eleks®

# Android Location and Sensors APIs

by Vitaliy Ishchuk, Android dev

# Agenda

Location Services

Sensors Overview

Sensors Framework

Motion Sensors

Position Sensors

Environment Sensor

# Location Services

Android gives your applications access to the location services supported by the device through classes in the **android.location** package. The central component of the location framework is the **LocationManager** system service, which provides APIs to determine location and bearing of the underlying device (if available).

As with other system services, you do not instantiate a LocationManager directly. Rather, you request an instance from the system by calling **getSystemService(Context.LOCATION_SERVICE)**. The method returns a handle to a new LocationManager instance.

# Location Services

Once your application has a LocationManager, your application is able to do three things:

- Query for the list of all **LocationProviders** for the last known user location.
- Register/unregister for periodic updates of the user's current location from a location provider (specified either by criteria or name).
- Register/unregister for a given Intent to be fired if the device comes within a given proximity (specified by radius in meters) of a given lat/long.

The **LocationProvider** class is the superclass of the different location providers which deliver the information about the current location. This information is stored in the Location class.

# Location Services

The Android's location APIs use three different providers to get location -

- **LocationManager.GPS_PROVIDER** — This provider determines location using satellites. Depending on conditions, this provider may take a while to return a location fix.
- **LocationManager.NETWORK_PROVIDER** — This provider determines location based on availability of cell tower and WiFi access points. Results are retrieved by means of a network lookup.
- **LocationManager.PASSIVE_PROVIDER** — This provider will return locations generated by other providers. You passively receive location updates when other applications or services request them without actually requesting the locations yourself.

# Challenges in Determining User Location

Some sources of error in the user location include:

- Multitude of location sources - GPS, Cell-ID, and Wi-Fi can each provide a clue to users location. Determining which to use and trust is a matter of trade-offs in accuracy, speed, and battery-efficiency.
- User movement - Because the user location changes, you must account for movement by re-estimating user location every so often.
- Varying accuracy - Location estimates coming from each location source are not consistent in their accuracy. A location obtained 10 seconds ago from one source might be more accurate than the newest location from another or same source.

These problems can make it difficult to obtain a reliable user location reading.

# Requesting Location Updates

Getting user location in Android works by means of callback. You indicate that you'd like to receive location updates from the **LocationManager** by calling requestLocationUpdates(), passing it a **LocationListener**. Your LocationListener must implement several callback methods that the Location Manager calls when the user location changes or when the status of the service changes.

For example, the following code shows how to define a LocationListener and request location updates. **Don`t forget to declare permission:**

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

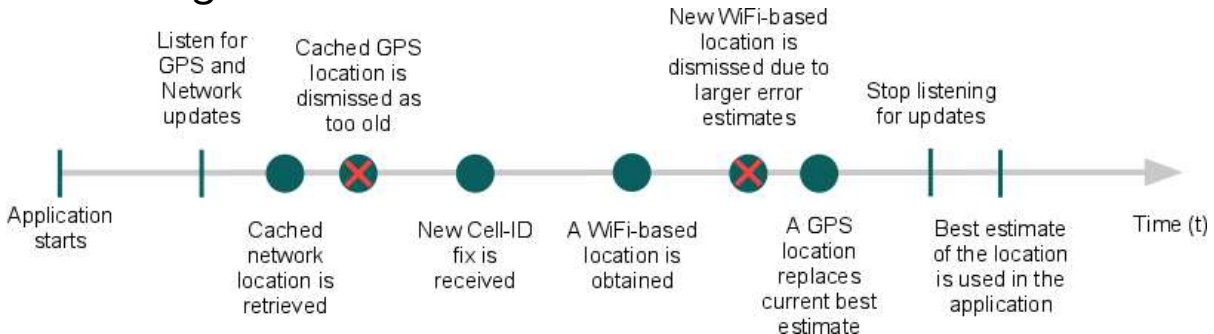
# Requesting Location Updates

```
// Acquire a reference to the system Location Manager
LocationManager locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);

// Define a listener that responds to location updates
LocationListener locationListener = new LocationListener() {

    // Called when the location has changed
    public void onLocationChanged(Location location) {}

    // Called when the provider status changes
    public void onStatusChanged(String provider, int status, Bundle extras) {}

    // Called when the provider is enabled by the user
    public void onProviderEnabled(String provider) {}

    // Called when the provider is disabled by the user
    public void onProviderDisabled(String provider) {}
  };

// Register the listener with the Location Manager to receive location updates
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0, locationListener);
```

# Flow for obtaining user location

Here's the typical flow of procedures for obtaining the user location:

1. Start application.
2. Sometime later, start listening for updates from desired location providers.
3. Maintain a "current best estimate" of location by filtering out new, but less accurate fixes.
4. Stop listening for location updates.
5. Take advantage of the last best location estimate.



# Google Location Services for Android.

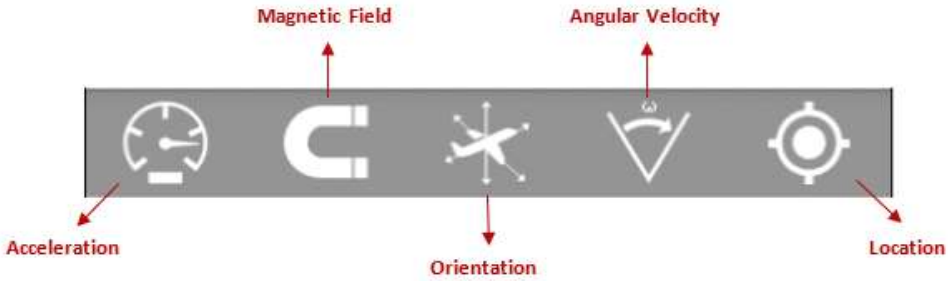
The Google Play services location APIs are preferred over the Android framework location APIs (android.location) as a way of adding location awareness to your app.

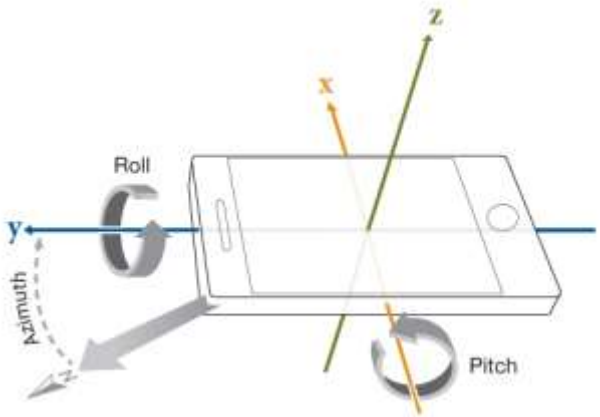
Using the Google Play services location APIs, your app can request the last known location of the user's device. In most cases, you are interested in the user's current location, which is usually equivalent to the last known location of the device.

**Note:** Google Play Services may not be installed on client device.

# Sensors

Most Android-powered devices have built-in sensors that measure motion, orientation, and various environmental conditions. These sensors are capable of providing raw data with high precision and accuracy, and are useful if you want to monitor three-dimensional device movement or positioning, or you want to monitor changes in the ambient environment near a device.



# Sensors

The Android sensor framework lets you access many types of sensors. Some of these sensors are **hardware-based** and some are **software-based**.

Hardware-based sensors are physical components built into a handset or tablet device. They derive their data by directly measuring specific environmental properties, such as acceleration, geomagnetic field strength, or angular change.

Software-based sensors are not physical devices, although they mimic hardware-based sensors. Software-based sensors derive their data from one or more of the hardware-based sensors and are sometimes called virtual sensors or synthetic sensors. The linear acceleration sensor and the gravity sensor are examples of software-based sensors.

# Sensors

The Android platform supports three broad categories of sensors:

- Motion sensors - These sensors measure acceleration forces and rotational forces along three axes. This category includes **accelerometers, gravity sensors, gyroscopes, and rotational vector sensors**.
- Environmental sensors - These sensors measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes **barometers, photometers, and thermometers**.
- Position sensors - These sensors measure the physical position of a device. This category includes **orientation sensors and magnetometers**.

# Sensors

You can access sensors available on the device and acquire raw sensor data by using the Android sensor framework. The sensor framework provides several classes and interfaces that help you perform a wide variety of sensor-related tasks. For example, you can use the sensor framework to do the following:

- Determine which sensors are available on a device.
- Determine an individual sensor's capabilities, such as its maximum range, manufacturer, power requirements, and resolution.
- Acquire raw sensor data and define the minimum rate at which you acquire sensor data.
- Register and unregister sensor event listeners that monitor sensor changes.

# Sensors Framework

You can access these sensors and acquire raw sensor data by using the **Android sensor framework**. The sensor framework is part of the **android.hardware** package and includes the following classes and interfaces:

### SensorManager

Use this class to create an instance of the sensor service. This class provides various methods for accessing and listing sensors, registering and unregistering sensor event listeners, and acquiring orientation information. This class also provides several sensor constants that are used to report sensor accuracy, set data acquisition rates, and calibrate sensors.

### Sensor

Use this class to create an instance of a specific sensor. This class provides various methods that let you determine a sensor's capabilities.

# Sensor Framework

## SensorEvent

The system uses this class to create a sensor event object, which provides information about a sensor event. A sensor event object includes the following information: the raw sensor data, the type of sensor that generated the event, the accuracy of the data, and the timestamp for the event.

## SensorEventListener

Use this interface to create two callback methods that receive notifications (sensor events) when sensor values change or when sensor accuracy changes.

# Sensor Framework

Use these sensor-related APIs to perform two basic tasks:

**Identifying sensors and sensor capabilities**

Identifying sensors and sensor capabilities at runtime is useful if your application has features that rely on specific sensor types or capabilities. For example, you may want to identify all of the sensors that are present on a device and disable any application features that rely on sensors that are not present.

**Monitor sensor events**

Monitoring sensor events is how you acquire raw sensor data. A sensor event occurs every time a sensor detects a change in the parameters it is measuring. A sensor event provides you with four pieces of information: the name of the sensor that triggered the event, the timestamp for the event, the accuracy of the event, and the raw sensor data that triggered the event.

# Sensor Availability

While sensor availability varies from device to device, it can also vary between Android versions. This is because the Android sensors have been introduced over the course of several platform releases. For example, many sensors were introduced in Android 1.5 (API Level 3), but some were not implemented and were not available for use until Android 2.3 (API Level 9). Likewise, several sensors were introduced in Android 2.3 (API Level 9) and Android 4.0 (API Level 14). Two sensors have been deprecated and replaced by newer, better sensors.

```
private SensorManager mSensorManager;
...
mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
List<Sensor> deviceSensors = mSensorManager.getSensorList(Sensor.TYPE_ALL);
```

# Monitoring Sensor Events

To monitor raw sensor data you need to implement two callback methods that are exposed through the SensorEventListener interface:

**onAccuracyChanged() - A sensor's accuracy changes.**

In this case the system invokes the onAccuracyChanged() method, providing you with a reference to the Sensor object that changed and the new accuracy of the sensor.

**onSensorChanged() - A sensor reports a new value.**

In this case the system invokes the onSensorChanged() method, providing you with a SensorEvent object. A SensorEvent object contains information about the new sensor data, including: the accuracy of the data, the sensor that generated the data, the timestamp at which the data was generated, and the new data that the sensor recorded.

# Monitoring Sensor Events

```
public class SensorActivity extends Activity implements SensorEventListener {
  private SensorManager mSensorManager;
  private Sensor mLight;

  @Override
  public final void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    mSensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    mLight = mSensorManager.getDefaultSensor(Sensor.TYPE_LIGHT);
  }

  @Override
  public final void onAccuracyChanged(Sensor sensor, int accuracy) {/* Do something here if sensor accuracy changes */.}

  @Override
  public final void onSensorChanged(SensorEvent event) {
    // The light sensor returns a single value. Many sensors return 3 values, one for each axis.
    float lux = event.values[0];
  }

  @Override
  protected void onResume() {
    super.onResume();
    mSensorManager.registerListener(this, mLight, SensorManager.SENSOR_DELAY_NORMAL);
  }

  @Override
  protected void onPause() {
    super.onPause();
    mSensorManager.unregisterListener(this);
  }
}
```

# Motion Sensors

The Android platform provides several sensors that let you monitor the motion of a device. Two of these sensors are always hardware-based (the accelerometer and gyroscope), and three of these sensors can be either hardware-based or software-based (the gravity, linear acceleration, and rotation vector sensors).

Motion sensors are useful for monitoring device movement, such as tilt, shake, rotation, or swing. The movement is usually a reflection of direct user input (for example play games), but it can also be a reflection of the physical environment in which the device is sitting (for example, moving with you while you drive your car). In the first case, you are monitoring motion relative to the device's frame of reference or your application's frame of reference; in the second case you are monitoring motion relative to the world's frame of reference.

# Using of the Motion Sensors

**Acceleration Sensor** measures the acceleration applied to the device, including the force of gravity. Accelerometer is a good sensor to use if you are monitoring device motion.

**Gravity Sensor** provides a three dimensional vector indicating the direction and magnitude of gravity. Almost the same as an Accelerometer.

**Gyroscope** measures the rate of rotation in rad/s around a device's x, y, and z axis. The sensor's coordinate system is the same as the one used for the acceleration sensor.

**Uncalibrated Gyroscope** is similar to the gyroscope, except that no gyro-drift compensation is applied to the rate of rotation.

**Linear Acceleration Sensor** provides you with a three-dimensional vector representing acceleration along each device axis, **excluding gravity**.

# Using of the Motion Sensors

**Rotation Vector** represents the orientation of the device as a combination of an angle and an axis, in which the device has rotated through an angle θ around an axis (x, y, or z).

**Step Counter Sensor** provides the number of steps taken by the user since the last reboot while the sensor was activated. The step counter has more latency (up to 10 seconds) but more accuracy than the step detector sensor.

**Step Detector Sensor** triggers an event each time the user takes a step. The latency is expected to be below 2 seconds.

# Position Sensors

The Android platform provides two sensors that let you determine the position of a device: the **geomagnetic field sensor** and the **orientation sensor**. The Android platform also provides a sensor that lets you determine **how close the face of a device** is to an object (known as the **proximity sensor**). The geomagnetic field sensor and the proximity sensor are hardware-based. Most handset and tablet manufacturers include a geomagnetic field sensor. Likewise, handset manufacturers usually include a proximity sensor to determine when a handset is being held close to a user's face (for example, during a phone call). **The orientation sensor is software-based and derives its data from the accelerometer and the geomagnetic field sensor.**

**Position sensors** are useful for determining a device's physical position in the world's frame of reference.

# Using of the Position Sensors

**Game Rotation Vector Sensor** is identical to the Rotation Vector Sensor, except it does not use the geomagnetic field. Therefore the Y axis does not point north but instead to some other reference. That reference is allowed to drift by the same order of magnitude as the gyroscope drifts around the Z axis.

**Geomagnetic Rotation Vector Sensor** is similar to the Rotation Vector Sensor, but it uses a magnetometer instead of a gyroscope.

**Orientation Sensor** lets you monitor the position of a device relative to the earth's frame of reference (specifically, magnetic north).

**Geomagnetic Field Sensor** lets you monitor changes in the earth's magnetic field.

**Uncalibrated Magnetometer** is similar to the geomagnetic field sensor, except that no hard iron calibration is applied to the magnetic field.

**Proximity Sensor** lets you determine how far away an object is from a device.

# Environment Sensors

The Android platform provides four sensors that let you monitor various environmental properties. You can use these sensors to monitor relative ambient **humidity**, **illuminance**, **ambient pressure**, and ambient temperature near an Android-powered device. All four environment sensors are hardware-based and are available only if a device manufacturer has built them into a device. With the exception of the light sensor, which most device manufacturers use to control screen brightness, environment sensors are not always available on devices. Because of this, it's particularly important that you verify at runtime whether an environment sensor exists before you attempt to acquire data from it.

Unlike most motion sensors and position sensors, which return a **multi-dimensional array of sensor values** for each SensorEvent, environment sensors **return a single sensor value for each data event.**

# Using of the Environment Sensors

**Light, Pressure, and Temperature Sensors** - The raw data you acquire from the light, pressure, and temperature sensors usually requires **no calibration, filtering, or modification, which makes them some of the easiest sensors to use.**

**Humidity Sensor** - You can acquire raw relative humidity data by using the humidity sensor the same way that you use the light, pressure, and temperature sensors.

eleks

# The end.