6...

Security and Application Deployment

Chapter Outcomes...

- Explain the given location based service.
- Write the steps to customize the given permissions for users.
- Explain features of the given android security service.
- Write the steps to publish the given android App.

Learning Objectives...

- To understand Basic Security Concepts in Android
- To learn Location Based Services and Application Deployment
- To study SMS Telephony
- To study Components of a Android Screen

6.0 INTRODUCTION

- Android location APIs make it easy for we to build location-aware applications, without needing to focus on the details of the underlying location technology. The Location object represents a geographic location.
- The Android SDK provides many helpful telephony utilities to handle making and receiving phone calls and SMS messages (with appropriate permissions) and tools to help with formatting phone numbers entered by the user or from other sources.
- These telephony utilities enable applications to work seamlessly with the device's core phone features. Developers might also integrate voice calls and messaging features into their own applications, resulting in compelling new features. Messaging is more popular than ever, so integrating text messaging into an application can add a familiar and exciting social feature that users will likely enjoy.
- Android incorporates industry-leading security features and works with developers and device implementers to keep the Android platform and ecosystem safe. A robust security model is essential to enable a vigorous ecosystem of apps and devices built on and around the Android platform and supported by cloud services.
- Android is designed to be open.Securing an open platform requires a strong security architecture and rigorous security programs. Android was designed with multilayered security that's flexible enough to support an open platform while still protecting all users of the platform.
- Android software development is the process by which new applications are created for devices running the Android operating system. In general, creating an Android app requires the SDK (Software Development Kit), an IDE (Integrated Development Environment) like Android Studio, the Java Software Development Kit (JDK) and a virtual device to test on.

6.1 SMS TELEPHONY

- Android devices can send and receive messages to or from any other phone that supports Short Message Service (SMS).
- Android offers the Messenger app that can send and receive SMS messages. A host of third-party apps for sending and receiving SMS messages are also available in Google Play.
- This section describes how to use SMS in our app. We can add code to our app to:
  1. Launch an SMS messaging app from our app to handle all SMS communication.
  2. Send an SMS message from within our app.
  3. Receive SMS messages in our app.

Sending and Receiving SMS Messages:

- Access to the SMS features of an Android device is protected by user permissions. Just as our app needs the user's permission to use phone features, so also does an app need the user's permission to directly use SMS features.
- However, our app doesn't need permission to pass a phone number to an installed SMS app, such as Messenger, for sending the message. The Messenger app itself is governed by user permission.
- We have two choices for sending SMS messages:

1. Use an implicit Intent to launch a messaging app such as Messenger, with the ACTION_SENDTO action.
   - This is the simplest choice for sending messages. The user can add a picture or other attachment in the messaging app, if the messaging app supports adding attachments.
   - Our app doesn't need code to request permission from the user.
   - If the user has multiple SMS messaging apps installed on the Android phone, the App chooser will appear with a list of these apps, and the user can choose which one to use. (Android smartphones will have at least one, such as Messenger.)
   - The user can change the message in the messaging app before sending it.
   - The user navigates back to our app using the Back button.
2. Send the SMS message using the sendTextMessage() method or other methods of the SmsManager class.
   - This is a good choice for sending messages from our app without having to use another installed app. Our code must ask the user for permission before sending the message if the user hasn't already granted permission.
   - The user stays in our app during and after sending the message.
   - We can manage SMS operations such as dividing a message into fragments, sending a multipart message, get carrier-dependent configuration values, and so on.

- To receive SMS messages, the best practice is to use the onReceive() method of the BroadcastReceiver class. The Android framework sends out system broadcasts of events such as receiving an SMS message, containing intents that are meant to be received using a BroadcastReceiver.
- Our app receives SMS messages by listening for the SMS_RECEIVED_ACTION broadcast. Most smartphones and mobile phones support what is known as "PDU mode" for sending and receiving SMS.
- PDU (Protocol Data Unit) contains not only the SMS message, but also metadata about the SMS message, such as text encoding, the sender, SMS service center address, and much more. To access this metadata, SMS apps almost always use PDUs to encode the contents of a SMS message.
- The sendTextMessage() and sendMultimediaMessage() methods of the SmsManager class encode the contents for we. When receiving a PDU, we can create an SmsMessage object from the raw PDU using createFromPdu().
- Example: Send SMS Example:

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.linearlayout.widget.LinearLayoutxmlns:android="http://schemas.android.com/apk/
res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MainActivity">
<TextView
android:id="@+id/fstTxt"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginLeft="100dp"
android:layout_marginTop="150dp"
android:text="Mobile No" />
<EditText
android:id="@+id/mblTxt"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginLeft="100dp"
android:ems="10"/>
<TextView
android:id="@+id/secTxt"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Message"
android:layout_marginLeft="100dp" />
<EditText
android:id="@+id/msgTxt"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginLeft="100dp"
android:ems="10" />
<Button
android:id="@+id/btnSend"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:layout_marginLeft="100dp"
android:text="Send SMS" />
</androidx.linearlayout.widget.LinearLayout>
```

MainActivity.java

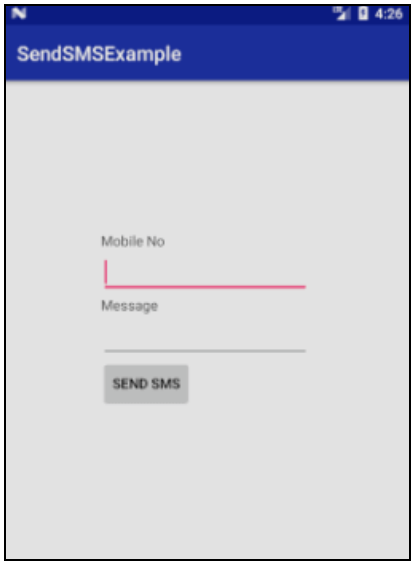
```
import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.telephony.SmsManager;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Toast;

public class MainActivityextends AppCompatActivity {
private EditTexttxtMobile;
```

```
private EditTexttxtMessage;
private Button btnSms;

    @Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
txtMobile= (EditText) findViewById(R.id.mblTxt);
txtMessage= (EditText) findViewById(R.id.msgTxt);
btnSms= (Button) findViewById(R.id.btnSend);
btnSms.setOnClickListener(new View.OnClickListener() {
public void onClick(View v) {
try {
SmsManagersmgr = SmsManager.getDefault();
smgr.sendTextMessage(txtMobile.getText().toString(), null,
txtMessage.getText().toString(), null, null);
Toast.makeText(MainActivity.this, "SMS Sent Successfully", Toast.LENGTH_SHORT).show();
                } catch (Exception e) {
Toast.makeText(MainActivity.this, "SMS Failed to Send, Please try again",
Toast.LENGTH_SHORT).show();
                }

            }

        });

    }
}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.smsdemo">
<uses-permission android:name="android.permission.SEND_SMS"/>
<application
android:allowBackup="true"
android:icon="@mipmap/ic_launcher"
android:label="@string/app_name"
android:roundIcon="@mipmap/ic_launcher_round"
android:supportsRtl="true"
android:theme="@style/AppTheme">
<activity android:name=".MainActivity">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
</manifest>
```

Output of Android Send SMS Example



6.2 LOCATION BASED SERVICES (LBSs)

- We have seen the rapid growth of mobile apps. The location based services is very well-liked, which is known as LBS.
- LBS app follows the location and offer extra services such as locating facilities close at hand, presenting suggestions for route planning and so on.
- In Location based service application map is the key components, which present the visual location of our location.

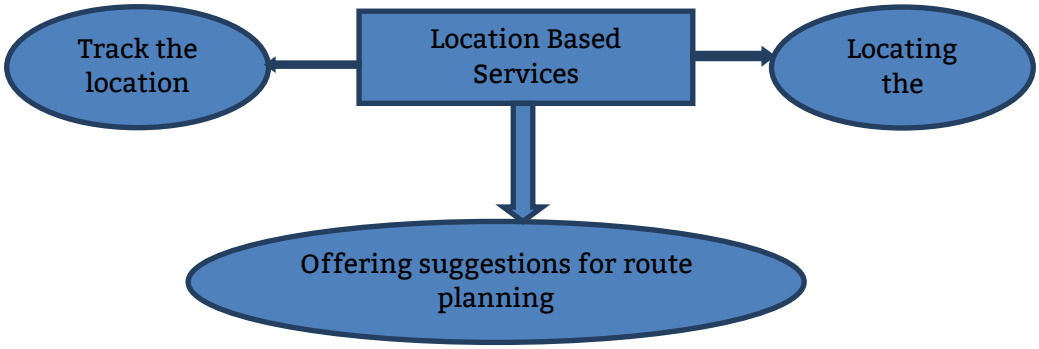


Fig. 6.1

6.2.1 Creating the Project

- To start the application we require to first creating an Android project so that we can display the Google Maps in the activity.

Step 1: Create an android project using android studio. Choose our project as Google Maps activity as follows:

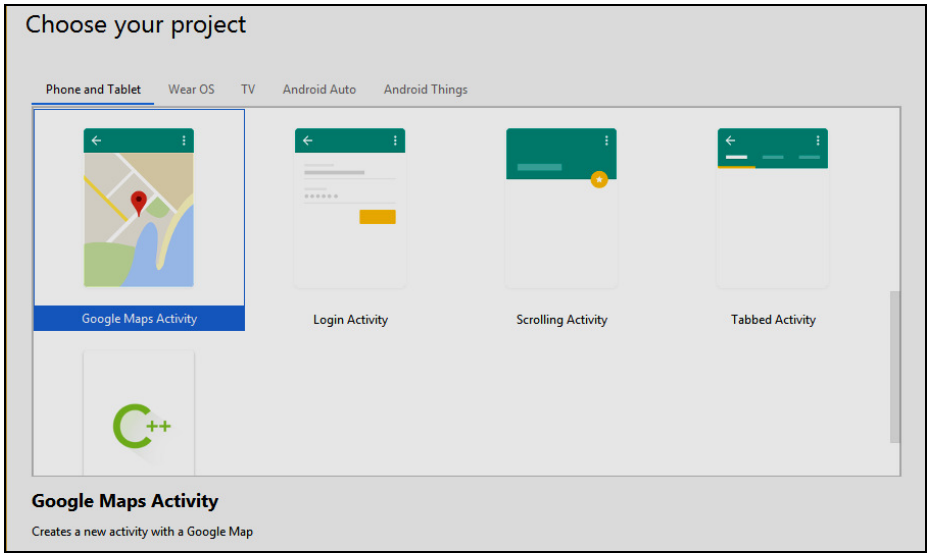


Fig. 6.2

Step 2: Write the name for the activity and then click finish.

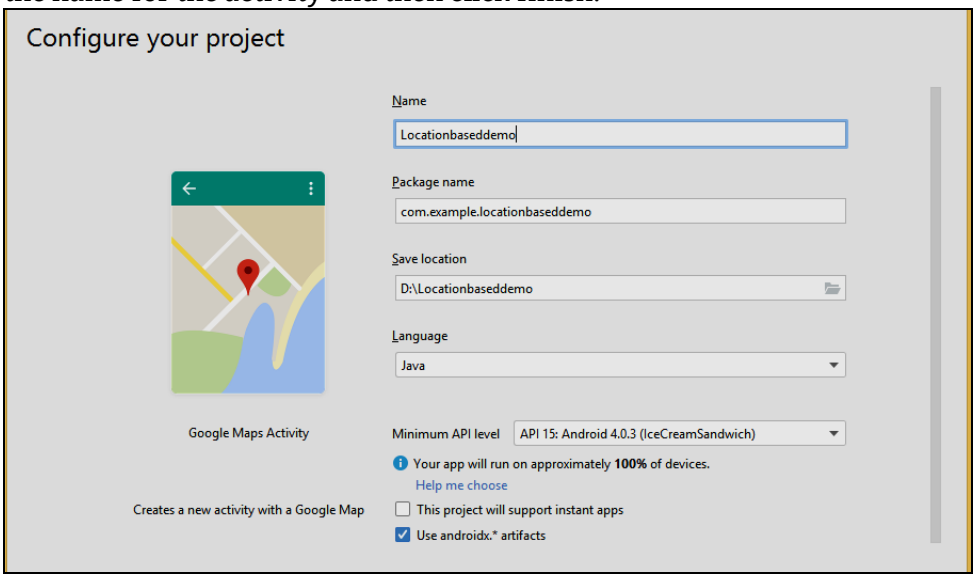


Fig. 6.3

Step 3: When the project is create, the added JAR file located below the Google APIs folder.

6.2.2 Getting the Maps API Key

- An API key is needed to access the Google Maps servers. This key is free and we can use it with any of our applications.

  Step 1: Open Google developer console and sign in with our gmail account: https://console. developers. google.com/project

  Step 2: Now create new project. We can create new project by clicking on the Create Project button and give name to our project.

Fig. 6.4

Step 3: Now click on APIs & Services and open **Dashboard** from it.

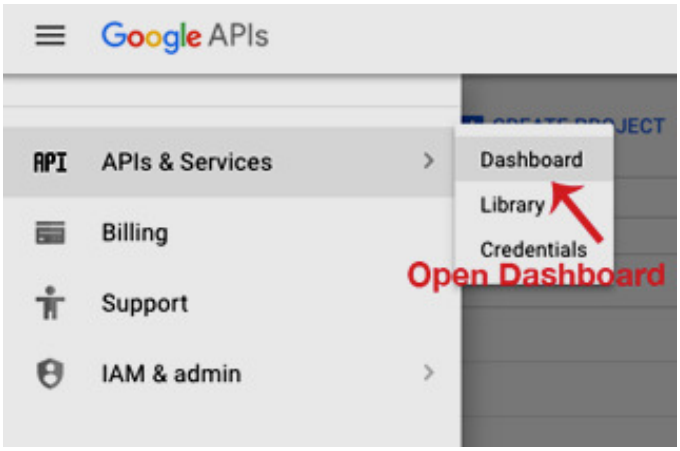


Fig. 6.5

Step 4: In this open Enabled APIs and serices.

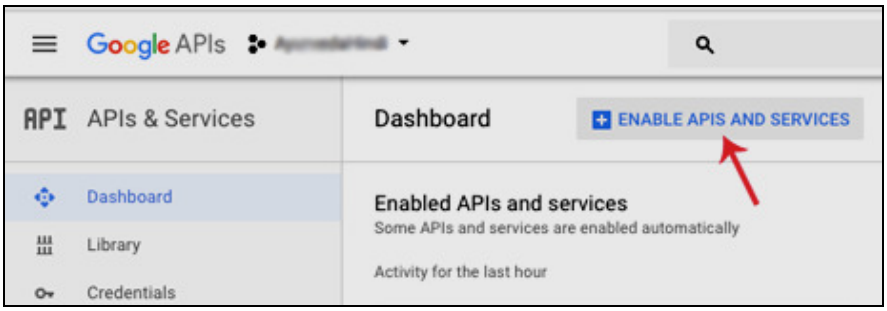


Fig. 6.6

Step 5: Now open Google Map Android API.

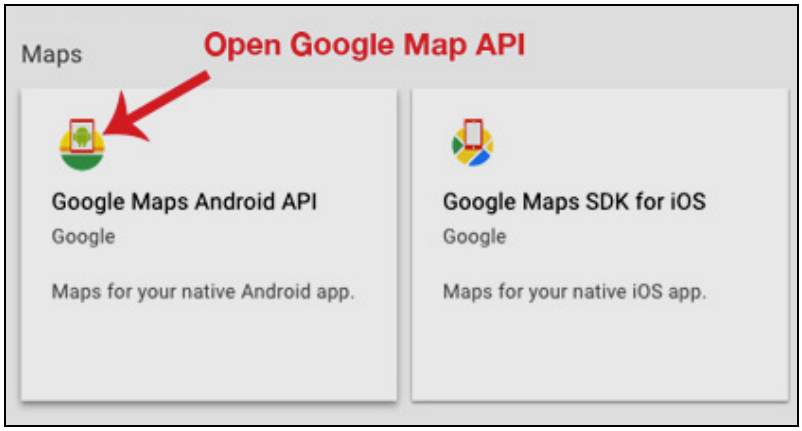


Fig. 6.7

Step 6: Now go to Credentials.

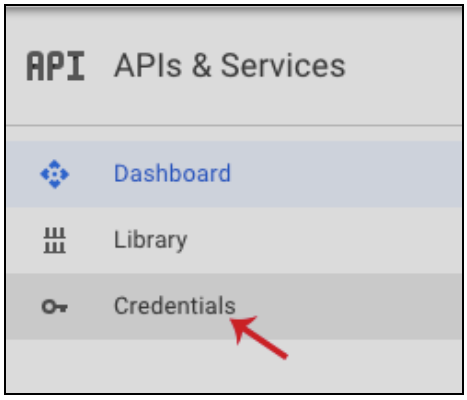


Fig. 6.8

Step 7: Here click on Create credentials and choose API key.

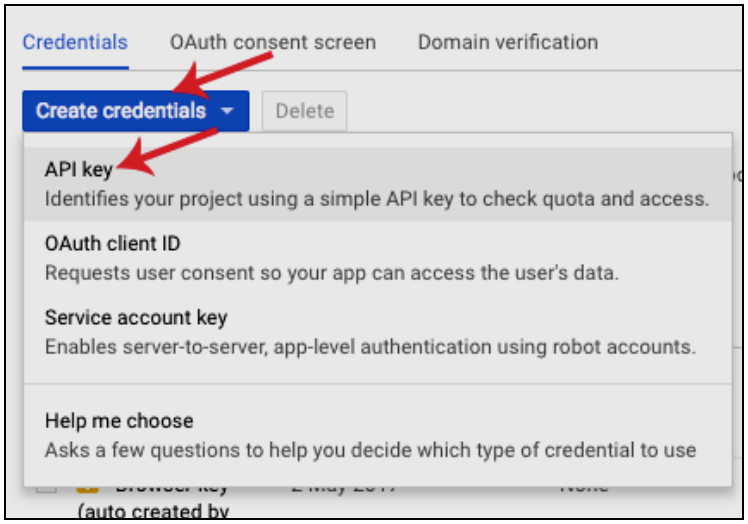


Fig. 6.9

Step 8: Now API our API key will be generated. Copy it and save it somewhere as we will need it when implementing Google Map in our Android project.

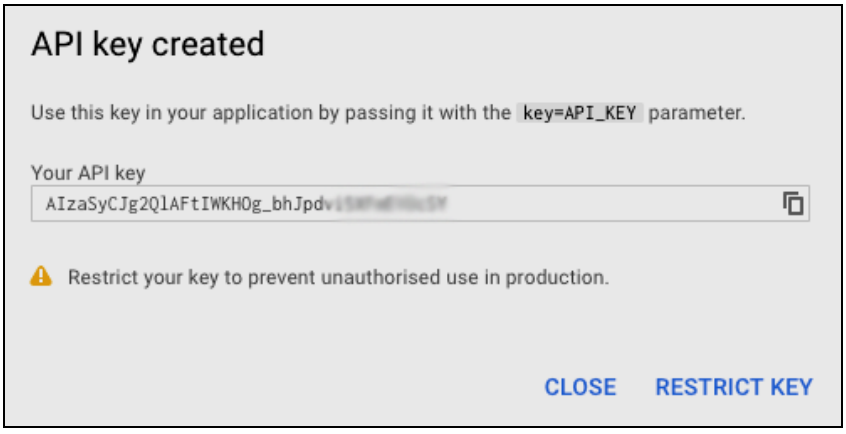


Fig. 6.10

6.2.3 Displaying the Maps

Step 1: Create a New Android Project and name it.

Step 2: Now select Google Maps Activity and then click Next and finish.

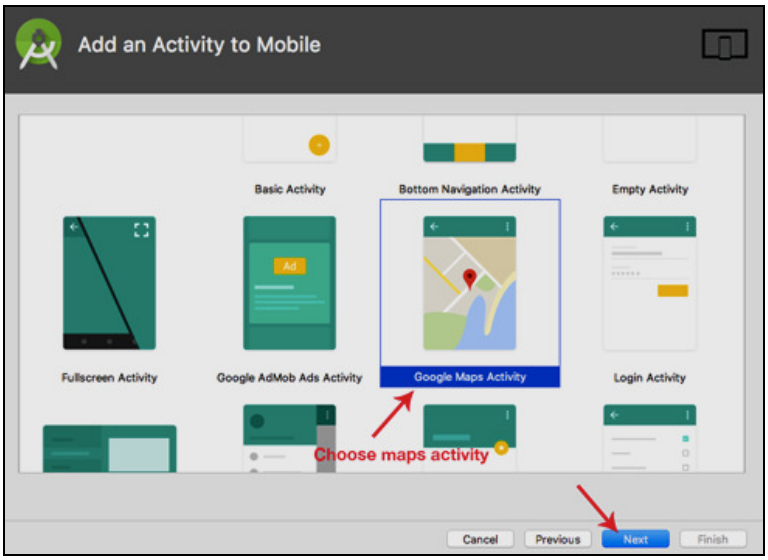


Fig. 6.11

Step 3: Now open google_maps_api.xml (debug) in values folde.

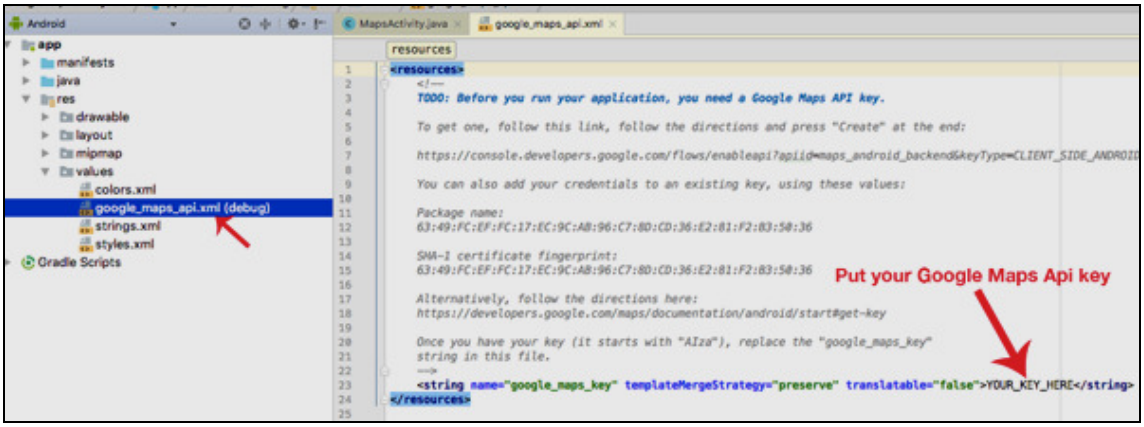


Fig. 6.12

Step 4: Here, enter our Google Maps API key in place of WER_KEY_HERE.

```
<resources>
<!--
    TODO: Before we run our application, we need a Google Maps API key.

    To get one, follow this link, follow the directions and press "Create" at the end:

https://console.developers.google.com/flows/enableapi?apiid=maps_android_backend&keyType
=CLIENT_SIDE_ANDROID&r=8B:49:70:2A:08:F2:23:14:CF:A1:FC:6F:6D:5B:60:3C:B6:85:98:F2%3Bcom
.example.abhishek.googlemaps

    We can also add our credentials to an existing key, using these values:

    Package name:
    8B:49:70:2A:08:F2:23:14:CF:A1:FC:6F:6D:5B:60:3C:B6:85:98:F2

    SHA-1 certificate fingerprint:
    8B:49:70:2A:08:F2:23:14:CF:A1:FC:6F:6D:5B:60:3C:B6:85:98:F2
```

```
    Alternatively, follow the directions here:
    https://developers.google.com/maps/documentation/android/start#get-key

    Once we have our key (it starts with "AIza"), replace the "google_maps_key"
string in this file.
    -->
<stringname="google_maps_key"templateMergeStrategy="preserve"translatable="false">AIzaSy
DV2_xy58r15K6TskZy4KWMuhUDVq67jqM</string>
</resources>
```

Step 6: Now open activity_maps.xml and add following code:

```
<?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"
xmlns:tools="http://schemas.android.com/tools"
android:id="@+id/map"
android:name="com.google.android.gms.maps.SupportMapFragment"
android:layout_width="match_parent"
android:layout_height="match_parent"
tools:context=".MapsActivity" />
```

Step 7: Add the following code in the MapsActivity.java.

```
import androidx.fragment.app.FragmentActivity;
import android.os.Bundle;
import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.OnMapReadyCallback;
import com.google.android.gms.maps.SupportMapFragment;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;
public class MapsActivityextends FragmentActivityimplements OnMapReadyCallback {
private GoogleMapmMap;
    @Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_maps);
SupportMapFragmentmapFragment = (SupportMapFragment) getSupportFragmentManager()
                .findFragmentById(R.id.map);
mapFragment.getMapAsync(this);
    }
@Override
public void onMapReady(GoogleMapgoogleMap) {
mMap= googleMap;

LatLngsydney = new LatLng(-34, 151);
mMap.addMarker(new MarkerOptions().position(sydney).title("Marker in Sydney"));
mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney));
    }
}
```

Step 8 : Add the following code in AndroidManifest.xml file

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

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

```
<application
android:allowBackup="true"
android:icon="@mipmap/ic_launcher"
android:label="@string/app_name"
android:roundIcon="@mipmap/ic_launcher_round"
android:supportsRtl="true"
android:theme="@style/AppTheme">

<meta-data
android:name="com.google.android.geo.API_KEY"
android:value="AIzaSyABJ6evNQ52va9Rucu_sU7Tjpxvb433-9A" />

<activity
android:name=".MapsActivity"
android:label="@string/title_activity_maps">
<intent-filter>
<action android:name="android.intent.action.MAIN" />

<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>

</manifest>
```

Step 9: Add the following in the strings,xml file:

```
<resources>
<string name="app_name">Locationdemo</string>
<string name="title_activity_maps">Map</string>
</resources>
```

Step 10: Now run the program.

Output:

6.2.4 Displaying the Zoom Control

- We can display Google Maps in our Android application. We can put the map to any preferred location. On the emulator there is no way to zoom in or out from a particular location, but it can possible on a real Android device where we can touch that map to zoom it. Hence we should learn how users can zoom in or out of the map using the built-in zoom controls.
- First, add a element to the main.xml file as shown below:

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

```
<RelativeLayoutxmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <com.google.android.maps.MapView
        android:id="@+id/mapView"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:enabled="true"
        android:clickable="true"
        android:apiKey="0l4sCTTyRmXTNo7k8DREHvEaLar2UmHGwnhZVHQ"
        />
    <LinearLayoutandroid:id="@+id/zoom"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        />
</RelativeLayout>
```

- We will use the element to hold the two zoom controls in Google Maps
- In the MapsActivity.java file, add the following imports:

```
importcom.google.android.maps.MapActivity;
importcom.google.android.maps.MapView;
importcom.google.android.maps.MapView.LayoutParams;
importandroid.os.Bundle;
importandroid.view.View;
importandroid.widget.LinearLayout;
public class MapsActivity extends MapActivity
{
    MapViewmapView;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mapView = (MapView) findViewById(R.id.mapView);
        LinearLayoutzoomLayout = (LinearLayout)findViewById(R.id.zoom);
        View zoomView = mapView.getZoomControls();
        zoomLayout.addView(zoomView,
            newLinearLayout.LayoutParams(
                LayoutParams.WRAP_CONTENT,
                LayoutParams.WRAP_CONTENT));
        mapView.displayZoomControls(true);
    }
    @Override
    protectedbooleanisRouteDisplayed() {
        // TODO Auto-generated method stub
        return false;
    }
}
```

Output:

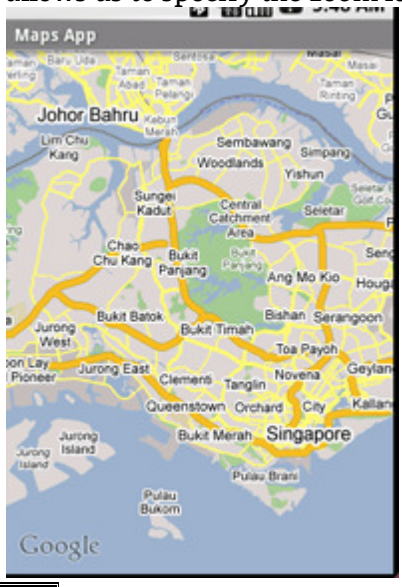
6.2.5 Navigating to a Specific Location

- Be default, the Google Maps displays the map of the United States when it is first loaded. However, you can also set the Google Maps to display a particular location. In this case, you can use the animateTo() method of the MapController class.
- The following code shows how this is done:

```
importcom.google.android.maps.GeoPoint;
importcom.google.android.maps.MapActivity;
importcom.google.android.maps.MapController;
importcom.google.android.maps.MapView;
importcom.google.android.maps.MapView.LayoutParams;
importandroid.os.Bundle;
importandroid.view.View;
importandroid.widget.LinearLayout;
public class MapsActivity extends MapActivity
{
    MapViewmapView;
    MapController mc;
    GeoPoint p;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mapView = (MapView) findViewById(R.id.mapView);
        LinearLayoutzoomLayout = (LinearLayout)findViewById(R.id.zoom);
        View zoomView = mapView.getZoomControls();
        zoomLayout.addView(zoomView,
            newLinearLayout.LayoutParams(
                LayoutParams.WRAP_CONTENT,
```

```
                LayoutParams.WRAP_CONTENT));
        mapView.displayZoomControls(true);
        mc = mapView.getController();
        String coordinates[] = {"1.352566007", "103.78921587"};
        doublelat = Double.parseDouble(coordinates[0]);
        doublelng = Double.parseDouble(coordinates[1]);
        p = new GeoPoint(
            (int) (lat * 1E6),
            (int) (lng * 1E6));
        mc.animateTo(p);
        mc.setZoom(17);
        mapView.invalidate();
    }
    @Override
    protectedbooleanisRouteDisplayed() {
        // TODO Auto-generated method stub
        return false;
    }
}
```

- To navigate the map to a particular location, you can use the animateTo() method of the MapController class (an instance which is obtained from the MapView object). The setZoom() method allows us to specify the zoom level in which the map is displayed.



6.2.6 Adding Markers

- Very often, we may wish to add markers to the map to indicate places of interests. Let's see how we can do this in Android. First, create a GIF image containing a pushpin (see Fig. 6.12) and copy it into the res/drawable folder of the project. For best effect, we should make the background of the image transparent so that it does not block off parts of the map when the image is added to the map.

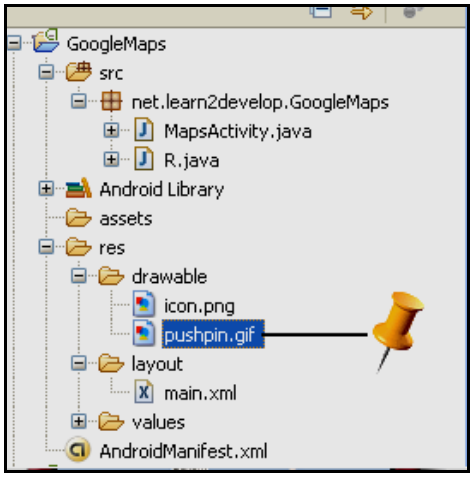


Fig. 6.12

- To add a marker to the map, you first need to define a class that extends the Overlay class:

```
importjava.util.List;
importcom.google.android.maps.GeoPoint;
importcom.google.android.maps.MapActivity;
importcom.google.android.maps.MapController;
importcom.google.android.maps.MapView;
importcom.google.android.maps.Overlay;
importcom.google.android.maps.MapView.LayoutParams;
importandroid.graphics.Bitmap;
importandroid.graphics.BitmapFactory;
importandroid.graphics.Canvas;
importandroid.graphics.Point;
importandroid.os.Bundle;
importandroid.view.View;
importandroid.widget.LinearLayout;
public class MapsActivity extends MapActivity
{
    MapViewmapView;
    MapController mc;
    GeoPoint p;
    classMapOverlay extends com.google.android.maps.Overlay
    {
        @Override
        publicboolean draw(Canvas canvas, MapViewmapView,
        boolean shadow, long when)
        {
            super.draw(canvas, mapView, shadow);
            //---translate the GeoPoint to screen pixels---
            Point screenPts = new Point();
            mapView.getProjection().toPixels(p, screenPts);
            //---add the marker---
            Bitmap bmp = BitmapFactory.decodeResource(
                getResources(), R.drawable.pushpin);
            canvas.drawBitmap(bmp, screenPts.x, screenPts.y-50, null);
            return true;
        }
```

```
    }
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        //...
    }
    @Override
    protectedbooleanisRouteDisplayed() {
        // TODO Auto-generated method stub
        return false;
    }
}
```

- In the MapOverlay class that we have defined, override the draw() method so that we can draw the pushpin image on the map. In particular, note that we need to translate the geographical location (represented by a GeoPoint object, p) into screen coordinates.
- As we want the pointed tip of the push pin to indicate the position of the location, we would need to deduct the height of the image (which is 50 pixels) from the y-coordinate of the point and draw the image at that location.

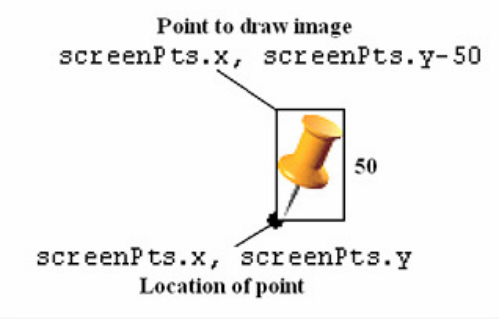
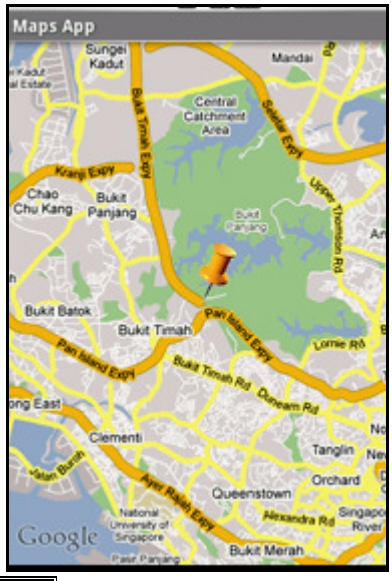


Fig. 6.13: Point to draw image

- To add the marker, create an instance of the MapOverlap class and add it to the list of overlays available on the MapView object:

```
@Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        //...
        mc.animateTo(p);
        mc.setZoom(17);
        //---Add a location marker---
        MapOverlaymapOverlay = new MapOverlay();
        List<Overlay>listOfOverlays = mapView.getOverlays();
        listOfOverlays.clear();
        listOfOverlays.add(mapOverlay);
        mapView.invalidate();
    }
```

Output:



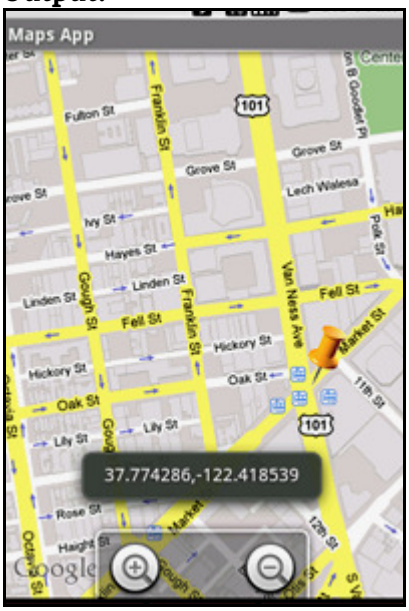
6.2.7 Getting the Location that was Touched

- After using Google Maps for a while, we may wish to know the latitude and longitude of a location corresponding to the position on the screen that we have just touched.
- Knowing this information is very useful as we can find out the address of a location, a process known as Geocoding (we will see how this is done in the next section).
- If we have added an overlay to the map, we can override the onTouchEvent() method within the Overlay class. This method is fired every time the user touches the map. This method has two parameters – MotionEvent and MapView.
- Using the MotionEvent parameter, we can know if the user has lifted his finger from the screen using the getAction() method.
- In the following code, if the user has touched and then lifted his finger, we will display the latitude and longitude of the location touched:

```
classMapOverlay extends com.google.android.maps.Overlay
    {
        @Override
        publicboolean draw(Canvas canvas, MapViewmapView,
        boolean shadow, long when)
        {
           //...
        }

        @Override
        publicbooleanonTouchEvent(MotionEvent event, MapViewmapView)
        {
            //---when user lifts his finger---
            if (event.getAction() == 1) {
                GeoPoint p = mapView.getProjection().fromPixels(
                    (int) event.getX(),
                    (int) event.getY());
                    Toast.makeText(getBaseContext(),
                        p.getLatitudeE6() / 1E6 + "," +
                        p.getLongitudeE6() /1E6 ,
                        Toast.LENGTH_SHORT).show();
```

```
            }
            return false;
        }
    }
```

Output:



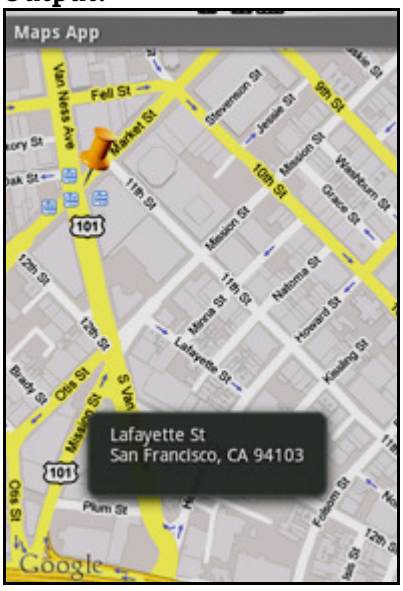
6.2.8 Geocoding and Reverse Geocoding

- If we know the latitude and longitude of a location, we can find out its address using a process known as Geocoding. Google Maps in Android supports this via the Geocoder class.
- The following code shows how we can find out the address of a location we have just touched using the getFromLocation() method:

```
classMapOverlay extends com.google.android.maps.Overlay
    {
        @Override
        publicboolean draw(Canvas canvas, MapViewmapView,
        boolean shadow, long when)
        {
          //...
        }

        @Override
        publicbooleanonTouchEvent(MotionEvent event, MapViewmapView)
        {
            //---when user lifts his finger---
            if (event.getAction() == 1) {
                GeoPoint p = mapView.getProjection().fromPixels(
                    (int) event.getX(),
                    (int) event.getY());

                Geocoder geoCoder = new Geocoder(
                    getBaseContext(), Locale.getDefault());
                try {
                    List<Address> addresses = geoCoder.getFromLocation(
```

```
                    p.getLatitudeE6()  / 1E6,
                    p.getLongitudeE6() / 1E6, 1);

                String add = "";
                if (addresses.size() > 0)
                {
                    for (inti=0; i<addresses.get(0).getMaxAddressLineIndex();
                         i++)
                       add += addresses.get(0).getAddressLine(i) + "n";
                }

                Toast.makeText(getBaseContext(), add, Toast.LENGTH_SHORT).show();
            }
            catch (IOException e) {
                e.printStackTrace();
            }
            return true;
        }
        else
            return false;
    }
}
```

Output:



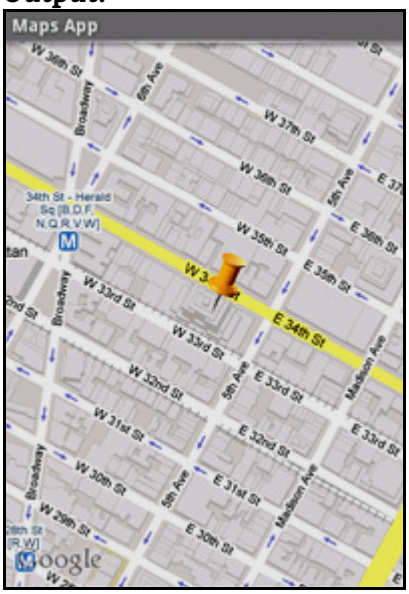
- If we know the address of a location but want to know its latitude and longitude, we can do so via reverse-Geocoding. Again, we can use the Geocoder class for this purpose.
- The following code shows how we can find the exact location of the Empire State Building by using the getFromLocationName() method:

```
Geocoder geoCoder = new Geocoder(this, Locale.getDefault());
    try {
        List<Address> addresses = geoCoder.getFromLocationName(
            "empire state building", 5);
        String add = "";
        if (addresses.size() > 0) {
            p = new GeoPoint(
```

```
                    (int) (addresses.get(0).getLatitude() * 1E6),
                    (int) (addresses.get(0).getLongitude() * 1E6));
            mc.animateTo(p);
            mapView.invalidate();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
```

- Once. the location is found, the above code navigates the map to the location.

Output:



6.2.9 Getting Location Data

- Nowadays, mobile devices are commonly equipped with GPS receivers. Because of the many satellites orbiting the earth, we can use a GPS receiver to find our location easily. However, GPS requires a clear sky to work and hence does not always work indoors or where satellites can not penetrate (such as a tunnel through a mountain).
- Another effective way to locate our position is through cell tower triangulation. When a mobile phone is switched on, it is constantly in contact with base stations surrounding it.
- By knowing the identity of cell towers, it is possible to translate this information into a physical location through the use of various databases containing the cell towers' identities and their exact geographical locations.
- The advantage of cell tower triangulation is that it works indoors, without the need to obtain information from satellites.
- However, it is not as precise as GPS because its accuracy depends on overlapping signal coverage, which varies quite a bit. Cell tower triangulation works best in densely populated areas where the cell towers are closely located.
- A third method of locating our position is to rely on Wi-Fi triangulation. Rather than connect to cell towers, the device connects to a Wi-Fi network and checks the service provider against databases to determine the location serviced by the provider.
- Of the three methods described here, Wi-Fi triangulation is the least accurate. On the Android platform, the SDK provides the LocationManager class to help our device determine the user's physical location.
- The following code shows how this is done in code.

```
MapsActivity.java
importandroid.content.Context;
importandroid.content.pm.PackageManager;
importandroid.location.Location;
importandroid.location.LocationListener;
importandroid.location.LocationManager;
import android.support.v4.app.ActivityCompat;
import android.support.v4.app.FragmentActivity;
importandroid.os.Bundle;
importandroid.widget.Toast;
importcom.google.android.gms.maps.CameraUpdateFactory;
importcom.google.android.gms.maps.GoogleMap;
importcom.google.android.gms.maps.OnMapReadyCallback;
importcom.google.android.gms.maps.SupportMapFragment;
importcom.google.android.gms.maps.model.LatLng;
public class MapsActivity extends FragmentActivity implements OnMapReadyCallback
{
final private int REQUEST_COURSE_ACCESS = 123;
booleanpermissionGranted = false;
privateGoogleMapmMap;
LocationManager lm;
LocationListenerlocationListener;
@Override
protected void onCreate(Bundle savedInstanceState)
{
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_maps);
// Obtain the SupportMapFragment and get notified
// when the map is ready to be used.
SupportMapFragmentmapFragment = (SupportMapFragment) getSupportFragmentManager()
.findFragmentById(R.id.map);
mapFragment.getMapAsync(this);
}
@Override
public void onPause()
{
super.onPause();
 //---remove the location listener---
 if (ActivityCompat.checkSelfPermission(this,
android.Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED&&ActivityCompat.checkSelfPermission( this,
android.Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) { ActivityCompat.requestPermissions(this, new
String[]{ android.Manifest.permission.ACCESS_COARSE_LOCATION}, REQUEST_COURSE_ACCESS);
return;
}
Else
{ permissionGranted = true;
 }
if(permissionGranted)
```

```
{ lm.removeUpdates(locationListener);
}
 }
 @
Override
public void onMapReady(GoogleMapgoogleMap)
{
mMap = googleMap;
lm = (LocationManager)getSystemService(Context.LOCATION_SERVICE);

locationListener = new MyLocationListener();
if (ActivityCompat.checkSelfPermission(this,
android.Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED&&ActivityCompat.checkSelfPermission( this,
android.Manifest.permission.ACCESS_COARSE_LOCATION) !=
PackageManager.PERMISSION_GRANTED)
 {
ActivityCompat.requestPermissions(this, new String[]
{ android.Manifest.permission.ACCESS_COARSE_LOCATION}, REQUEST_COURSE_ACCESS); return; }
Else
{
permissionGranted = true;
}
if(permissionGranted)
{
lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, locationListener);
}
 }
 @Override
public void onRequestPermissionsResult(intrequestCode, String[] permissions, int[]
grantResults)
{
switch (requestCode)
 {
case REQUEST_COURSE_ACCESS: if (grantResults[0] == PackageManager.PERMISSION_GRANTED)
 {
permissionGranted = true;
 }
else
{
permissionGranted = false;
}
break;
default: super.onRequestPermissionsResult(requestCode, permissions, grantResults);
}
 }
private class MyLocationListener implements LocationListener
{
public void onLocationChanged(Location loc)
{
```

```
if (loc != null)
{
Toast.makeText(getBaseContext(),
"Location changed :Lat: " + loc.getLatitude() + " Lng: " + loc.getLongitude(),
Toast.LENGTH_SHORT).show(); LatLng p = new LatLng( (int) (loc.getLatitude()),
(int) (loc.getLongitude()));
mMap.moveCamera(CameraUpdateFactory.newLatLng(p));
mMap.animateCamera(CameraUpdateFactory.zoomTo(7));
}
 }
public void onProviderDisabled(String provider) { }
public void onProviderEnabled(String provider) { }
public void onStatusChanged(String provider, int status, Bundle extras) { }
 }
}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.jfdimarzio.locationservices" >
<!--
 The ACCESS_COARSE/FINE_LOCATION permissions are not required to use
 Google Maps Android API v2, but we must specify either coarse or fine
location permissions for the 'MyLocation' functionality.
 -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<application
android:allowBackup="true"
android:icon="@mipmap/ic_launcher"
android:label="@string/app_name"
android:supportsRtl="true"
android:theme="@style/AppTheme" >
<!--
 The API key for Google Maps-based APIs is defined as a string resource
 (See the file "res/values/google_maps_api.xml").
 Note that the API key is linked to the encryption
key used to sign the APK.
 We need a different API key for each encryption key,
including the release key that is used to
sign the APK for publishing.
 We can define the keys for the debug and release targets in
src/debug/ and src/release/.
 -->
<meta-data
android:name="com.google.android.geo.API_KEY"
android:value="@string/google_maps_key" />
<activity
android:name=".MapsActivity"
android:label="@string/title_activity_maps" >
<intent-filter>
```

```
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
</manifest>
```

6.2.10 Monitoring a Location

- One very cool feature of the LocationManager class is its ability to monitor a specific location. This is achieved using the addProximityAlert() method.
- The following code snippet shows how to monitor a particular location such that if the user is within a five-meter radius from that location, our application will fire an intent to launch the web browser:

```
importandroid.app.PendingIntent;
importandroid.content.Intent;
importandroid.net.Uri;
 //---use the LocationManager class to obtain locations data---
lm = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
 //---PendingIntent to launch activity if the user is within
 // some locations---
PendingIntentpendingIntent = PendingIntent.getActivity(
this, 0, new
Intent(android.content.Intent.ACTION_VIEW,
Uri.parse("http://www.amazon.com")), 0);
lm.addProximityAlert(37.422006, -122.084095, 5, -1, pendingIntent);
```

- The addProximityAlert() method takes five arguments namely, Latitude, Longitude, Radius (in meters), Expiration (duration for which the proximity alert is valid, after which it is deleted; –1 for no expiration), Pending intent
- Note that if the Android device's screen goes to sleep, the proximity is also checked once every fourminutes in order to preserve the battery life of the device.

6.3 ANDROID SECURITY MODEL

- Android is a multi-process system, in which each application (and parts of the system) runs in its own process.
- Most security between applications and the system is enforced at the process level through standard Linux facilities, such as user and group IDs that are assigned to applications.
- Additional finer-grained security features are provided through a "permission" mechanism that enforces restrictions on the specific operations that a particular process can perform, and per-URI permissions for granting ad-hoc access to specific pieces of data.
- The Android security model is primarily based on a sandbox and permission mechanism. Each application is running in a specific Dalvik virtual machine with a unique user ID assigned to it, which means the application code runs in isolation from the code of all others applications. As a consequence, one application has not granted access to other applications' files.
- Android application has been signed with a certificate with a private key Know the owner of the application is unique. This allows the author of The application will be identified if needed.
- When an application is installed in The phone is assigned a user ID, thus avoiding it from affecting it Other applications by creating a sandbox for it.

- This user ID is permanent on which devices and applications with the same user ID are allowed to run in a single process. This is a way to ensure that a malicious application has Can not access/compromise the data of the genuine application.
- It is mandatory for an application to list all the resources it will Access during installation. Terms are required of an application, in The installation process should be user-based or interactive Check with the signature of the application.
- The purpose of a permission is to protect the privacy of an Android user. Android apps must request permission to access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and internet).
- Depending on the feature, the system might grant the permission automatically or might prompt the user to approve the request.
- Permissions are divided into several protection levels. The protection level affects whether runtime permission requests are required. There are three protection levels that affect third-party apps: *normal*, *signature* and *dangerous* permissions.
- Normal permissions cover areas where our app needs to access data or resources outside the app's sandbox, but where there's very little risk to the user's privacy or the operation of other apps.
- For example, permission to set the time zone is a normal permission. If an app declares in its manifest that it needs a normal permission, the system automatically grants the app that permission at install time. The system doesn't prompt the user to grant normal permissions, and users cannot revoke these permissions.

1. **Signature Permissions:** The system grants these app permissions at install time, but only when the app that attempts to use a permission is signed by the same certificate as the app that defines the permission.
2. **Dangerous Permissions:** Cover areas where the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps. For example, the ability to read the user's contacts is a dangerous permission. If an app declares that it needs a dangerous permission, the user has to explicitly grant the permission to the app. Until the user approves the permission, our app cannot provide functionality that depends on that permission. To use a dangerous permission, our app must prompt the user to grant permission at runtime.

Android Threat:

- However, the Android operating system also revealed some of its faults for the user may be attacked and stolen personal information.
- Some security vulnerabilities on Android:
    1. **Leaking Information to Logs:** Android provides centralized logging via the Log API, which can displayed with the "logcat" command. While logcat is a debugging tool, applications with the READ_LOGS permission can read these log messages. The Android documentation for this permission indicates that "the logs can contain slightly private information about what is happening on the device, but should never contain the user's private information."
    2. **SDcard Use:** Any application that has access to read or write data on the SDcard can read or write any other application's data on the SDcard.
    3. **Unprotected Broadcast Receivers:** Applications use broadcast receiver components to receive intent messages. Broadcast receivers define "intent filters" to subscribe to specific event types are public. If the receiver is not protected by a permission, a malicious application can forge messages.

4. **Intent Injection Attacks:** Intent messages are also used to start activity and service components. An intent injection attack occurs if the in-tent address is derived from untrusted input.
5. **Wifi Sniffing:** This may disrupt the data being transmitted from A device like many web sites and applications does not have security measures strict security. The application does not encrypt the data and therefore it can be Blocked by a listener on unsafe lines.

6.3.1 Declaring and Using Permissions

- The purpose of a permission is to protect the privacy of an Android user. Android apps must request permission to access sensitive user data (such as contacts and SMS), as well as certain system features (such as camera and internet). Depending on the feature, the system might grant the permission automatically or might prompt the user to approve the request.
- A central design point of the Android security architecture is that no app, by default, has permission to perform any operations that would adversely impact other apps, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or emails), reading or writing another app's files, performing network access, keeping the device awake, and so on.

Permission Approval:

- An app must publicize the permissions it requires by including <uses-permission> tags in the app manifest. For example, an app that needs to send SMS messages would have this line in the manifest:

```
<manifestxmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.snazzyapp">
    <uses-permissionandroid:name="android.permission.SEND_SMS"/>
    <application ...>
        ...
    </application>
</manifest>
```

- If our app lists normal permissions in its manifest (that is, permissions that don't pose much risk to the user's privacy or the device's operation), the system automatically grants those permissions to our app.
- If our app lists dangerous permissions in its manifest (that is, permissions that could potentially affect the user's privacy or the device's normal operation), such as the SEND_SMS permission above, the user must explicitly agree to grant those permissions.

Request Prompts for Dangerous Permissions

- Only dangerous permissions require user agreement. The way Android asks the user to grant dangerous permissions depends on the version of Android running on the user's device, and the system version targeted by our app.

Runtime Requests:

- If the device is running Android 6.0 (API level 23) or higher, *and* the app's targetSdkVersion is 23 or higher, the user isn't notified of any app permissions at install time. Our app must ask the user to grant the dangerous permissions at runtime.
- When our app requests permission, the user sees a system dialog (as shown in Fig. 6.14, left) telling the user which permission group our app is trying to access. The dialog includes a Deny and Allow button.
- If the user denies the permission request, the next time our app requests the permission, the dialog contains a checkbox that, when checked, indicates the user doesn't want to be prompted for the permission again

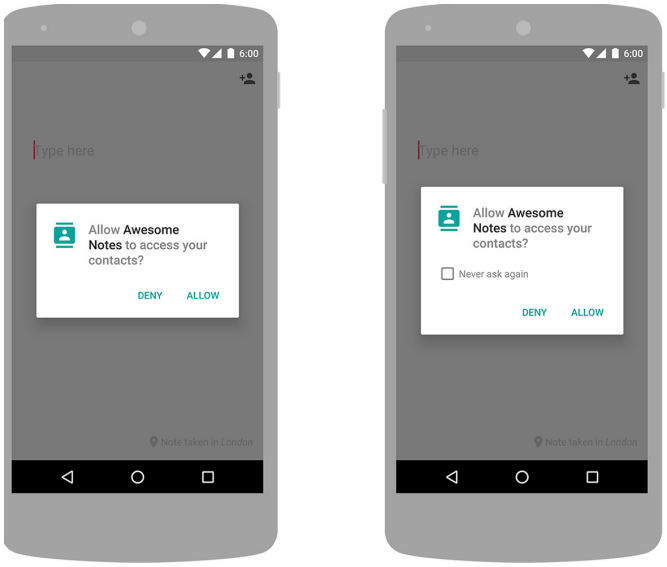


Fig. 6.14

- If the user checks the Never ask again box and taps Deny, the system no longer prompts the user if we later attempt to requests the same permission.
- Even if the user grants our app the permission it requested we cannot always rely on having it. Users also have the option to enable and disable permissions one-by-one in system settings.
- We should always check for and request permissions at runtime to guard against runtime errors (SecurityException).

Request prompts to Access Sensitive user Information:

- Some apps depend on access to sensitive user information related to call logs and SMS messages. If we want to request the permissions specific to call logs and SMS messages and publish our app to the Play Store, we must prompt the user to set our app as the *default handler* for a core system function before requesting these runtime permissions.

Permissions for Optional Hardware Features:

- Access to some hardware features (such as Bluetooth or the camera) require an app permission. However, not all Android devices actually have these hardware features. So if our app requests the CAMERA permission, it's important that we also include the <uses-feature> tag in our manifest to declare whether or not this feature is actually required. For example:

  ```
  <uses-featureandroid:name="android.hardware.camera"android:required="false"/>
  ```

  If we declare android:required="false" for the feature, then Google Play allows our app to be installed on devices that don't have the feature.
  We then must check if the current device has the feature at runtime by calling PackageManager. has SystemFeature(), and gracefully disable that feature if it's not available.
- If we does not provide the <uses-feature> tag, then when Google Play sees that our app requests the corresponding permission, it assumes our app requires this feature. So it filters our app from devices without the feature, as if we declared android:required="true" in the <uses-feature> tag.

Permission Enforcement:

- Permissions aren't only for requesting system functionality. Services provided by apps can enforce custom permissions to restrict who can use them.

Activity Permission Enforcement:

- Permissions applied using the android:permission attribute to the <activity> tag in the manifest restrict who can start that Activity. The permission is checked during Context.startActivity() and Activity.startActivityForResult(). If the caller doesn't have the required permission then SecurityException is thrown from the call.

Service Permission Enforcement:

- Permissions applied using the android:permission attribute to the <service> tag in the manifest restrict who can start or bind to the associated Service.
- The permission is checked during Context.startService(), Context.stopService() and Context.bindService(). If the caller doesn't have the required permission then SecurityException is thrown from the call.

Broadcast Permission Enforcement:

- Permissions applied using the android:permission attribute to the <receiver> tag restrict who can send broadcasts to the associated BroadcastReceiver.
- The permission is checked *after* Context.sendBroadcast() returns, as the system tries to deliver the submitted broadcast to the given receiver.
- As a result, a permission failure doesn't result in an exception being thrown back to the caller; it just doesn't deliver the Intent.
- In the same way, a permission can be supplied to Context.registerReceiver() to control who can broadcast to a programmatically registered receiver.
- Going the other way, a permission can be supplied when calling Context.sendBroadcast() to restrict which broadcast receivers are allowed to receive the broadcast.
- Note that both a receiver and a broadcaster can require a permission. When this happens, both permission checks must pass for the intent to be delivered to the associated target.

Content Provider Permission Enforcement:

- Permissions applied using the android:permission attribute to the <provider> tag restrict who can access the data in a ContentProvider, (Content providers have an important additional security facility available to them called URI permissions which is described next.)
- Unlike the other components, there are two separate permission attributes we can set: android:readPermission restricts who can read from the provider, and android:writePermission restricts who can write to it. Note that if a provider is protected with both a read and write permission, holding only the write permission doesn't mean we can read from a provider.
- The permissions are checked when we first retrieve a provider (if we don't have either permission, a SecurityException is thrown), and as we perform operations on the provider.
- Using ContentResolver.query() requires holding the read permission; using ContentResolver.insert(), ContentResolver.update(), ContentResolver.delete() requires the write permission. In all of these cases, not holding the required permission results in a SecurityException being thrown from the call.

URI Permissions:

- The standard permission system described so far is often not sufficient when used with content providers. A content provider may want to protect itself with read and write permissions, while its direct clients also need to hand specific URIs to other apps for them to operate on.
- A typical example is attachments in a email app. Access to the emails should be protected by permissions, since this is sensitive user data. However, if a URI to an image attachment is given to an image viewer, that image viewer no longer has permission to open the attachment since it has no reason to hold a permission to access all email.
- The solution to this problem is per-URI permissions: when starting an activity or returning a result to an activity, the caller can set Intent.FLAG_GRANT_READ_URI_PERMISSION and/or Intent.FLAG_GRANT_WRITE_URI_PERMISSION.
- This grants the receiving activity permission access the specific data URI in the intent, regardless of whether it has any permission to access data in the content provider corresponding to the intent.

- This mechanism allows a common capability-style model where user interaction (such as opening an attachment or selecting a contact from a list) drives ad-hoc granting of fine-grained permission. This can be a key facility for reducing the permissions needed by apps to only those directly related to their behavior.
- To build the most secure implementation that makes other apps accountable for their actions within our app, we should use fine-grained permissions in this manner and declare our app's support for it with the android:grantUriPermissions attribute or <grant-uri-permissions> tag.
- More information can be found in the Context.grantUriPermission(), Context.revokeUriPermission(), and Context.checkUriPermission() methods.

Other Permission Enforcement:

- Arbitrarily fine-grained permissions can be enforced at any call into a service. This is accomplished with the Context.checkCallingPermission() method. Call with a desired permission string and it returns an integer indicating whether that permission has been granted to the current calling process.
- Note that this can only be used when we are executing a call coming in from another process, usually through an IDL interface published from a service or in some other way given to another process.
- There are a number of other useful ways to check permissions. If we have the process ID (PID) of another process, we can use the Context.checkPermission() method to check a permission against that PID.
- If we have the package name of another app, we can use the PackageManager.checkPermission() method to find out whether that particular package has been granted a specific permission.

Protection Levels:

- Permissions are divided into several protection levels. The protection level affects whether runtime permission requests are required.
- There are three protection levels that affect third-party apps namely, normal, signature and dangerous permissions as explained below:

1. Normal Permissions:

- Normal permissions cover areas where our app needs to access data or resources outside the app's sandbox, but where there's very little risk to the user's privacy or the operation of other apps. For example, permission to set the time zone is a normal permission.
- If an app declares in its manifest that it needs a normal permission, the system automatically grants the app that permission at install time. The system doesn't prompt the user to grant normal permissions, and users cannot revoke these permissions.

2. Signature Permissions:

- The system grants these app permissions at install time, but only when the app that attempts to use a permission is signed by the same certificate as the app that defines the permission.

3. Dangerous Permissions:

- Dangerous permissions cover areas where the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps.
- For example, the ability to read the user's contacts is a dangerous permission. If an app declares that it needs a dangerous permission, the user has to explicitly grant the permission to the app. Until the user approves the permission, our app cannot provide functionality that depends on that permission.

Special Permissions:

- There are a couple of permissions that don't behave like normal and dangerous permissions. SYSTEM_ALERT_WINDOW and WRITE_SETTINGS are particularly sensitive, so most

apps should not use them. If an app needs one of these permissions, it must declare the permission in the manifest, *and* send an intent requesting the user's authorization. The system responds to the intent by showing a detailed management screen to the user.

- For details on how to request these permissions, see the SYSTEM_ALERT_WINDOW and WRITE_SETTINGS reference entries.
- All permissions provided by the Android system can be found at Manifest.permission.

Permission Groups:

- Permissions are organized into groups related to a device's capabilities or features. Under this system, permission requests are handled at the group level and a single permission group corresponds to several permission declarations in the app manifest.
- For example, the SMS group includes both the READ_SMS and the RECEIVE_SMS declarations. Grouping permissions in this way enables the user to make more meaningful and informed choices, without being overwhelmed by complex and technical permission requests.

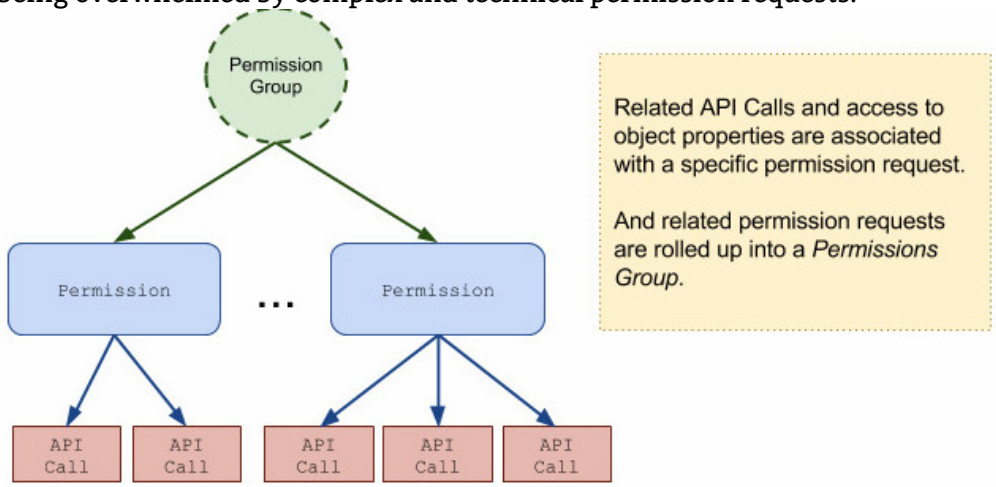


Fig. 6.15

- All dangerous Android permissions belong to permission groups. Any permission can belong to a permission group regardless of protection level. However, a permission's group only affects the user experience if the permission is dangerous.
- If the device is running Android 6.0 (API level 23) and the app's targetSdkVersion is 23 or higher, the following system behavior applies when our app requests a dangerous permission:
- If the app doesn't currently have any permissions in the permission group, the system shows the permission request dialog to the user describing the permission group that the app wants access to.
- The dialog doesn't describe the specific permission within that group.
- For example, if an app requests the READ_CONTACTS permission, the system dialog just says the app needs access to the device's contacts. If the user grants approval, the system gives the app just the permission it requested.
- If the app has already been granted another dangerous permission in the same permission group, the system immediately grants the permission without any interaction with the user. For example, if an app had previously requested and been granted the READ_CONTACTS permission, and it then requests WRITE_CONTACTS, the system immediately grants that permission without showing the permissions dialog to the user.

6.3.2 Using Custom Permission

- This document describes how app developers can use the security features provided by Android to define their own permissions. By defining custom permissions, an app can share its resources and capabilities with other apps.
- Android is a privilege-separated operating system, in which each app runs with a distinct system identity (Linux user ID and group ID). Parts of the system are also separated into distinct identities. Linux thereby isolates apps from each other and from the system.
- Apps can expose their functionality to other apps by defining permissions which those other apps can request. They can also define permissions which are automatically made available to any other apps which are signed with the same certificate

App Signing:

- All APKs must be signed with a certificate whose private key is held by their developer. This certificate identifies the author of the app.
- The certificate does *not* need to be signed by a certificate authority; it is perfectly allowable, and typical, for Android apps to use self-signed certificates.
- The purpose of certificates in Android is to distinguish app authors. This allows the system to grant or deny apps access to signature-level permissions and to grant or deny an app's request to be given the same Linux identity as another app.

User IDs and File Access:

- At install time, Android gives each package a distinct Linux user ID. The identity remains constant for the duration of the package's life on that device.
- On a different device, the same package may have a different UID; what matters is that each package has a distinct UID on a given device.
- Because security enforcement happens at the process level, the code of any two packages cannot normally run in the same process, since they need to run as different Linux users.
- We can use the sharedUserId attribute in the AndroidManifest.xml's manifest tag of each package to have them assigned the same user ID. By doing this, for purposes of security the two packages are then treated as being the same app, with the same user ID and file permissions.
- Note that in order to retain security, only two apps signed with the same signature (and requesting the same sharedUserId) will be given the same user ID.
- Any data stored by an app will be assigned that app's user ID, and not normally accessible to other packages.

Defining and Enforcing Permissions:

- To enforce our own permissions, we must first declare them in our AndroidManifest.xml using one or more <permission> elements.
- For example, an app that wants to control who can start one of its activities could declare a permission for this operation as follows:

```
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.myapp">    <permission
      android:name="com.example.myapp.permission.DEADLY_ACTIVITY"
      android:label="@string/permlab_deadlyActivity"
      android:description="@string/permdesc_deadlyActivity"
      android:permissionGroup="android.permission-group.COST_MONEY"
      android:protectionLevel="dangerous"/>
    ...
</manifest>
```

- The protectionLevel attribute is required, telling the system how the user is to be informed of apps requiring the permission, or who is allowed to hold that permission, as described in the linked documentation.
- The android:permissionGroup attribute is optional, and only used to help the system display permissions to the user.
- In most cases we should set this to a standard system group (listed in android.Manifest.permission_group), although we can define a group ourself. It is preferable to use an existing group, as this simplifies the permission UI shown to the user.
- We need to supply both a label and description for the permission. These are string resources that the user can see when they are viewing a list of permissions (android:label) or details on a single permission (android:description).
- The label should be short; a few words describing the key piece of functionality the permission is protecting. The description should be a couple of sentences describing what the permission allows a holder to do.
- Our convention is a two-sentence description: the first sentence describes the permission, and the second sentence warns the user of the type of things that can go wrong if an app is granted the permission.
- Here is an example of a label and description for the CALL_PHONE permission:

```
<stringname="permlab_callPhone">directly call phone numbers</string>
<stringname="permdesc_callPhone">Allows the app to call
    phone numbers without our intervention. Malicious apps may
    cause unexpected calls on our phone bill. Note that this does not
    allow the app to call emergency numbers.</string>
```

Create a Permission Group:

- As shown in the previous section, we can use the android:permissionGroup attribute to help the system describe permissions to the user.
- In most cases we will want to set this to a standard system group (listed in android.Manifest.permission_group), but we can also define our own group with <permission-group>.
- The <permission-group> element defines a label for a set of permissions—both those declared in the manifest with <permission> elements and those declared elsewhere. This affects only how the permissions are grouped when presented to the user. The <permission-group> element does not specify the permissions that belong to the group, but it gives the group a name.
- We can place a permission in the group by assigning the group name to the <permission> element's permissionGroup attribute.
- The <permission-tree> element declares a namespace for a group of permissions that are defined in code.

Custom Permission Recommendations:

- Apps can define their own custom permissions and request custom permissions from other apps by defining <uses-permission> elements. However, we should carefully assess whether it is necessary for our app to do so.
- If we are designing a suite of apps that expose functionality to one another, try to design the apps so that each permission is defined only once. We must do this if the apps are not all signed with the same certificate. Even if the apps are all signed with the same certificate, it's a best practice to define each permission once only.

- If the functionality is only available to apps signed with the same signature as the providing app, we may be able to avoid defining custom permissions by using signature checks. When one of our apps makes a request of another of our apps, the second app can verify that both apps are signed with the same certificate before complying with the request.

6.4 APPLICATION DEPLOYMENT

- Publishing is the general process that makes our Android applications available to users. When we publish an Android application we perform two main tasks:
  1. **We prepare the application for release:** During the preparation step we build a release version of our application, which users can download and install on their Android-powered devices.
  2. **We release the application to users:** During the release step we publicize, sell, and distribute the release version of our application to users.

6.4.1 Preparing Our App For Release

- Preparing our application for release is a multi-step process that involves the following tasks:

1. Configuring our Application for Release:

- At a minimum we need to remove Log calls and remove the android:debuggable attribute from our manifest file. We should also provide values for the android:versionCode and android:versionName attributes, which are located in the <manifest> element. We may also have to configure several other settings to meet Google Play requirements or accommodate whatever method we are using to release our application.
- If we are using Gradle build files, we can use the release build type to set our build settings for the published version of our app.

2. Building and Signing a Release Version of Our Application.

- We can use the Gradle build files with the *release* build type to build and sign a release version of our application.

3. Testing the Release Version of Our Application.

- Before we distribute our application, we should thoroughly test the release version on at least one target handset device and one target tablet device.

4. Updating Application Resources for Release.

- We need to be sure that all application resources such as multimedia files and graphics are updated and included with our application or staged on the proper production servers.

5. Preparing Remote Servers and Services that Our Application Depends on.

- If our application depends on external servers or services, we need to be sure they are secure and production ready.
- We may have to perform several other tasks as part of the preparation process. For example, we will need to get a private key for signing our application.
- We will also need to create an icon for our application, and we may want to prepare an End User License Agreement (EULA) to protect our person, organization, and intellectual property.
- When we are finished preparing our application for release we will have a signed .apk file that we can distribute to users.

Releasing our app to Users:

- We can release our Android applications several ways. Usually, we release applications through an application marketplace such as Google Play, but we can also release applications on our own website or by sending an application directly to a user.

Releasing through an app Marketplace:

- If we want to distribute our apps to the broadest possible audience, releasing through an app marketplace such as Google Play is ideal.
- Google Play is the premier marketplace for Android apps and is particularly useful if we want to distribute our applications to a large global audience.
- However, we can distribute our apps through any app marketplace we want or we can use multiple marketplaces.

Releasing our apps on Google Play:

- Google Play is a robust publishing platform that helps us to publicize, sell, and distribute our Android applications to users around the world.
- When we release our applications through Google Play we have access to a suite of developer tools that let we analyze our sales, identify market trends, and control who our applications are being distributed to.
- We also have access to several revenue-enhancing features such as in-app billing and application licensing. The rich array of tools and features, coupled with numerous end-user community features, makes Google Play the premier marketplace for selling and buying Android applications.
- Releasing our application on Google Play is a simple process that involves three basic steps:

Preparing Promotional Materials.:

- To fully leverage the marketing and publicity capabilities of Google Play, we need to create promotional materials for our application, such as screenshots, videos, graphics, and promotional text.

Configuring Options and Uploading Assets:

- Google Play lets us to target our application to a worldwide pool of users and devices. By configuring various Google Play settings, we can choose the countries we want to reach, the listing languages we want to use, and the price we want to charge in each country.
- We can also configure listing details such as the application type, category, and content rating. When we are done configuring options we can upload our promotional materials and our application as a draft (unpublished) application.

Publishing the Release Version of our Application:

- If we are satisfied that our publishing settings are correctly configured and our uploaded application is ready to be released to the public, we can simply click Publish in the Play Console and within minutes our application will be live and available for download around the world.

6.4.2 Signing of Application

- Application signing allows developers to identify the author of the application and to update their application without creating complicated interfaces and permissions.
- Every application that is run on the Android platform must be signed by the developer. Applications that attempt to install without being signed will be rejected by either Google Play or the package installer on the Android device.
- On Google Play, application signing bridges the trust Google has with the developer and the trust the developer has with their application.
- Developers know their application is provided, unmodified, to the Android device; and developers can be held accountable for behavior of their application.
- On Android, application signing is the first step to placing an application in its Application Sandbox. The signed application certificate defines which user ID is associated with which application; different applications run under different user IDs. Application signing ensures that one application cannot access any other application except through well-defined IPC.

- When an application (APK file) is installed onto an Android device, the Package Manager verifies that the APK has been properly signed with the certificate included in that APK.
- If the certificate (or, more accurately, the public key in the certificate) matches the key used to sign any other APK on the device, the new APK has the option to specify in the manifest that it will share a UID with the other similarly-signed APKs.
- Applications can be signed by a third-party (OEM, operator, alternative market) or self-signed. Android provides code signing using self-signed certificates that developers can generate without external assistance or permission.
- Applications do not have to be signed by a central authority. Android currently does not perform CA verification for application certificates.
- Applications are also able to declare security permissions at the Signature protection level, restricting access only to applications signed with the same key while maintaining distinct UIDs and Application Sandboxes.
- A closer relationship with a shared Application Sandbox is allowed via the shared UID feature where two or more applications signed with same developer key can declare a shared UID in their manifest.

APK Signing Schemes:

- Android supports following three application signing schemes:
  1. **v1 Scheme:** based on JAR signing
  2. **v2 Scheme:** APK Signature Scheme v2, which was introduced in Android 7.0.
  3. **v3 Scheme:** APK Signature Scheme v3, which was introduced in Android 9.
- For maximum compatibility, sign applications with all schemes, first with v1, then v2, and then v3. Android 7.0+ and newer devices install apps signed with v2+ schemes more quickly than those signed only with v1 scheme. Older Android platforms ignore v2+ signatures and thus need apps to contain v1 signatures.

1. JAR signing (v1 scheme):

- APK signing has been a part of Android from the beginning. It is based on signed JAR. For details on using this scheme, see the Android Studio documentation on Signing our app.
- v1 signatures do not protect some parts of the APK, such as ZIP metadata. The APK verifier needs to process lots of untrusted (not yet verified) data structures and then discard data not covered by the signatures. This offers a sizeable attack surface.
- Moreover, the APK verifier must uncompress all compressed entries, consuming more time and memory. To address these issues, Android 7.0 introduced APK Signature Scheme v2.

2. APK Signature Scheme v2 & v3 (v2+ scheme):

- Devices running Android 7.0 and later support APK signature scheme v2 (v2 scheme) and later. (v2 scheme was updated to v3 in Android P to include additional information in the signing block, but otherwise works the same.)
- The contents of the APK are hashed and signed, then the resulting APK Signing Block is inserted into the APK. For details on applying the v2+ scheme to an app.
- During validation, v2+ scheme treats the APK file as a blob and performs signature checking across the entire file. Any modification to the APK, including ZIP metadata modifications, invalidates the APK signature. This form of APK verification is substantially faster and enables detection of more classes of unauthorized modifications.
- The new format is backwards compatible, so APKs signed with the new signature format can be installed on older Android devices (which simply ignore the extra data added to the APK), as long as these APKs are also v1-signed.

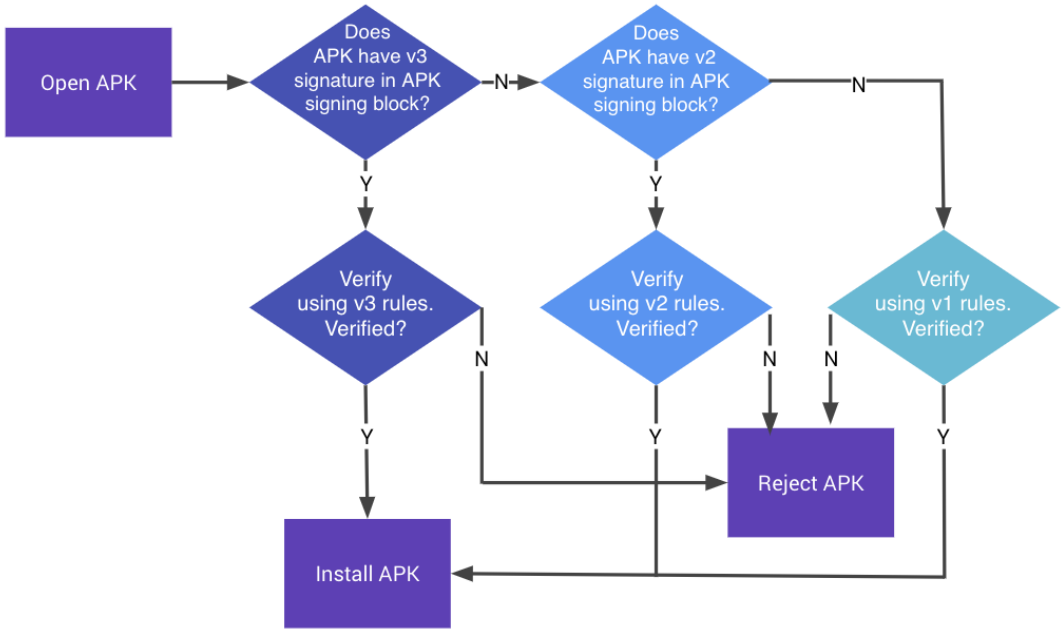


Fig. 6.16

- Whole-file hash of the APK is verified against the v2+ signature stored in the APK Signing Block. The hash covers everything except the APK Signing Block, which contains the v2+ signature.
- Any modification to the APK outside of the APK Signing Block invalidates the APK's v2+ signature. APKs with stripped v2+ signature are rejected as well, because their v1 signature specifies that the APK was v2-signed, which makes Android 7.0 and newer refuse to verify APKs using their v1 signatures.

6.4.3 Deploying App on Google Play Store

- Follow the following steps to developing app on Google Play Store:

Step 1: Create a Developer Account:

- Before we can publish any app on Google Play, we need to create a Developer Account. We can easily sign up for one using our existing Google Account.

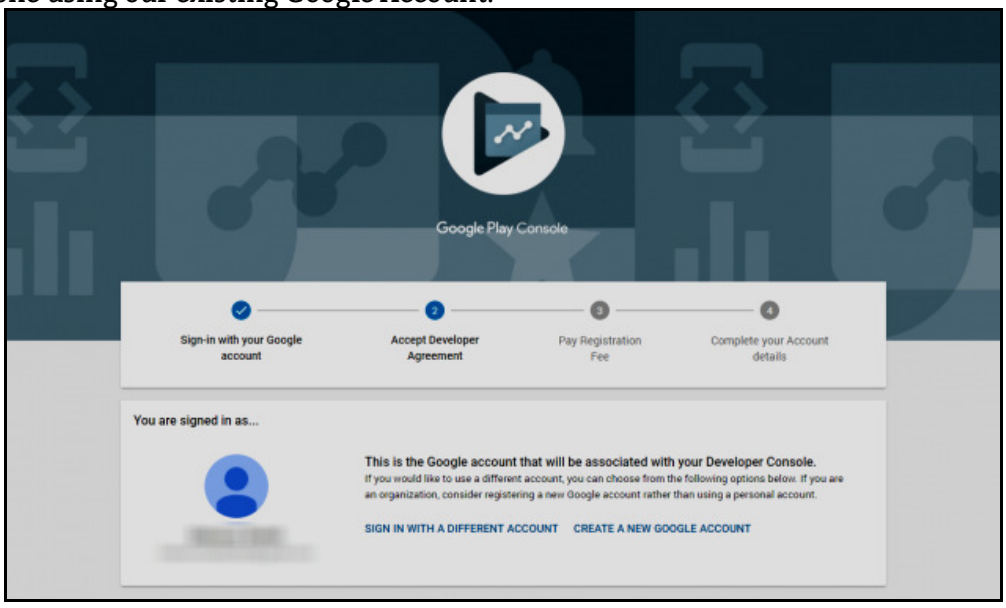


Fig. 6.17

- The sign up process is fairly straightforward, and we will need to pay a one-time registration fee. After we've reviewed and accepted the Developer Distribution Agreement, we can proceed to make the payment using our credit or debit card.
- To finish the sign up process, fill out all our necessary account details, including our Developer Name, which will be visible to our customers on Google Play. We can always add more details later.
- Also, do remember that it can take up to 48 hours for our registration to be fully processed.

Step 2: Link Our Merchant Account:

- If we want to publish a paid app or plan to sell in-app purchases, we need to create a payments center profile, i.e. a merchant account. Here's how we can do that:
  (i) Sign in to our Play Console.
  (ii) Click on Download Reports – Financial.
  (iii) Select 'Set up a merchant account now'.
  (iv) Fill out our business information.
- Once we create the profile, it will be automatically linked to our developer account.
- A merchant account will let we we manage our app sales and monthly payouts, as well as analyze our sales reports right in our Play Console.

Step 3: Create an App:

- Now that we have set up our Play Console, we can finally add our app. Here's how to do that:
  (i) Navigate to the 'All applications' tab in the menu.
  (ii) Click on 'Create Application'.
  (iii) Select our app's default language from the drop-down menu.
  (iv) Type in a title for our app.
  (v) Click on "Create".
- The title of our app will show on Google Play after we've published. Don't worry too much about it at this stage; we can always change the name later.
- After we have created our app, we'll be taken to the store entry page. Here, we will need to fill out all the details for our app's store listing.

Step 4: Prepare Store Listing:

- Before we can publish our app, we need to prepare its store listing. These are all the details that will show up to customers on our app's listing on Google Play.
- Note: We don't necessarily have to complete this step before moving on to the next one. We can always save a draft and revisit it later when we're ready to publish.
- The information required for our store listing is divided into several categories:
  (i) Our app's title and description should be written with a great user experience in mind.
  (ii) Use the right keywords, but don't overdo it. Make sure our app doesn't come across as spam-y or promotional, or it will risk getting suspended on the Play Store.

Graphic Assets:

- Under graphic assets, we can add screenshots, images, videos, promotional graphics, and icons that showcase our app's features and functionality.
- Some parts under graphic assets are mandatory, like screenshots, a feature graphic, and a high resolution icon. Others are optional, but we can add them to make our app look more attractive to users.
- There are specific requirements for each graphic asset that we upload, such as the file format and dimensions.

Languages and Translations:

- We can also add translations of our app's information in the store listing details, along with in-language screenshots and other localized images.

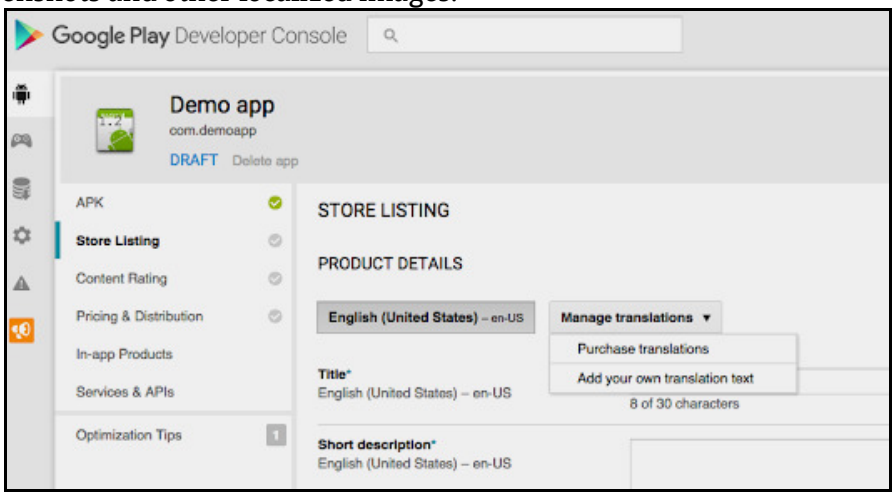


Fig. 6.18

- There's also an option for users to view automated translations of our app's information using Google.
- Translate (with the exception of Armenian, Raeto-romance, Tagalog, and Zulu), in case we don't add our own translations.

Categorization:

- This part requires we to select the appropriate type and category our app belongs to. From the drop-down menu, we can pick either app or game for the application type.

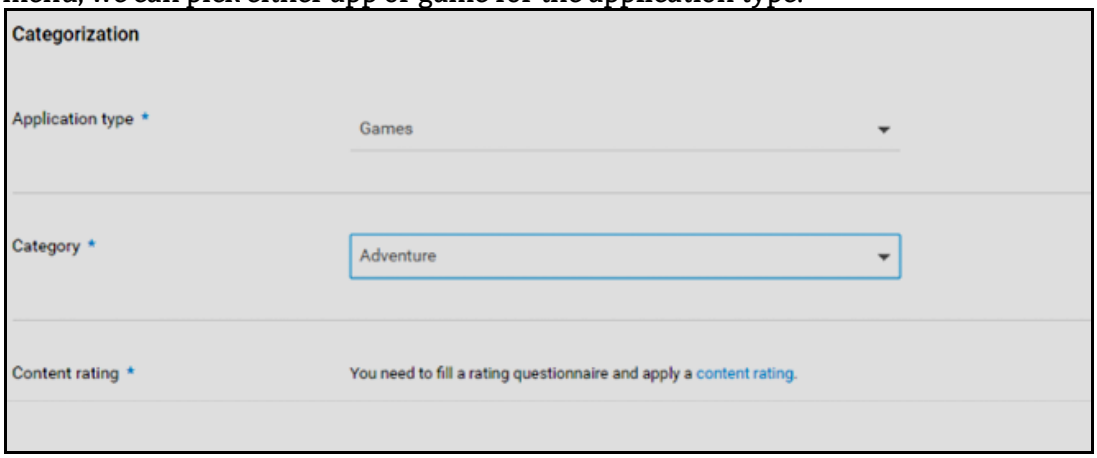


Fig. 6.19

- There are various categories for each type of app available on the Play Store. Pick the one our app fits into best.
- In order to rate our content, we'll need to upload an APK first. We can skip this step for later.

Contact Details:

- This part requires we to enter contact details to offer our customers access to support regarding our app.
- We can add multiple contact channels here, like an email, website, and phone number, but providing a contact email is mandatory for publishing an app.

Privacy Policy:

- For apps that request access to sensitive user data or permissions, we need to enter a comprehensive privacy policy that effectively discloses how our app collects, uses and shares that data.

- We must add a URL linking to our privacy policy in our store listing and within our app. Make sure the link is active and relevant to our app.
- We're now done with the store listing. Go ahead and click on 'Save Draft' to save our details. We can always skip some steps and come back to them later before we publish our app.

Step 5: Upload APK to an App Release:

- Now that we have prepared the ground to finally upload our app, it's time to dig out our APK file.
- The Android Package Kit (or APK, for short) is the file format used by the Android operating system to distribute and install apps. Simply put, our APK file contains all the elements needed for our app to actually work on a device.
- Google offers us to multiple ways to upload and release our APK. Before we upload the file, however, we need to create an app release.
- To create a release, select the app we created in Step 3. Then, from the menu on the left side, navigate to 'Release management' -> 'App releases.'
- Here, we need to select the type of release we want to upload our first app version to. We can choose between an internal test, a closed test, an open test, and a production release.
- The first three releases allow we to test out our app among a select group of users before we make it go live for everyone to access.
- This is a safer option because we can analyze the test results and optimize or fix our app accordingly if we need to before rolling it out to all users.
- However, if we create a production release, our uploaded app version will become accessible to everyone in the countries we choose to distribute it in.
- Once we've picked an option, click on 'Create release.'
- Next, follow the on-screen instructions to add our APK files, and name and describe our release.

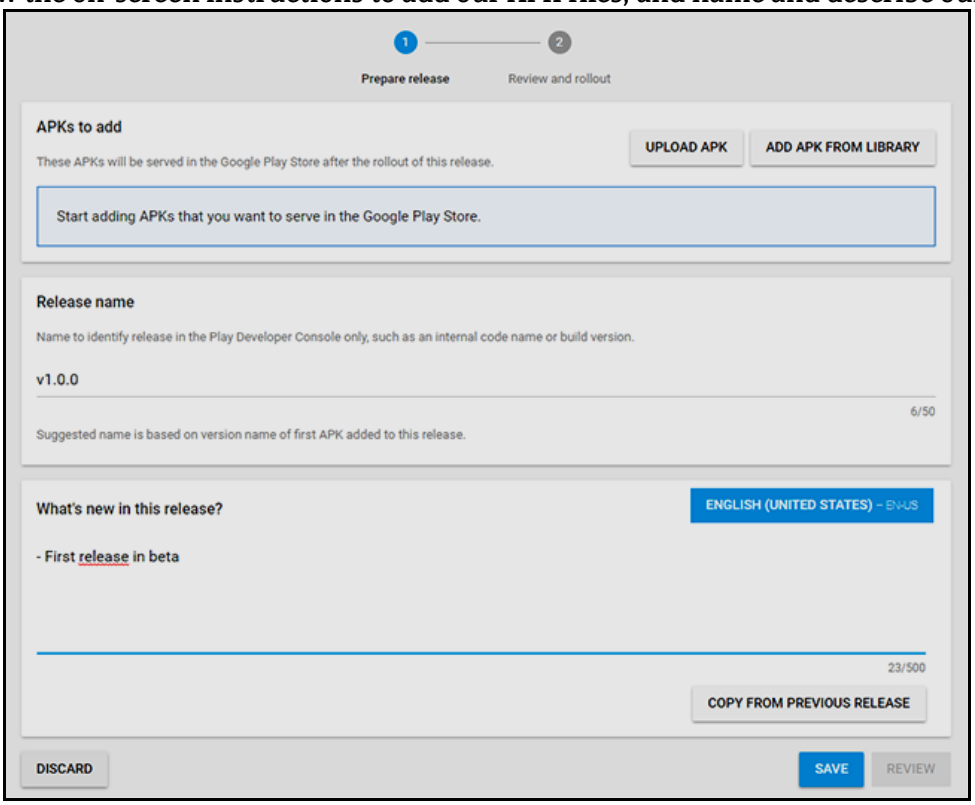


Fig. 6.19

- After we are done, press Save.

Step 6: Provide an Appropriate Content Rating:

- If we don't assign a rating to our app, it will be listed as 'Unrated'. Apps that are 'Unrated' may get removed from Google Play.
- To rate our app, we need to fill out a content rating questionnaire. We can access it when we select our app in the Play Console, and navigate to 'Store presence' – 'Content rating' on the left menu.
- Make sure we enter accurate information. Misrepresentation of our app's content can lead to suspension or removal from the Play Store.
- An appropriate content rating will also help we get to the right audience, which will eventually improve our engagement rates.

Step 7: Set Up Pricing & Distribution

- Before we can fill out the details required in this step, we need to determine our app's monetization strategy.
- Once we know how our app is going to make money, we can go ahead and set up our app as free or paid.

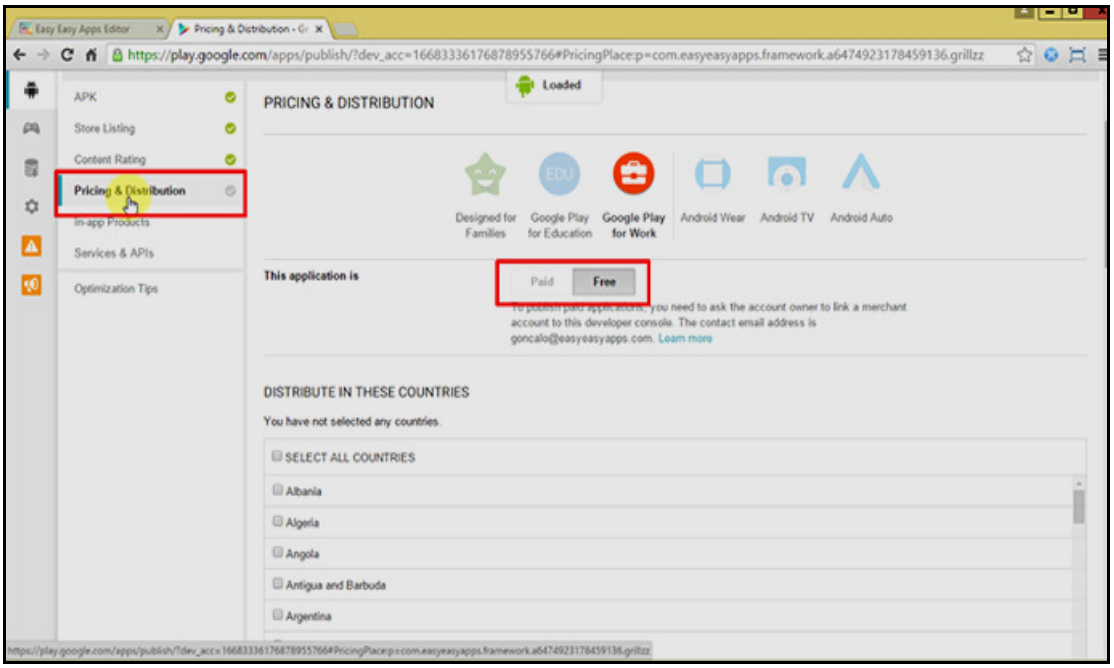


Fig. 6.20

- Remember, we can always change our app from paid to free later, but we cannot change a free app to paid.
- For that, we'll need to create a new app and set its price.
- We can also choose the countries we wish to distribute our app in, and opt-in to distribute to specific Android devices and programs too.

Step 8: Rollout Release to Publish Our App:

- We are almost done. The final step involves reviewing and rolling out our release after making sure we have taken care of everything else.
- Before we review and rollout our release, make sure the store listing, content rating, and pricing and distribution sections of our app each have a green check mark next to them.
- Once we're sure we've filled out those details, select our app and navigate to 'Release management' – 'App releases.' Press 'Edit release' next to our desired release, and review it.
- Next, click on 'Review' to be taken to the 'Review and rollout release' screen. Here, we can see if there are any issues or warnings we might have missed out on.
- Finally, select 'Confirm rollout.' This will also publish our app to all users in our target countries on Google Play.

Practice Questions

1. What is SMS?
2. What is security?
3. What is meant by application development.
4. Explain the term SMS telephony in detail.
5. Describe the term location based services in detail.
6. Explain the following terms with example.
   (i) Displaying the map
   (ii) Displaying zoom control
7. How to monitoring location in Android?
8. Describe Android security model with permissions.
9. How to create and deplaying application on Google Play Store?
10. Describe the term signing of application with example.
11. How to custom permission in Android security model?
12. With the help of example explain how to get locations in Android ?