EMBEDDED SYSTEMS PROGRAMMING 2017-18

Accessing Hardware: Android

HARDWARE: ANDROID (1/4)

- Access to sensors provided through the packages `android.hardware` & `android.hardware.camera2`

- **`SensorManager`** and **`Sensor`** classes
  Access to accelerometers, magnetometers, gyroscopes, light sensors, temperature sensors, ... No positioning!

- **`Camera`** class (deprecated)
  `CameraManager` and **`CameraDevice`** classes
  Access to camera(s). Require permissions

HARDWARE: ANDROID (2/4)

- Audio recording and playback is possible via classes in the `android.media` package.

- **`MediaRecorder`** class
  Captures audio (& video) from a device.
  Requires permissions

- **`MediaPlayer`** class
  Performs playback of audio (& video) files (& streams)

- **`AudioTrack`** class (Android 1.5+)
  Low-latency playback of PCM audio streams pushed to the class

- **`AudioManager`** class
  Manages audio sources, audio output and volume.
  Requires permissions

HARDWARE: ANDROID (3/4)

- Access to location services provided through the classes in the `android.location` package

- **`LocationManager`** class
  Allows applications to obtain periodic updates of the device's geographical location, or to be notified when the device enters the proximity of a given location.
  Requires permissions

HARDWARE: ANDROID (4/4)

- Battery status monitoring provided via classes in the `android.os` and `java.lang.Object` packages

- Changes in the status of the battery are notified to applications via **intents**

- **`BatteryManager`** class
  Contains constants that describes the attributes of the battery

- **`ApplicationErrorReport.BatteryInfo`** class
  A battery usage report about an application that is consuming too much energy (Android 4.0+)

USING PERMISSIONS (1/4)

- Several hardware resources and system functions cannot be accessed without acquiring **permissions**

- Permissions are asked in the app manifest (**`<uses-permission>`** tag)

- *Normal* (not "risky") permissions are granted automatically

- *Dangerous* permissions are granted in different ways depending on the Android version

USING PERMISSIONS (2/4)

- **API Level ≤ 22** (Android 5.1 and below):
  the system asks the user to grant permissions when the user installs the app. No way to selectively refuse permissions. No way to revoke permissions

- **API Level ≥ 23** (Android 6.0+):
  the app must check every single permission at run time; if the permission is not available, it must be requested from the user. The user can revoke permissions selectively at any time

SENSORMANAGER CLASS: KEY METHODS

- **`List<Sensor> getSensorList(int type)`**
  Returns all available sensors of type `type`

- **`Sensor getDefaultSensor(int type)`**
  Returns the default sensor for the type `type`

- **`boolean registerListener(SensorEventListener listener, Sensor sensor, int rate)`**
  Registers a SensorEventListener for the sensor `sensor`. The rate for change notifications, expressed in microseconds, is given by `rate`

- **`void unregisterListener(SensorEventListener listener, Sensor sensor)`**
  Unregisters a listener for the specified sensor

- **`void unregisterListener(SensorEventListener listener)`**
  Unregisters a listener for all sensors

SENSOR CLASS: SOME HARDWARE SENSOR TYPES

`TYPE_ACCELEROMETER`	Accelerometer
`TYPE_AMBIENT_TEMPERATURE`	Temperature sensor (Android 4.0+)
`TYPE_GRAVITY`	Gravity sensor
`TYPE_GYROSCOPE`	Gyroscope (Android 2.3+)
`TYPE_HEART_RATE`	Heart rate monitor (Android 4.4W+)
`TYPE_LIGHT`	Light sensor
`TYPE_LINEAR_ACCELERATION`	Lin. acceleration sensor (Android 2.3+)
`TYPE_MAGNETIC_FIELD`	Magnetic field sensor
`TYPE_MOTION_DETECT`	Motion detect sensor (Android 7.0+)
`TYPE_PRESSURE`	Pressure sensor
`TYPE_PROXIMITY`	Proximity sensor
`TYPE_RELATIVE_HUMIDITY`	Humidity sensor (Android 4.0+)
`TYPE_ROTATION_VECTOR`	Rotation vector sensor (Android 2.3+)

SENSOR CLASS: KEY METHODS

- **`int getType()`**
  Returns the type of the sensor

- **`float getMaximumRange()`**
  Returns the maximum range of the sensor

- **`float getResolution()`**
  Returns the resolution of the sensor

- **`int getMinDelay()`**
  Returns the minimum delay allowed between two events in microseconds.
  Returns zero if the sensor only returns a value when a change occurs

- **`float getPower()`**
  Returns the power in mA consumed by the sensor while in use

SENSOREVENTLISTENER INTERFACE: METHODS

- **`abstract void onAccuracyChanged (Sensor sensor, int accuracy)`**
  Called when the accuracy of sensor `sensor` has changed.
  The new accuracy is specified in `accuracy`

- **`abstract void onSensorChanged (SensorEvent event)`**
  Called when sensed value has changed.
  All the information about the event are contained in `event`

- The interface may be implemented by an activity

SENSOREVENT CLASS: FIELDS

- **`int accuracy`**
  Accuracy of values reported in the event

- **`Sensor sensor`**
  Sensor that generated the event

- **`long timestamp`**
  Time (expressed in nanoseconds) when the event happened

- **`final float[] values`**
  Values reported by the event.
  The length and contents of the array depends on the sensor type. For instance, if the sensor is an accelerometer three values are reported, corresponding to the accelerations in m/s^2 along the three axes

SENSORMANAGER: USE

1. Obtain an instance of `SensorManager` by calling `Context.getSystemService(SENSOR_SERVICE)`.
   Do not directly instantiate objects from this class!

2. Obtain instances of the desired sensors by calling `getDefaultSensor()` or `GetSensorList()`

3. Register to receive notifications of sensor changes by calling the `registerListener()` method of `SensorManager`.
   Your class must implement the `SensorEventListener` interface

EXAMPLE (1/3)

```
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.util.Log;
import android.widget.TextView;

public class AccLogger extends Activity implements SensorEventListener
{
    final String tag = "AccLogger";
    SensorManager sm = null;
    TextView xAccView = null;
    TextView yAccView = null;
    TextView zAccView = null;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

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

        // Get references to UI objects
        xAccView = (TextView)findViewById(R.id.xbox);
        yAccView = (TextView)findViewById(R.id.ybox);
        zAccView = (TextView)findViewById(R.id.zbox);
    }
                                                                    ...
```

EXAMPLE (2/3)

```
...

    @Override
    protected void onResume()
    {
        super.onResume();
        Sensor Accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

        // register this class as a listener for the accelerometer sensor
        sm.registerListener((SensorEventListener) this, Accel, SensorManager.SENSOR_DELAY_UI);
    }

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

                                                                                    ...
```

EXAMPLE (3/3)

```
...

    public void onSensorChanged(SensorEvent event)
    {
        // Java's synchronized keyword is used to ensure mutually exclusive
        // access to the sensor. See also
        // http://download.oracle.com/javase/tutorial/essential/concurrency/locksync.html
        synchronized(this)
        {
            // The SensorEvent object holds informations such as
            // the sensor's type, the time-stamp, accuracy and of course
            // the sensor's data.
            if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER)
            {
                 // IMPORTANT NOTE: The axes are swapped when the device's
                 // screen orientation changes. To access the unswapped values,
                 // use indices 3, 4 and 5 in values[]
                xAccView.setText("X: " + event.values[0]);
                yAccView.setText("Y: " + event.values[1]);
                zAccView.setText("Z: " + event.values[2]);
            }
        }
    }

    public void onAccuracyChanged(Sensor sensor, int accuracy)
    {
        Log.d(tag,"onAccuracyChanged: " + sensor + ", accuracy: " + accuracy);
    }
```

CAMERA CLASS: KEY METHODS (1/2)

- **`static int getNumberOfCameras()`**
  Returns the number of physical cameras available

- **`static void getCameraInfo(int cameraId, Camera.CameraInfo cameraInfo)`**
  Returns information about a particular camera

- **`static Camera open(int cameraId)`**
  Creates a new `Camera` object to access a particular hardware camera

- **`final void release()`**
  Disconnects and releases the `Camera` object resources

- **`Camera.Parameters getParameters()`**
  Returns the current settings for the `Camera` instance

- **`void setParameters(Camera.Parameters params)`**
  Changes the settings for the `Camera` instance

CAMERA CLASS: KEY METHODS (2/2)

- **`final void setPreviewDisplay(SurfaceHolder holder)`**
  Sets the surface to be used for live preview

- **`final void startPreview()`**
  Starts capturing and drawing preview frames to the screen

- **`final void stopPreview()`**
  Stops capturing and drawing preview frames to the screen

- **`final void takePicture(Camera.ShutterCallback shutter, Camera.PictureCallback raw, Camera.PictureCallback postview, Camera.PictureCallback jpeg)`**
  Starts an asynchronous image capture. The camera service will invoke several callbacks to the application at different stages during the capture process.
  If the application does not need a particular callback, a `null` can be passed as the corresponding parameter

CAMERA: PERMISSIONS (1/3)

- To access a camera, the **CAMERA** permission must be **declared in** `AndroidManifest.xml`

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

- Also add one or more **<uses-feature>** manifest elements to declare the camera features used by the application

```
<uses-feature android:name="android.hardware.camera" />

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

CAMERA: PERMISSIONS (2/3)

- Additionally, in Android 6.0+ permissions must be checked/requested at run time, as appropriate

```
…

protected void onCreate(Bundle savedInstanceState)
{
    …
    if (ActivityCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
            != PackageManager.PERMISSION_GRANTED)
    {
        tv.setText(R.string.perm_denied);
        requestCameraPermission();
    }
}

private void requestCameraPermission()
{
    if (ActivityCompat.shouldShowRequestPermissionRationale(this,
            Manifest.permission.CAMERA))
    {
        // Show dialog to the user explaining why the permission is required
    }

        ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.CAMERA},
                REQUEST_CAMERA_PERMISSION);
}
```

CAMERA: PERMISSIONS (3/3)

```
public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
                                       @NonNull int[] grantResults) {
    if (requestCode == REQUEST_CAMERA_PERMISSION)
    {
        if (grantResults.length != 1 ||
            grantResults[0] != PackageManager.PERMISSION_GRANTED)
        {
            Log.i(TAG, "CAMERA permission has been DENIED.");

            // Handle lack of permission here
        }
        else
        {
            Log.i(TAG, "CAMERA permission has been GRANTED.");

            // You can now access the camera
        }
    }
    else
    {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    }
}
```

CAMERA: USE

1. Obtain an instance of `Camera` by invoking the `Camera.open()` method

2. If necessary: get existing settings with `getParameters()`, modify them and save modifications by calling `setParameters(Camera.Parameters)`

3. Pass a fully initialized `SurfaceHolder` to `setPreviewDisplay(SurfaceHolder)`

4. Call `startPreview()` to start updating the preview surface

5. When appropriate, call `takePicture(...)` to start the capture process. Wait for the callbacks to provide the actual image data

6. To take more photos, call `startPreview()` again

7. When you are done, call `release()`

CAMERA2 PACKAGE (1/2)

- API Level ≥ 21 (Android 5.0+)

- Provides access to all the stages in the image capture pipeline. Multiple capture requests can be in flight at once

- Captures JPEG images and DNG (i.e., RAW) images, if supported by the camera hardware

- Manual control of focus, exposure, white balance, noise reduction, aberration correction, color correction, …

CAMERA2 PACKAGE (2/2)

- **`CameraManager`** class
  Allows to enumerate, query, and open available camera devices

- **`CameraDevice`** class
  Instances represent single hardware cameras.
  To acquire an image, configure a `CameraCaptureSession`, then a `CaptureRequest`, and finally process the `TotalCaptureResult`

- For further details, study the Camera2Basic, Camera2Raw, and Camera2Video sample apps

MEDIARECORDER CLASS

- An instance of `MediaRecorder` can
  - capture both audio and video from any source supported by the device (cameras, mikes, ...),
  - encode media data,
  - save encoded data to the local storage

MEDIARECORDER CLASS: KEY METHODS (1/2)

- **`void setAudioSource(int audio_source)`**
  Sets the audio/video source to be used for recording.
  See `MediaRecorder.AudioSource` for a list of defined audio sources

- **`void setOutputFile(String path)`**
  Sets where the output file should be produced

- **`void setOutputFormat(int output_format)`**
  Sets the format of the output file.
  See `MediaRecorder.OutputFormat` for a list of defined file formats

- **`void setAudioEncoder(int audio_encoder)`**
  Sets the format audio should be encoded to.
  See `MediaRecorder.AudioEncoder` for a list of defined encoders

- **`setVideoSource(...)`** and **`setVideoEncoder(...)`** methods also exist

MEDIARECORDER CLASS: KEY METHODS (2/2)

- **`void prepare()`**
  Prepares the MediaRecorder for recording.
  Returns when the object is ready

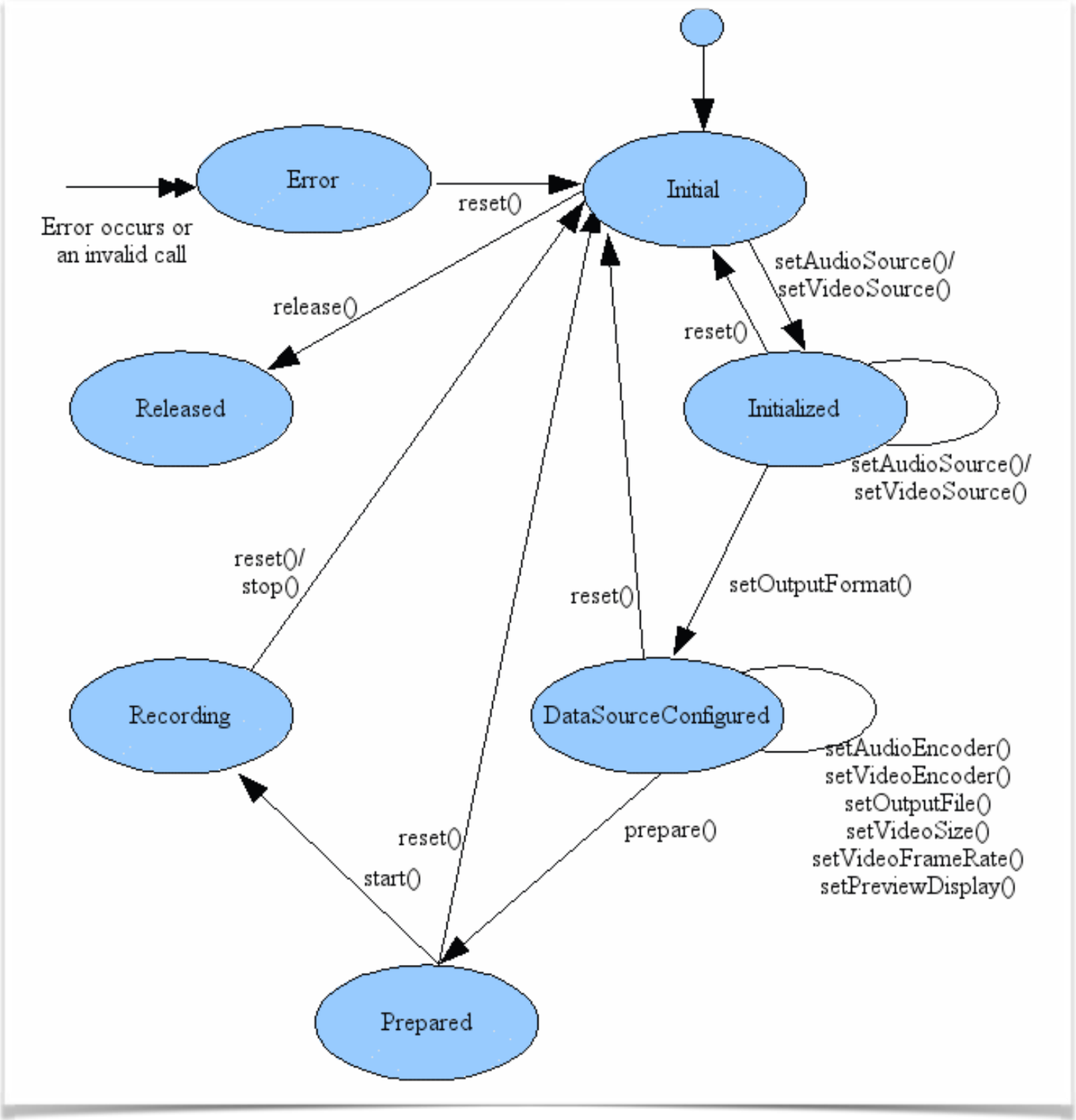
- **`void start()`**
  Starts recording

- **`int getMaxAmplitude()`**
  Returns the maximum audio amplitude sampled since the last call to this method, or 0 if it is the first call

- **`void stop()`**
  Stops recording

- **`void release()`**
  Releases resources associated with the MediaRecorder

MEDIARECORDER: STATES



State diagram from
developer.android.com

MEDIARECORDER: USE

1. Instantiate a MediaRecorder
2. Set the source(s)
3. Set the output file name and format
4. Set the encoder(s)
5. Call `prepare()`
6. Call `start()` to start capture
7. When you are done, call `stop()` and `release()`

MEDIAPLAYER CLASS

- An instance of `MediaPlayer` can
  - fetch data from the local storage or the network (streaming),
  - decode and play both audio and video
- Data should be properly encoded in one of the formats supported by the device

MEDIAPLAYER CLASS: KEY METHODS (1/2)

- **`void setAudioStreamType(int streamtype)`**
  Sets the stream where decoded audio is to be sent

- **`void setDisplay(SurfaceHolder sh)`**
  Sets the surface to be used for video playback

- **`void setDataSource(String path)`**
  `void setDataSource(Context context, Uri uri)`
  Sets the source of media data

- **`prepare()`**
  Prepares the `MediaPlayer` for playback. Returns when the object is ready

- **`start()`**, **`stop()`**
  Starts (resp., stops) playback

- **`release()`**
  Releases resources associated with the MediaPlayer

MEDIAPLAYER.PREPARE

- The call to `prepare()` can take a long time to complete if it involves fetching and/or decoding data

- Do not call `prepare()` from your application's main (i.e., UI) thread: **spawn another thread** that runs it and notifies the main thread when done

- Luckily, the **`prepareAsync()`** method does exactly this

MEDIAPLAYER CLASS: KEY METHODS (2/2)

- **`boolean isPlaying()`**
  Tells whether media data are being played

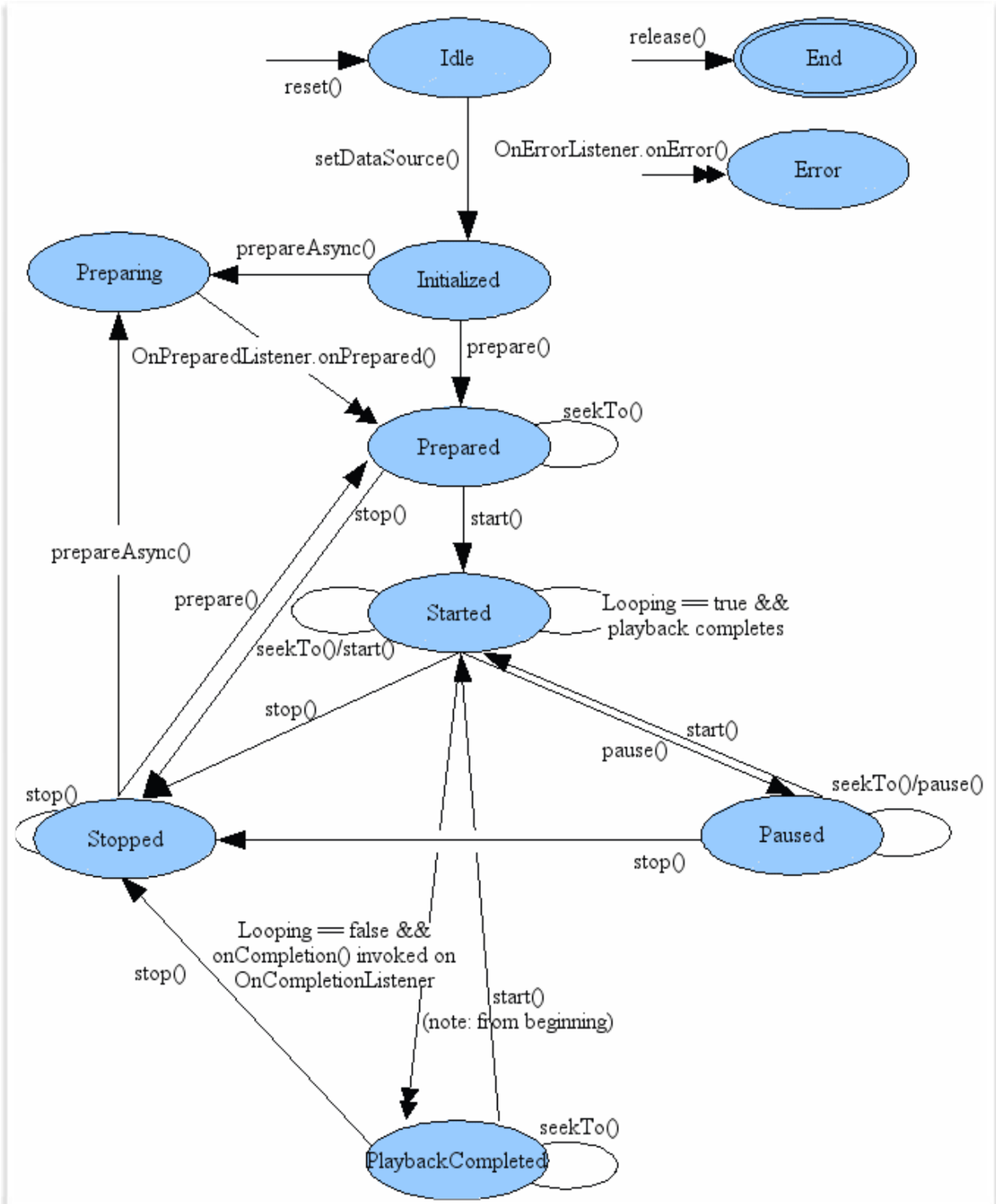
- **`int getDuration()`**
  Gets the duration of the media file in milliseconds

- **`int getCurrentPosition()`**
  Gets the current playback position in milliseconds

- **`void seekTo(int msec)`**
  Seeks to specified time position

- **`void pause()`**
  Pauses playback

MEDIAPLAYER: STATES



State diagram from
developer.android.com

MEDIAPLAYER AND THE APPLICATION LIFECYCLE

- If you do not want to play media in the background and the activity receives a call to `onPause()`, you must call `release()`.
  When your activity is resumed/restarted, a new `MediaPlayer` must be prepared

- If you want to play media in the background, then you should **execute the `MediaPlayer` object inside a service**.
  Remember to properly manage **audio focus**

AUDIOTRACK CLASS

- Playback only
- PCM audio only
- Only data from a memory buffer

- Low-latency playback
- Data can be provided on the fly, e.g., while playback is already in progress

AUDIOTRACK CLASS: MODES

- **Static mode**
  Ensures the smallest latency possible
  The sound must entirely fit into the memory buffer

- **Stream mode**
  New data can be fed to AudioTrack while playback is in progress.
  Can be used if the sound is too big to fit into the memory buffer, or it is not fully available when playback starts

AUDIOTRACK CLASS: KEY METHODS (1 OF 2)

- **`static int getMinBufferSize(int sampleRateInHz, int channelConfig, int audioFormat)`**
  Returns the minimum buffer size required for the successful creation of an AudioTrack object.
  Note: in stream mode the minimum size does not guarantee a smooth playback under load

- **`AudioTrack(int streamType, int sampleRateInHz, int channelConfig, int audioFormat, int bufferSizeInBytes, int mode)`**
  Class constructor.

- **`int write(short[] audioData, int offsetInShorts, int sizeInShorts)`**
  Writes audio data into the memory buffer. In streaming mode, will block until all data has been written. Returns the number of shorts that were written

- **`void flush()`**
  Removes all audio data from the memory buffer

AUDIOTRACK CLASS: KEY METHODS (2 OF 2)

- **`int getPlayState()`**
  Returns the playback state: stopped, paused, or playing

- **`void play()`**
  Starts playback

- **`void pause()`**
  Pauses playback

- **`void stop()`**
  Waits for the memory buffer content to be consumed completely, then stops playback.
  Note: for an immediate stop, use pause(), followed by flush()

- **`int getPlaybackHeadPosition()`**
  Returns the playback position, expressed in samples. This is a continuously advancing counter

- **`int setPlaybackHeadPosition(int positionInFrames)`**
  Sets the playback position, expressed in samples. Works only in static mode

AUDIOMANAGER CLASS: SOME METHODS

- **`void setMode(int mode)`**
  Sets the audio mode (audio routing, including the telephony layer).
  Should be used by applications that need to override the platform-wide management of audio settings

- **`void adjustVolume(int direction, int flags)`**
  Adjusts the volume of the most relevant stream.
  Should be used by applications need to override the platform-wide management of audio settings

- **`boolean isMicrophoneMute()`**
  Checks whether the microphone mute is on or off

- **`boolean isSpeakerphoneOn()`**
  Checks whether the speakerphone is on or off

PERMISSIONS (1/3)

- To record audio, the **`RECORD_AUDIO`** permission must be declared in `AndroidManifest.xml`

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

- Also ask for the **`MODIFY_AUDIO_SETTINGS`** permission if you need to modify global audio settings

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

PERMISSIONS (2/3)

- To record media data on external storage (e.g., to a microSD card), the **WRITE_EXTERNAL_STORAGE** permission must be acquired

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

PERMISSIONS (3/3)

- To keep the screen from dimming or the processor from sleeping during video playback, the **`WAKE_LOCK`** permission must be declared in `AndroidManifest.xml`

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

- **Related methods:** `MediaPlayer.setScreenOnWhilePlaying()`, `MediaPlayer.setWakeMode()`

AUDIO CAPURING & PLAYBACK: EXAMPLES

See

- Develop > Guide > Audio & Video > MediaRecorder
- Develop > Guide > Audio & Video > MediaPlayer
- The “media” portion of the API Demos app

LOCATIONMANAGER CLASS: KEY METHODS

- **`List<String> getAllProviders()`**
  Returns a list of the names of all known location providers

- **`LocationProvider getProvider(String name)`**
  Returns the information associated with the location provider `name`

- **`void requestLocationUpdates(String provider, long minTime, float minDistance, LocationListener listener)`**
  Registers the current activity to be notified periodically by the named provider. Periodically, the supplied `LocationListener` will be called

- **`Location getLastKnownLocation(String provider)`**
  Returns a Location indicating the data from the last known location fix obtained from `provider`. This can be done without starting the provider, hence without consuming battery power

- **`void removeUpdates(LocationListener listener)`**
  Removes any current registration for location updates of the current activity with the given `LocationListener`

LOCATIONLISTENER INTERFACE: METHODS

- **`abstract void onLocationChanged(Location location)`**
  Called when the location has changed

- **`abstract void onProviderDisabled(String provider)`**
  Called when the location provider is disabled by the user

- **`abstract void onProviderEnabled(String provider)`**
  Called when the location provider is enabled by the user

- **`abstract void onStatusChanged(String provider, int status, Bundle extras)`**
  Called when the provider status changes (example: the provider becomes temporarily unavailable because there is no GPS signal)
  `extras` contain provider-specific status variables

LOCATION CLASS: KEY METHODS

- **`double getLatitude()`**
  `double getLongitude()`
  Return the latitude and longitude contained in the `Location` instance (the "fix")

- **`double getAltitude()`**
  Returns the altitude of the fix.
  If `hasAltitude()` is `false`, 0.0 is returned

- **`float getBearing()`**
  Returns the direction of travel in degrees East of true North.
  If `hasBearing()` is `false`, 0.0 is returned

- **`float getSpeed()`**
  Returns the speed of the device over ground, in m/s.
  If `hasSpeed()` is `false`, 0.0 is returned

- **`long getTime()`**
  Returns the UTC time of the fix, in milliseconds since January 1, 1970

LOCATION SERVICES: PERMISSIONS

- To receive location updates, appropriate permissions must be declared in `AndroidManifest.xml`

- To receive location updates from the GPS: **`ACCESS_FINE_LOCATION`** permission

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

- To receive location estimates based on network information: **`ACCESS_COARSE_LOCATION`** permission

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

LOCATIONMANAGER: USE

1. Obtain an instance of `LocationManager` by calling `Context.getSystemService(LOCATION_SERVICE)`.
   Do not directly instantiate objects of this class!

2. Implement a `LocationListener` that responds to location updates

3. Register the listener by calling `LocationManager`'s method `requestLocationUpdates(...)`

4. When you are done, call `removeUpdates(...)`

BATTERY: INTENTS (1/2)

- Relevant battery state changes are broadcast by Android via intents called **implicit broadcasts**. Such Intents are defined as constants in the `Intent` class.
- **`public static final String ACTION_BATTERY_CHANGED`**
  Contains the charging state, level, and other info about the battery
- **`public static final String ACTION_BATTERY_LOW`**
  Indicates the battery is running low.
  A "Low battery warning" system dialog is also shown to the user
- **`public static final String ACTION_BATTERY_OKAY`**
  Indicates the battery is no longer low
- **`public static final String ACTION_POWER_DISCONNECTED`**
  External power has been removed : the device is now running on batteries
- **`public static final String ACTION_POWER_CONNECTED`**
  External power has been connected: the device is no longer running on batteries

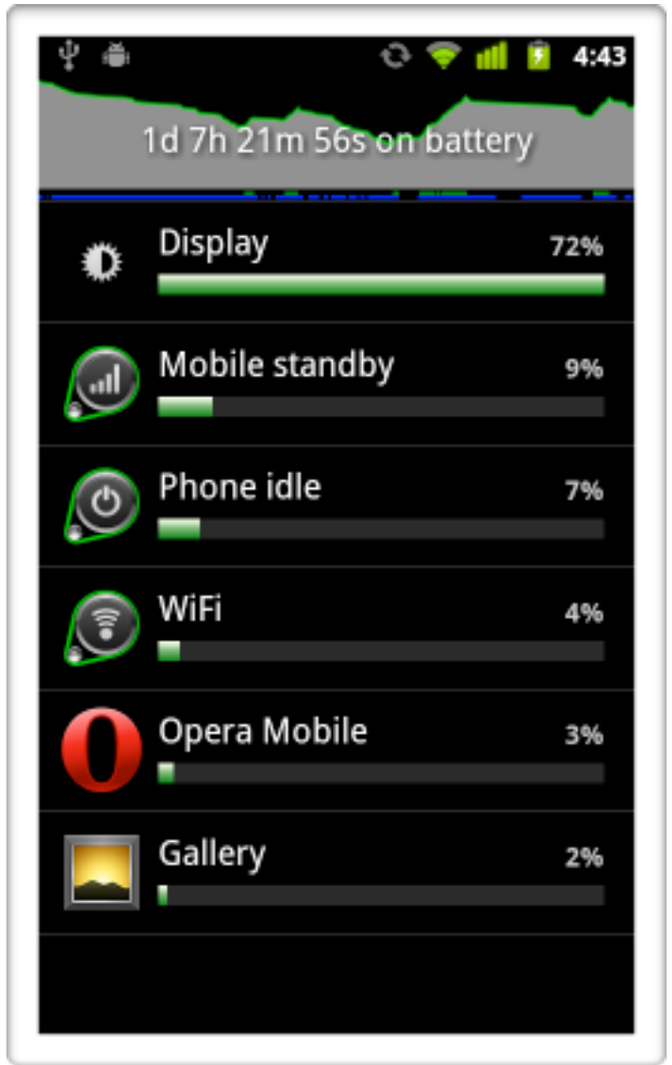
BATTERY: INTENTS (2/2)

- The **`ACTION_BATTERY_CHANGED`** sticky intent must be received by registering a **broadcast receiver**.
  Android 8.0+: you must register for the intent at runtime, not in the manifest

- All other intents can be also received through application components declared in the manifest

POWER USAGE SUMMARY

- To show power usage information to the user, there is a system activity that can be launched with the intent **ACTION_POWER_USAGE_SUMMARY**



BATTERYMANAGER CLASS: KEY CONSTANTS & STRINGS

- Integer constants for the extended intent datum **`EXTRA_HEALTH`**:
  `BATTERY_HEALTH_COLD` (Android 3.0+), `BATTERY_HEALTH_DEAD`, `BATTERY_HEALTH_GOOD`, `BATTERY_HEALTH_OVERHEAT`, `BATTERY_HEALTH_OVER_VOLTAGE`, `BATTERY_HEALTH_UNKNOWN`, `BATTERY_HEALTH_UNSPECIFIED_FAILURE`
- Integer constants for the extended intent datum **`EXTRA_PLUGGED`**:
  `BATTERY_PLUGGED_AC`, `BATTERY_PLUGGED_USB`
- Integer constants for the extended intent datum **`EXTRA_STATUS`**:
  `BATTERY_STATUS_CHARGING`, `BATTERY_STATUS_DISCHARGING`, `BATTERY_STATUS_FULL`, `BATTERY_STATUS_NOT_CHARGING`, `BATTERY_STATUS_UNKNOWN`
- The extended intent datum **`EXTRA_LEVEL`** is an integer that contains the current battery level (from 0 to `EXTRA_SCALE`)

EXAMPLE

- This example implements and registers a receiver for the `ACTION_BATTERY_CHANGED` intent

```
public class BatteryActivity extends Activity
{
    private BroadcastReceiver myBatteryReceiver = new BroadcastReceiver()
    {
        public void onReceive(Context arg0, Intent arg1)
        {
            if(arg1.getAction().equals(Intent.ACTION_BATTERY_CHANGED))
            {
                int status = arg1.getIntExtra(BatteryManager.EXTRA_STATUS,
                                              BatteryManager.BATTERY_STATUS_UNKNOWN);

                if(status == BatteryManager.BATTERY_STATUS_DISCHARGING)
                {
                    int level  = arg1.getIntExtra(BatteryManager.EXTRA_LEVEL, 0);
                    ... // manages the fact that the app is running on batteries
                }
            }
        }
    };

    public void onCreate(Bundle savedInstanceState)
    {
        ...
        this.registerReceiver(this.myBatteryReceiver,
          new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
    }
}
```

LAST MODIFIED: APRIL 13, 2018