SESSION #6

LOCATION BASED SERVICES

OUTLINE

- Building Apps with Location & Maps
  - Making Your App Location-Aware
    - https://developer.android.com/training/building-location.html
  - Adding Maps
    - https://developer.android.com/training/maps/index.html
- Getting started with maps: https://developers.google.com/maps/documentation/android-api/start
- Setting Up Google Play Services
- https://developers.google.com/android/guides/setup

OUTLINE

MAKING YOUR APP LOCATION-AWARE

- Getting the Last Known Location
  - Learn how to retrieve the last known location of an Android device, which is usually equivalent to the user's current location.
- Changing Location Settings
  - Learn how to detect and apply system settings for location features.
- Receiving Location Updates
  - Learn how to request and receive periodic location updates.
- Displaying a Location Address
  - Learn how to convert a location's latitude and longitude into an address (reverse geocoding).
- Creating and Monitoring Geofences
  - Learn how to define one or more geographic areas as locations of interest, called geofences, and detect when the user is close to or inside a geofence.

OUTLINE

ADDING MAPS

- Add maps to your app
  - With Google Maps Android API v2, you can embed maps into an activity as a fragment with a simple XML snippet.
- Customize the map
  - Add markers onto the map to indicate special points of interest for your users. You can define custom colors or icons for your map markers to match your app's look and feel.
- Control the user's view
  - Give your users a different view of the world with the ability to control the rotation, tilt, zoom, and pan properties of the "camera" perspective of the map.
- Add Street View to your app
  - Embed Street View into an activity and let your users explore the world through panoramic 360-degree views.

LOCATION-AWARE APPS

- ✧ Knowing user's location in an Android app can be extremely useful:
  - Mobile users take their devices with them everywhere, using them on the go.
  - Developers we can capitalize on that by providing a more contextual experience based on their current location
- ✧ The location APIs facilitate adding location awareness to your app with automated location tracking, *geofencing*, and activity recognition.
- ✧ There are two ways to access location services in android:
  - Android Location Service API
  - Fused Location Provider available with the Google Location Services API.

  The Google Play services location APIs are preferred over the Android framework location APIs (**android.location**).

LOCATION PROVIDERS

- Android provides access to various location technologies such as cell tower location, Wi-Fi, and GPS. Android uses these three location providers:
  - **GPS Provider**: GPS gives the most accurate location, uses the most power, and works best outdoors. This provider uses a combination of GPS and assisted GPS (aGPS), which returns GPS data collected by cellular towers.
  - **Network Provider** – Providers a combination of WiFi and Cellular data, including a GPS data collected by cell towers. It uses less power than the GPS Provider, but returns location data of varying accuracy.
  - **Passive Provider** – A "piggyback" option using providers requested by other applications or Services to generate location data in an application. This is a less reliable but power-saving option ideal for applications that don't require constant location updates to work.

LOCATION PERMISSIONS

- A location-aware application needs access a device's hardware sensors in order to receive GPS, Wi-Fi, and cellular data.
- This access is controlled through appropriate permissions in the application's Android Manifest. The permission you choose determines the accuracy of the location returned by the API:
  - **ACCESS_FINE_LOCATION**: Allows an application access to GPS. Required for the GPS Provider and Passive Provider options (Passive Provider needs permission to access GPS data collected by another application or Service). Allows an app to access precise location.
  - **ACCESS_COARSE_LOCATION**: Allows an application access to Cellular and Wi-Fi location. Required for Network Provider if ACCESS_FINE_LOCATION is not set. Allows an app to access approximate location.

FUSED LOCATION SERVICES

- The Fused Location Provider handles changes in provider status inside the application.
  - The Fused Location Provider is part of Google Play Services, which includes the new Google Location Services APIs.
  - **Google Play Services** must be installed and configured properly in the application for the Fused Location Provider API to work.
- An application using the Fused Location Provider switches dynamically between providers to select the best provider available at any given moment.
- The Fused Location Provider API provides a variety of other tools to empower location-aware applications, including ***geofencing*** and ***activity monitoring***.

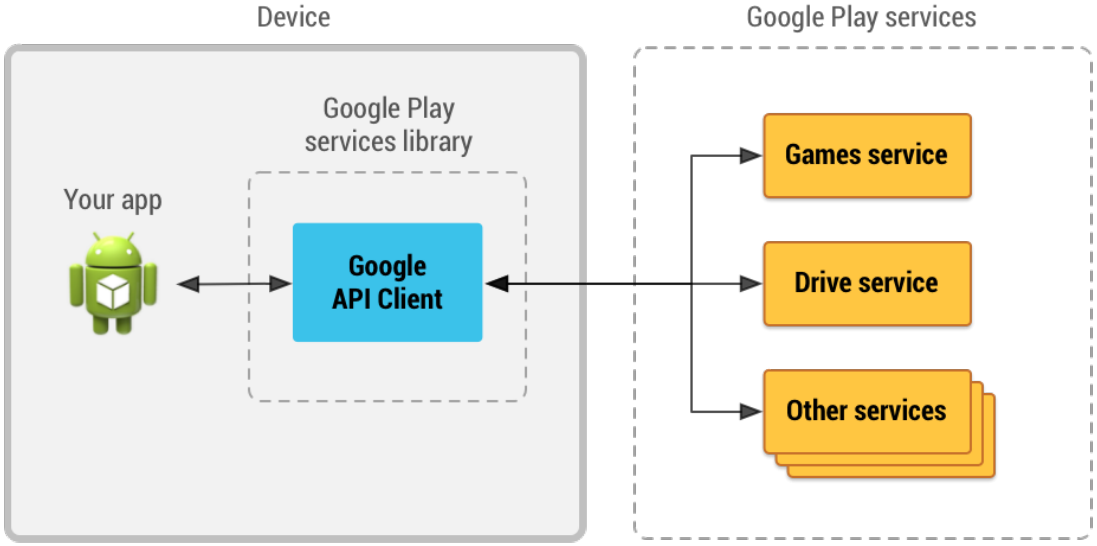
GETTING USER'S LOCATION

- To get the user's location using the Fused Location Provider, we need to:
  - Add the Google Play Services component, and configure the application to use it.
  - Define a **LocationClient** and implement the required interfaces to interact with the Client.
  - For continuous updates, define a **LocationRequest** with requirements for location updates, and provide an instance of **LocationListener** to listen for location changes.
  - Stop location updates when application enters the background.
- To access the fused location provider, app's project must include Google Play services. Download and install the Google Play services component via the SDK Manager and add the library to the project.

GOOGLE PLAY

ACCESS

- When you want to make a connection to one of the Google APIs provided in the Google Play services library, you need to create an instance of *GoogleApiClient* ("Google API Client").
- The Google API Client provides a common entry point to all the Google Play services and manages the network connection between the user's device and each Google service.



CONNECTION

- In your activity's *onCreate()* method, create an instance of Google API Client using **GoogleApiClient.Builder**. Use the builder to add the **LocationServices** API.
- The following function *buildGoogleApiClient()* method, called from the activity's *onCreate()* method, connects with the Google Plays Services:

```
protected synchronized void buildGoogleApiClient() {
    mGoogleApiClient = new GoogleApiClient.Builder(this)
        .addConnectionCallbacks(this)
        .addOnConnectionFailedListener(this)
        .addApi(LocationServices.API)
        .build();
}
```

[1] More about connecting with Google play services: https://developers.google.com/android/guides/api-client#Starting

GOOGLE PLAY

SET UP

- To make the Google Play services APIs available to your app:

1. Open the build.gradle file inside your application module directory.
2. Add a new build rule under dependencies for the latest version of play-services. For example:

```
apply plugin: 'com.android.application'
  ...
  dependencies {
     compile 'com.google.android.gms:play-services:8.3.0‘
     compile 'com.google.android.gms:play-services-maps:8.3.0’
  }
```

   Be sure you update this version number each time Google Play services is updated.

3. Save the changes and click **Sync Project with Gradle Files** in the toolbar.

GETTING LAST KNOWN LOCATION

- When your app is connected to Google Play services and the location services API you can use the fused location provider's *getLastLocation()* method to retrieve the device location.
- To request the last location, call the *getLastLocation()* method, passing it your instance of the GoogleApiClient object. Do this in the *onConnected()* callback provided by Google API Client, which is called when the client is ready.

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener {
    ...
    @Override
    public void onConnected(Bundle connectionHint) {
        mLastLocation = LocationServices.FusedLocationApi.getLastLocation(
                mGoogleApiClient);
        if (mLastLocation != null) {
            mLatitudeText.setText(String.valueOf(mLastLocation.getLatitude()));
            mLongitudeText.setText(String.valueOf(mLastLocation.getLongitude()));
        }
    }
}
```

- The *getLastLocation()* method returns a *Location* object from which you can retrieve the latitude and longitude coordinates of a geographic location.

LOCATION UPDATES (I)

- To request periodic updates with the best available location it is necessary to perform a location request.
- The options for a location request are defined using a *LocationRequest*.
  - **Update interval** - *setInterval()*: This method sets the rate in milliseconds at which your app prefers to receive location updates.
  - **Fastest update interval** - *setFastestInterval()* : This method sets the fastest rate in milliseconds at which your app can handle location updates. You need to set this rate because other apps also affect the rate at which updates are sent.
  - **Priority** - *setPriority()*: This method sets the priority of the request, which gives the Google Play services location services a strong hint about which location sources to use. The following values are supported:
    - **PRIORITY_BALANCED_POWER_ACCURACY** - Use this setting to request location precision to within a city block, which is an accuracy of approximately 100 meters.
    - **PRIORITY_HIGH_ACCURACY** - Use this setting to request the most precise location possible.
    - **PRIORITY_LOW_POWER** - Use this setting to request city-level precision, which is an accuracy of approximately 10 kilometers. This is considered a coarse level of accuracy, and is likely to consume less power.
    - **PRIORITY_NO_POWER** - Use this setting if you need negligible impact on power consumption, but want to receive location updates when available.

LOCATION UPDATES (II)

- Create the location request and set the parameters:

```
protected void createLocationRequest() {
    LocationRequest mLocationRequest = new LocationRequest();
    mLocationRequest.setInterval(10000);
    mLocationRequest.setFastestInterval(5000);
    mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
}
```

The priority of PRIORITY_HIGH_ACCURACY, combined with the ACCESS_FINE_LOCATION permission setting that you've defined in the app manifest, and a fast update interval of 5000 milliseconds (5 seconds), causes the fused location provider to return location updates that are accurate to within a few meters.

- Now that you've set up a location request containing your app's requirements for the location updates, you can start the regular updates by calling *requestLocationUpdates()*. Do this in the *onConnected()*.

```
@Override
public void onConnected(Bundle connectionHint) {
    ...
    if (mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}
```

```
protected void startLocationUpdates() {
    LocationServices.FusedLocationApi.requestLocationUpdates(
            mGoogleApiClient, mLocationRequest, this);
}
```

LOCATION UPDATES (III)

- Depending on the form of the request, the fused location provider either invokes the *LocationListener.onLocationChanged()* callback method and passes it a Location object, or issues a **PendingIntent** that contains the location in its extended data.

```
public class MainActivity extends ActionBarActivity implements
        ConnectionCallbacks, OnConnectionFailedListener, LocationListener {
    ...
    @Override
    public void onLocationChanged(Location location) {
        mCurrentLocation = location;
        mLastUpdateTime = DateFormat.getTimeInstance().format(new Date());
        updateUI();
    }

    private void updateUI() {
        mLatitudeTextView.setText(String.valueOf(mCurrentLocation.getLatitude()));
        mLongitudeTextView.setText(String.valueOf(mCurrentLocation.getLongitude()));
        mLastUpdateTimeTextView.setText(mLastUpdateTime);
    }
}
```

LOCATION UPDATES (IV)

- Consider whether you want to stop the location updates when the activity is no longer in focus, such as when the user switches to another app or to a different activity in the same app.
  - This can be handy to reduce power consumption, provided the app doesn't need to collect information even when it's running in the background.

- To stop location updates, call *removeLocationUpdates()*, passing it your instance of the GoogleApiClient object and a LocationListener:

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

```
@Override
public void onResume() {
    super.onResume();
    if (mGoogleApiClient.isConnected() && !mRequestingLocationUpdates) {
        startLocationUpdates();
    }
}
```

```
protected void stopLocationUpdates() {
    LocationServices.FusedLocationApi.removeLocationUpdates(
            mGoogleApiClient, this);
}
```

LOCATION ADDRESS

- Although latitude and longitude are useful for calculating distance or displaying a map position, in many cases the address of the location is more useful.
- Using the **Geocoder** class in the Android framework location APIs, you can convert an address to the corresponding geographic coordinates.
- The *getFromLocation()* method provided by the **Geocoder** class accepts a latitude and longitude, and returns a list of addresses.
  - The method is synchronous, and may take a long time to do its work, so you should not call it from the main, UI thread of your app.

```
try {
    addresses = geocoder.getFromLocation(
            location.getLatitude(),
            location.getLongitude(),
            // In this sample, get just a single address.
            1);
} catch (IOException ioException) {
    // Catch network or other I/O problems.
    errorMessage = getString(R.string.service_not_available);
    Log.e(TAG, errorMessage, ioException);
} catch (IllegalArgumentException illegalArgumentException) {
    // Catch invalid latitude or longitude values.
    errorMessage = getString(R.string.invalid_lat_long_used);
    Log.e(TAG, errorMessage + ". " +
            "Latitude = " + location.getLatitude() +
            ", Longitude = " +
            location.getLongitude(), illegalArgumentException);
}
```

GEOFENCES

DEFINITION

- **Geofencing** combines awareness of the user's current location with awareness of the user's proximity to locations that may be of interest.
  - To mark a location of interest, you specify its latitude and longitude.
  - To adjust the proximity for the location, you add a radius.
- The latitude, longitude, and radius define a geofence, creating a circular area, or fence, around the location of interest.

GEOFENCES

CREATION

- First, use **Geofence.Builder** to create a geofence, setting the desired radius, duration, and transition types for the geofence.

```
mGeofenceList.add(new Geofence.Builder()
    // Set the request ID of the geofence. This is a string to identify this
    // geofence.
    .setRequestId(entry.getKey())

    .setCircularRegion(
            entry.getValue().latitude,
            entry.getValue().longitude,
            Constants.GEOFENCE_RADIUS_IN_METERS
    )
    .setExpirationDuration(Constants.GEOFENCE_EXPIRATION_IN_MILLISECONDS)
    .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER |
            Geofence.GEOFENCE_TRANSITION_EXIT)
    .build());
```

This example shows the use of two geofence **triggers**. The **GEOFENCE_TRANSITION_ENTER** transition triggers when a device enters a geofence, and the **GEOFENCE_TRANSITION_EXIT** transition triggers when a device exits a geofence. Specifying **INITIAL_TRIGGER_ENTER** tells Location services that **GEOFENCE_TRANSITION_ENTER** should be triggered if the device is already inside the geofence.

GEOFENCES

ADDITION

- To add geofences, use the *GeoencingApi.addGeofences()* method. Provide the Google API client, the **GeofencingRequest** object, and the **PendingIntent**.

```
public class MainActivity extends FragmentActivity {
    ...
    LocationServices.GeofencingApi.addGeofences(
                mGoogleApiClient,
                getGeofencingRequest(),
                getGeofencePendingIntent()
        ).setResultCallback(this);
```

```
public class MainActivity extends FragmentActivity {
    ...
    private PendingIntent getGeofencePendingIntent() {
        // Reuse the PendingIntent if we already have it.
        if (mGeofencePendingIntent != null) {
            return mGeofencePendingIntent;
        }
        Intent intent = new Intent(this, GeofenceTransitionsIntentService.class);
        // We use FLAG_UPDATE_CURRENT so that we get the same pending intent back when
        // calling addGeofences() and removeGeofences().
        return PendingIntent.getService(this, 0, intent, PendingIntent.
                FLAG_UPDATE_CURRENT);
    }
```

HANDLING TRANSITIONS (I)

- When Location Services detects that the user has entered or exited a geofence, it sends out the Intent contained in the **PendingIntent** you included in the request to add geofences.
- This Intent is received by a service like **GeofenceTransitionsIntentService**, which obtains the geofencing event from the intent, determines the type of Geofence transitions, and determines which of the defined geofences was triggered. It then sends a notification as the output.

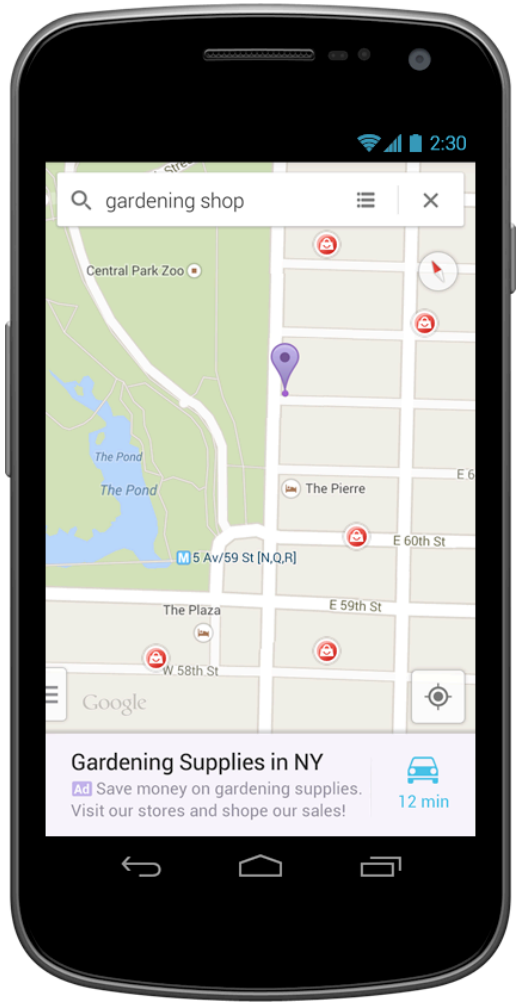
```
public class GeofenceTransitionsIntentService extends IntentService {
    ...
     protected void onHandleIntent(Intent intent) {
         GeofencingEvent geofencingEvent = GeofencingEvent.fromIntent(intent);
         if (geofencingEvent.hasError()) {
             String errorMessage = GeofenceErrorMessages.getErrorString(this,
                     geofencingEvent.getErrorCode());
             Log.e(TAG, errorMessage);
             return;
         }

         // Get the transition type.
         int geofenceTransition = geofencingEvent.getGeofenceTransition();
```

HANDLING TRANSITIONS (II)

```
        // Test that the reported transition was of interest.
        if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_ENTER ||
                geofenceTransition == Geofence.GEOFENCE_TRANSITION_EXIT) {

            // Get the geofences that were triggered. A single event can trigger
            // multiple geofences.
            List triggeringGeofences = geofencingEvent.getTriggeringGeofences();

            // Get the transition details as a String.
            String geofenceTransitionDetails = getGeofenceTransitionDetails(
                    this,
                    geofenceTransition,
                    triggeringGeofences
            );

            // Send notification and log the transition details.
            sendNotification(geofenceTransitionDetails);
            Log.i(TAG, geofenceTransitionDetails);
        } else {
            // Log the error.
            Log.e(TAG, getString(R.string.geofence_transition_invalid_type,
                    geofenceTransition));
        }
    }
```

GOOGLE MAPS

- Google Maps mapping technologies are a ubiquitous complement to mobile devices.
  - Mobile devices use such applications to locate devices and to display changing location information.
- Android has powerful, built-in technology that displays location data on maps using location hardware that may be available on the device:
  - Using the built-in maps application to quickly add mapping functionality.
  - Working with the Maps API to control a map's display.
  - Using a variety of techniques to add graphical overlays.



APPLICATION (I)

- The simplest way to work with maps in is to leverage the built-in maps application.
  - When you use the maps application, the map will not be part of your application. Instead, your application will launch the maps application and load the map externally.
- Working with the maps application is as easy as creating an Intent with an appropriate URI, setting the action to ActionView, and calling the StartActivity method.

```
var geoUri = Android.Net.Uri.Parse ("geo:42.374260,-71.120824");
var mapIntent = new Intent(Intent.ActionView, geoUri);
StartActivity (mapIntent);
```

APPLICATION (II)

- In addition to specifying latitude and longitude, the URI scheme for maps supports several other options:
  - geo:latitude,longitude - Opens the maps application centered at the lat/lon position.
  - geo:latitude,longitude?z=zoom - Opens the maps application centered at a lat/lon and zoomed to the specified level.
    - The zoom level can range from 1 to 23: 1 displays the entire Earth and 23 is the closest zoom level.
  - geo:0,0?q=my+street+address - Opens the maps application to the location of a street address.
  - geo:0,0?q=business+near+city - Opens the maps application and displays the annotated search results.
- You can also load street views using the ActionView intent and specifyng the google.streetview URI scheme:

  google.streetview:cbll=lat,lng&cbp=1,yaw,,pitch,zoom&mz=mapZoom

API

- Using the Maps application is great, but sometimes you want to include maps directly in your application.
- The Maps API is suitable for cases where you want to maintain more control over the mapping experience:
  - Programmatically changing the viewpoint of the map.
  - Adding and customizing markers.
  - Annotating a map with overlays.
- The Google Maps Android API v2 is based on Google Play Services.
- To use this API you need to perform some configuration steps:
  - Installing the Google Play Services SDK.
  - Create a Google Maps project
  - Obtaining a Maps API key

  https://developers.google.com/maps/documentation/android-api/start

API KEY – google_maps_api.xml

```
<resources>
    <!--
    TODO: Before you run your application, you 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=F5:30:6F:CC:63:18:F7:7C:
3B:04:D5:C1:CC:B1:63:5F:B4:DB:6D:54%3Bcat.eps.jgervas.basicmap

    You can also add your credentials to an existing key, using this line:
    F5:30:6F:CC:63:18:F7:7C:3B:04:D5:C1:CC:B1:63:5F:B4:DB:6D:
54;cat.eps.jgervas.basicmap

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

    Once you have your key (it starts with "AIza"), replace the "google_maps_key"
    string in this file.
    -->
    <string name="google_maps_key" templateMergeStrategy="preserve"
translatable="false">YOUR_API_KEY</string>
</resources>
```

GOOGLEMAP CLASS

- The **GoogleMap** class is the main API that a application will use to display and interact with a Google Maps.
- This class has the following responsibilities:
  - Interacting with Google Play services to authorize the application with the Google web service.
  - Downloading, caching, and displaying the map tiles.
  - Displaying UI controls such as pan and zoom to the user.
  - Drawing markers and geometric shapes on maps.
- The GoogleMap is added to an Activity in one of two ways:
  - **MapFragment** - The MapFragment is a specialized Fragment that acts as host for the GoogleMap object.
  - **MapView** - The MapView is a specialized View subclass which can act as a host for a GoogleMap object.

MAP FRAGMENT

- You can attach MapFragment to the activity defining a Fragment object in the layout. In this element, set the android:name attribute to "com.google.android.gms.maps.MapFragment".

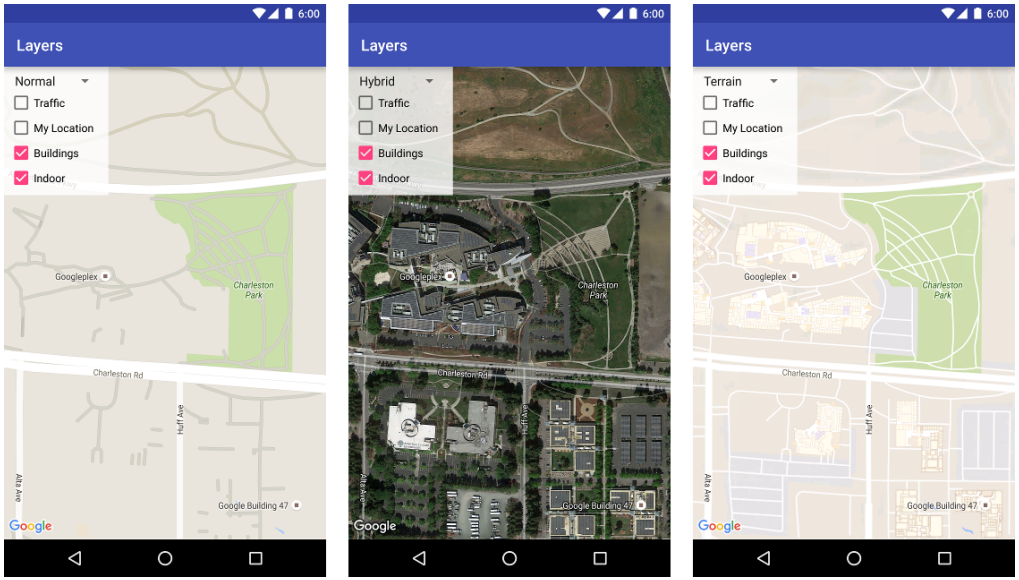
```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
    android:name="com.google.android.gms.maps.MapFragment"
    android:id="@+id/map"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

- You can also add a MapFragment to an Activity in code. To do this, create a new MapFragment instance, and then call *FragmentTransaction.add()* to add the Fragment to the Activity:

```
mMapFragment = MapFragment.newInstance();
FragmentTransaction fragmentTransaction = getFragmentManager().beginTransaction();
fragmentTransaction.add(R.id.my_container, mMapFragment);
fragmentTransaction.commit();
```

MAPS TYPES

- There are five different types of maps:
  - Normal: This is the default map type. It shows roads and important natural features along with some man-made points of interest (such as buildings).
  - Satellite: This map shows satellite photography.
  - Hybrid: This map shows satellite photography and road maps.
  - Terrain: This primarily shows topographical features with some roads.
  - None: This map does not load any tiles, it is rendered as an empty grid.



MAPS TYPES

- The GoogleMap.MapType property is used to set or change which type of map is displayed. The following code shows how to display a satellite map:

```
MapFragment mapFrag = (MapFragment)FragmentManager.FindFragmentById(Resource.Id.my_mapfragment);
GoogleMap map = mapFrag.Map;
if (map != null) {
    map.MapType = GoogleMap.MapTypeSatellite;
}
```

- You can also set the type of a map, calling the GoogleMap object's setMapType() method, passing one of the type constants defined in GoogleMap:

```
@Override
public void onMapReady(GoogleMap map) {
    // Sets the map type to be "hybrid"
    map.setMapType(GoogleMap.MAP_TYPE_HYBRID);
}
```

PROPERTIES

- GoogleMap defines several properties that can control the functionality and the appearance of the map.
- One way to configure a GoogleMap object is by setting values the UiSettings property of the map object. This code sample shows how to configure a GoogleMap to display the zoom controls and a compass:

```
MapFragment mapFrag = (MapFragment)
FragmentManager.FindFragmentById(Resource.Id.my_mapfragment_container);
GoogleMap map = mapFrag.Map;
if (map != null) {
    map.UiSettings.ZoomControlsEnabled = true;
    map.UiSettings.CompassEnabled = true;
}
```

- Other alternative way to configure the initial state of a GoogleMap is to pass a GoogleMapOptions object when creating a MapFragment.

DRAWING

✧ The Android Maps API v2 provides API's for drawing the following items on a map:

- **Markers:** A special icons that are used to identify a single location on a map.
- **Overlays:** A image that can be used to identify a collection of locations or area on the map.
- **Lines, Polygons, and Circles** - These are APIs that allow Activities to add shapes to a map.

MARKERS

- The Maps API provides a Marker class which encapsulates all of the data about a single location on a map.
  - By default they use a standard icon provided by Google Maps.
  - It is possible to customize the appearance of a marker and to respond to user clicks.
- To add a marker to a map, it is necessary create a new MarkerOptions object and then call the AddMarker() method on a GoogleMap instance. This method will return a Marker object.

```
MapFragment mapFrag = (MapFragment)
FragmentManager.FindFragmentById(Resource.Id.my_mapfragment_container);
GoogleMap map = mapFrag.Map;
if (map != null) {
    MarkerOptions markerOpt1 = new MarkerOptions();
    markerOpt1.SetPosition(new LatLng(50.379444, 2.773611));
    markerOpt1.SetTitle("Vimy Ridge");
    map.AddMarker(marker1);
}
```

- It is possible to customize the icon used by the marker by calling the MarkerOptions.InvokeIcon() method when adding the marker to the map.

INFO WINDOWS

- Info windows are special windows that popup to display information to the user when they tap a specific marker.
  - By default the info window will display the contents of the marker's title. If the title has not been assigned, then no info window will appear.
  - Only one info window may be shown at a time.
- It is possible to customize the info window by implementing the GoogleMap.IInfoWindowAdapter interface. There are two important methods on this interface:
  - public View GetInfoWindow(Marker marker): This method is called to get a custom info window for a marker. If it returns null , then the default window rendering will be used. If this method returns a View, then that View will be placed inside the info window frame.
  - public View GetInfoContents(Marker marker) - This method will only be called if GetInfoWindow returns null . This method can return a null value if the default rendering of the info window contents is to be used. Otherwise, this method should return a View with the contents of the info window.

GROUND OVERLAYS

- Unlike markers, which identify a specific location on a map, a GroundOverlay is an image that used to identify a collection of locations or an area on the map.
- Adding a ground overlay to a map is very similar to adding a marker to a map. First, a GroundOverlayOptions object is created. This object is then passed as a parameter to the GoogleMap.AddGroundOverlay() method, which will return a GroundOverlay object:

```
BitmapDescriptor image =
BitmapDescriptorFactory.FromResource(Resource.Drawable.polarbear);
GroundOverlayOptions groundOverlayOptions = new GroundOverlayOptions()
        .Position(position, 150, 200)
        .InvokeImage(image);
GroundOverlay myOverlay = _map.AddGroundOverlay(groundOverlayOptions);
```

LINES, CIRCLES AND POLYGONS

- There are three simple types of geometric figures that can be added to a map:
  - Polyline - This is a series of connected line segments. It can mark a path on a map or form any shape required.
  - Polygon - This is an closed shape for marking areas on a map.
  - Circle - This will draw a circle on the map.

```
PolylineOption rectOptions = new PolylineOption();
rectOptions.Add(new LatLng(37.35, -122.0));
rectOptions.Add(new LatLng(37.45, -122.0));
rectOptions.Add(new LatLng(37.45, -122.2));
rectOptions.Add(new LatLng(37.35, -122.2));
rectOptions.Add(new LatLng(37.35, -122.0)); // close the polyline - this makes a rectangle.
myMap.AddPolyline(rectOptions);

CircleOptions circleOptions = new CircleOptions ();
circleOptions.InvokeCenter (new LatLng(37.4, -122.1));
circleOptions.InvokeRadius (1000);
myMap.AddCircle (CircleOptions);
```

EVENTS (I)

✧ There are three types of interactions a user may have with a map:

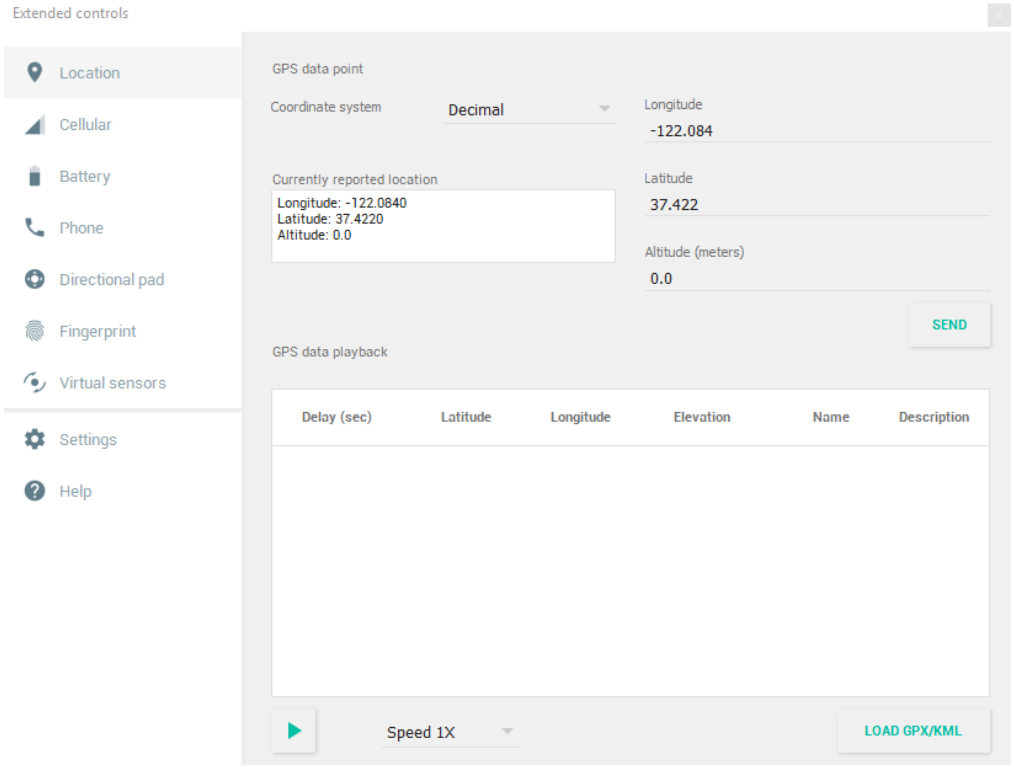
- **Marker Click** - The user clicks on a marker.
  - When the user clicks on a marker the MarkerClick event will be raised, and a GoogleMap.MarkerClickEventArgs passed in.
- **Marker Drag** - The user has long-clicked on a marker
  - This event is raised when the user wishes to drag the marker. By default, markers are not draggable. A marker can be set as draggable by setting the Marker.Draggable property to true or by invoking the MarkerOptions.Draggable() method with true as a parameter.
    - GoogleMap.MarkerDragStart(object sender, GoogleMap.MarkerDragStartEventArgs e)
    - GoogleMap.MarkerDrag(object sender, GoogleMap.MarkerDragEventArgs e)
    - GoogleMap.MarkerDragEnd(object sender, GoogleMap.MarkerDragEndEventArgs e)
- **Info Window Click** - The user has clicked on an info window.
  - Only one info window can be displayed at a time. When the user clicks on an info window in a map, the map object will raise an InfoWindowClick event.

EVENTS (II)

```
/** this will be the marker click listener shows a popup window above the maps marker The
popup will contain 3 buttons: Chat, Track, SMS  */

public void onClick_marker(){
    map.setOnMarkerClickListener(new OnMarkerClickListener(){
         @Override public boolean onMarkerClick(    Marker marker){
             if (!marker.getSnippet().equals(myMarker.getSnippet())) {
                 map.setInfoWindowAdapter(new
EventMarkerInfoWindow( (LayoutInflater)getSystemService(LAYOUT_INFLATER_SERVICE)));
                 marker.showInfoWindow();
             }
             return true;
         }
     });
}

@Override
public void onInfoWindowClick(Marker marker) {
    Toast.makeText(getBaseContext(),
            "Info Window clicked@" + marker.getId(),
            Toast.LENGTH_SHORT).show();
}
```

EXAMPLES

✧ Test in emulator … ¿?



EXAMPLES

- Location services: https://github.com/googlesamples/android-play-location

- Maps: https://github.com/googlemaps/android-samples