to introduce a map service many years ago, it is hard to say whether or not they ever anticipated having a version available for integration into mobile applications. When the first web based version of what would eventually be called Google Maps was introduced in 2005, the iPhone had yet to ignite the smartphone revolution and the company that was developing the Android operating system would not be acquired by Google for another six months. Whatever aspirations Google had for the future of Google Maps, it is remarkable to consider that all of the power of Google Maps can now be accessed directly via Android applications using the Google Maps Android API.

This session is intended to provide an overview of the Google Maps system and Google Maps Android API. The chapter will provide an overview of the different elements that make up the API, detail the steps necessary to configure a development environment to work with Google Maps and then work through some code examples.

## The Elements of the Google Maps Android API

The Google Maps Android API consists of a core set of classes that combine to provide mapping capabilities in Android applications. The key elements of a map are as follows:

* **GoogleMap** – The main class of the Google Maps Android API. This class is responsible for downloading and displaying map tiles and for displaying and responding to map controls. The GoogleMap object is not created directly by the application but is instead created when MapView or MapFragment instances are created. A reference to the GoogleMap object can be obtained within application code via a call to the getMap() method of a MapView or MapFragment instance.
* **MapView** - A subclass of the View class, this class provides the view canvas onto which the map is drawn by the GoogleMap object, allowing a map to be placed in the user interface layout of an activity.
* **MapFragment** – A subclass of the Fragment class, this class allows a map to be placed within a Fragment in an Android layout.
* **Marker** – The purpose of the Marker class is to allow locations to be marked on a map. Markers are added to a map by obtaining a reference to the GoogleMap object associated with a map and then making a call to the addMarker() method of that object instance. The position of a marker is defined via Longitude and Latitude. Markers can be configured in a number of ways, including specifying a title, text and an icon. Markers may also be made to be “draggable”, allowing the user to move the marker to different positions on a map.
* **Shapes** – The drawing of lines and shapes on a map is achieved through the use of the Polyline, Polygon and Circle classes.
* **UiSettings** – The UiSettings class provides a level of control from within an application of which user interface controls appear on a map. Using this class, for example, the application can control whether or not the zoom, current location and compass controls appear on a map. This class can also be used to configure which touch screen gestures are recognized by the map.
* **My Location Layer** – When enabled, the My Location Layer displays a button on the map which, when selected by the user, centers the map on the user’s current geographical location. If the user is stationary, this location is represented on the map by a blue marker. If the user is in motion the location is represented by a chevron indicating the user’s direction of travel

## Getting Ready to use the Google Maps Android API

The use of the Google Maps Android API in an Android application is somewhat unusual in that there is more work involved in setting up the environment than there is in actually writing the Java code. Each step must, however, be performed carefully and with great attention to detail to ensure that maps will function within an application. Even the slightest error or missed step will prevent Maps from appearing within an application.

### Installing the Google APIs

The APIs for using Google Maps are not included in the standard Android SDK. Instead, they must be downloaded separately before maps can be integrated into an application. To install the Google APIs, start the Android SDK Manager (Window -> Android SDK Manager within Eclipse) and locate the section for the latest SDK from the list of packages. From within this category, make sure that the Status entry for Google APIs is set to Installed as illustrated in Figure 1:

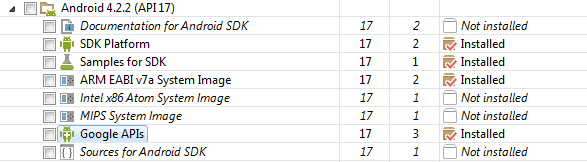
[](http://www.techotopia.com/index.php/Image:Android_google_apis_installed.png)

Figure 1

If the package is listed as not installed, select the check box next to the package name before clicking on the Install packages… button.

### Downloading the Google Play Services SDK

The Google Maps Android API is part of the Google Play services SDK, which will need to be downloaded and integrated into any Eclipse project for which maps are to be included. The SDK can be downloaded using the Android SDK Manager.

To perform the SDK download, return to the Android SDK Manager, scroll down to the section entitled Extras and unfold it if necessary as outlined in Figure 2:

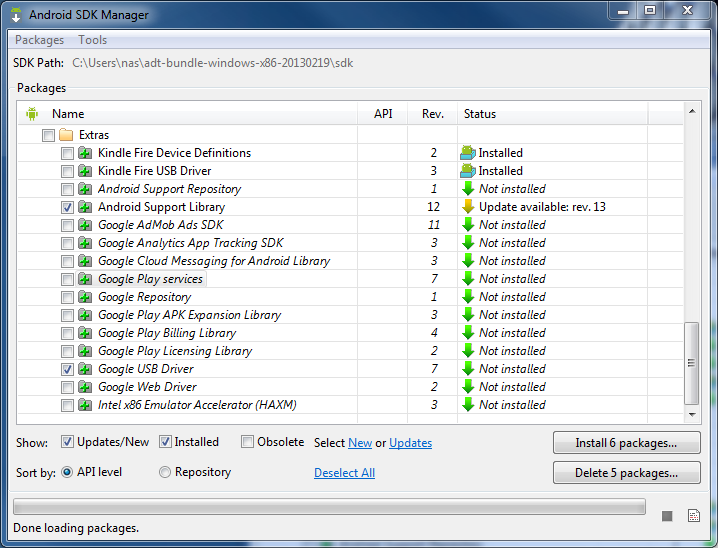
[](http://www.techotopia.com/index.php/Image:Android_install_google_play_services.png)

Figure 2

Enable the checkbox next to the Google Play services item in the list before clicking on the Install packages… button to initiate the installation process. Once the download has completed, the SDK will have been installed into <sdk path>/extras/google/google\_play\_services, where <sdk\_path> is the location into which the Android ADT bundle was installed in the chapter entitled Setting up an Android Development Environment.

### Adding the Google Play Services Library Project to the Eclipse Workspace

The Google Play services library project will need to be added to your Eclipse workspace in order for it to be available for use in projects. To achieve this, select the Eclipse File -> Import… menu option. In the resulting dialog, select the Android -> Existing Android Code into Workspace option before clicking on the Next > button.

On the Import Project panel, click on the Browse… button next to the Root Directory: text box and navigate to the following file system location (where <sdk path> is the path into which the Android SDK was previously installed on your system):

<sdk path>/extras/google/google\_play\_services/libproject

Once the libproject folder has been selected, click on OK in the browser window, followed by Finish in the Import Projects window. On completion of a successful import, the google-play-services\_lib project should appear in the Project Explorer panel.

### Adding the Google Play Services Library to a Project Build Path

The Google Play services library project will need to be added to the build path of any Eclipse project requiring Google Maps functionality. To add map support to a project, locate the project within the Eclipse Package Explorer panel, right click on the project name and select the Properties option from the resulting menu.

Within the Properties dialog, select the Android category from the left hand panel and, within the right hand panel, click on the Add… button in the Library section. This will display the Project Selection dialog (Figure 3) with the previously imported google-play-services\_lib library project listed.

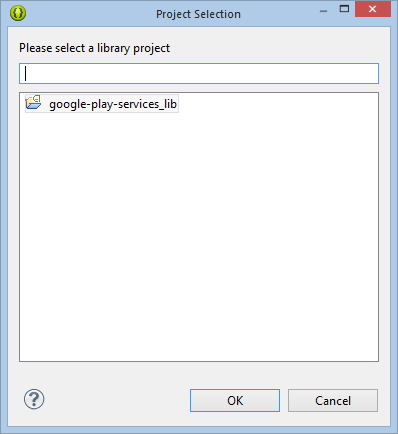
[](http://www.techotopia.com/index.php/Image:Android_add_google_play_library.png)

Figure 3

With the google-play-services\_lib project selected, click on OK to close the selection dialog followed by the Apply button in the properties panel. The library project should then be listed within the Library section of the properties panel as illustrated in Figure 4:

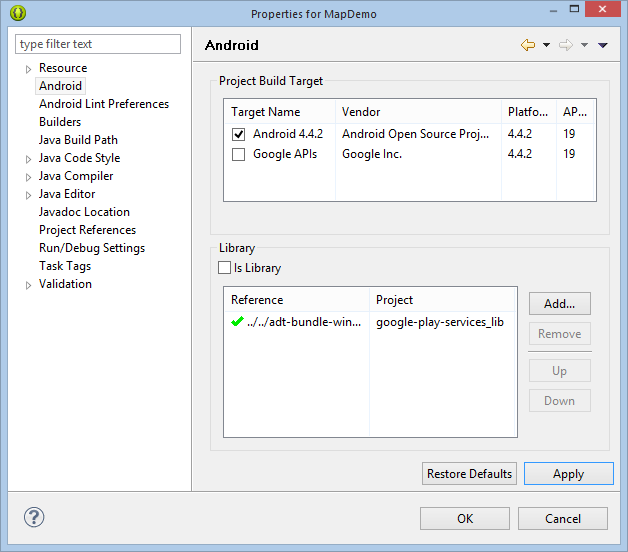
[](http://www.techotopia.com/index.php/Image:Android_google_play_library_added.png)

Figure 4

### Obtaining Your Developer Signature

Before an application can make use of the Google Maps Android API, it must first be registered within the Google APIs Console. Before an app can be registered, however, the developer signature (also referred to as the SHA-1 fingerprint) associated with your development environment must be obtained. This can be found from within the Eclipse Preferences dialog. Access this dialog by selecting the Eclipse Window -> Preferences menu option. Once loaded, click on the small right facing arrow next to Android in the left hand panel to unfold the Android sub-categories. From this list, select the Build item to display the build settings as shown in Figure 42-5.

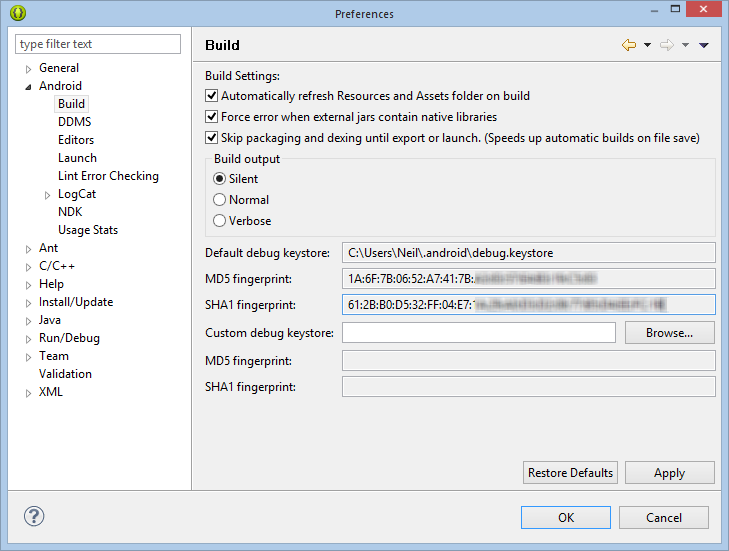
[](http://www.techotopia.com/index.php/Image:Eclipse_sha_fingerprint.png)

Figure 5

The SHA1 fingerprint is listed under the default debug keystore path next to the SHA1 fingerprint: label and consists of 20 sets of 2 digit hexadecimal numbers separated by colons.

### Registering the Project in the Google APIs Console

The next step is to create a project in the Google APIs Console and enable it to make use of the Google Maps Android API service. The Google APIs Console can be accessed using the following URL where you will be required to sign in using your existing Google account information:

https://code.google.com/apis/console/

Currently, the generated API key in this example is configured only for use with an application with the package name of com.example.AndroidTest. Additional applications may be added to the API key by clicking on the Edit allowed Android applications… link and entering additional lines consisting of the SHA-1 fingerprint and the application package name.

At this point, the development environment is set up to enable maps to be used within a specific application. The next step is to set up the application itself to use maps. This begins with making some additions to the application’s Android manifest file.

## Adding Map Support to the AndroidManifest.xml File

Before maps can be used in an application, some additional entries need to be added to the application’s Android Manifest file. Within Eclipse, locate the AndroidManifest.xml file for the project for which maps support is required and for which the package name was used when generating the API key. Immediately before the </application> tag, enter the following lines (where <api\_key> is replaced by the API Key generated for your development system and application in the API Console):

<meta-data

android:name="com.google.android.maps.v2.API\_KEY"

android:value="<api\_key>"/>

<meta-data

android:name="com.google.android.gms.version"

android:value="@integer/google\_play\_services\_version" />

In addition, a number of permissions need to be added to the manifest file. The following permissions are mandatory, where <package\_name> is replaced by the application package name (for example com.example.AndroidTest):

<permission

android:name="<package\_name>.permission.MAPS\_RECEIVE"

android:protectionLevel="signature"/>

<uses-permission

android:name="<package\_name>.permission.MAPS\_RECEIVE"/>

<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"/>

The following permissions are optional, but are necessary if the application is required to obtain the current location of the device:

<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION"/>

<uses-permission android:name="android.permission.ACCESS\_FINE\_LOCATION"/>

Finally, an entry needs to be added to the manifest file to indicate that the maps features are only available when the application is running on a device that supports OpenGL ES version 2 or later:

<uses-feature

android:glEsVersion="0x00020000"

android:required="true"/>

Bringing these settings together, a typical AndroidManifest.xml file with Google Maps support might, therefore, read as follows:

<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.AndroidTest"

android:versionCode="1"

android:versionName="1.0" >

<permission

android:name="com.example.AndroidTest.permission.MAPS\_RECEIVE"

android:protectionLevel="signature"/>

<uses-permission

android:name="com.example.AndroidTest.permission.MAPS\_RECEIVE"/>

<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"/>

<uses-permission

android:name="android.permission.ACCESS\_COARSE\_LOCATION"/>

<uses-permission

android:name="android.permission.ACCESS\_FINE\_LOCATION"/>

<uses-feature

android:glEsVersion="0x00020000"

android:required="false"/>

<uses-sdk

android:minSdkVersion="19"

android:targetSdkVersion="19" />

<application

android:allowBackup="true"

android:icon="@drawable/ic\_launcher"

android:label="@string/app\_name"

android:theme="@style/AppTheme" >

<activity

android:name="com.example.AndroidTest.AndroidTestActivity"

android:label="@string/app\_name" >

<intent-filter>

<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />

</intent-filter>

</activity>

<meta-data

android:name="com.google.android.maps.v2.API\_KEY"

android:value="<api\_key>"/>

<meta-data

android:name="com.google.android.gms.version"

android:value="@integer/google\_play\_services\_version" />

</application>

</manifest>

## Checking for Google Play Services Support

Since the use of the Google Maps Android API requires that the target device support Google Play services, it is important that the application check to make sure those services are available and enabled on a device before attempting to render a map. This can be achieved by making a call to the isGooglePlayServicesAvailable() method contained within the GooglePlayServicesUtil package.

When called, this method will return a status value indicating whether the service is available, missing, invalid, disabled or requires an update. Some of the error states are categorized as being user recoverable. Whether or not the error is recoverable or not can be identified by a call to the GooglePlayServicesUtil isUserRecoverableError() method, passing through the status value as an argument.

In the event that the failure is user recoverable, a call to the getErrorDialog() method will display a dialog to the user providing the option either to enable Google Play on the device or, if necessary, download and install the Google Play services package from the Google Play Store.

|  |  |  |
| --- | --- | --- |
|  |  |  |

## Understanding Geocoding and Reverse Geocoding

It is impossible to talk about maps and geographical locations without first covering the subject of Geocoding. Geocoding can best be described as the process of converting a textual based geographical location (such as a street address) into geographical coordinates expressed in terms of longitude and latitude.

Geocoding can be achieved using the Android Geocoder class. An instance of the Geocoder class can, for example, be passed a string representing a location such as a city name, street address or airport code. The Geocoder will attempt to find a match for the location and return a list of Address objects that potentially match the location string, ranked in order with the closest match at position 0 in the list. A variety of information can then be extracted from the Address objects, including the longitude and latitude of the potential matches.

The following code, for example, requests the location of the National Air and Space Museum in Washington, D.C.:

double latitude;

double longitude;

List<Address> geocodeMatches = null;

try {

geocodeMatches =

new Geocoder(this).getFromLocationName(

"600 Independence Ave SW, Washington, DC 20560", 1);

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

if (!geocodeMatches.isEmpty())

{

latitude = geocodeMatches.get(0).getLatitude();

longitude = geocodeMatches.get(0).getLongitude();

}

Note that the value of 1 is passed through as the second argument to the getFromLocationName() method. This simply tells the Geocoder to return only one result in the array. Given the specific nature of the address provided, there should only be one potential match. For more vague location names, however, it may be necessary to request more potential matches and allow the user to choose the correct one.

The above code is an example of forward-geocoding in that coordinates are calculated based on a text location description. Reverse-geocoding, as the name suggests, involves the translation of geographical coordinates into a human readable address string. Consider, for example, the following code:

List<Address> geocodeMatches = null;

String Address1;

String Address2;

String State;

String Zipcode;

String Country;

try {

geocodeMatches =

new Geocoder(this).getFromLocation(38.8874245, -77.0200729, 1);

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

if (!geocodeMatches.isEmpty())

{

Address1 = geocodeMatches.get(0).getAddressLine(0);

Address2 = geocodeMatches.get(0).getAddressLine(1);

State = geocodeMatches.get(0).getAdminArea();

Zipcode = geocodeMatches.get(0).getPostalCode();

Country = geocodeMatches.get(0).getCountryName();

}

In this case the Geocoder object is initialized with latitude and longitude values via the getFromLocation() method. Once again, only a single matching result is requested. The text based address information is then extracted from the resulting Address object.

It should be noted that the geocoding is not actually performed on the Android device, but rather on a server to which the device connects when a translation is required and the results subsequently returned when the translation is complete. As such, geocoding can only take place when the device has an active internet connection.

## Adding a Map to an Application

The simplest way to add a map to an application is to specify it in the user interface layout XML file for an activity. The following example layout file shows a MapFragment instance added as the child of a RelativeLayout view:

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

tools:context=".AndroidTestActivity" >

<fragment

android:id="@+id/map"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:name="com.google.android.gms.maps.MapFragment"/>

</RelativeLayout>

Assuming that the onCreate() method for the activity contains the standard code to load the layout resource file, the application should run and display a map:

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_maps\_demo);

}

In the event that a map is not displayed, check the following areas:

* If the application is running on an emulator, make sure that the emulator is running a version of Android that includes the Google APIs.
* Check the LogCat output for any areas relating to authentication problems with regard to the Google Maps API. This usually means the API key was entered incorrectly or that the application package name does not match that specified when the API key was generated.
* Verify that the correct entries and permissions have been added to the Manifest file, including all references to the package name and API key.
* Verify within the Google API Console that the Google Maps Android API has been enabled in the Services panel.

## Displaying the User’s Current Location

The user’s current location may be displayed on the map by obtaining a reference to the GoogleMap object associated with the displayed map and calling the setMyLocationEnabled() method of that instance, passing through a value of true. The following code, for example, obtains a reference to the GoogleMap object from the previous example, verifies that the map instance exists and then enables the My Location Layer:

import com.google.android.gms.maps.MapFragment;

.

.

.

GoogleMap map;

map = ((MapFragment)

getFragmentManager().findFragmentById(R.id.map)).getMap();

if (map != null)

map.setMyLocationEnabled(true);

## Changing the Map Type

The type of map displayed can be modified dynamically by making a call to the setMapType() method of the corresponding GoogleMap object, passing through one of the following values:

* **GoogleMap.MAP\_TYPE\_NONE** – An empty grid with no mapping tiles displayed.
* **GoogleMap.MAP\_TYPE\_NORMAL** – The standard view consisting of the classic road map.
* **GoogleMap.MAP\_TYPE\_SATELLITE** – Displays the satellite imagery of the map region.
* **GoogleMap.MAP\_TYPE\_HYBRID** – Displays satellite imagery with the road maps superimposed.
* **GoogleMap.MAP\_TYPE\_TERRAIN** – Displays topographical information such as contour lines and colors.

The following code, for example, switches a map to Terrain mode:

map.setMapType(GoogleMap.MAP\_TYPE\_TERRAIN);

Alternatively, the map type may be specified in the XML layout file in which the map is embedded using the map:mapType property together with a value of none, normal, hybrid, satellite or terrain. In order to be able to use this directive, however, it is first necessary to add the map namespace to the XML resource file in which the directives are being used:

xmlns:map="http://schemas.android.com/apk/res-auto"

For example:

<?xml version="1.0" encoding="utf-8"?>

<fragment xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:map="http://schemas.android.com/apk/res-auto"

android:id="@+id/map"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

map:mapType="hybrid"

android:name="com.google.android.gms.maps.MapFragment"/>

## Displaying Map Controls to the User

The Google Maps Android API provides a number of controls that may be optionally displayed to the user consisting of zoom in and out buttons, a “my location” button and a compass.

Whether or not the zoom and compass controls are displayed may be controlled either programmatically or within the map element in XML layout resources. In order to configure the controls programmatically, a reference to the UiSettings object associated with the GoogleMap object must be obtained:

UiSettings mapSettings;

mapSettings = map.getUiSettings();

The zoom controls are enabled and disabled via calls to the setZoomControlsEnabled() method of the UiSettings object. For example:

mapSettings.setZoomControlsEnabled(true);

Alternatively, the map:uiZoomControls property may be set within the map element of the XML resource file:

map:uiZoomControls="false"

The compass may be displayed either via a call to the setCompassEnabled() method of the UiSettings instance, or through XML resources using the map:uiCompass property.

The “My Location” button will only appear when My Location mode is enabled as outlined earlier in this chapter. The button may be prevented from appearing even when in this mode via a call to the setMyLocationButtonEnabled() method of the UiSettings instance.

## Handling Map Gesture Interaction

The Google Maps Android API is capable of responding to a number of different user interactions. These interactions can be used to change the area of the map displayed, the zoom level and even the angle of view (such that a 3D representation of the map area is displayed for certain cities).

### Map Zooming Gestures

Support for gestures relating to zooming in and out of a map may be enabled or disabled using the setZoomControlsEnabled() method of the UiSettings object associated with the GoogleMap instance. For example, the following code enables zoom gestures for our example map:

UiSettings mapSettings;

mapSettings = map.getUiSettings();

mapSettings.setZoomGesturesEnabled(true);

The same result can be achieved within an XML resource file by setting the map:uiZoomGestures property to true or false.

When enabled, zooming will occur when the user makes pinching gestures on the screen. Similarly, a double tap will zoom in whilst a two finger tap will zoom out. One finger zooming gestures, on the other hand, are performed by tapping twice but not releasing the second tap and then sliding the finger up and down on the screen to zoom in and out respectively.

### Map Scrolling/Panning Gestures

A scrolling, or panning gesture allows the user to move around the map by dragging the map around the screen with a single finger motion. Scrolling gestures may be enabled within code via a call to the setScrollGesturesEnabled() method of the UiSettings instance:

UiSettings mapSettings;

mapSettings = map.getUiSettings();

mapSettings.setScrollGesturesEnabled(true);

Alternatively, scrolling on a map instance may be enabled in an XML resource layout file using the map:uiScrollGestures property.

### Map Tilt Gestures

Tilt gestures allow the user to tilt the angle of projection of the map by placing two fingers on the screen and moving them up and down to adjust the tilt angle. Tilt gestures may be enabled or disabled via a call to the setTiltGesturesEnabled() method of the UiSettings instance, for example:

UiSettings mapSettings;

mapSettings = map.getUiSettings();

mapSettings.setTiltGesturesEnabled(true);

Tilt gestures may also be enabled and disabled using the map:uiTiltGestures property in an XML layout resource file.

### Map Rotation Gestures

By placing two fingers on the screen and rotating then in a circular motion, the user may rotate the orientation of a map when map rotation gestures are enabled. This gesture support is enabled and disabled in code via a call to the setRotateGesturesEnabled() method of the UiSettings instance, for example:

UiSettings mapSettings;

mapSettings = map.getUiSettings();

mapSettings.setRotateGesturesEnabled(true);

Rotation gestures may also be enabled or disabled using the map:uiRotateGestures property in an XML layout resource file.

## Creating Map Markers

Markers are used to notify the user of locations on a map and take the form of either a standard or custom icon. Markers may also include a title and optional text (referred to as a snippet) and may be configured such that they can be dragged to different locations on the map by the user. When tapped by the user an info window will appear displaying additional information about the marker location.

Markers are represented by instances of the Marker class and are added to a map via a call to the addMarker() method of the corresponding GoogleMap object. Passed through as an argument to this method is a MarkerOptions class instance containing the various options required for the marker such as the title and snippet text. The location of a marker is defined by specifying latitude and longitude values, also included as part of the MarkerOptions instance. For example, the following code adds a marker including a title, snippet and a position to a specific location on the map:

LatLng MUSEUM = new LatLng(38.8874245, -77.0200729);

Marker museum = map.addMarker(new MarkerOptions()

.position(MUSEUM)

.title("Museum")

.snippet("National Air and Space Museum"));

When executed, the above code will mark the location specified which, when tapped, will display an info window containing the title and snippet as shown in Figure 10:

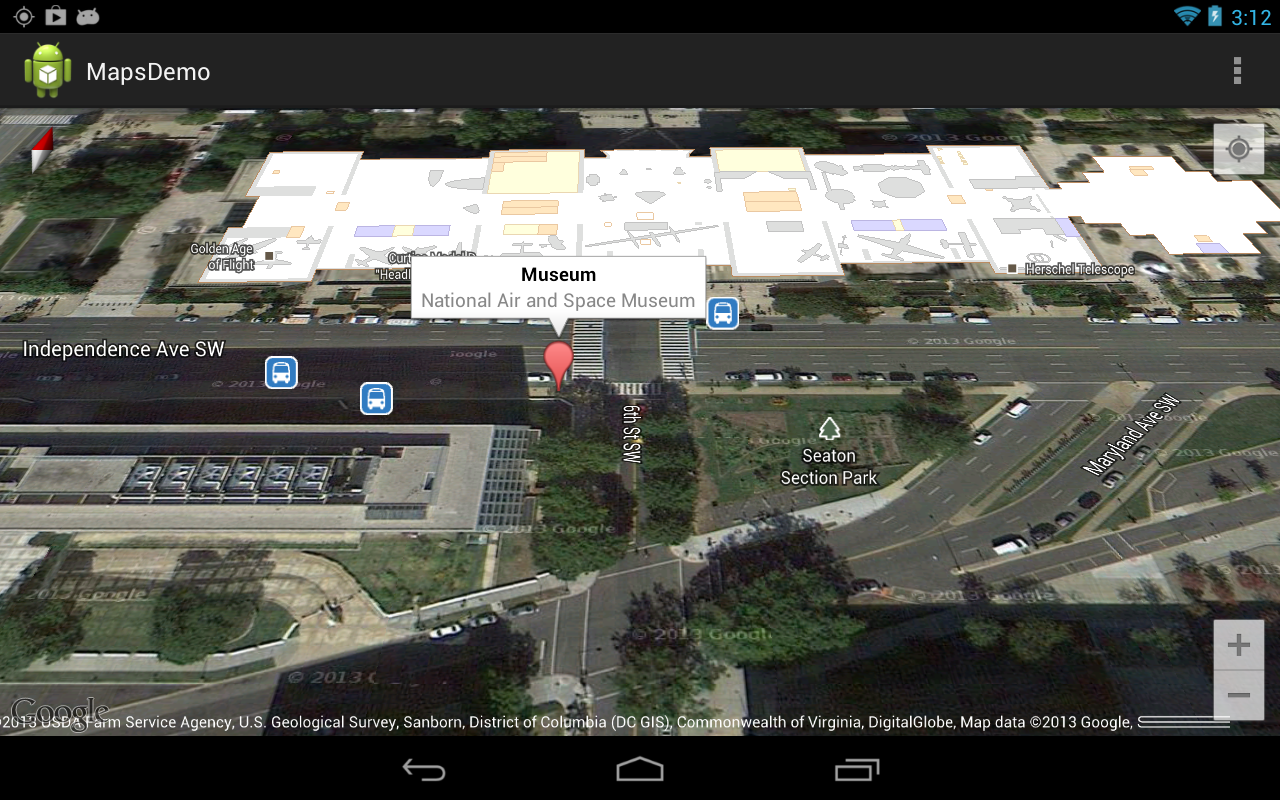
[](http://www.techotopia.com/index.php/Image:Google_map_marker.png)

Figure 10

## Controlling the Map Camera

Because Android device screens are flat and the world is a sphere, the Google Maps Android API uses the Mercator projection to represent the earth on a flat surface. The default view of the map is presented to the user as though through a camera suspended above the map and pointing directly down at the map. The Google Maps Android API allows the target, zoom, bearing and tilt of this camera to be changed in real-time from within the application:

* **Target** – The location of the center of the map within the device display specified in terms of longitude and latitude.
* **Zoom** – The zoom level of the camera specified in levels. Increasing the zoom level by 1.0 doubles the width of the amount of the map displayed.
* **Tilt** – The viewing angle of the camera specified as a position on an arc spanning directly over the center of the viewable map area measured in degrees from the top of the arc (this being the nadir of the arc where the camera points directly down to the map).
* **Bearing** – The orientation of the map in degrees measured in a clockwise direction from North.

Camera changes are made by creating an instance of the CameraUpdate class with the appropriate settings. CameraUpdate instances are created by making method calls to the CameraUpdateFactory class. Once a CameraUpdate instance has been created, it is applied to the map via a call to the moveCamera() method of the GoogleMap instance. To obtain a smooth animated effect as the camera changes, the animateCamera() method may be called instead of moveCamera().

A summary of CameraUpdateFactory methods is as follows:

* **CameraUpdateFactory.zoomIn()** – Provides a CameraUpdate instance zoomed in by one level.
* **CameraUpdateFactory.zoomOut()** - Provides a CameraUpdate instance zoomed out by one level.
* **CameraUpdateFactory.zoomTo(float)** - Generates a CameraUpdate instance that changes the zoom level to the specified value.
* **CameraUpdateFactory.zoomBy(float)** – Provides a CameraUpdate instance with a zoom level increased or decreased by the specified amount.
* **CameraUpdateFactory.zoomBy(float, Point)** - Creates a CameraUpdate instance that increases or decreases the zoom level by the specified value.
* **CameraUpdateFactory.newLatLng(LatLng)** - Creates a CameraUpdate instance that changes the camera's target latitude and longitude.
* **CameraUpdateFactory.newLatLngZoom(LatLng, float)** - Generates a CameraUpdate instance that changes the camera's latitude, longitude and zoom.
* **CameraUpdateFactory.newCameraPosition(CameraPosition)** - Returns a CameraUpdate instance that moves the camera to the specified position. A CameraPosition instance can be obtained using CameraPosition.Builder().

The following code, for example, zooms in the camera by one level using animation:

map.animateCamera(CameraUpdateFactory.zoomIn());

The following code, on the other hand, moves the camera to a new location and adjusts the zoom level to 10 without animation:

private static final LatLng MUSEUM =

new LatLng(38.8874245, -77.0200729);

map.moveCamera(CameraUpdateFactory.newLatLngZoom(MUSEUM, 10));

Finally, the next code example uses CameraPosition.Builder() to create a CameraPosition object with changes to the target, zoom, bearing and tilt. This change is then applied to the camera using animation:

CameraPosition cameraPosition = new CameraPosition.Builder()

.target(MUSEUM)

.zoom(10)

.bearing(70)

.tilt(25)

.build();

map.animateCamera(CameraUpdateFactory.newCameraPosition(cameraPosition));