Week 7 - Location and Maps

A few years ago my family spent some time in Montreal, Canada and just as we arrived it began to snow and snow and snow. Of course, we quickly discovered that my two daughters had forgotten to pack their snow boots, so we needed to buy boots and we needed to buy them from a store that was close to us. I whipped out my mobile phone and ran a search for boots in the local area.

The application came back with a list of shoe stores plotted on a map. It also allowed me to get directions from my current location to the stores. We followed those directions, bought some boots and saved our vacation.

I'm sure many of you could recount similar experiences. Sometimes the information we need depends critically on where we are at the time we need that information. To help out in these situations, Android includes support for location and maps.

In today's lesson, I'll talk about some of that support: about what location information is, and the classes your applications will use to get it. I talk about maps, which allow you to take location information and display it visually. I'll finish up by going over the classes that Android provides, to let you display and customize maps.

Location

Generally speaking, mobile applications can benefit from being location-aware. To know where things are at a particular moment Android allows applications to determine and manipulate location information. Earlier I gave an example of using location capabilities to find stores near my current location, and then to get directions from my current location to one of those stores. Applications can also use this capability to do things like define a geographical area, or **Geofence**, and then to initiate actions when the user enters or exits the Geofence.

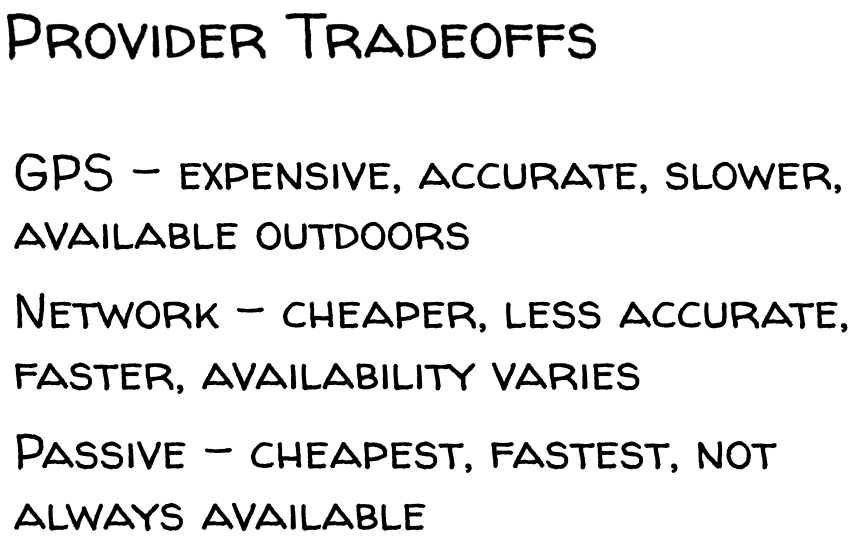
Android provides several support classes to make this possible. One is the **Location** class. A Location represents a position on the Earth. A Location instance contains information such as latitude, longitude, a time stamp and, optionally, an estimated accuracy, altitude, speed and bearing. Location information comes from Location Providers - devices can have access to multiple Location Providers.

The actual data may come from sources such as GPS satellites, cell phone towers and WiFi access points. Specifically, applications can request information from the **Network provider,** the **GPS provide**r and the **Passive provider.**

The **Network provider**, determines location based on cell tower and WiFi access points. If you want to use this provider, then you must declare either the **ACCESS_COARSE_LOCATION permission** or the **ACCESS_FINE_LOCATION**

permission. The **GPS provider** gets its location data from GPS satellites. To use this provider, you must declare the **ACCESS_FINE_LOCATION** permission. The **Passive provider** doesn't actually turn on any devices - it returns locations that happen to have been calculated through the requests of other applications. So using this provider requires that you declare the **ACCESS_FINE_LOCATION** permission.

While you can get location information from each of these different sources, each offers a differing trade-offs with respect to cost, accuracy, availability, and timeliness of the data.



Let's look more closely at some of the providers available in Android.

The GPS provider relies on communicating with a satellite - it is generally the most expensive but gives the most accurate readings. It also takes the longest amount of time to provide that very accurate reading, and the user needs to have a clear view of the sky when they're communicating with the GPS satellite.

The Network provider is cheaper than the GPS provider, but may give less accurate readings. It takes less time to return location information than GPS, but it's useful only when you're in the range of a cell tower or WiFi access point.

The Passive provider is the cheapest to use - you're essentially just reusing measurements that already have been taken. So it's fast, but it may turn out that there are no recent readings when your application asks for that information.

Location support classes

One way to access location information is to use the **LocationManager** class, which is a System service for accessing location data. You first acquire a reference to the location manager by calling the Context class's getSystemService method, passing in an ID for the service: **Context.LOCATION_SERVICE**.

Once you have a reference to the LocationManager, you can use it to get location information. For instance, you can determine the last reading taken by a particular provider. You can register for location updates to find out when new location information is acquired. And you can also register to receive intents when the device nears or moves away from a given geographic region.

If you want to be informed when new locations are determined, you can implement and use a **LocationListener**. The location listener interface defines the callback methods that are called when location changes or when the status of a location provider changes. A LocationListener interface includes the following methods:

- **void onLocationChanged(Location location**): called when a new location is determined
- **void onProviderDisabled(String provider)**: called when the user disables a particular provider
- **void onProviderEnabled(String provider)**: called when the user enables a particular provider
- **void onStatusChanged(String provider, int status, Bundle extras):**

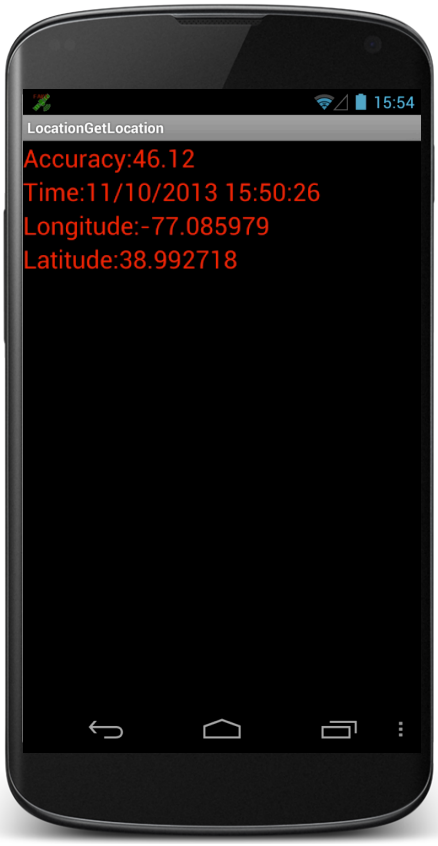
If your application cannot get a recent reading from the system, it will need to acquire its own reading and will normally perform the following steps:

1. Start listening for updates from location providers by registering a Location listener
2. Maintain and update a current best estimate as it begins to receive location updates
3. Determine when the current best estimate is good enough
4. Stop listening for location updates by unregistering the Location listener
5. Finally, use that best estimate as the current location.

When you're determining whether your location is good enough, there are several factors that you might want to consider. For example, for how long should you keep measuring? A navigation system might need continuous measurement while a restaurant finder application might need just a single measurement. Another question is, how accurate a measurement do you actually need? Again, a navigation system needs to know your location to say, within ten meters or so. A restaurant application, might just

need to know what city you're in, in which case you'd only need to know your location to within a kilometer or so. Of course, the choices you make here will affect battery usage. The example application, called **LocationGetLocation**, first acquires and displays the last known location from all the providers on the device. If these readings are too old, or have too low accuracy, then the application acquires and displays new readings from all the providers.

Let's give **LocationGetLocation** a run. When it starts up, the best previous location estimate from the device is displayed using red text. This reading is either not recent enough or doesn't have enough accuracy, so the application acquires new location estimates. These new readings are displayed using grey text.



Let's look at the source code for this application's main activity.
package course.examples.Location.GetLocation;

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.List;
import java.util.Locale;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
```

```
import android.app.Activity;
import android.content.Context;
import android.graphics.Color;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;

public class LocationGetLocationActivity extends Activity {

      private static final long ONE_MIN = 1000 * 60;
      private static final long TWO_MIN = ONE_MIN * 2;
      private static final long FIVE_MIN = ONE_MIN * 5;
      private static final long MEASURE_TIME = 1000 * 30;
      private static final long POLLING_FREQ = 1000 * 10;
      private static final float MIN_ACCURACY = 25.0f;
      private static final float MIN_LAST_READ_ACCURACY = 500.0f;
      private static final float MIN_DISTANCE = 10.0f;


      // Views for display location information
      private TextView mAccuracyView;
      private TextView mTimeView;
      private TextView mLatView;
      private TextView mLngView;

      private int mTextViewColor = Color.GRAY;

      // Current best location estimate
      private Location mBestReading;

      // Reference to the LocationManager and LocationListener
      private LocationManager mLocationManager;
      private LocationListener mLocationListener;

      private final String TAG = "LocationGetLocationActivity";

      private boolean mFirstUpdate = true;

      @Override
      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);

            setContentView(R.layout.main);

            mAccuracyView = (TextView) findViewById
                  (R.id.accuracy_view);
            mTimeView = (TextView) findViewById(R.id.time_view);
            mLatView = (TextView) findViewById(R.id.lat_view);
            mLngView = (TextView) findViewById(R.id.lng_view);
```

```
// Acquire reference to the LocationManager
if (null == (mLocationManager = (LocationManager)
      getSystemService(Context.LOCATION_SERVICE)))
      finish();

// Get best last location measurement
mBestReading = bestLastKnownLocation
      (MIN_LAST_READ_ACCURACY, FIVE_MIN);

// Display last reading information
if (null != mBestReading) {

      updateDisplay(mBestReading);

} else {

      mAccuracyView.setText("No Initial Reading Available");

}

mLocationListener = new LocationListener() {

      // Called back when location changes

      public void onLocationChanged(Location location) {

            ensureColor();

      // Determine whether new location is better than
      // current best estimate

            if (null == mBestReading
                  || location.getAccuracy() <
                  mBestReading.getAccuracy()) {

                  // Update best estimate
                  mBestReading = location;

                  // Update display
                  updateDisplay(location);

                  if (mBestReading.getAccuracy() <
                        MIN_ACCURACY)
                        mLocationManager.removeUpdates
                        (mLocationListener);
            }
      }

      public void onStatusChanged(String provider, int
            status, Bundle extras) {
            // NA
      }

      public void onProviderEnabled(String provider) {
```

```
                // NA
            }

            public void onProviderDisabled(String provider) {
                // NA
            }
        };
    }

    @Override
    protected void onResume() {
        super.onResume();

        // Determine whether initial reading is
        // "good enough"

        if (mBestReading.getAccuracy() > MIN_LAST_READ_ACCURACY
                || mBestReading.getTime() <
                System.currentTimeMillis()- TWO_MIN) {

            // Register for network location updates
            mLocationManager.requestLocationUpdates(
                LocationManager.NETWORK_PROVIDER, POLLING_FREQ,
                MIN_DISTANCE, mLocationListener);

            // Register for GPS location updates
            mLocationManager.requestLocationUpdates(
                LocationManager.GPS_PROVIDER, POLLING_FREQ,
                MIN_DISTANCE, mLocationListener);

            // Schedule a runnable to unregister location
            // listeners

            Executors.newScheduledThreadPool(1).schedule(new
                Runnable() {

                @Override
                public void run() {

                    Log.i(TAG, "location updates cancelled");

                    mLocationManager.removeUpdates
                        (mLocationListener);

                }
            }, MEASURE_TIME, TimeUnit.MILLISECONDS);
        }
    }

    // Unregister location listeners

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

```
      mLocationManager.removeUpdates(mLocationListener);
}

// Get the last known location from all providers
// return best reading is as accurate as minAccuracy and
// was taken no longer then minTime milliseconds ago

private Location bestLastKnownLocation(float minAccuracy,
      long minTime) {

      Location bestResult = null;
      float bestAccuracy = Float.MAX_VALUE;
      long bestTime = Long.MIN_VALUE;

      List<String> matchingProviders =
            mLocationManager.getAllProviders();

      for (String provider : matchingProviders) {

            Location location =
                  mLocationManager.getLastKnownLocation(provider);

            if (location != null) {
                  float accuracy = location.getAccuracy();
                  long time = location.getTime();
                  if (accuracy < bestAccuracy) {
                        bestResult = location;
                        bestAccuracy = accuracy;
                        bestTime = time;
                  }
            }
      }

      // Return best reading or null
      if (bestAccuracy > minAccuracy || bestTime < minTime) {
            return null;
      } else {
            return bestResult;
      }
}

// Update display
private void updateDisplay(Location location) {

      mAccuracyView.setText("Accuracy:" + location.getAccuracy
            ());
      mTimeView.setText("Time:"new SimpleDateFormat
            ("MM/dd/yyyy HH:mm:ss", Locale.getDefault()).format
            (new Date(location.getTime())));
      mLatView.setText("Longitude:" + location.getLongitude());
      mLngView.setText("Latitude:" + location.getLatitude());

}
```

```
	private void ensureColor() {
		if (mFirstUpdate) {
			setTextViewColor(mTextViewColor);
			mFirstUpdate = false;
		}
	}

	private void setTextViewColor(int color) {
		mAccuracyView.setTextColor(color);
		mTimeView.setTextColor(color);
		mLatView.setTextColor(color);
		mLngView.setTextColor(color);
	}
}
```

Scrolling to the **onCreate** method, we see that the code acquires a reference to the LocationManager. Next, it calls a method called bestLastKnownLocation, which will find the last known location from every location provider, and return the most accurate of these measurements that also meets certain criteria. If no readings meets those criteria, the method returns null.

Next, the code displays information about the last reading. The code continues by defining a LocationListener. In the listener's onLocationChange method the code determines whether the new location is better than the current best estimate. If so, the code then updates the best estimate, and then updates the display. If the accuracy is less than min_accuracy, then the current location is considered good enough, so the code unregisters the location listener.

Scrolling down, you see the **onResume** method, which checks whether the current best estimate is of low accuracy, or was taken more than two minutes ago. If so, the code registers Listeners for both the Network provider, and for the GPS provider. Then, the code schedules a **Runnable** that will unregister the listeners after a fixed period of time. The code will also unregister the LocationListener if the activity's onPause method is called.

Here are some tips you can use to save battery power when you're creating location-aware applications.

- **Always check the last-known measurement.** If that's good enough, then there's no need to take new measurements.
- **Return updates as infrequently as possible and limit the total measurement time.** Some applications, such as an application that tracks a jogger, need to update more frequently and need to keep measuring while the application is running because the user's location is changing. Applications like the one we just saw though will need a single good measurement so they can measure infrequently and for less time.
- **Use the least accurate measurement necessary, and only use GPS if you really need to**.
- **Turn off the updates in OnPause**.

Maps

Locations indicate real places around us, and so it often makes sense to visualize locations using maps, which are visual representations of area. Android provides mapping support through the Google Maps Android v2 API, which provides several different kinds of maps:

- **Normal maps**, which look like traditional road maps.
- **Satellite maps**, which display aerial photographs of an area.
- **Hybrids maps**, which combine satellite photographs and road maps.
- **Terrain maps**, which display topographic details such as elevation.

Android allows your application to customize maps in several ways. For instance, you can change the area of the map that's visible to the user. You can add icons, called **Markers**, at specific places on the map. And you can add overlay images on top of the map. You can make the map respond to gestures, such as a two-finger Stretch and Pinch to zoom. And a two-finger Rotation to rotate the map. You can also have the map indicate the user's current location, e.g., by placing a special marker on the map.

Map Support Classes

To display maps, your application will probably use some of the following classes:

- **GoogleMap**, which represents and manages the map itself
- **MapFragmen**t, which displays a Google map within a fragment
- **Camera**, which defines the part of the map that's visible on the screen, and defines the viewpoint from which the user is seeing the map.
- **Marker**, which represents icons, sometimes with popup windows, that indicate locations on the map and allows your application to display information associated with those locations.

To set up and run a maps application, you'll need to take some extra steps.
- Configure the Google Play services SDK.
- Obtain an API key that identifies your application.
- Specify permissions, settings, and the API key in the AndroidManifest.XML file.
- Add the map to your application.

These steps are described in more detail at:
See: https://developers.google.com/maps/documentation/android/start

In order to use Maps, you'll need to include several permissions:

Internet permission, so that map images can be downloaded from Google Map servers:
<uses-permission android:name="android.permission.INTERNET"/>

Network state permission. Which the Maps API uses to determine whether it can download data:
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

Write external storage permission because map data needs to be written to the device's external storage area:
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

Google read services permission, so the Maps API can access Google Web Services:
<uses-permission
android:name="com.google.android.providers.gsf.permission.READ_GSERVICES"/>

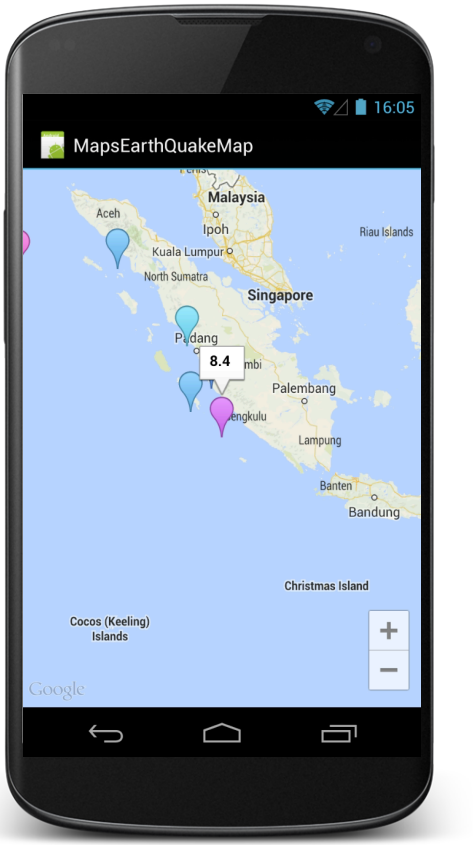
Coarse and fine location permissions. If your map needs to acquire location information, e.g. to display the user's current location then you'll need one or more of these:
<uses-permission
android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

Our next example application is called MapEarthquakeMap, which acquires earthquake data from a server and displays it on a map using clickable markers. Let's take a look, but first, let me take you back to the lesson on networking.

Recall that I showed you the NetworkingAndroid HTTPClientJSON application, which issued a network request for earthquake data from the api.geonames.org web service. The request returned a list of earthquakes that had occurred in a particular geographic region, and the application displayed that data in a simple ListView.

As you can see, the application gets the earthquake data, and then presents it in a list view. Now this is certainly fine - the data I wanted is there on the screen. But the user interface isn't all that helpful to me. For example, I can't really visualize where on the Earth these locations really are. And I can't easily distinguish major earthquakes from lesser strength earthquakes.

So let's now look at the MapEarthquakeMap application. And we'll see the same data but this time it's presented in a map. Here goes:



As you can see, instead of a list view showing lots of text the earthquake data now appears as a set of markers on a map of the world. The location of the marker tells us where the earthquake occurred. And the color of the marker indicates the magnitude of the earthquake. Markers that are more red in tone indicate higher magnitude earthquakes. Markers that are more blue in tone indicate lower magnitude earthquakes. If I touch a marker, a pop-up window will appear showing the magnitude of that earthquake.

Let's take a look at the source code for this application. Here's the application's main activity. In **OnCreate**, the application sets the content view to the main.xml file in the res/layout directory. Lets open that file now:

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

And here you can see that the entire layout is comprised of a single fragment that is provided by the com.google.android.gms.maps.MapFragment class.

Turning back to the main activity, the application creates and starts an asyncTask, which acquires the earthquake data in the do in background method. And then it parses it and updates the map In the onPostExecute method. Let's scroll down to the onPostExecute method now:

```
package course.examples.Maps.EarthQuakeMap;

import java.io.IOException;
import java.util.List;

import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.methods.HttpGet;

import android.app.Activity;
import android.net.http.AndroidHttpClient;
import android.os.AsyncTask;
import android.os.Bundle;
import android.util.Log;

import com.google.android.gms.maps.CameraUpdateFactory;
import com.google.android.gms.maps.GoogleMap;
import com.google.android.gms.maps.MapFragment;
import com.google.android.gms.maps.model.BitmapDescriptorFactory;
import com.google.android.gms.maps.model.LatLng;
import com.google.android.gms.maps.model.MarkerOptions;

public class MapsEarthquakeMapActivity extends Activity {

      // Coordinates used for centering the Map

      private static final double CAMERA_LNG = 87.0;
      private static final double CAMERA_LAT = 17.0;

      // The Map Object
      private GoogleMap mMap;

      // URL for getting the earthquake
      // replace with your own user name

      private final static String UNAME = "aporter";
      private final static String URL = "http://api.geonames.org/
earthquakesJSON?north=44.1&south=-9.9&east=-22.4&west=55.2&username="
                  + UNAME;

      public static final String TAG = "MapsEarthquakeMapActivity";

      // Set up UI and get earthquake data
      @Override
      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);

            setContentView(R.layout.main);
```

```
        new HttpGetTask().execute(URL);

    }

    private class HttpGetTask extends
                AsyncTask<String, Void, List<EarthQuakeRec>> {

        AndroidHttpClient mClient = AndroidHttpClient.newInstance
                ("");

        @Override
        protected List<EarthQuakeRec> doInBackground(String...
                params) {

                HttpGet request = new HttpGet(params[0]);
                JSONResponseHandler responseHandler = new
                        JSONResponseHandler();

                try {

                        // Get Earthquake data in JSON format
                        // Parse data into a list of EarthQuakeRecs

                        return mClient.execute(request, responseHandler);

                } catch (ClientProtocolException e) {
                        Log.i(TAG, "ClientProtocolException");
                } catch (IOException e) {
                        Log.i(TAG, "IOException");
                }

                return null;

        }

        @Override
        protected void onPostExecute(List<EarthQuakeRec> result) {

                // Get Map Object
                mMap = ((MapFragment) getFragmentManager
                        ().findFragmentById(R.id.map)).getMap();

                if (null != mMap) {

                        // Add a marker for every earthquake

                        for (EarthQuakeRec rec : result) {

                                // Add a new marker for this earthquake
                                // and set the Marker's position,
                                // set the title of the Marker's
                                // information window and set the color for
                                // the Marker.
                                mMap.addMarker(new MarkerOptions()
```

```
                                    .position(new LatLng(rec.getLat(),
                                    rec.getLng())).title(String.valueOf
                                    (rec.getMagnitude()))
                                    .icon(BitmapDescriptorFactory
                                    .defaultMarker(getMarkerColor
                                    (rec.getMagnitude())));
                        }

                        // Center the map
                        // Should compute map center from the actual data
                        mMap.moveCamera(CameraUpdateFactory.newLatLng(
                              new LatLng(CAMERA_LAT, CAMERA_LNG)));
                  }

                  if (null != mClient)
                        mClient.close();

            }

            // Assign marker color
            private float getMarkerColor(double magnitude) {

                  if (magnitude < 6.0) {
                        magnitude = 6.0;
                  } else if (magnitude > 9.0) {
                        magnitude = 9.0;
                  }

                  return (float) (120 * (magnitude - 6));
            }

      }

}
```

This method begins by getting a reference to the Google Map underlying the MapFragment. Next, it iterates through the result data for each **EarthQuakeRec** in that result data.

Each new marker is created by creating a **MarkerOptions** object and passing it to the Google maps **adMarker** method. As we've seen with other API's the MarkerOptions class uses a **fluent interface**, so the code first creates a new empty MarkerOption object, and then sets the marker's position, by calling the **position** method, and then it tacks on the text that will appear in a pop-up information window when the user touches the marker by calling the **title** method.

In this case the title simply displays the earthquake's magnitude. And after that the code sets the color of the marker by tacking on a call to the **icon** method, passing in the default marker, but setting it's color to reflect the earthquake's magnitude.

Finally the code calls the **massMoveCamera** method to center the map at a particular location. To keep this example simple, I precomputed the map center, but it would be better and more robust to compute the center based on the actual data returned by the webservice.

That's all for our lesson on location and maps. Please join me next time when we'll discuss data management.

Week 7 - Sensors

Handheld devices allow for context-aware computing, which means that applications can respond or behave differently based on contextual factors, such as the location where they're being used, how they're being held by their user, how much ambient light there is, or how fast the user is traveling. To do this, applications read information from the sensors that come built into today's hand-held devices.

In this lesson, I'll talk about the sensors that Android devices can support and how applications can access these sensors. Next, I'll discuss **SensorEvents**, the class that Android uses to represent sensor readings and **SensorEventListeners**, which are used to transfer information from a sensor, to your application. After that, I'll discuss some common techniques that are used to smooth out or filter sensor values, so that applications can use the values in a variety of different ways. As we go through the lesson, I'll demonstrate several example applications that make use of common sensors.

Sensors are hardware components that measure the physical environment around the device. They come in three flavors:
- sensors that measure **motion**, e.g. how fast the device is moving
- sensors that measure the **position** of the device, e.g. where you are in the world or the **orientation** of the device
- sensors that measure the local environment, e.g. **illumination, air pressure, or humidity**

For example, my device has a **three-axis accelerometer**, which measures the forces exerted on the device, such as when I shake it. It also has a **3-axis magnetic field sensor**, which can be used to measure it's position or orientation relative to the earth's magnetic field - we'll see it in action later in one of the example applications. Lastly, my device has a **barometer** that measures atmospheric pressure.

SensorManager & Sensor classes

For an application to use a sensor, it needs a reference to the **SensorManager**, which is the system service that manages sensors. To get a reference to the SensorManager call the **getSystemService** method, passing in the value Context.SENSOR_SERVICE.

To access a specific sensor, use the **SensorManager's getDefaultSensor** method, passing in a constant corresponding to the desired sensor. Some of those sensor type constants include:
- Sensor.TYPE_ACCELEROMETER for the accelerometer
- Sensor.TYPE_MAGNETIC_FIELD for the magnetic field sensor
- Sensor.TYPE_PRESSURE for the barometer

SensorEvent & SensorEventListener

If an application wants to receive information from a sensor, it must implement a **SensorEventListener,** an interface that defines callback methods that are invoked when either a sensor's accuracy changes or the sensor acquires a new reading.

When a sensor's accuracy changes, Android calls the **onAccuracyChanged** method, passing the sensor that changed and it's new accuracy. When a sensor has a new reading, the **onSensorChanged** method is called, passing in the SensorEvent corresponding to the new reading.

Before you can receive SensorEvents however you must register a SensorEventListener, and when you're done with the Sensor, you'll must unregister the Sensor and the SensorEventListener to avoid wasting battery power.

To register a SensorEventListener for a given sensor you call the **registerListener** method, passing in the SensorEventListener that will be called back to for the sensor you want to listen too and the rate at which you want the sensor to be polled.

To unregister a listener for all sensors with which it's registered you can, for instance, call the **unregisterListener** method passing in the sensorEventListener, and passing in a bitmask indicating the sensors you no longer want to listen to.

Sensor readings are represented as instances of the **sensorEvent class**. The data this class holds, depends on the specific kind of sensor but will include:
• sensor type
• time-stamp
• accuracy of the reading
• measurement data for the new reading

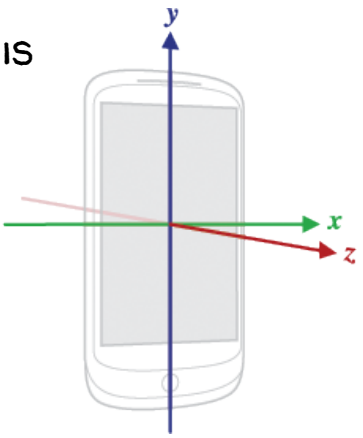
To make sense of the data, you'll need to know how measurements are interpreted for the specific sensor. For instance, many sensors use a 3-Dl coordinate system. When the default orientation is portrait and when the device is lying flat face-up on a table, the axes of the coordinate system are as shown here. The positive X-axis runs from to right to left, the positive Y-axis runs from the top to the bottom of the device, and the positive Z-axis runs perpendicularly upward from the screen of the device.

The coordinate system is oriented relative to the device, it moves and rotates with the device.

Sensor Coordinate System

When default orientation is portrait & the device is lying flat, face-up on a table, axes run

- X – right to left
- Y – top to bottom
- Z – Down to up



Our first example application for this lesson is called **SensorShowValues**. This application displays the raw values that it receives from the device's accelerometer. As you can see, this application displays three text views, with a number in each, corresponding to the X, Y, and Z values being read from this device's accelerometer
.
The greatest force is now being exerted on the Y axis, but I'm failing to hold the device perfectly straight up and down, and my hand shakes a bit and so the numbers will dance around a bit. I will now rotate the device counter-clockwise 90 degrees, around the Z-axis. Now the greatest force is being exerted on the X-axis. Let's rotate the device another 90 degrees, and now you see again, that the greatest force is being exerted on the Y-axis. But this time that force is negative, and that's because the y axis is now upside down. And finally, i'll rotate the device another 90 degrees, and again you'll see that the greatest force is exerted on the X-axis, and that that force is operating in the negative direction.

Let's look at the source code for this application's main activity.

```
package course.examples.Sensors.ShowValues;

import android.app.Activity;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.widget.TextView;

public class SensorRawAccelerometerActivity extends Activity
implements
           SensorEventListener {

     private static final int UPDATE_THRESHOLD = 500;
     private SensorManager mSensorManager;
     private Sensor mAccelerometer;

     private TextView mXValueView, mYValueView, mZValueView;
     private long mLastUpdate;
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);

      mXValueView = (TextView) findViewById(R.id.x_value_view);
      mYValueView = (TextView) findViewById(R.id.y_value_view);
      mZValueView = (TextView) findViewById(R.id.z_value_view);

      // Get reference to SensorManager
      mSensorManager = (SensorManager) getSystemService
      (SENSOR_SERVICE);

      // Get reference to Accelerometer
      if (null == (mAccelerometer = mSensorManager
                  .getDefaultSensor(Sensor.TYPE_ACCELEROMETER)))
            finish();

}

// Register listener
@Override
protected void onResume() {
      super.onResume();

      mSensorManager.registerListener(this, mAccelerometer,
                  SensorManager.SENSOR_DELAY_UI);

      mLastUpdate = System.currentTimeMillis();

}

// Unregister listener
@Override
protected void onPause() {
      mSensorManager.unregisterListener(this);
      super.onPause();
}

// Process new reading
@Override
public void onSensorChanged(SensorEvent event) {

      if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {

            long actualTime = System.currentTimeMillis();

            if (actualTime - mLastUpdate > UPDATE_THRESHOLD) {

                  mLastUpdate = actualTime;

                  float x = event.values[0], y = event.values[1],
                        z = event.values[2];
```

```
                        mXValueView.setText(String.valueOf(x));
                        mYValueView.setText(String.valueOf(y));
                        mZValueView.setText(String.valueOf(z));

                  }
            }
      }

      @Override
      public void onAccuracyChanged(Sensor sensor, int accuracy) {
            // N/A
      }
}
```

Notice that this class implements the **SensorEventListener** interface so we can receive callbacks from the **SensorManager**. In **onCreate** we get the reference to SensorManager. Next, we get a reference to the device's accelerometer by calling **SensorManager.getDefaultSensor**, passing in the type Sensor.TYPE_ACCELEROMETER.

In the **onResume** method, the application registers this class as a listener for accelerometer event, by calling the **registerListener** method. The last parameter, **SensorManager.sensor_delay_UI**, corresponds to a relatively low-frequency holding rate.

Next, the **onPause** method unregisters this class as a listener for any and all sensors to which it may be listening.

Scrolling down, we now come to the **onSensorChanged** method. This method first checks to make sure that this event is an accelerometer reading. Next, it checks that a certain amount of time has passed since the last reading was displayed. And if so, the code records the accelerometer's x, y and z values, and then it displays those values on the screen.

Filtering sensor values

In the example application that we just looked at, I tried to hold the device perfectly straight up. And if I'd been able to do that, the accelerometer would ideally have reported values around x equals 0 meters per second squared, y equals 9.81 meters per second squared and Z equals 0 meters per second squared. But as you saw in the example application, the accelerometers values fluctuated.

All applications will experience this kind of thing due to natural user movement, non flat surfaces, electrical noise and so forth. When creating sensor-enabled applications, developers will often apply transforms to the raw data to smooth it out. Two common kinds of transforms are called **low-pass filters** and **high-pass filters**.

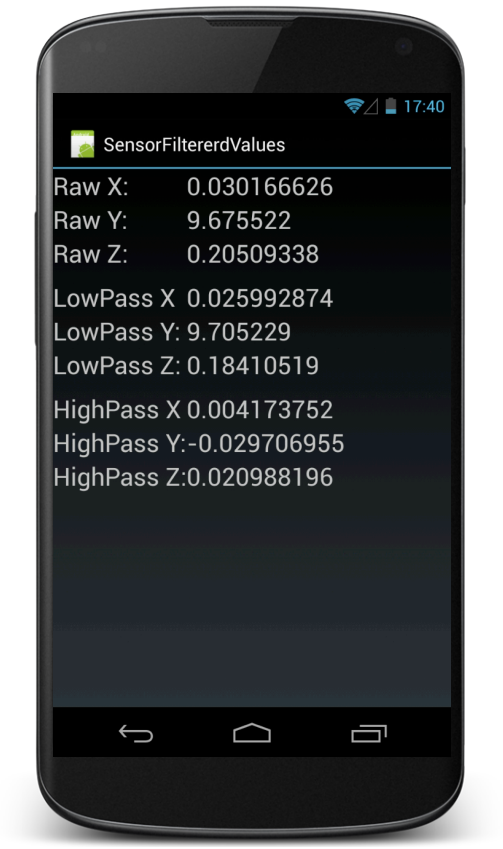
Low-pass filters are used to deemphasize small transient force changes while emphasizing the long-term constant forces. You might use a low-pass filter when your application needs to pay attention to the constant force of gravity for example, and you don't want to be affected just because your hands shakes a little. A real life example of this would be something like a carpenter's level. The bubble needs to move based on gravity, not based on small hand twitches.

In contrast, you use a high-pass filter when you want to emphasize the transient force changes, and you want to deemphasize the constant force components. You might use a high-pass filter when your application should ignore the constant force of gravity for example, but should respond to the specific moves that the user makes. A real life example of this might be a percussion instrument like a set or maracas. You don't really care about gravity here, you care about how the user is shaking the instrument.

The next application is called **SensorFilteredAccelerometer**. This application applies both low pass and high pass filters to the raw accelerometer values and displays the filtered values.

I'll start up the SensorFilteredAccelerometer application. As you can see, this application displays 9 text views with numbers in them. These numbers correspond to the x, y, and z values being read from the device's accelerometer. The raw values, after applying a low pass filter and those raw values after applying a high pass filter.

If we let the application run for a while, we'll see that the low pass values begin to approximate our ideal accelerometer readings roughly 0 for the x and z axes, and roughly 9.81 for the y axis. At the same time, you can see that the high-pass values all tend toward 0. If I rotate the device counterclockwise, you see the high pass x value go positive. And if I rotate the device clockwise, you'll see the high pass x value go negative.



Let's look at the source code for this application' main activity.

```
package course.examples.Sensors.ShowValues;

import android.app.Activity;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.widget.TextView;

public class SensorFilteredValuesActivity extends Activity implements
          SensorEventListener {

     // References to SensorManager and accelerometer

     private SensorManager mSensorManager;
     private Sensor mAccelerometer;

     // Filtering constant
```

```
private final float mAlpha = 0.8f;

// Arrays for storing filtered values
private float[] mGravity = new float[3];
private float[] mAccel = new float[3];

private TextView mXValueView, mYValueView, mZValueView,
                     mXGravityView, mYGravityView,
                     mZGravityView, mXAccelView, mYAccelView,
                     mZAccelView;

private long mLastUpdate;

@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      setContentView(R.layout.main);

      mXValueView = (TextView) findViewById(R.id.x_value_view);
      mYValueView = (TextView) findViewById(R.id.y_value_view);
      mZValueView = (TextView) findViewById(R.id.z_value_view);

      mXGravityView = (TextView) findViewById
            (R.id.x_lowpass_view);
      mYGravityView = (TextView) findViewById
            (R.id.y_lowpass_view);
      mZGravityView = (TextView) findViewById
            (R.id.z_lowpass_view);

      mXAccelView = (TextView) findViewById
            (R.id.x_highpass_view);
      mYAccelView = (TextView) findViewById
            (R.id.y_highpass_view);
      mZAccelView = (TextView) findViewById
            (R.id.z_highpass_view);

      // Get reference to SensorManager
      mSensorManager = (SensorManager) getSystemService
            (SENSOR_SERVICE);

      // Get reference to Accelerometer
      if (null == (mAccelerometer = mSensorManager
                  .getDefaultSensor(Sensor.TYPE_ACCELEROMETER)))
            finish();

      mLastUpdate = System.currentTimeMillis();
}

// Register listener
@Override
protected void onResume() {
      super.onResume();
```

```
      mSensorManager.registerListener(this, mAccelerometer,
            SensorManager.SENSOR_DELAY_UI);

}

// Unregister listener
@Override
protected void onPause() {
      super.onPause();

      mSensorManager.unregisterListener(this);
}

// Process new reading
@Override
public void onSensorChanged(SensorEvent event) {
      if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
            long actualTime = System.currentTimeMillis();
            if (actualTime - mLastUpdate > 500) {
                  mLastUpdate = actualTime;
                  float rawX = event.values[0];
                  float rawY = event.values[1];
                  float rawZ = event.values[2];

                  // Apply low-pass filter
                  mGravity[0] = lowPass(rawX, mGravity[0]);
                  mGravity[1] = lowPass(rawY, mGravity[1]);
                  mGravity[2] = lowPass(rawZ, mGravity[2]);
                  // Apply high-pass filter
                  mAccel[0] = highPass(rawX, mGravity[0]);
                  mAccel[1] = highPass(rawY, mGravity[1]);
                  mAccel[2] = highPass(rawZ, mGravity[2]);

                  mXValueView.setText(String.valueOf(rawX));
                  mYValueView.setText(String.valueOf(rawY));
                  mZValueView.setText(String.valueOf(rawZ));

                  mXGravityView.setText(String.valueOf(mGravity
                        [0]));
                  mYGravityView.setText(String.valueOf(mGravity
                        [1]));
                  mZGravityView.setText(String.valueOf(mGravity
                        [2]));

                  mXAccelView.setText(String.valueOf(mAccel[0]));
                  mYAccelView.setText(String.valueOf(mAccel[1]));
                  mZAccelView.setText(String.valueOf(mAccel[2]));
            }
      }
}

// Deemphasize transient forces
private float lowPass(float current, float gravity) {
      return gravity * mAlpha + current * (1 - mAlpha);
```

```
	}

	// Deemphasize constant forces
	private float highPass(float current, float gravity) {
		return current - gravity;
	}

	@Override
	public void onAccuracyChanged(Sensor sensor, int accuracy) {
		// NA
	}
}
```

Notice again that this class implements the SensorEventListener interface, so it can receive callbacks from the sensor manager.

In **onCreate** the application gets a reference to the SensorManager. Next, it gets a reference to the device's accelerometer by calling SensorManager.getDefaultSensor passing in the type constant that corresponds to the accelerometer.

In **onResume** the application registers this class as a listener for accelerometer events by calling the registerListener method.

Next, **onPause** unregisters this class as a listener for any sensors to which it may be listening.

Scrolling down, we now come to the **onSensorChanged** method. As before, this method first checks to make sure that this event is an accelerometer reading. And then it checks that a certain amount of time has passed since the last reading was displayed. If it has, the code records the accelerometer's X, Y and Z values, and then applies the low pass filter to each of the raw values, after which the code applies the high-pass filter to each of the raw values.

Let's look at the code for the filters. Here's the **lowPass** method, which computes the low-pass filtered values. This method takes 2 parameters: the current reading and the long term average. It then computes the filter value, as, as a kind of weighted average. In this case, the filtered value equals 80% of the long term average plus 20% of the current reading. Over time, this calculation moves towards the ideal values that we talked about earlier.

Scrolling down, here's the highPass method which computes the high-pass filtered values. And this method also takes 2 parameters - the current reading, and the long term average which is actually computed by the low pass method, that we just talked about. This code then subtracts the long-term average from the current reading and therefore represents the part of the reading that is not due to gravity.

The next example application is called **SensorCompass** and uses the device's accelerometer and magnetometer to orient a compass arrow towards magnetic north.

I'll start up the SensorCompass application. As you can see, it displays a green circle with a white arrow. Right now, this arrow points towards magnetic north. However if I begin to rotate the device, you see that the arrow continues to point towards the north which of course is exactly what a compass should do.

Let's look at the source code for this application's main activity:

```
package course.examples.compass;

import android.app.Activity;
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.hardware.Sensor;
import android.hardware.SensorEvent;
import android.hardware.SensorEventListener;
import android.hardware.SensorManager;
import android.os.Bundle;
import android.view.View;
import android.widget.RelativeLayout;
```

```
public class CompassActivity extends Activity implements
SensorEventListener {

      @SuppressWarnings("unused")
      private String TAG = "SensorCompass";

      // Main View
      private RelativeLayout mFrame;

      // Sensors & SensorManager
      private Sensor accelerometer;
      private Sensor magnetometer;
      private SensorManager mSensorManager;

      // Storage for Sensor readings
      private float[] mGravity = null;
      private float[] mGeomagnetic = null;

      // Rotation around the Z axis
      private double mRotationInDegress;

      // View showing the compass arrow
      private CompassArrowView mCompassArrow;

      @Override
      protected void onCreate(Bundle savedInstanceState) {

            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);

            mFrame = (RelativeLayout) findViewById(R.id.frame);

            mCompassArrow = new CompassArrowView(getApplicationContext
());

            mFrame.addView(mCompassArrow);

            // Get a reference to the SensorManager
            mSensorManager = (SensorManager) getSystemService
                  (SENSOR_SERVICE);

            // Get a reference to the accelerometer
            accelerometer = mSensorManager
                  .getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

            // Get a reference to the magnetometer
            magnetometer = mSensorManager
                        .getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);

            // Exit unless both sensors are available
            if (null == accelerometer || null == magnetometer)
                  finish();
```

```
}

@Override
protected void onResume() {
      super.onResume();


      // Register for sensor updates

      mSensorManager.registerListener(this, accelerometer,
            SensorManager.SENSOR_DELAY_NORMAL);

      mSensorManager.registerListener(this, magnetometer,
            SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
protected void onPause() {
      super.onPause();

      // Unregister all sensors
      mSensorManager.unregisterListener(this);

}

@Override
public void onSensorChanged(SensorEvent event) {

      // Acquire accelerometer event data

      if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {

            mGravity = new float[3];
            System.arraycopy(event.values, 0, mGravity, 0, 3);

      }

      // Acquire magnetometer event data

      else if (event.sensor.getType() ==
            Sensor.TYPE_MAGNETIC_FIELD) {

            mGeomagnetic = new float[3];
            System.arraycopy(event.values, 0, mGeomagnetic, 0, 3);

      }

      // If we have readings from both sensors then
      // use the readings to compute the device's orientation
      // and then update the display.

      if (mGravity != null && mGeomagnetic != null) {

            float rotationMatrix[] = new float[9];
```

```
            // Users the accelerometer and magnetometer readings
            // to compute the device's rotation with respect to
            // a real world coordinate system

            boolean success = SensorManager.getRotationMatrix
                  (rotationMatrix,null, mGravity, mGeomagnetic);

            if (success) {

                  float orientationMatrix[] = new float[3];

                  // Returns the device's orientation given
                  // the rotationMatrix

                  SensorManager.getOrientation(rotationMatrix,
                        orientationMatrix);

                  // Get the rotation, measured in radians, around
                  // the Z-axis. Note: This assumes the device is
                  // held flat and parallel to the ground

                  float rotationInRadians = orientationMatrix[0];

                  // Convert from radians to degrees
                  mRotationInDegress = Math.toDegrees
                        (rotationInRadians);

                  // Request redraw
                  mCompassArrow.invalidate();

                  // Reset sensor event data arrays
                  mGravity = mGeomagnetic = null;

            }
      }

}

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
      // N/A
}

public class CompassArrowView extends View {

      Bitmap mBitmap = BitmapFactory.decodeResource(getResources
            (), R.drawable.arrow);
      int mBitmapWidth = mBitmap.getWidth();

      // Height and Width of Main View
      int mParentWidth;
      int mParentHeight;
```

```
            // Center of Main View
            int mParentCenterX;
            int mParentCenterY;

            // Top left position of this View
            int mViewTopX;
            int mViewLeftY;

            public CompassArrowView(Context context) {
                  super(context);
            };

            // Compute location of compass arrow
            @Override
            protected void onSizeChanged(int w, int h, int oldw,
            int oldh) {
                  mParentWidth = mFrame.getWidth();
                  mParentHeight = mFrame.getHeight();

                  mParentCenterX = mParentWidth / 2;
                  mParentCenterY = mParentHeight / 2;

                  mViewLeftY = mParentCenterX - mBitmapWidth / 2;
                  mViewTopX = mParentCenterY - mBitmapWidth / 2;
            }

            // Redraw the compass arrow
            @Override
            protected void onDraw(Canvas canvas) {

                  // Save the canvas
                  canvas.save();

                  // Rotate this View
                  canvas.rotate((float) -mRotationInDegress,
                        mParentCenterX, mParentCenterY);

                  // Redraw this View
                  canvas.drawBitmap(mBitmap, mViewLeftY, mViewTopX,
                        null);

                  // Restore the canvas
                  canvas.restore();

            }
      }
}
```

Lets scroll to the **onCreate** method. As with the other applications, this one begins by setting up the user interface, and in particular, it creates a custom view that holds the compass arrow and then it adds that view to the activity's main view. It then gets a reference to the sensor manager. After that, it gets a reference to the device's

accelerometer and it gets a reference to the device's magnetometer by calling **SensorManager.getdefaultsensor** and passing in the appropriate type constants.

In the **onResume** method, the code registers this class a listener for accelerometer events and for magnetometer events, by calling the **registerListener** method.

The **onPause** method unregisters this class as a listener for all sensors.

The **onSensorChange** method processes the incoming sensor events. First it deteimines whether the event is an accelerometer or a magnetometer event and then copies the appropriate event data. Next, if there are readings from each of the 2 sensors, the code calls the **SensorManager.getRotationMatrix** method passing in the sensor readings and an array in which to store the rotation matrix.

If that method iss successful, the code calls the **SensorManager.getOrientation** method passing in the rotation matrix that we just acquired from the call to get rotation matrix. It also passes in another array called **orientationMatrix**. When this method returns, orientation matrix will hold the information the application needs to determine how the device is oriented with respect to the earth's magnetic north.

The code then grabs the result value from the orientationMatrix and, since this value is measured in radians, the code then converts the radian value to degrees. Then, the code invalidates the **compassArrowView** and clears the arrays that holds the sensor readings.

Let's look at the **compassArrowView** to see how it uses the new orientation information. Scrolling down to the **onDraw** method, the code first saves the current canvas and then rotates this view on the canvas by an amount equal to (-1)*mRotation in degrees. So basically, the idea here is that if the device is pointing say 90 degrees away from north, then the compass arrow must rotate back 90 degrees in order for the compass arrow to keep pointing north.

That's all for this lesson on sensors. See you next time when we'll talk about location and maps.

Week 8 - Data Management

Handheld systems can generate and manipulate large amounts of data. Android provides a number of support classes that allow you to manage data across multiple application sessions. In today's lesson, I'll talk about several of these support classes. I'll begin by talking about the **SharedPreferences** class, which allows applications to store and manage small amounts of primitive data. Next, I'll talk about writing files to both **internal and external storage**. Last, I'll discuss the creation and use of complex **SQLite databases**.

Your applications will typically use the **SharedPreferences** class when you want to store small amounts of primitive data, such as a user name. Your applications will typically use **internal device storage**, when you need to manage small to medium amounts of data that should remain private to the application, such as temporary files that are used by the application. Your applications will typically use **external storage** when you want to store larger amounts of non-private data, such as songs or video files. And your application will typically use **databases** when you intend to store small to large to amounts of private structured data.

SharedPreferences

SharedPreferences are essentially persistent maps and like any map they hold **key-value pairs of simple data types**, e.g. strings and floats. SharedPreferences are automatically persisted across application sessions, which allows a user to create information, exit the application and restart it later, getting access to the information they created earlier. SharedPreferences are often used for long-term storage of customizable application data, such as a username, favorite WiFi networks, specific user options or preferences.

To associate a SharedPreferences object with an activity, you can use the **Activity.getPreferences** method, passing in an access mode as a parameter. For example, this mode can be mode_private, indicating that the data is private to the calling application:

```
Activity.getPreferences (int mode)
mode = MODE_PRIVATE
```

If you want a SharedPreferences object that is not associated with a specific activity, then you can use the context.getSharedPreferences method, to retrieve a named SharedPreferences object:

```
•Context.getSharedPreferences (String name, int mode)
name = name of the SharedPreferences file
mode = MODE_PRIVATE
```

With the above method, you pass in a name for the SharedPreferences object and an access mode, such as the mode_private that we saw earlier.

Once you've acquired a SharedPreferences object, you can edit that object:
• **`SharedPreferences.edit()`** - returns an instance of **`SharedPreferences.Editor`**

You can then add or change the values of the SharedPreferences object:
• **`SharedPreferences.Editor.putInt(String key, int value)`**
• **`SharedPreferences.Editor.putString(String key, String value)`**
• **`SharedPreferences.Editor.remove(String key)`**

After making the desired changes, you can make them permanent by calling the method:
• **`SharedPreferences.Editor.commit()`**

At this point, the SharedPreferences object is saved, and an application can exit secure that the data can be retrieved during a later session.

To read these values later, an application can get a SharedPreferences object and use various methods to read out the stored values:

• **`getAll()`** - get the SharedPreferences values
• **`getBoolean(String key, ...)`** - get a particular Boolean value
• **`getString(String key, ...)`** - get a particular String value

The first example application is called **DataManagementSharedPreferences**. It has a button labeled Play, which when presses displays a random number. The application keeps track of the highest number seen so far, and saves that number across different user sessions.

When this application starts it shows that the high score is currently zero. When I press the Play button a new number is displayed in the center of the screen. When the number is higher than the previous high score, the high score display is updated to show the new number.

Now, remember this high score. I'll quit the application and restart it: observe the current high score is the same as it was the last time we ran the application. Now let me hit the Play button a few more times to increase the high score. I'll stop the application again and restart it ... again the application displays the high score from the last use of the application. I can also hit the Reset button to clear the high score.

Let's look at the source code for this application's main activity:

```
package course.examples.DataManagement.SharedPreferences;

import java.util.Random;

import android.app.Activity;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;

public class SharedPreferenceReadWriteActivity extends Activity {
      private static String HIGH_SCORE = "high_score";

      @Override
      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            final SharedPreferences prefs = getPreferences
                  (MODE_PRIVATE);
```

```
setContentView(R.layout.main);

// High Score
final TextView highScore = (TextView) findViewById
      (R.id.high_score_text);
highScore.setText(String.valueOf(prefs.getInt(
      HIGH_SCORE, 0)));

//Game Score
final TextView gameScore = (TextView) findViewById
      (R.id.game_score_text);
gameScore.setText(String.valueOf("0"));

// Play Button
final Button playButton = (Button) findViewById
      (R.id.play_button);
playButton.setOnClickListener(new OnClickListener() {

      @Override
      public void onClick(View v) {

            Random r = new Random();
            int val = r.nextInt(1000);
            gameScore.setText(String.valueOf(val));

            // Get Stored High Score
            if (val > prefs.getInt(HIGH_SCORE, 0)) {

                  // Get and edit high score
                  SharedPreferences.Editor editor =
                        prefs.edit();
                  editor.putInt(HIGH_SCORE, val);
                  editor.commit();

                  highScore.setText(String.valueOf(val));

            }
      }
});

// Reset Button
final Button resetButton = (Button) findViewById
      (R.id.reset_button);
resetButton.setOnClickListener(new OnClickListener() {

      @Override
      public void onClick(View v) {

            // Set high score to 0
            SharedPreferences.Editor editor = prefs.edit();
                  editor.putInt(HIGH_SCORE, 0);
                  editor.commit();
```

```
                        highScore.setText(String.valueOf("0"));
                        gameScore.setText(String.valueOf("0"));
                    }
            });
        }
}
```

In **onCreate,** the code acquires the **SharedPreferences** object for this activity. Next, when the user clicks the Play button, the code generates a new score, which it stores in a variable called Val. After that, the code calls **getInt** on the Preferences object, to retrieve the current high score.

If Val is greater than the current high score, then we need to update the high score. So the code calls the **edit** method on the Preferences object, which returns a **SharedPreferences.Editor** object.

Next, the code calls **putInt** on the editor object to update the high score, to the current value. And finally, the code calls **commit** on the editor, to save the current high score.

User Preferences

The Shared Preferences class is often used to store an application's user preferences. Android provides a **PreferenceFragment** class to display and modify user preferences.

This next example application, **DataManagementPreferenceActivity**, does just this. In this case, the preference is the name the application uses when addressing the user. Let's take a look.

When the application starts up it presents a single button, labeled View User Name, that presents the Preference Fragment, letting me view and change my current user name. As you can see, my User Name has not yet been set, so I'll click on this area. A dialog box pops up, asking me to enter my User Name. I'll enter the new name and click on the Submit. The dialog box closes, and my new User Name is displayed.

Now I'll close the application, and restart it - observe that my User Name is unchanged - the information persisted across user sessions.

Let's see source code for the application's **ViewAndUpdatePreferencesActivity**, which is the one that starts when the user clicks the View User Name button.

```
package course.examples.DataManagement.PreferenceActivity;

import android.app.Activity;
import android.content.SharedPreferences;
import
android.content.SharedPreferences.OnSharedPreferenceChangeListener;
import android.os.Bundle;
import android.preference.Preference;
import android.preference.PreferenceFragment;

public class ViewAndUpdatePreferencesActivity extends Activity {

      private static final String USERNAME = "uname";

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);

            setContentView(R.layout.user_prefs_fragment);
      }

      // Fragment that displays the username preference
      public static class UserPreferenceFragment extends
            PreferenceFragment {

            protected static final String TAG = "UserPrefsFragment";
            private OnSharedPreferenceChangeListener mListener;
            private Preference mUserNamePreference;

            @Override
            public void onCreate(Bundle savedInstanceState) {
                  super.onCreate(savedInstanceState);

                  // Load the preferences from an XML resource
                  addPreferencesFromResource(R.xml.user_prefs);
```

```
                // Get the username Preference
                mUserNamePreference = (Preference)
                      getPreferenceManager().findPreference(USERNAME);

                // Attach a listener to update summary when username
                // changes
                mListener = new OnSharedPreferenceChangeListener() {
                      @Override
                      public void onSharedPreferenceChanged(
                            SharedPreferences sharedPreferences, String
                                  key) {
                            mUserNamePreference.setSummary
                                  (sharedPreferences.getString(
                                  USERNAME, "None Set"));
                      }
                };

                // Get SharedPreferences object managed by the
                // PreferenceManager for this Fragment
                SharedPreferences prefs = getPreferenceManager()
                      .getSharedPreferences();

                // Register a listener on the SharedPreferences object
                prefs.registerOnSharedPreferenceChangeListener
                      (mListener);

                // Invoke callback manually to display the current
                // username
                mListener.onSharedPreferenceChanged(prefs, USERNAME);
          }
     }
}
```

The onCreate method first calls **setContentView**, passing in an XML file called **User_prefs_fragment.xml**. And this layout file instantiates and displays an instance of the **UserPreferenceFragment** class, which is defined further down in this same file. This class's **onCreate** method first calls the **AddPreferencesFromResource** method, passing in the **user_preps.xml** file from the res/xml directory.

Let's open that file:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/
android"
    android:key="@+id/pref_screen" >

    <EditTextPreference
        android:dialogMessage="Enter Your User Name"
        android:dialogTitle="Change User Name"
        android:key="uname"
        android:negativeButtonText="Cancel"
```

```
        android:positiveButtonText="Submit"
        android:title="User Name"
        >
    </EditTextPreference>
</PreferenceScreen>
```

As you can see, this file defines a preference screen resource. This preference screen, contains one preference that's displayed in an edit text box with the key “u_name”, and i title “User Name”.

When the user clicks on the EditTextBox to change the User Name, a dialog box pops up with the title “Change User Name”, a message saying “Enter your User Name”, and two buttons labelled “Cancel” and “Submit”.

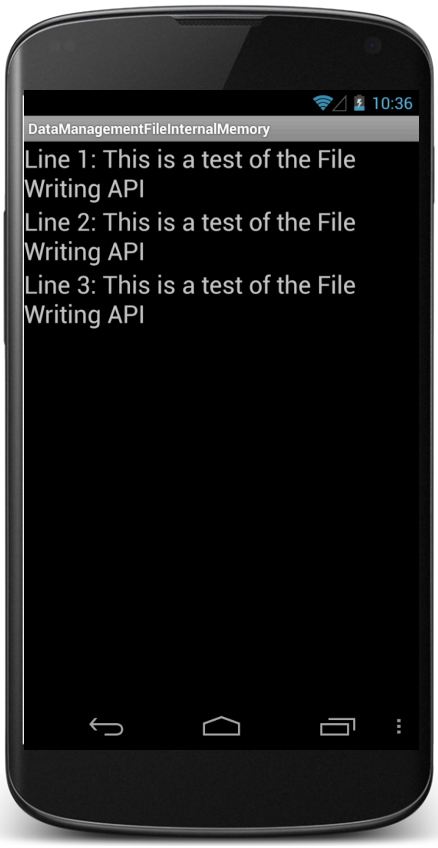
Internal Storage

Android also supports the use of files. **File** is a class that represents a file system entity identified by a **path name**. In Android, storage areas are classified as being either **internal** or **externa**l. Historically, this distinguished between the internal flash memory on a device, and the removable external memory cards attached to the device. Today, not all external memory is removable. Internal memory, is usually used for smaller data sets that are private to an application. External memory, is usually reserved for larger non-private data sets, such as music files and pictures.

The next example applications use files to store information. One of the methods they use are **OpenFileOutput**, which opens a private file for **writing**. This method will also create a physical file if it doesn't already exist. Another method is **OpenFileInput**, which opens a private file for **reading**. There are, of course, many other file related methods, so please look at the documentation for more information.

The next example application is called **DataManagementFileInternalMemory.** When it starts it checks whether a particular text file exists. If it does not the application creates that file and writes some text into it. Then the application opens that same file, reads the text from it, and displays it. Let's take a look at this application:

As you can see, the application has displayed some text on the screen.



Let's look at the source code for this application to find out where that text came from:

```
package course.examples.DataManagement.FileInternal;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.LinearLayout;
import android.widget.TextView;
import course.examples.Files.FileWriteAndRead.R;

public class InternalFileWriteReadActivity extends Activity {
```

```
private final static String fileName = "TestFile.txt";
private String TAG = "InternalFileWriteReadActivity";

@Override
public void onCreate(Bundle savedInstanceState) {

      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      LinearLayout ll = (LinearLayout) findViewById
            (R.id.linearLayout);

      // Check whether fileName already exists in directory used
      // by the openFileOutput() method.
      // If the text file doesn't exist, then create it now

      if (!getFileStreamPath(fileName).exists()) {
            try {
                  writeFile();
            } catch (FileNotFoundException e) {
                  Log.i(TAG, "FileNotFoundException");
            }
      }

      // Read the data from the text file and display it
      try {
            readFile(ll);
      } catch (IOException e) {
            Log.i(TAG, "IOException");
      }
}

private void writeFile() throws FileNotFoundException {

      FileOutputStream fos = openFileOutput(fileName,
            MODE_PRIVATE);
      PrintWriter pw = new PrintWriter(new BufferedWriter(
                  new OutputStreamWriter(fos)));
      pw.println(
            "Line 1: This is a test of the File Writing API");
      pw.println(
            "Line 2: This is a test of the File Writing API");
      pw.println(
            "Line 3: This is a test of the File Writing API");
      pw.close();

}

private void readFile(LinearLayout ll) throws IOException {
      FileInputStream fis = openFileInput(fileName);
      BufferedReader br = new BufferedReader(
            new InputStreamReader(fis));
      String line = "";
```

```
            while (null != (line = br.readLine())) {
                  TextView tv = new TextView(this);
                  tv.setTextSize(24);
                  tv.setText(line);
                  ll.addView(tv);
            }
            br.close();
      }
}
```

Here's the application's main activity. In the **onCreate method,** the code first gets the file path string associated with the file name, **TestFile.txt**. If that file does not exist the WriteFile method is called. Let's scroll down and look at that method:

This method first calls the **OpenFileOutput** method, which returns a file output stream. The code then writes three lines of text to the text file and than closes the file.

Let's scroll back up to the **onCreate method**. The code continues by calling the **readFile** method, passing in a LinearLayout for displaying the text. The readFile method now opens the text file for input and reads lines of text from the file. Each line is placed in a TextView, which is added to the linear layout.

External Storage

Android also allows applications to write to external memory. When you do this, however, you must consider the additional twist that external memory can be removable, e.g. an SD card. Removable media can appear or disappear without warning so, before you write to external memory you first need to determine its state.

Now one way to do this is by using a method of the **Environment** class: **getExternalStorageState**(), which returns a **String** indicating the current state of the device's external memory. Some of those values are shown here:

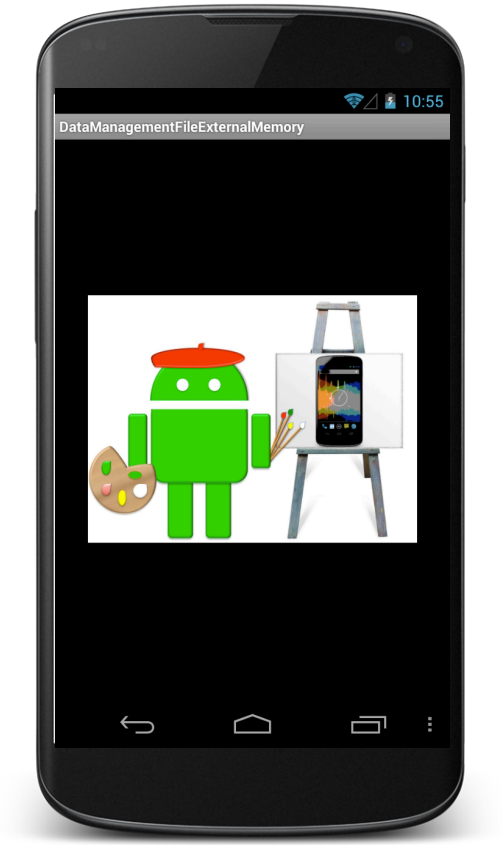
```
String Environment.getExternalStorageState()
```

- MEDIA_MOUNTED - present & mounted with read/write access
- MEDIA_MOUNTED_READ_ONLY - present & mounted with read-only access
- MEDIA_REMOVED - not present
- etc.

If your application wishes to write to external memory it will need to request the WRITE_EXTERNAL_STORAGE permission in the application's AndroidManifest.xml file:

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

The next example application, **DataManagementFileExternalMemory**, reads an image file from the **res/raw directory**, copies the file to external storage, reads the image data back, and displays it on the screen. Let's take a look at that application right now:



When I start the application, an image is displayed on the screen.

Let's look at the source code for the application's main activity. I

```
package course.examples.DataManagement.FileExternal;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import android.app.Activity;
import android.net.Uri;
import android.os.Bundle;
```

```
import android.os.Environment;
import android.util.Log;
import android.widget.ImageView;
import course.examples.Files.FileWriteAndRead.R;

public class ExternalFileWriteReadActivity extends Activity {
      private final String fileName = "painter.png";
      private String TAG = "ExternalFileWriteReadActivity";

      @Override
      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.main);

            if (Environment.MEDIA_MOUNTED.equals(Environment
                        .getExternalStorageState())) {

                  File outFile = new File(getExternalFilesDir
                        (Environment.DIRECTORY_PICTURES),
                        fileName);

                  if (!outFile.exists())
                        copyImageToMemory(outFile);

                  ImageView imageview = (ImageView) findViewById
                        (R.id.image);
                  imageview.setImageURI(Uri.parse("file://" +
                        outFile.getAbsolutePath()));
            }
      }

      private void copyImageToMemory(File outFile) {
            try {

                  BufferedOutputStream os = new BufferedOutputStream(
                        new FileOutputStream(outFile));
                  BufferedInputStream is = new BufferedInputStream(
                        getResources().openRawResource(
                        R.raw.painter));
                  copy(is, os);

            } catch (FileNotFoundException e) {
                  Log.e(TAG, "FileNotFoundException");
            }
      }

      private void copy(InputStream is, OutputStream os) {
            final byte[] buf = new byte[1024];
            int numBytes;
            try {
                  while (-1 != (numBytes = is.read(buf))) {
                        os.write(buf, 0, numBytes);
                  }
            } catch (IOException e) {
```

```
                e.printStackTrace();
            } finally {
                try {
                    is.close();
                    os.close();
                } catch (IOException e) {
                    Log.e(TAG, "IOException");
                }
            }
        }
}
```

In the **onCreate** method, the code checks the external storage state to ensure that the media is mounted, and that it is readable and writable. Then the code gets the external file directory where pictures are normally stored and constructs a new file object pointing to that directory. Next, it checks whether the file actually exists on the external memory.

The code then calls the **copyImageDataToMemory** method. Let's look at that method.

This method starts by creating a new output stream, that will be used to write to a file on the external memory. Then, it creates an input stream so it can read image data from the res/raw directory. And finally, it copies the data from the input stream to the output stream. When the method completes there will be a new file in external memory. Back in the onCreate method, the code gets a reference to an image view, and inserts the image URI for the file that was just created.

If your application creates temporary files, then you may consider writing these to cache directories instead. Cache files are temporary files that may be deleted by the system, when storage is low. Also, they are removed when the application is uninstalled.

You can get access to the Cache directory, by using the **getCacheDir** method of the Context class. This method returns the absolute path to an application-specific directory that can be used for temporary files. You can also use the **getExternalCacheDir** method of the context class, which returns a file object representing a directory for Cache files in external storage.

SQLite Databases

When your application reads and uses larger amounts of structured and complex data, you may want to put that data in a database. Android provides an implementation of SQLite, which allows applications to create and use **in-memory relational databases**.

SQLite is designed to operate within a very small footprint, say, less than 300 kB of storage. SQLIte is however, a full-fledged relational database. It implements most of the

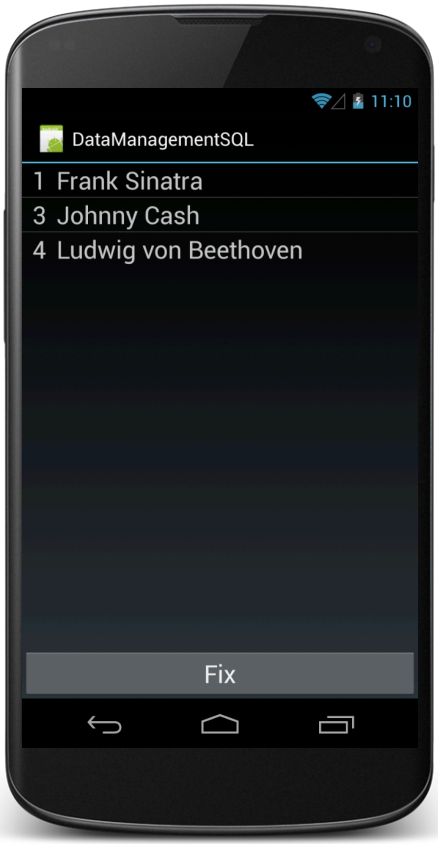
SQL92 standard, and it **supports highly reliable ACID transitions**, which means that it's **transactions are atomic, consistent, isolated, and durable**.

To use an SQL database in your applications, Android recommends that you use a helper class called **SQLiteOpenHelper**, which you can subclass. In the constructor you must call the super class's constructor, passing in some information about the database you want to create. Next, override the **onCreate** method and the **onUpgrade** method. In **onCreate**, you'll execute one or more **CREATE_TABLE** commands, which define the database's structure and layout.

After that, use the **SQLiteOpenHelper** methods to open and return the underlying **SQLite database**, and to execute operations on it.

Our last example application is called **DataManagementSQL**, which creates an SQLite database and then inserts several records into the database, **some with errors**. The application will also display a button labeled **Fix** - when the user presses it the application will delete and update some of those records that were just inserted and then display the updated database records on the screen.

Let's take a look at this application now:



As you can see, it displays four database records, each containing a record ID, and the name of an artist. The fix button is at the bottom of the screen. When I press it record number two, Lady Gaga, will be deleted, and record number three, Johnny Cash, will be updated to correctly spell the artist's first name. Let me hit the Fix button now. As you can see, record number two has been deleted, and record number three now shows the correct spelling of Johnny Cash's first name.

Let's look at the source code for this application's main activity.

```
package course.examples.DataManagement.DataBaseExample;

import android.app.ListActivity;
import android.content.ContentValues;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.SimpleCursorAdapter;

public class DatabaseExampleActivity extends ListActivity {

     private SQLiteDatabase mDB = null;
     private DatabaseOpenHelper mDbHelper;
     private SimpleCursorAdapter mAdapter;

     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);

          // Create a new DatabaseHelper
          mDbHelper = new DatabaseOpenHelper(this);

          // Get the underlying database for writing
          mDB = mDbHelper.getWritableDatabase();

          // start with an empty database
          clearAll();

          // Insert records
          insertArtists();

          // Create a cursor
```

```
        Cursor c = readArtists();
        mAdapter = new SimpleCursorAdapter(this,
             R.layout.list_layout, c,
             DatabaseOpenHelper.columns, new int[] {
                  R.id._id, R.id.name }, 0);

        setListAdapter(mAdapter);

        Button fixButton = (Button) findViewById
             (R.id.fix_button);
        fixButton.setOnClickListener(new OnClickListener() {

             @Override
             public void onClick(View v) {
                  // execute database operations
                  fix();

                  // Redisplay data
                  mAdapter.getCursor().requery();
                  mAdapter.notifyDataSetChanged();
             }
        });

    }

    // Insert several artist records
    private void insertArtists() {
        ContentValues values = new ContentValues();
        values.put(DatabaseOpenHelper.ARTIST_NAME,
             "Frank Sinatra");
        mDB.insert(DatabaseOpenHelper.TABLE_NAME,
             null, values);

        values.clear();
        values.put(DatabaseOpenHelper.ARTIST_NAME,
             "Lady Gaga");
        mDB.insert(DatabaseOpenHelper.TABLE_NAME,
             null, values);

        values.clear();
        values.put(DatabaseOpenHelper.ARTIST_NAME,
             "Jawny Cash");
        mDB.insert(DatabaseOpenHelper.TABLE_NAME,
             null, values);

        values.clear();
```

```
            values.put(DatabaseOpenHelper.ARTIST_NAME,
                  "Ludwig von Beethoven");
            mDB.insert(DatabaseOpenHelper.TABLE_NAME,
                  null, values);
      }

      // Returns all artist records in the database
      private Cursor readArtists() {
            return mDB.query(DatabaseOpenHelper.TABLE_NAME,
                  DatabaseOpenHelper.columns, null,
                  new String[] {}, null, null, null);
      }

      // Modify the contents of the database
      private void fix() {
            // Sorry Lady Gaga :-(
            mDB.delete(DatabaseOpenHelper.TABLE_NAME,
                  DatabaseOpenHelper.ARTIST_NAME + "=?",
                  new String[] { "Lady Gaga" });

            // fix the Man in Black
            ContentValues values = new ContentValues();
                  values.put(DatabaseOpenHelper.ARTIST_NAME,
                  "Johnny Cash");

            mDB.update(DatabaseOpenHelper.TABLE_NAME, values,
                  DatabaseOpenHelper.ARTIST_NAME + "=?",
                  new String[] { "Jawny Cash" });
      }

      // Delete all records
      private void clearAll() {
            mDB.delete(DatabaseOpenHelper.TABLE_NAME, null, null);
      }

      // Close database
      @Override
      protected void onDestroy() {
            mDB.close();
            mDbHelper.deleteDatabase();
            super.onDestroy();
      }
}
```

In the **onCreate** method, we begin by creating a new **DatabaseOpenHelper** instance. And that class is a subclass of **SQLiteOpenHelper**.

Let's take a look at that class. The constructor for this class calls the superclass's constructor, passing in information such as the name of the database, and a version number. This class's **onCreate** method receives an **SQLiteDatabase** object and then calls its **ExecSQL** method, passing in a string, which corresponds to an actual SQL command that will create a table named **Artists**. This table will contain two fields, an integer ID, and a string for the artist's name. This class also has a **DeleteDatabase** method, which simply deletes the database.

Going back to the main activity, the **onCreate** method continues by getting a reference to the underlying data base that can be used for reading and writing. Next, it calls the **clearAll** method, which just deletes every record in the database.

After that, the code calls the **insertArtist** method, which inserts a number of records into the database. This method first creates a **ContentValues** object, and then puts information into that object corresponding, in this case, to the artist's name, Frank Sinatra. Next, it inserts the record into the Artist table by calling the **insert** method. The I**D** field is auto generated by the database, so the application doesn't need to include it here. Next, the code clears out the **values** object, and then adds a second record for Lady Gaga, a third record for Johnny Cash, and a fourth record for Ludwig van Beethoven.

Going back up to the **onCreate** method, the code calls the **readArtist** method, which reads all the records in the database, and returns a **cursor** object. A **cursor** is essentially an iterator over a set of records returned by a **query** operation. This cursor is used to create an **Adapter** for the **ListView** that will display these records on the display.

And finally, the code **sets a Listener** on the Fix button. When the user presses this button, the code calls the **Fix** method.

Let's scroll down to that method. The **Fix** method first calls the **Delete** method, which finds the record with the artist name Lady Gaga, and then deletes it. And after that, the method creates a **ContentValues** object that has the correct spelling of the name Johnny. It then performs an **update** operation that first finds the record with the misspelled name, and then replaces it with the correct spelling.

When you need to **debug** your database applications, there are several things you might need to know:

The database files are stored in the **data/data/package name/databases** directory. You can examine these files by **first opening a shell** to the emulator or to your device with ADB. For instance, if your emulator's name is emulator-5554, then you can issue this command to open a shell.

Once the shell is open, you can then use the **SQLite3** command to open the connection to the database itself. From there, you can type, for example, **.help** to get more information about the specific commands this program understands.

Examining the Database Remotely

Databases stored in

/data/data/<package name>/databases/

Can examine database with sqlite3

adb -s emulator-5554 shell

sqlite3 /data/data/
course.examples.DataManagement.Data
BaseExample/databases/artist_dbd

That's all for our lesson on Data Management. Please join me next time, when we'll discuss the ContentProvider class.

Week 8 - Services

Much of our discussion of Android so far has focused around the **Activity** class and the associated UI objects. We've also talked about **Broadcast Receivers** and **Content Providers**. The fourth fundamental Android component is the **Service Class**, which is designed to support longer-running operations that are not usually visible to the user, such as **downloading large data files from the network, or synchronizing on-device information with the network server**.

In this lesson I'll begin by giving an overview of the **Service class**, and then I'll talk about how you implement relatively simple services that can be started by clients. I'll finish up by talking about how you implement more complex services that clients can bind to and interact with.

Services do not interact directly with users, so they lack user interfaces. Services have two main uses:

- They allow you to work in the background even if the services application terminates, i.e. **performing background processing**
- They allow code in one process to interact with code in another process, i.e. **supporting remote method execution**

Components that want to use a service, but don't need to directly interact with it, can start that service by calling the **Context** class's **startService** method, passing in an intent associated with that specific service.

```
Context.startService(Intent intent)
```

Once started, the service can run in the background **indefinitely**. Like any component, Android can kill a service if it needs that service's resources.

Started services are usually designed to perform a single operation, after which they terminate themselves without transferring any results back to the component that started them. By default, services run in the main thread of their hosting application. Depending on how you implement and use a service, you may need to create a separate thread for it.

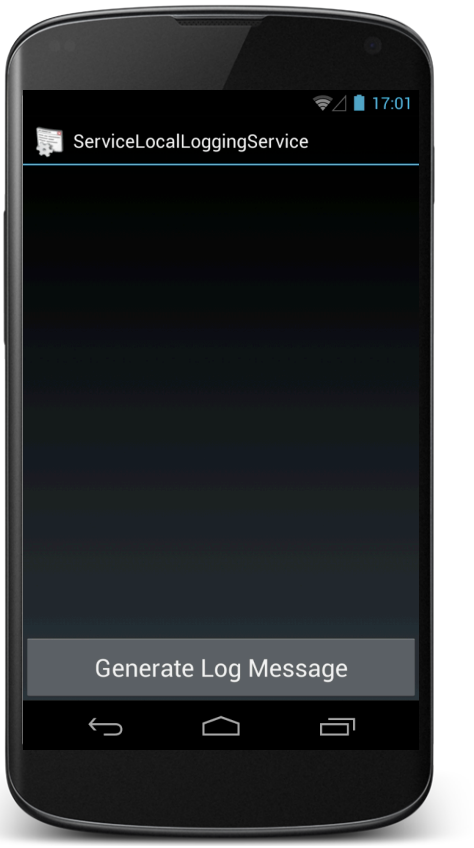
In contrast, components that want to use a service, but do need to directly interact with it, can bind to that service by calling the **bindService** method, passing in an Intent associated with a specific service, a **ServiceConnection** object, which implements callback methods when the client is connected to or disconnected from the service, and flags that control the behavior of the binding operation.

```
Context.bindService( Intent service, ServiceConnection conn,
int flags)
```

Binding to a service allows a component to send requests to and receive responses from a service running in either the same process or a different one. At binding time, if the service is not already started, it will be started. Once bound, the service will continue running as long as at least one client remains bound to it.

Implementing Started Services

Here's my device, now I'll start the **LocalLoggingService** application. As you can see, this application displays a single button labeled Generate Log Message.



When I click this button, a service will be started, and that service will write a message to the log. When I hit the button the LogCat view shows a log message that was written by application.

Let's look at the source code for this application's **LoggingServiceClient** class.

```
package course.examples.Services.LocalLoggingService;

import android.app.Activity;
import android.content.Intent;
```

```
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class LoggingServiceClient extends Activity {
     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);
          final Button messageButton = (Button) findViewById
               (R.id.message_button);
          messageButton.setOnClickListener
               new OnClickListener() {
               public void onClick(View v) {
                    // Create an Intent for starting the
                    //   LoggingService
                    Intent startServiceIntent =
                         new Intent(getApplicationContext(),
                         LoggingService.class);

                    // Put Logging message in intent
                    startServiceIntent.
                         putExtra(LoggingService.EXTRA_LOG,
                         "Log this message");

                    // Start the Service
                    startService(startServiceIntent);
               }
          });
     }
}
```

In **onCreate**, the code first creates an Intent that will be used to start the **LoggingService** object. Next, the code adds the message to be logged as an extra to the **startServiceIntent**. And lastly, the code calls the **startService** method, passing in the Intent.

Now, I'll open up a **LoggingService** class, which extends **IntentService**.

```
package course.examples.Services.LocalLoggingService;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;
```

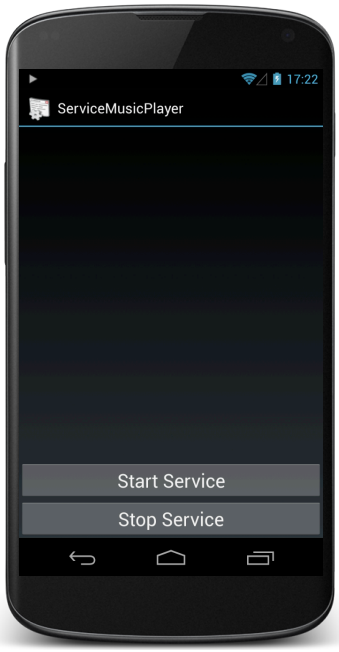
```
public class LoggingService extends IntentService {
     public static String EXTRA_LOG =
          "course.examples.Services.Logging.MESSAGE";
     private static final String TAG = "LoggingService";
     public LoggingService() {
          super(TAG);
     }

     @Override
     protected void onHandleIntent(Intent intent) {
          Log.i(TAG, intent.getStringExtra(EXTRA_LOG));
     }
}
```

It is started by the system and thereafter it queues incoming intents and hands them off, one at a time, to the **onHandleIntent** method. In this case, that method simply writes the incoming message to the log.

Our next example application is called **MusicPlayingServiceExample**, where a client activity starts a service that plays a music file. The service plays the music in what's called the **Foreground state**, which means the service will be doing something that the user be aware of and, therefore, Android should refrain from killing this service if the system gets low on memory. As with any service, the music playing service will keep playing even if the client activity pauses or terminates.

Let's watch this application run:



As you can see, there are now two buttons shown on the display. The top one is labeled Start Service, and the bottom one is labeled Stop Service. Pressing these buttons starts or stops a service that is hosted by this application. Let me start by pressing the start button, and as you can hear, when I press the start button, music started playing. You also notice that there's also a notification up in the status bar.

Now, I'll back out of this application and, even though I've killed the application, the service is still playing the music. Pulling down on the notification drawer, and clicking on the Notification View which restarts the application, I'll press the Stop Service button. The service and the music that it was playing both stop.

Let's look at the source code for this application in MusicServiceClient.java:

package course.examples.Services.MusicService;

```
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class MusicServiceClient extends Activity {

      @Override
      public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);

            setContentView(R.layout.main);

            // Intent used for starting the MusicService
            final Intent musicServiceIntent = new Intent
                  (getApplicationContext(), MusicService.class);

            final Button startButton = (Button) findViewById
                  (R.id.start_button);
            startButton.setOnClickListener(new OnClickListener() {
                  public void onClick(View src) {

                        // Start the MusicService using the Intent
                        startService(musicServiceIntent);

                  }
            });

            final Button stopButton = (Button) findViewById
                  (R.id.stop_button);
            stopButton.setOnClickListener(new OnClickListener() {
                  public void onClick(View src) {

                        // Stop the MusicService using the Intent
```

```
                stopService(musicServiceIntent);
            }
        });
    }
}
```

In onCreate, this code first creates an Intent, that will be used to start the **MusicService** class. Next the code sets up listeners for the startButton and for the stopButton. The startButton and listener calls the **startService** method, passing in the **musicServiceIntent**. And the stopButton calls the **stopService** method, also passing in the same musicServiceIntent.

Let's take a look now at the **MusicService class**:

```
package course.examples.Services.MusicService;

import android.app.Notification;
import android.app.PendingIntent;
import android.app.Service;
import android.content.Intent;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;
import android.os.IBinder;

public class MusicService extends Service {

    @SuppressWarnings("unused")
    private final String TAG = "MusicService";

    private static final int NOTIFICATION_ID = 1;
    private MediaPlayer mPlayer;
    private int mStartID;

    @Override
    public void onCreate() {
        super.onCreate();

        // Set up the Media Player
        mPlayer = MediaPlayer.create(this, R.raw.badnews);

        if (null != mPlayer) {

            mPlayer.setLooping(false);

            // Stop Service when music has finished playing
            mPlayer.setOnCompletionListener(
                    new OnCompletionListener() {

                    @Override
                    public void onCompletion(MediaPlayer mp) {

                    // stop Service if it was started with this ID
```

```
                // Otherwise let other start commands proceed
                    stopSelf(mStartID);
                }
            });
        }

        // Create a notification area notification so the user
        // can get back to the MusicServiceClient

        final Intent notificationIntent = new Intent(
            getApplicationContext(), MusicServiceClient.class);
        final PendingIntent pendingIntent =
            PendingIntent.getActivity( this, 0,
            notificationIntent, 0);
        final Notification notification = new Notification.Builder(
                getApplicationContext())
                .setSmallIcon(android.R.drawable.ic_media_play)
                .setOngoing(true).setContentTitle(
                "Music Playing")
                .setContentText("Click to Access Music Player")
                .setContentIntent(pendingIntent).build();

        // Put this Service in a foreground state, so it won't
        // readily be killed by the system
        startForeground(NOTIFICATION_ID, notification);
}

@Override
public int onStartCommand(Intent intent, int flags, int startid)
{    if (null != mPlayer) {
            // ID for this start command
            mStartID = startid;
            if (mPlayer.isPlaying()) {
                // Rewind to beginning of song
                mPlayer.seekTo(0);
            } else {
                // Start playing song
                mPlayer.start();
            }
        }
        // Don't automatically restart this Service if it is killed
        return START_NOT_STICKY;
}
@Override
public void onDestroy() {
        if (null != mPlayer) {
            mPlayer.stop();
            mPlayer.release();
        }
}
// Can't bind to this Service
@Override
public IBinder onBind(Intent intent) {
        return null;
```

```
        }
}
```

Here, in onCreate this code begins by setting up a new **MediaPlayer**. It also attaches an **onCompletionListener** to the MediaPlayer, which will stop the service when the music finishes playing. Next, the code creates a **notification area notification**, so that the user can exit the music player service client, but still have a way to get back to the client in order to shut down the music player.

Continuing down, the code calls the **startForeground** method, which puts this service in a foreground state that it is less likely to be killed if the system needs more resources. Next the code overrides the **onStartCommand** method, which is called when a client calls the start service command. The code in this method checks to see whether the media player exists, and if so, it then saves the current start command ID and then makes the media player play the song from the beginning. This method ends by returning a value that tells the system what to do if the service is killed by the system. In this case, that value is **START_NOT_STICKY,** which means that the system **should not automatically restart** the service if it gets killed.

Implementing Bound Services

Some services work independently of other components. They get started, they do their work, and they quit. Other services are instead meant to receive requests and to provide responses to other components. Sometimes those components are running in the same application or process as the service and sometimes they are not. In these cases, components will need to bind to the service. That is, they need to open and maintain a connection to the service so they can interact with it.

If the component and the service are running in different processes, then there are two common approaches for designing a service and for **binding** to that service. The first approach is to use the **Messenger** class. The second is to define a remote interface to the service using the **AIDL** language and tools. Let's talk about each of these one at a time.

The **Messenger class** manages a handler. Using a Messenger allows messages to be sent from one component to another component **across process boundaries**. Once sent, messages are then queued and processed sequentially by the receiving component. You should use this approach only if your service can tolerate **sequential access**.

For this approach to work, you'll have to implement the client components and the service so that they can send and receive Messenger messages.

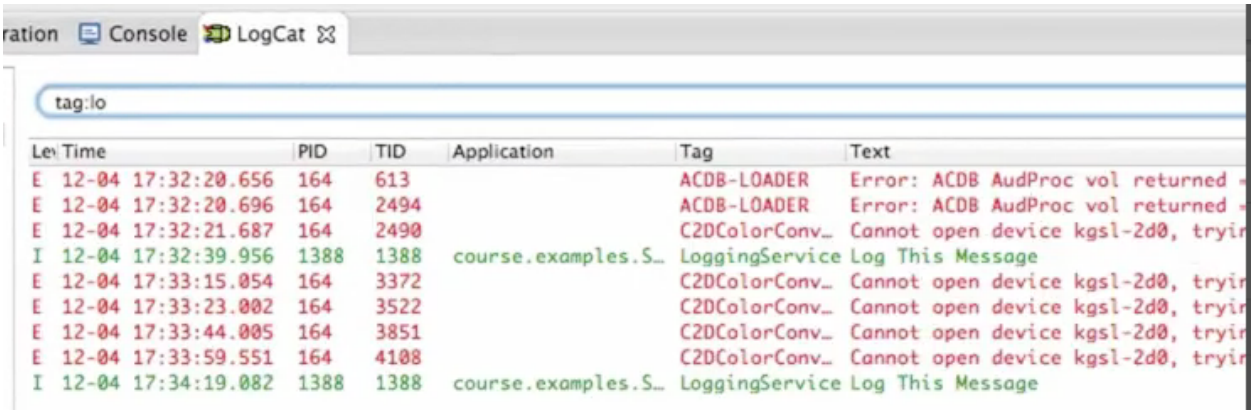
On the **service side**, your service will have to create a **Handler** for receiving and processing specific messages. The service will also have to create a Messenger that supplies back to any client a **Binder** object that binds to this service.

On the **client side**, the client binds to the service and eventually gets back a binder object that it uses to create its own Messenger, which can send messages to the Handler, where they are processed by the service.

Our next example applications are called **LoggingServiceWithMessengerClient** and **LoggingServiceWithMessenger**. With these applications, a client sends log messages to a remote logging service, which is implemented in a separate application. A remote logging service takes these messages and then writes them to the log.

Let's run these applications now. First, I'll start the **LoggingServiceWithMessenger** application:

As you can see, there's a single button, labeled Generate Log Message. Let me click on that now. Let's take a look at the log console to make sure that the message was actually written. In my IDE, open the LogCat view. And here you can see the log message that was written by the log in service.



Let's look a the source code for this application, **LoggingServiceWithMessenger**, starting with the **LoggingService** class:

```
package course.examples.Services.LoggingServiceWithMessenger;

import android.app.Service;
import android.content.Intent;
import android.os.Handler;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.util.Log;

public class LoggingService extends Service {

	private final static String MESSAGE_KEY =
		"course.examples.Services.Logging.MESSAGE";
	private final static int LOG_OP = 1;

	private static final String TAG = "LoggingService";

	// Messenger Object that receives Messages from connected clients
	final Messenger mMessenger = new Messenger(new
		IncomingMessagesHandler());

	// Handler for incoming Messages
	static class IncomingMessagesHandler extends Handler {

		@Override
		public void handleMessage(Message msg) {
			switch (msg.what) {
			case LOG_OP:
				Log.i(TAG, msg.getData().getString(MESSAGE_KEY));
				break;
			default:
				super.handleMessage(msg);
			}
		}
	}
```

```
	// Returns the Binder for the mMessenger, which allows
	// the client to send Messages through the Messenger
	@Override
	public IBinder onBind(Intent intent) {
		return mMessenger.getBinder();
	}
}
```

As you can see, this class extends the **Service** class. The code creates a Messenger object passing in a new **IncomingMessageHandler**, defined here. The **handleMessage** method receives messages that are sent to the handler. In this case, if the **message.what** field is **log_op**, then this code writes the message to the log. Lastly, this service defines an **onBind** method, which associates the Binder with the mMessenger object and returns it. This value will eventually be passed back to clients, that bind to this service.

Now let's look at the **LoggingServiceWithMessengerClient** application, starting with the **LoggingServiceClient** class:

```
package course.examples.Services.LoggingServiceWithMessengerClient;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.os.Message;
import android.os.Messenger;
import android.os.RemoteException;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class LoggingServiceClient extends Activity {

	private final static String MESSAGE_KEY =
		"course.examples.Services.Logging.MESSAGE";
	private final static int LOG_OP = 1;
	private final static String TAG = "LoggingServiceClient";

	// Intent used for binding to LoggingService
	private final static Intent mLoggingServiceIntent = new
		Intent("course.examples.Services.
		LoggingServiceWithMessenger.LoggingService");
	private Messenger mMessengerToLoggingService;
	private boolean mIsBound;

	// Object implementing Service Connection callbacks
```

```
private ServiceConnection mConnection = new ServiceConnection() {

      public void onServiceConnected(ComponentName className,
            IBinder service) {

            // Messenger object connected to the LoggingService
            mMessengerToLoggingService = new Messenger(service);
            mIsBound = true;
      }

      public void onServiceDisconnected(
            ComponentName className) {

            mMessengerToLoggingService = null;
            mIsBound = false;

      }
};

@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      final Button buttonStart = (Button) findViewById
            (R.id.buttonStart);
      buttonStart.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                  if (mIsBound) {
                        // Send Message to the Logging Service
                        logMessageToService();
                  }
            }
      });
}

// Create a Message and sent it to the LoggingService
// via the mMessenger Object

private void logMessageToService() {

      // Create Message
      Message msg = Message.obtain(null, LOG_OP);
      Bundle bundle = new Bundle();
      bundle.putString(MESSAGE_KEY, "Log This Message");
      msg.setData(bundle);

      try {
      // Send Message to LoggingService using Messenger
            mMessengerToLoggingService.send(msg);
      } catch (RemoteException e) {
            Log.e(TAG, e.toString());
      }
}
```

```
	// Bind to LoggingService
	@Override
	protected void onResume() {
		super.onResume();
		bindService(mLoggingServiceIntent, mConnection,
				Context.BIND_AUTO_CREATE);
	}

	// Unbind from the LoggingService
	@Override
	protected void onPause() {
		if (mIsBound) unbindService(mConnection);
		super.onPause();
	}
}
```

This code starts by creating an Intent that will be used for binding to the logging service. Next, the code defines an object that implements the **ServiceConnection** callback methods. In particular, in the **onServiceConnected** method the code creates a **Messenger** object passing in the binder that was returned from the logging service when this class binds to the logging service. This class will use this Messenger to communicate with the login service.

Now scrolling down to onCreate, the code sets a Button listener that calls the **logMessageToService** method when this button is clicked, which creates a new **Message object** containing a message code, and the text to be written to the log. Then it sends that message to the logging service using the messenger object's **send** method.

Continuing on, in **onResume** the code calls **bindService**, passing in the right intent, the callback object, and a flag that tells Android to create the service if it doesn't already exist.

In **onPause**, the code unbinds from the logging service.

Implementing bound services with a messenger as we just did is often the simplest way to go as long as you can tolerate having only sequential access to the service.

When that doesn't work, though, then you will want to develop an **AIDL** interface to your service. AIDL stands for the **Android Interface Definition Language**. When you use this language to implement your service, you normally do the following things:

1. Define the service's remote interface using AIDL.
2. Implement the methods of that remote interface.
3. Implement the various service life cycle and callback methods.
4. Implement the client methods.

Let's talk about these steps. To define the service's remote interface, you'll need to create an **interface definition** using the AIDL language in a **.AIDL file**, which defines the methods through which components can interact with your service.

AIDL is similar in many ways to the syntax that you use in Java when you create interfaces. For example, you can declare methods that need to be implemented if an object wants to say that it conforms to that interface. One way in which it differs however, is that **in AIDL, you can't declare static fields**. Another difference is that non-primitive parameters to methods defined by the interface, **require a directional tag**, which indicates how data is copied into and out of the methods.

The choices for the directional tag are:
- **In** - data is only transferred into the remote method.
- **Out** - data is only returned out of the remote method
- **Inout** - data is transferred both into and out of the remote method.

The parameters and return types of these remote methods are limited to the following types: you can have **Java primitive types such as int, float, and boolean**, you can have a **string and/or a character sequence**.

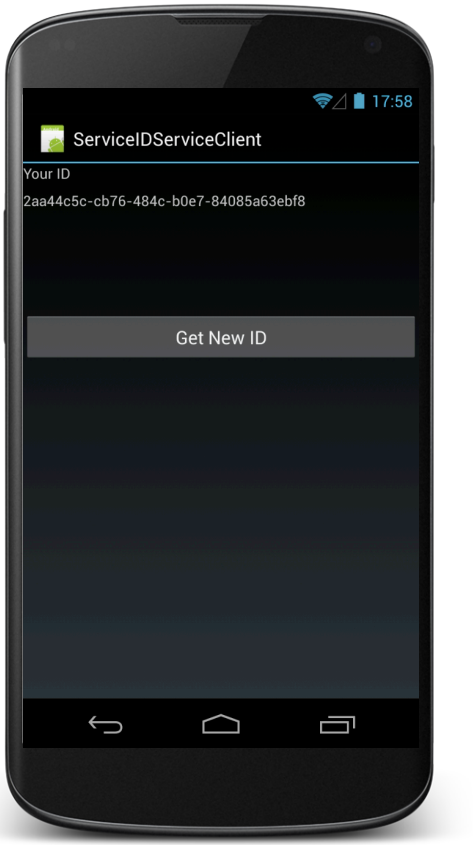
If you define other AIDL interfaces you can also use those as well, and you can use any class that implements the **Parcelable** interface. You can also have **lists**, whose elements are any of those types that I just mentioned. And these lists can be defined as **generic classes that take a type parameter**. And finally, you can also use **maps** of any of the types I've discussed. AIDL **does not allow for generic map types**.

Here's a simple example of an AIDL interface from an upcoming example. This interface is called **KeyGenerator** and it defines a single method called **getKey** that takes no parameters and returns a string:

```
interface KeyGenerator {
     String getKey();
}
```

Our next example applications are called **ServiceWithIPCExampleClient** and **ServiceWithIPCExampleService**, where a client in one application binds to a service hosted in another application. The client then retrieves a unique ID from the service. For this example, we're going to assume that, for whatever reason, we want the service to handle ID requests concurrently, so we've decided to use AIDL to define a remote interface to the service.

Let's see these applications in action. Starting the **ServiceWithIPCExampleClient** application:



As you can see the application presents a single button labeled Get New ID, when I press this button, this application will bind to a service to retrieve a unique ID. This application will then display that new ID. So here it goes. And there's my new ID.

Let's look at the source code for these applications. So here are the applications open in the IDE. Let's start by looking at the **ServiceWithIPCExampleService** application. Now I'll start by opening the keygenerator.aidl file.

```
package course.examples.Services.KeyCommon;

interface KeyGenerator {
    String getKey();
}
```

As we saw earlier this file defines the interface that clients can use to interact with this service. That is they can call one method, **getKey** and then we'll get back a string.

Now let's look at the KeyGeneratorImpl class, implements the remote methods defined in the AIDL KeyGenerator interface:

```
package course.examples.Services.KeyService;

import java.util.HashSet;
import java.util.Set;
import java.util.UUID;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import course.examples.Services.KeyCommon.KeyGenerator;

public class KeyGeneratorImpl extends Service {

      // Set of already assigned IDs
      // Note: These keys are not guaranteed to be unique if the
      // Service is killed and restarted.

      private final static Set<UUID> mIDs = new HashSet<UUID>();

      // Implement the Stub for this Object
      private final KeyGenerator.Stub mBinder =
            new KeyGenerator.Stub() {

            // Implement the remote method
            public String getKey() {

                  UUID id;

                  // Acquire lock to ensure exclusive access to mIDs
                  // Then examine and modify mIDs

                  synchronized (mIDs) {

                        do {
                              id = UUID.randomUUID();
                        } while (mIDs.contains(id));

                        mIDs.add(id);
                  }
                  return id.toString();
            }
      };

      // Return the Stub defined above
      @Override
      public IBinder onBind(Intent intent) {
            return mBinder;
      }
}
```

First, the class defines a set of IDs that it has already assigned. And know that this code will not guarantee that all keys are unique if this service happens to be killed and then

restarted. To do that, **you would need to store and restore the already assigned keys**. Now, in order to implement this interface, you need to implement the remote methods inside an abstract inner class called **Stub**. That's just the name choice that the AIDL compiler makes, and **you have to respect that choice**. Here the code makes a new **KeyGenerator.Stub** instance, and then defines the **getKey** method in line.

The methods of this class **need to be thread safe**. So this class first synchronizes on the mIDs object. And then it generates new IDs, until it finds one that hasn't already been assigned. It then stores the ID, and returns a string version of it to the client.

Finally, down in the **onBind** method, the code returns the **Stub** class just defined above, which will eventually be given to the client so it can interact with this service.

Let's look at the **ServiceWithIPCExampleClient** application's main activity:

```
package course.examples.Services.KeyClient;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.TextView;
import course.examples.Services.KeyCommon.KeyGenerator;

public class KeyServiceUser extends Activity {

      protected static final String TAG = "KeyServiceUser";
      private KeyGenerator mKeyGeneratorService;
      private boolean mIsBound;

      @Override
      public void onCreate(Bundle icicle) {
            super.onCreate(icicle);

            setContentView(R.layout.main);

            final TextView output = (TextView) findViewById
                  (R.id.output);

            final Button goButton = (Button) findViewById
                  (R.id.go_button);
            goButton.setOnClickListener(new OnClickListener() {
```

```
            public void onClick(View v) {

                  try {

                        // Call KeyGenerator and get a new ID
                        if (mIsBound)
                              output.setText
                                    (mKeyGeneratorService.getKey());

                  } catch (RemoteException e) {

                        Log.e(TAG, e.toString());

                  }
            }
      });
}

// Bind to KeyGenerator Service
@Override
protected void onResume() {
      super.onResume();

      if (!mIsBound) {

            Intent intent = new Intent(KeyGenerator.class.
                  getName());
            bindService(intent, this.mConnection,
                  Context.BIND_AUTO_CREATE);

      }
}

// Unbind from KeyGenerator Service
@Override
protected void onPause() {
      if (mIsBound) {
            unbindService(this.mConnection);
      }
      super.onPause();
}

private final ServiceConnection mConnection =
      new ServiceConnection() {

      public void onServiceConnected(ComponentName className,
            IBinder iservice) {
            mKeyGeneratorService =
                  KeyGenerator.Stub.asInterface(iservice);
            mIsBound = true;

      }

      public void onServiceDisconnected(ComponentName className)
```

```
            {
                  mKeyGeneratorService = null;
                  mIsBound = false;
            }
      };
}
```

In **onCreate**, the code sets up a button that will make a call to the **KeyGeneratorService.getKey** method, and then display the result in a text view.

In **onResume**, the code binds to the **KeyGenerato**r service.

And in **onPause**, the code unbinds from the KeyGenerator service.

Scrolling down, the code creates the **ServiceConnection** object. In particular, in the **onServiceConnected** method, the code takes the **Stub** that was returned from the KeyGenerator service. And calls the **KeyGenerator.Stub.asInterface** method, which returns an object that essentially represents the connection to the KeyGenerator service.

So that brings us to the end of our discussion on services. And in fact, that brings us to the end of the course. You did it. Congratulations!

It's been my pleasure, indeed, my honor to be your instructor. And I truly hope that you're leaving this class with much, much more than what you brought to it. I know I am. Now if you remember back to the start of the class, I asked you to write a note to yourself, spelling out what you hoped to learn from this course. When you get a chance, take a look at that note, and let us know how things turned out. We'd love to hear.

So that's all for now. Maybe our paths will cross again. I certainly hope so. Until then, however, keep on learning. So long.

Week 8 - Content Providers

So far in this course we've talked in detail about only two of the four fundamental components of Android, Activities and Broadcast receivers. In this lesson, we'll discuss another: Content providers.

Content providers represent centralized repositories of structured data. They encapsulate different data sets and they specify and enforce the permissions needed to access that data. In many ways the **ContentProvider** class resembles the databases we covered in the lesson on data storage, but content providers are specifically designed to allow data sharing across between applications.

Applications that want to access a particular content provider do so by using a companion class called **ContentResolver**, which presents a database-style interface that lets applications read and write the data stored in a content provider. A ContentResolver, therefore, supports methods such as **query, insert, update, delete**, etc. ContentResolvers also provide additional services such as notifying registered observers when data in the content provider changes.

When using a content resolver, your application will first get a reference to the current context's ContentResolver by calling the Context class's **getContentResolve**r method. Together, the content provider and the content resolver make it possible for code running in one process to access data managed in another process.

To give some examples, Android comes with a number of standard content providers. For instance, there's a content provider that stores browser information such as your bookmarks and your browsing history. There's one for keeping track of the telephone calls that you make. There's one managing contact information, another for keeping track of your pictures, songs and videos, and another for keeping track of the words you type into various applications so your device can improve predictive typing over time.

ContentProvider Data Model

Data represented logically as database tables

_ID	artist
13	Lady Gaga
44	Frank Sinatra
45	Elvis Presley
53	Barbara Streisand

Logically, the data managed by a content provider is represented as a database table. For example, a content provider for managing a list of artists might have data elements or columns for each record such as an ID field, and an artist name field. Applications identify the data they want and the appropriate content provider through a URI.

Let's look at that format. Content URIs start with the string “content://”, which is the **URI's scheme** and indicates that the URI refers to data that is managed by a content provider. The URI also specifies it's **authority** by naming the specific content provider carrying the desired data. The URI can also specify a **path**, containing segments that indicate the specific data set containing the desired data. Finally URI can have an **ID** that identifies a specific record within the desired data set.

Format

content://authority/path/id

content – scheme indicating data that is managed by a content provider

authority – id for the content provider

path – 0 or more segments indicating the type of data to be accessed

id – a specific record being requested

For example, an application that wants to access the contacts content provider might use the following URI:

```
ContactsContract.Contacts.CONTENT_URI =
"content://com.android.contacts/contacts/"
```

This URI specifies a **content scheme**. It specifies com.android.contacts as it's **authority**. Its **path** has the single string "contacts", which basically corresponds to a logical table within the content provider's database. And in this case, **there's no ID** field, so that URI is interpreted as referring to the entire table, not to a single record within the table.

An application can use this URI with a **ContentResolver** method, such as the **query** method, to retrieve some or all of the contact records managed by the **content provider**. As you can see in the next slide, the **query** method takes several parameters, including a URI, something like what we just saw, an array of strings which specifies the specific columns within the database that should be retrieved from the ContentProvider, and several more strings that are used to perform the SQL query to be processed by the ContentProvider. These strings can be used to retrieve particular subsets of the data, and to specify an ordering criteria for the results.

ContentResolver.query()

```
Cursor query (
    Uri uri,                    // ContentProvider Uri
    String[] projection         // Columns to retrieve
    String selection            // SQL selection pattern
    String[] selectionArgs      // SQL pattern args
    String sortOrder            // Sort order
    )
```

Returns a Cursor for iterating over the set of results

An application example that uses this method, **ContentProviderWithCursorAdapter**, extracts contact information from the contacts ContentProvider and displays each contact's name and photo.

Let's see that application in action (above). As you can see it presents the names and photos of some of my contacts.

Let's look at the source code for this application's main activity:

```
package
course.examples.ContentProviders.ContactsListWithAdapter;
import android.app.ListActivity;
import android.content.ContentResolver;
import android.database.Cursor;
import android.os.Bundle;
import android.provider.ContactsContract.Contacts;

public class ContactsListExample extends ListActivity {
     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);

          // Contact data
          String columnsToExtract[] = new String[]
               { Contacts._ID, Contacts.DISPLAY_NAME,
               Contacts.PHOTO_THUMBNAIL_URI };

          // Get the ContentResolver
          ContentResolver contentResolver =
               getContentResolver();

          // filter contacts with empty names
          String whereClause = "((" + Contacts.DISPLAY_NAME +
               " NOTNULL) AND ("+ Contacts.DISPLAY_NAME +
               " != '' ) AND (" + Contacts.STARRED + "== 1))";

          // sort by increasing ID
          String sortOrder = Contacts._ID + " ASC";

          // query contacts ContentProvider
          Cursor cursor = contentResolver.query
               (Contacts.CONTENT_URI, columnsToExtract,
               whereClause, null, sortOrder);

          // pass cursor to custom list adapter
          setListAdapter(new ContactInfoListAdapter(this,
               R.layout.list_item, cursor, 0));
     }
}
```

In **onCreate** the code first defines the columns that it would request from the contact ContentProvider. There are, of course, many different pieces of data associated with each contact, but in this application we are interested in only the contact's name and thumbnail photo.

Next, the code gets a reference to to context's **ContentResolver**. Then the code defines a string for **filtering** out contacts with missing, empty, or unstarred names. Next, the code creates a string defining an **ascending sorting order** for the records based on the _ID fields.

Then the code issues a call to the **query** method, where the parameters include the **URI**, which is defined by the contact class called **Contacts**, the **columns** to extract, the **where clause** to filter out specific contacts, and the **string defining the sort order**. This method returns a **cursor**, which will allow the application to iterate over the results of this query.

Finally, the code **creates and sets a new adapter** which will be used by the **listView** to display the contact info. The adapter is a **ContactInfoListAdapter,** which is defined by this application.

Let's open that class now:

```
package
     course.examples.ContentProviders.ContactsListWithAdapter;

import java.io.FileNotFoundException;
import java.io.InputStream;

import android.content.Context;
import android.database.Cursor;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.net.Uri;
import android.provider.ContactsContract.Contacts;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.ResourceCursorAdapter;
import android.widget.TextView;

public class ContactInfoListAdapter extends
     ResourceCursorAdapter {

     private Bitmap mNoPictureBitmap;
```

```
private String TAG = "ContactInfoListAdapter";

public ContactInfoListAdapter(Context context, int layout,
     Cursor c, int flags) {
     super(context, layout, c, flags);

     // default thumbnail photo
     mNoPictureBitmap = BitmapFactory.decodeResource
          (context.getResources(),
          R.drawable.ic_contact_picture);
}

// Create and return a new contact data view
@Override
public View newView(Context context, Cursor cursor,
     ViewGroup parent) {

     LayoutInflater inflater = (LayoutInflater)
          context.getSystemServiceContext.
          LAYOUT_INFLATER_SERVICE);
     return inflater.inflate(R.layout.list_item, parent,
     false);

}

// Update and return a contact data view
@Override
public void bindView(View view, Context context,
     Cursor cursor) {

     TextView textView = (TextView) view.findViewById
          (R.id.name);
     textView.setText(cursor.getString(cursor
          .getColumnIndex(Contacts.DISPLAY_NAME)));

     // default thumbnail photo
     ImageView imageView = (ImageView) view.findViewById
          (R.id.photo);
     Bitmap photoBitmap = mNoPictureBitmap;

     // Get actual thumbnail photo if it exists
     String photoContentUri = cursor.getString(cursor
          .getColumnIndex(Contacts.PHOTO_THUMBNAIL_URI));

     if (null != photoContentUri) {
          InputStream input = null;
```

```
                try {

                     // Read thumbnail data from input stream
                     input = context.getContentResolver()
                          .openInputStream
                          (Uri.parse(photoContentUri));

                     if (input != null) {
                          photoBitmap =
                          BitmapFactory.decodeStream(input);
                     }

                } catch (FileNotFoundException e) {
                     Log.i(TAG, "FileNotFoundException");
                }
           }

           // Set thumbnail image
           imageView.setImageBitmap(photoBitmap);
     }
}
```

In this class's constructor, the code finds and stores a default image for contacts that have no thumbnail photo. When the **ListView** needs views to display, it will first call the **newView** method to get a brand new view, and then it will call the **bindView** method to add data to that view.

Let's look closely these two methods. First, the **newView** method gets a reference to the **LayoutInflater** service and calls the **inflate** method to create the new view based on an XLM resource.

Down in the **bindView** method the code fills in that view. First, it displays the contact's name within a TextView. Then it stores a reference to an ImageView within this view, and it stores the default photo in the **photoBitmap** variable. Next the code checks whether the is an actual photo stored for this contact. If so, it gets the URI associated with that photo and **opens an input stream** to read the photo data into memory. Next, it turns that data into a bitmap which it stores in the **photoBitmap** variable. And finally, the code sets the photoBitmap as the image bitmap for the image view object we saw earlier.

When an application queries a **ContentProvider**, that operation can take a while to complete and we generally try to avoid doing intensive operations on the main thread to avoid slowing an application's responsiveness. To prevent this when we use ContentProviders, Android provides the **CursorLoader class**, which uses an **AsyncTask** so that queries will be performed on a background thread instead of the main thread.

To use a **CursorLoader**, you first need to create an object that implements the **LoaderManager's LoaderCallback interface**. At run time, you must create and initialize the CursorLoader with the help of that object that implements the loader callbacks. To do this, the application will call the **LoaderManager's initLoader** method to create and initialize a loader. That method takes several parameters, including an ID, arguments, and the object that implements the LoaderCallbacks.

```
Loader<D> initLoader(int id, Bundle args, LoaderCallbacks<D>
callback)
```

After **initLoade**r is called, several callbacks will occur. The first callback will be to the onCreateLoader method.

```
Loader<D> onCreateLoader(int id, Bundle args)
```

In this method, we'll create and return a new loader for the specified ID. When that loader finishes loading its data, Android calls the next callback method, onLoadFinished.

```
void onLoadFinished(Loader<D> loader, D data)
```

This method receives the newly created loader and a cursor containing the relevant data.

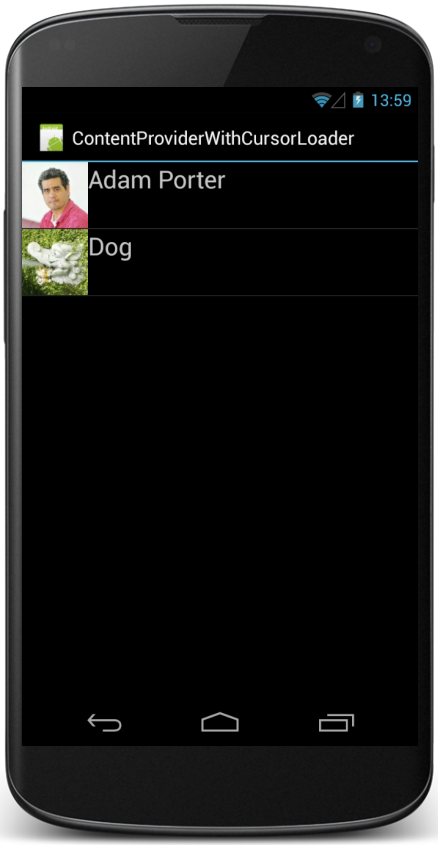
When a previously created loader is reset, the onLoaderReset method is called.

```
void onLoaderReset (Loader<D> loader)
```

In this method, applications will typically remove any references they have to the previous loader's data.

Our next example application is called **ContentProviderWithCursorLoader**. This application produces the same result as our previous example. It extracts contact information from the **Android Contacts ContentProvider** and it then displays each contact's name and photo if the photo is available. However, this application performs the **ContentProvider Query** in a background thread using the **CursorLoader.**

Let's see that application in action. As expected we see the same data as with the previous application.



Let's take a look at the source code for this application's main activity.

```
package course.examples.ContentProviders.ContactsList;

import android.app.ListActivity;
import android.app.LoaderManager;
import android.content.CursorLoader;
import android.content.Loader;
import android.database.Cursor;
import android.os.Bundle;
import android.provider.ContactsContract.Contacts;

public class ContactsListExample extends ListActivity implements
     LoaderManager.LoaderCallbacks<Cursor> {

     private ContactInfoListAdapter mAdapter;

     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
```

```
        // Create and set empty adapter
        mAdapter = new ContactInfoListAdapter(this,
             R.layout.list_item, null, 0);
        setListAdapter(mAdapter);

        // Initialize the loader
        getLoaderManager().initLoader(0, null, this);

    }

    // Contacts data items to extract
    static final String[] CONTACTS_ROWS = new String[]{
        Contacts._ID,
        Contacts.DISPLAY_NAME,
        Contacts.PHOTO_THUMBNAIL_URI };

    // Called when a new Loader should be created
    // Returns a new CursorLoader

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {

        // String used to filter contacts with empty or
        //  missing names or are unstarred
        String select = "((" + Contacts.DISPLAY_NAME +
             " NOTNULL) AND ("+ Contacts.DISPLAY_NAME +
             " != '' ) AND (" + Contacts.STARRED + "== 1))";

        // String used for defining the sort order
        String sortOrder = Contacts._ID + " ASC";

        return new CursorLoader(this, Contacts.CONTENT_URI,
             CONTACTS_ROWS, select, null, sortOrder);
    }

    // Called when the Loader has finished loading its data
    @Override
    public void onLoadFinished(Loader<Cursor> loader,
        Cursor data) {

        // Swap the new cursor into the List adapter
        mAdapter.swapCursor(data);
    }

    // Called when the last Cursor provided to onLoadFinished()
```

```
        // is about to be closed

        @Override
        public void onLoaderReset(Loader<Cursor> loader) {

            // set List adapter's cursor to null
            mAdapter.swapCursor(null);
        }
}
```

In **onCreate**, this application creates and sets an empty adapter, because there's no data to show at this point. Then the code gets **loaderManager** and calls the **initLoader** method on it, passing in the id zero, null for the arguments, and the reference "**this**" as the object to call back as the loader progresses.

As before, the first callback method to be called will be the **onCreateLoade**r method that sets up the same filters that we saw before: we don't want contacts with empty or missing names and we don't want contacts that are unstarred. We do want this data to be sorted in a ascending order by ID. The code creates and returns a new **CursorLoader**, passing in the information that we just set up.

The **query** that provides the data for this loader will be performed on a background thread. And when that query is finished, the **onLoadFinish** method will be called. This method takes the **Cursor** that is passed in and swaps it into the ListAdapter used by this application.

Finally, when the cursor is about to be closed, the **onLoaderReset** method is called. In this case, that method simply swaps in a null cursor to the list view's list adapter.

In addition to the query method, some common content resolver methods include the **delete, insert** and **update** methods. The **delete** method takes three parameters, a URI specifying the data to be deleted, a string specifying a pattern for selecting which rows to delete, and another array of strings, providing arguments for that selection pattern. This method returns the number of rows that were deleted.

```
int delete( Uri url, String where, String[] selectArgs)
```

The **insert** method takes a URI indicating the data set into which you want to insert a new row. It also takes an object of the content values type, which holds the fields for that new row. And this method returns the URI of the newly inserted row:

```
Uri insert (Uri uri, ContentValues values)
```

The **update** method also takes a URI and a content values object, just like Insert did. But like the delete method, it also takes a string selection pattern and an array of

selection string arguments for indicating which specific rows to update. And this method returns the number of rows that were updated:

```
int update(Uri uri, ContentValues values, String where, String[]
selectionArgs)
```

Our next example application is called **ContentProviderInsertContacts.** This application reads several contacts from the Android contacts content provider, inserts several new contacts into the content provider and displays the old and the new contacts. When the application exits, it deletes all these new contacts.

Let's run this application:

Initially, this application displays a single button labeled Insert Contacts. I'll hit the Home button now and start the People application. Here you can see some contacts that I have on this device. I'll quit this application and restart our sample application. Again you can see the Insert Contacts button. When I hit this button, some new contacts will be inserted into the contacts content provider. So, here it goes.

Here you can see that I've inserted four new contacts into the contacts content provider. Now just to be sure let me hit the home button again and restart the people application. Now as you can see, the application does indeed Show those new contacts that we

programmatically inserted into the contacts content provider. So that means that other applications can see, display, and even modify these new contacts.

Let's look at the source code for this application's main activity.

```
package course.examples.ContentProviders.ContactsListInsertContacts;

import java.util.ArrayList;
import java.util.List;

import android.accounts.Account;
import android.accounts.AccountManager;
import android.app.ListActivity;
import android.app.LoaderManager;
import android.content.ContentProviderOperation;
import android.content.CursorLoader;
import android.content.Loader;
import android.content.OperationApplicationException;
import android.database.Cursor;
import android.os.Bundle;
import android.os.RemoteException;
import android.provider.ContactsContract;
import
      android.provider.ContactsContract.CommonDataKinds.StructuredName;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.Data;
import android.provider.ContactsContract.RawContacts;
import android.util.Log;
import android.widget.SimpleCursorAdapter;
import course.examples.ContentProviders.ContactsListWithInsDel.R;

public class DisplayActivity extends ListActivity implements
      LoaderManager.LoaderCallbacks<Cursor> {

      public final static String[] mNames = new String[] {
            "Android Painter",
            "Steve Ballmer",
            "Steve Jobs",
            "Larry Page" };

      private static final String columnsToExtract[] = new String[] {
            Contacts._ID, Contacts.DISPLAY_NAME, Contacts.STARRED };
      private static final String columnsToDisplay[] = new String[]
      { Contacts.DISPLAY_NAME };
      private static final int[] resourceIds = new int[] { R.id.name };
      private static final String TAG = "ContactsListDisplayActivity";

      private Account[] mAccountList;
      private String mType;
      private String mName;
      private SimpleCursorAdapter mAdapter;

      @Override
```

```
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);

      // Get Account information
      mAccountList = AccountManager.get(this).getAccountsByType
            ("com.google");
      mType = mAccountList[0].type;
      mName = mAccountList[0].name;

      // Insert new contacts
      insertAllNewContacts();

      // Create and set empty list adapter
      mAdapter = new SimpleCursorAdapter(this,
            R.layout.list_layout, null,
            columnsToDisplay, resourceIds, 0);
      setListAdapter(mAdapter);

      // Initialize a CursorLoader
      getLoaderManager().initLoader(0, null, this);
}

// Insert all new contacts into Contacts ContentProvider
private void insertAllNewContacts() {

      // Set up a batch operation on Contacts ContentProvider
      ArrayList<ContentProviderOperation> batchOperation =
            new ArrayList<ContentProviderOperation>();

      for (String name : mNames) {
            addRecordToBatchInsertOperation(name, batchOperation);
      }

      try {

            // Apply all batched operations
            getContentResolver().applyBatch
                  (ContactsContract.AUTHORITY, batchOperation);

      } catch (RemoteException e) {
            Log.i(TAG, "RemoteException");
      } catch (OperationApplicationException e) {
            Log.i(TAG, "RemoteException");
      }

}

// Insert named contact into Contacts ContentProvider
private void addRecordToBatchInsertOperation(String name,
      List<ContentProviderOperation> ops) {

      int position = ops.size();

      // First part of operation
```

```
            ops.add(ContentProviderOperation.newInsert
                  (RawContacts.CONTENT_URI)
                  .withValue(RawContacts.ACCOUNT_TYPE, mType)
                  .withValue(RawContacts.ACCOUNT_NAME, mName)
                  .withValue(Contacts.STARRED, 1).build());

            // Second part of operation
            ops.add(ContentProviderOperation.newInsert
                  (Data.CONTENT_URI)
                  .withValueBackReference(Data.RAW_CONTACT_ID, position)
                  .withValue(Data.MIMETYPE,
                  StructuredName.CONTENT_ITEM_TYPE)
                  .withValue(StructuredName.DISPLAY_NAME,
                  name).build());
      }

      // Remove all newly-added contacts when activity is destroyed
      @Override
      protected void onDestroy() {
            deleteAllNewContacts();
            super.onDestroy();
      }

      private void deleteAllNewContacts() {
            for (String name : mNames) {
                  deleteContact(name);
            }
      }

      private void deleteContact(String name) {
            getContentResolver().delete
                  (ContactsContract.RawContacts.CONTENT_URI,
                  ContactsContract.Contacts.DISPLAY_NAME + "=?",
                  new String[] { name });
      }

      public Loader<Cursor> onCreateLoader(int id, Bundle args) {
            String select = "((" + Contacts.DISPLAY_NAME +
            " NOTNULL) AND (" + Contacts.DISPLAY_NAME +
            " != '' ) AND (" + Contacts.STARRED + "== 1))";
            return new CursorLoader(this, Contacts.CONTENT_URI,
                  columnsToExtract, select, null,
                  Contacts._ID + " ASC");
      }

      public void onLoadFinished(Loader<Cursor> loader, Cursor data) {
            mAdapter.swapCursor(data);
      }

      public void onLoaderReset(Loader<Cursor> loader) {
            mAdapter.swapCursor(null);
      }
}
```

In **OnCreate**, the code first gets information about the accounts registered on this device. Next the code calls the **insertAllNewContacts** method, which inserts the four new contacts. After this, the code sets up and uses a **CursorLoader** to get and display the appropriate contact information.

Let's look at how this application inserts the new contacts. The **insertAllNewContacts** method sets up a batch operation in which it inserts all of the new contacts at once. It starts by creating an ArrayList of content provider operations. And it then calls the **addRecordToBatchInsertOperation** method for each of the new contacts. Now, in this method, the code first adds some information to the raw contacts table, such as the account's name, type and the fact that this contact should be **starred, which means it's treated as a favorite contact**. Next, the method adds the new contact's name to the data table. And after all the new contacts have been added to the batch operation, the insert all new contacts method then called the applied batch method to commit the entire batch operation.

Implementing ContentProviders

If you want to create your own content provider, then you'll need to do the following things:

1. Implement a storage system for the data. You'll often do this by creating an **SQL Lite database**, but other approaches will work just as well.
2. Define what is called a **Contract Class** that defines constants and other information that other applications will need in order to use your Content Provider.
3. Define a **ContentProvider subclass** implementing methods such as delete and insert.
4. Declare and configure your Content Provider in the AndroidManifest.xml file for its application.

The next example application is called **ContentProviderCustom**. This application defines a content provider for simple ID and string pairs.

Let's first take a look at the source code for this application's content provider itself, which is in the StringsContentProvider.java file.

```
package course.examples.ContentProviders.StringContentProvider;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.database.Cursor;
import android.database.MatrixCursor;
import android.net.Uri;
import android.util.SparseArray;
```

```
// Note: Currently, this data does not persist across device reboot
public class StringsContentProvider extends ContentProvider {

      // Data storage
      private static final SparseArray<DataRecord> db =
            new SparseArray<DataRecord>();

      @SuppressWarnings("unused")
      private static final String TAG = "StringsContentProvider";

      // Delete some or all data items
      @Override
      public synchronized int delete(Uri uri, String selection,
                  String[] selectionArgs) {

            int numRecordsRemoved = 0;

            // If last segment is the table name, delete all data items
            if (isTableUri(uri)) {

                  numRecordsRemoved = db.size();
                  db.clear();

            // If last segment is the digit, delete the data item with
            // that ID
            } else if (isItemUri(uri)) {
                  Integer requestId = Integer.parseInt
                        (uri.getLastPathSegment());
                  if (null != db.get(requestId)) {
                        db.remove(requestId);
                        numRecordsRemoved++;
                  }
            }

            //return number of items deleted
            return numRecordsRemoved;
      }

      // Return MIME type for given uri
      @Override
      public synchronized String getType(Uri uri) {
            String contentType = DataContract.CONTENT_ITEM_TYPE;
            if (isTableUri(uri)) {
                  contentType = DataContract.CONTENT_DIR_TYPE;
            }
            return contentType;
      }

      // Insert specified value into ContentProvider
      @Override
      public synchronized Uri insert(Uri uri, ContentValues value) {
```

```
      if (value.containsKey(DataContract.DATA)) {

            DataRecord dataRecord = new DataRecord
                  (value.getAsString(DataContract.DATA));
            db.put(dataRecord.getID(), dataRecord);

            // return Uri associated with newly-added data item
            return Uri.withAppendedPath(DataContract.CONTENT_URI,
                  String.valueOf(dataRecord.getID()));
      }
      return null;
}

// Return all or some rows from ContentProvider based on
// specified Uri.
// All other parameters are ignored

@Override
public synchronized Cursor query(Uri uri, String[] projection,
      String selection, String[] selectionArgs,
      String sortOrder) {

      // Create simple cursor
      MatrixCursor cursor = new MatrixCursor
            (DataContract.ALL_COLUMNS);

      if (isTableUri(uri)) {

            // Add all rows to cursor
            for (int idx = 0; idx < db.size(); idx++) {

                  DataRecord dataRecord = db.get(db.keyAt(idx));
                  cursor.addRow(new Object[] { dataRecord.getID(),
                              dataRecord.getData() });

            }
      } else if (isItemUri(uri)){

            // Add single row to cursor
            Integer requestId = Integer.parseInt
                  (uri.getLastPathSegment());

            if (null != db.get(requestId)) {
                  DataRecord dr = db.get(requestId);
                  cursor.addRow(new Object[] { dr.getID(),
                        dr.getData() });
            }
      }
      return cursor;
}

// Ignore request
@Override
public synchronized int update(Uri uri, ContentValues values,
```

```
            String selection, String[] selectionArgs) {
            return 0;
      }

      // Initialize ContentProvider
      // Nothing to do in this case
      @Override
      public boolean onCreate() {
            return true;
      }

      // Does last segment of the Uri match a string of digits?
      private boolean isItemUri(Uri uri) {
            return uri.getLastPathSegment().matches("\\d+");
      }

      // Is the last segment of the Uri the name of the data table?
      private boolean isTableUri(Uri uri) {
            return uri.getLastPathSegment().equals
                  (DataContract.DATA_TABLE);
      }
}
```

Here you can see that this class extends the **ContentProvider** class. First you can see that the actual data is stored in a simple data structure of the type SparseArray. **So your content provider doesn't always have to be implemented using a database**.

Next the code defines a number of methods. First, there's the **delete** method, which removes either a single record from the database or removes all of them. Also, this method **doesn't process any of the SQL constructs**. This method tests to see whether the URI that was passed refers to the whole table or to a single record within the table. If it refers to the whole table, then the code clears the entire sparse array. And if it refers to only a single record, then the code removes that single record from the sparse array. The code returns the number of records that were deleted.

Next, the code implements the **getType** method, which uses the **DataContract** class to return the mime type of the data associated with a given URI.

Next, the code implements the **insert** method, which extracts the data for a given record, puts it in the data record object and then stores the data record object in the sparse array. It returns a new URI that can be used to retrieve this new record.

Next the code implements the **query** method, which begins by creating a **MatrixCursor object**, which is just a simple cursor that will return all of the data associated with each record. If the URI refers to all of the records, the code will insert every record into this cursor. If instead the URI refers to a single record then the code will add only that record to the cursor. The code returns the cursor it just created.

The last method is **onCreate** and normally you would initialize the data storage system here but for this simple example no initialization is necessary - the code simply returns true to indicate that the initialization completed properly.

Before we move on, let's open up the AndroidManifest.xml file for this application.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
     "http://schemas.android.com/apk/res/android"
package="course.examples.ContentProviders.StringContentProvider"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="13"
        android:targetSdkVersion="19" />

    <application
        android:allowBackup="false"
        android:icon="@drawable/icon"
        android:label="@string/app_name" >
        <provider
           android:name="course.examples.ContentProviders.
               StringContentProvider.StringsContentProvider"
           android:authorities="course.examples.
               ContentProviders.StringContentProvider"
            android:exported="true" >
        </provider>
    </application>
</manifest>
```

Here you can see that we've added a provider tag. And within this tag, we've specified the name of the class that implements this content provider. We've added the authority portion of the URI that will have to be used to access this content provider, and we've set the exported attribute to true to allow other applications to access this content provider.

Our last example application is called **ContentProviderCustomUser**. This application is separate from the content provider custom application that we just looked at. However, it reads data from that content provider and displays the records in a ListView.

Let's look at this application in action:

And as you see, the application displays two data records.

Let's look at the source code for this application to see where that data came from. Here's the **ContentProviderCustomUser** application's main activity.

```
package
course.examples.ContentProviders.StringContentProviderUser;

import android.app.ListActivity;
import android.content.ContentResolver;
import android.content.ContentValues;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.widget.SimpleCursorAdapter;
import
     course.examples.ContentProviders.StringContentProvider.
     DataContract;
```

```
public class CustomContactProviderDemo extends ListActivity {

     @SuppressWarnings("unused")
     private static final String TAG =
          "CustomContactProviderDemo";

     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          ContentResolver contentResolver =
               getContentResolver();
          ContentValues values = new ContentValues();

          // Insert first record
          values.put(DataContract.DATA, "Record1");
          Uri firstRecordUri = contentResolver.insert
               (DataContract.CONTENT_URI, values);
          values.clear();

          // Insert second record
          values.put(DataContract.DATA, "Record2");
          contentResolver.insert(DataContract.CONTENT_URI,
               values);
          values.clear();

          // Insert third record
          values.put(DataContract.DATA, "Record3");
          contentResolver.insert(DataContract.CONTENT_URI,
               values);

          // Delete first record
          contentResolver.delete(firstRecordUri, null, null);

          // Create and set cursor and list adapter
          Cursor c = contentResolver.query
               (DataContract.CONTENT_URI, null, null, null,
               null);

          setListAdapter(new SimpleCursorAdapter(this,
               R.layout.list_layout, c,
               DataContract.ALL_COLUMNS, new int[]
               { R.id.idString, R.id.data }, 0));
     }
}
```

The **onCreate** method begins by getting a reference to the content resolver for this context. Next, it creates a **ContentValues** object. After that, the code adds a value for record one. And then calls the **insert** method to insert that record into the content provider. The code then does the same thing for a second, and a third record. And then, it deletes the first record that it had previously inserted, by calling the **delete** method. After all of this, the code then queries the content provider to extract all of the records that are currently stored in the content provider and then it displays those records in a ListView.

So that's all for our lesson on ContentProviders. Please join me next time for a discussion of the last fundamental component of Android, the **Service** class.