Google Map View

Using the Google Maps library, you can create your own map-viewing Activity. In this tutorial, you'll create a simple map application in two parts. In Part 1, you'll create an app that shows a map the user can pan and zoom. In Part 2, you'll add overlay items that mark points of interest.

This tutorial requires that you have the external Google Maps library installed in your SDK environment. The Maps library is included with the Google APIs add-on, which you can install using the Android SDK and AVD Manager. To learn how, see Adding SDK Components.

After installing the Google APIs add-on in your SDK, set your project properties to use the build target called "Google APIs by Google Inc.". See the instructions for setting a build target inCreating and Managing Projects in Eclipse or Creating and Managing Projects on the Command Line, as appropriate for your environment.

You will also need to set up a new AVD that uses the same Google APIs deployment target. See Creating and Managing Virtual Devices for more information.

For reference material, see the Google Maps library documentation.

Part 1: Creating a Map Activity

1. Start a new project named *HelloGoogleMaps*.
2. Because the Maps library is not a part of the standard Android library, you must declare it in the Android Manifest. Open the `AndroidManifest.xml` file and add the following as a child of the `<application>` element:

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

3. You also need access to the Internet in order to retrieve map tiles, so you must also request the `INTERNET` permission. In the manifest file, add the following as a child of the `<manifest>`element:

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

4. While you're in the manifest, give the map some more space by getting rid of the title bar with the "NoTitleBar" theme:

```
<activity android:name=".HelloGoogleMaps" android:label="@string/app_name"

     android:theme="@android:style/Theme.NoTitleBar">
```

5. Open the `res/layout/main.xml` file and add a single `com.google.android.maps.MapView` as the root node:

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

```
<com.google.android.maps.MapView

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

    android:id="@+id/mapview"

    android:layout_width="fill_parent"

    android:layout_height="fill_parent"

    android:clickable="true"

    android:apiKey="Your Maps API Key goes here"

/>
```

The `android:clickable` attribute defines whether you want to allow user-interaction with the map. If this is "false" then touching the map does nothing.

The `android:apiKey` attribute holds the Maps API Key for your application, which proves your application and signer certificate has been registered with the Maps service. This is required in order to receive the map data, even while you are developing. Registration to the service is free and it only takes a couple minutes to register your certificate and get a Maps API Key.

Go now to get a key. For instructions, read Obtaining a Maps API Key. For the purpose of this tutorial, you should register with the SDK debug certificate, which will only be valid while your application is signed with the debug key (once you sign with your private key, you will need a new API key). When you get your key, insert it for the value of `android:apiKey`.

6. Now open the `HelloGoogleMaps.java` file. For this Activity, extend `MapActivity` (instead of `android.app.Activity`):

```
public class HelloGoogleMaps extends MapActivity {
```

This is a special sub-class of `Activity`, provided by the Maps library, which provides important map capabilities.

7. Inside every `MapActivity`, the `isRouteDisplayed()` method is required, so override this method:

```
@Override

protected boolean isRouteDisplayed() {
```

```
    return false;

}
```

This method is required for some accounting from the Maps service to see if you're currently displaying any route information. In this case, you're not, so return false.

8. Now add the standard `onCreate()` callback method to the class:

```
@Override

public void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);

}
```

This loads the layout file created above. In fact, this is now a workable application that will display map tiles and allow the user to pan around the map. But there's no ability to zoom. Fortunately, there's a very simple zoom feature built into the `MapView` class, which you can summon with `setBuiltInZoomControls(boolean)`. Do this at the end of the`onCreate()` method:

```
    MapView mapView = (MapView) findViewById(R.id.mapview);

    mapView.setBuiltInZoomControls(true);
```

9. That's all there is to it. Run the application. (Remember, you must have an AVD configured to use the Google APIs target, or be using a development device that includes the Maps library.)

Part 2: Adding Overlay Items

So, now you have a map, but in many cases you'll also want to create your own map markers and lay-overs. That's what you'll do now. In order to do so, you must implement the`ItemizedOverlay` class, which can manage a whole set of `Overlay` (which are the individual items placed on the map).

1. Create a new Java class named `HelloItemizedOverlay` that implements `ItemizedOverlay`.

When using Eclipse, right-click the package name in the Eclipse Package Explorer, and select **New** > **Class**. Fill-in the Name field as *HelloItemizedOverlay*. For the Superclass, enter "com.google.android.maps.ItemizedOverlay". Click the checkbox for *Constructors from superclass*. Click Finish.

2. First, you need an `OverlayItem` `ArrayList`, in which you'll put each of the `OverlayItem` objects you want on the map. Add this at the top of the `HelloItemizedOverlay` class:

```
private ArrayList<OverlayItem> mOverlays = new ArrayList<OverlayItem>();
```

3. Now define the `HelloItemizedOverlay` constructors. The constructor must define the default marker for each of the `OverlayItem`s. In order for the `Drawable` to actually get drawn, it must have its bounds defined. Most commonly, you want the center-point at the bottom of the image to be the point at which it's attached to the map coordinates. This is handled for you with the `boundCenterBottom()` method. Wrap this around our defaultMarker, so the super constructor call looks like this:

```
public HelloItemizedOverlay(Drawable defaultMarker) {

  super(boundCenterBottom(defaultMarker));

}
```

4. In order to add new `OverlayItem`s to the ArrayList, you need a new method:

```
public void addOverlay(OverlayItem overlay) {

    mOverlays.add(overlay);

    populate();

}
```

Each time you add a new `OverlayItem` to the ArrayList, you must call `populate()` for the `ItemizedOverlay`, which will read each of the `OverlayItem`s and prepare them to be drawn.

5. When the `populate()` method executes, it will call `createItem(int)` in the `ItemizedOverlay` to retrieve each `OverlayItem`. You must override this method to properly read from the ArrayList and return the `OverlayItem` from the position specified by the given integer. Your override method should look like this:

```
@Override

protected OverlayItem createItem(int i) {
```

```
  return mOverlays.get(i);

}
```

6. You must also override the `size()` method to return the current number of items in the ArrayList:

```
@Override

public int size() {

  return mOverlays.size();

}
```

7. Now set up the ability to handle touch events on the overlay items. First, you're going to need a reference to the application `Context` as a member of this class. So add `Context mContext` as a class member, then initialize it with a new class constructor:

```
public HelloItemizedOverlay(Drawable defaultMarker, Context context) {

  super(boundCenterBottom(defaultMarker));

  mContext = context;

}
```

This passes the `defaultMarker` up to the default constructor to bound its coordinates and then initialize `mContext` with the given `Context`.

Then override the `onTap(int)` callback method, which will handle the event when an item is tapped by the user:

```
@Override

protected boolean onTap(int index) {

  OverlayItem item = mOverlays.get(index);

  AlertDialog.Builder dialog = new AlertDialog.Builder(mContext);
```

```
    dialog.setTitle(item.getTitle());

    dialog.setMessage(item.getSnippet());

    dialog.show();

    return true;

}
```

This uses the member `android.content.Context` to create a new `AlertDialog.Builder` and uses the tapped `OverlayItem`'s title and snippet for the dialog's title and message text. (You'll see the `OverlayItem` title and snippet defined when you create it below.)
You're now done with the `HelloItemizedOverlay` class and can start using it to add items on the map.

Go back to the `HelloGoogleMaps` class. In the following procedure, you'll create an `OverlayItem` and add it to an instance of the `HelloItemizedOverlay` class, then add the`HelloItemizedOverlay` to the `MapView` using a `GeoPoint` to define its coordinates on the map.

1. First, you need the image for the map overlay. If you don't have one handy, use the Android on the right. Drag this image (or your own) into the `res/drawable/` directory of your project.

2. At the end of your existing `onCreate()` method, instantiate :

```
List<Overlay> mapOverlays = mapView.getOverlays();

Drawable drawable =
this.getResources().getDrawable(R.drawable.androidmarker);

HelloItemizedOverlay itemizedoverlay = new HelloItemizedOverlay(drawable,
this);
```

All overlay elements on a map are held by the `MapView`, so when you want to add some, you have to get a list from the `getOverlays()` method. Then instantiate the `Drawable` used for the map marker, which was saved in the `res/drawable/` directory. The constructor for `HelloItemizedOverlay` (your custom `ItemizedOverlay`) takes the Drawable in order to set the default marker for all overlay items.

3. Now create a `GeoPoint` that defines the map coordinates for the first overlay item, and pass it to a new `OverlayItem`:

```
GeoPoint point = new GeoPoint(19240000,-99120000);


OverlayItem overlayitem = new OverlayItem(point, "Hola, Mundo!", "I'm in
Mexico City!");
```

`GeoPoint` coordinates are specified in microdegrees (`degrees * 1e6`). The `OverlayItem` constructor accepts the `GeoPoint` location, a string for the item's title, and a string for the item's snippet text, respectively.

4. All that's left is to add this `OverlayItem` to your collection in the `HelloItemizedOverlay` instance, then add the `HelloItemizedOverlay` to the MapView:

```
itemizedoverlay.addOverlay(overlayitem);


mapOverlays.add(itemizedoverlay);
```

5. Now run the application.

You should see the following:

When you tap the overlay item, you'll see the dialog appear.

Because the `ItemizedOverlay` class uses an `java.util.ArrayList` for all of the `OverlayItem`s, it's easy to add more. Try adding another one. Before the `addOverlay()` method is called, add these lines:

```
GeoPoint point2 = new GeoPoint(35410000, 139460000);


OverlayItem overlayitem2 = new OverlayItem(point2, "Sekai, konichiwa!",
"I'm in Japan!");
```

Run the application again. (You probably need to move the map to find the new overlay item.)