Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Android Development

Camera, Media

Playing media

Media

- Several different capabilities
  - DTMF generator
  - Ringtone Player and Manager
  - Face Detector
  - EXIF Interface
  - Async and Jet Player
  - Media Metadata Retriever
- Most important classes for us
  - MediaPlayer
  - MediaRecorder
- The API is changing continuously
  - Always check the new capabilities
- Also check the AudioManager class

Supported formats

- Network protocols
  - RTSP (RTP, SDP)
  - HTTP progressive streaming
  - HTTP live streaming (API level 11 and above)
- Formats

Format / Codec	Encoder	Decoder	Container Formats
BMP		•	BMP (.bmp)
GIF		•	GIF (.gif)
JPEG	•	•	JPEG (.jpg)
PNG	•	•	PNG (.png)
WebP	Android 4.0+ Lossless Android 10+	Android 4.0+ Android 4.2.1+	WebP (.webp)
HEIF		(Android 8.0+)	HEIF (.heic; .heif)

Supported formats – Audio

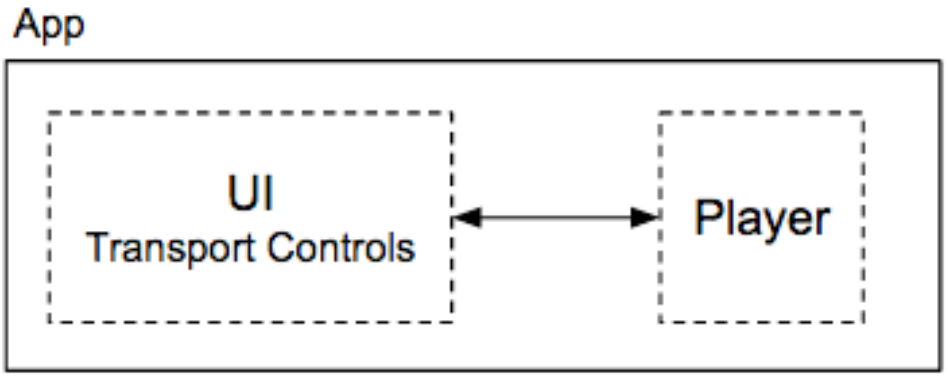
Format / Codec	Encoder	Decoder	Supported File Type(s) / Container Formats
AAC LC	•	•	• 3GPP (.3gp) • MPEG-4 (.mp4, .m4a) • ADTS raw AAC (.aac, decode in Android 3.1+, encode in Android 4.0+, ADIF not supported) • MPEG-TS (.ts, not seekable, Android 3.0+)
HE-AACv1 (AAC+)	(Android 4.1+)	•	
HE-AACv2 (enhanced AAC+)		•	
AAC ELD (enhanced low delay AAC)	(Android 4.1+)	(Android 4.1+)	
AMR-NB	•	•	3GPP (.3gp)
AMR-WB	•	•	3GPP (.3gp)
FLAC	(Android 4.1+)	(Android 3.1+)	FLAC (.flac) only
GSM		•	GSM(.gsm)
MIDI		•	• Type 0 and 1 (.mid, .xmf, .mxmf) • RTTTL/RTX (.rtttl, .rtx) • OTA (.ota) • iMelody (.imy)
MP3		•	MP3 (.mp3)
Opus		(Android 5.0+)	Matroska (.mkv)
PCM/WAVE	(Android 4.1+)	•	WAVE (.wav)
Vorbis		•	• Ogg (.ogg) • Matroska (.mkv, Android 4.0+)

Supported formats – Video

Format / Codec	Encoder	Decoder	Container Formats
H.263	•	•	• 3GPP (.3gp) • MPEG-4 (.mp4)
H.264 AVC Baseline Profile (BP)	(Android 3.0+)	•	• 3GPP (.3gp) • MPEG-4 (.mp4) • MPEG-TS (.ts, AAC audio only, not seekable, Android 3.0+)
H.264 AVC Main Profile (MP)	(Android 6.0+)	•	
H.265 HEVC		(Android 5.0+)	MPEG-4 (.mp4)
MPEG-4 SP		•	3GPP (.3gp)
VP8	(Android 4.3+)	(Android 2.3.3+)	• WebM (.webm) • Matroska (.mkv, Android 4.0+)
VP9		(Android 4.4+)	• WebM (.webm) • Matroska (.mkv, Android 4.0+)

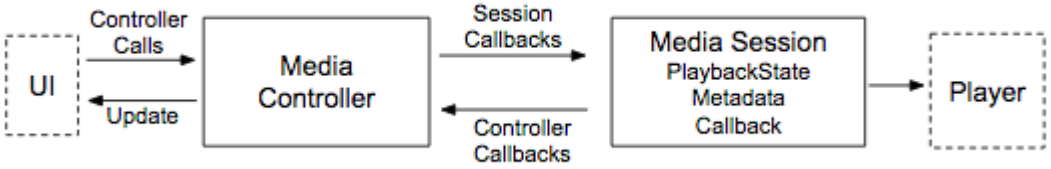
Media app architecture

- Multimedia application that plays audio or video usually has two parts:
  - A player that takes digital media in and renders it as video and/or audio
  - A UI with transport controls to run the player and optionally display the player's state



Media session and controller

- The Android framework defines two classes, a media session and a media controller, that impose a well-defined structure for building a media player app.
- The media session and media controller communicate with each other using predefined callbacks
  - That correspond to standard player actions
  - Extensible for our app



Options

- In Android you can build your own player from the ground up
- Or you can choose from these options:
  - The MediaPlayer class provides the basic functionality for a bare-bones player that supports the most common audio/video formats and data sources.
  - ExoPlayer is an open source library that exposes the lower-level Android audio APIs
    - ExoPlayer supports high-performance features like DASH and HLS streaming that are not available in MediaPlayer.
    - You can customize the ExoPlayer code, making it easy to add new components.
    - ExoPlayer can only be used with Android version 4.1 and higher.

Before start

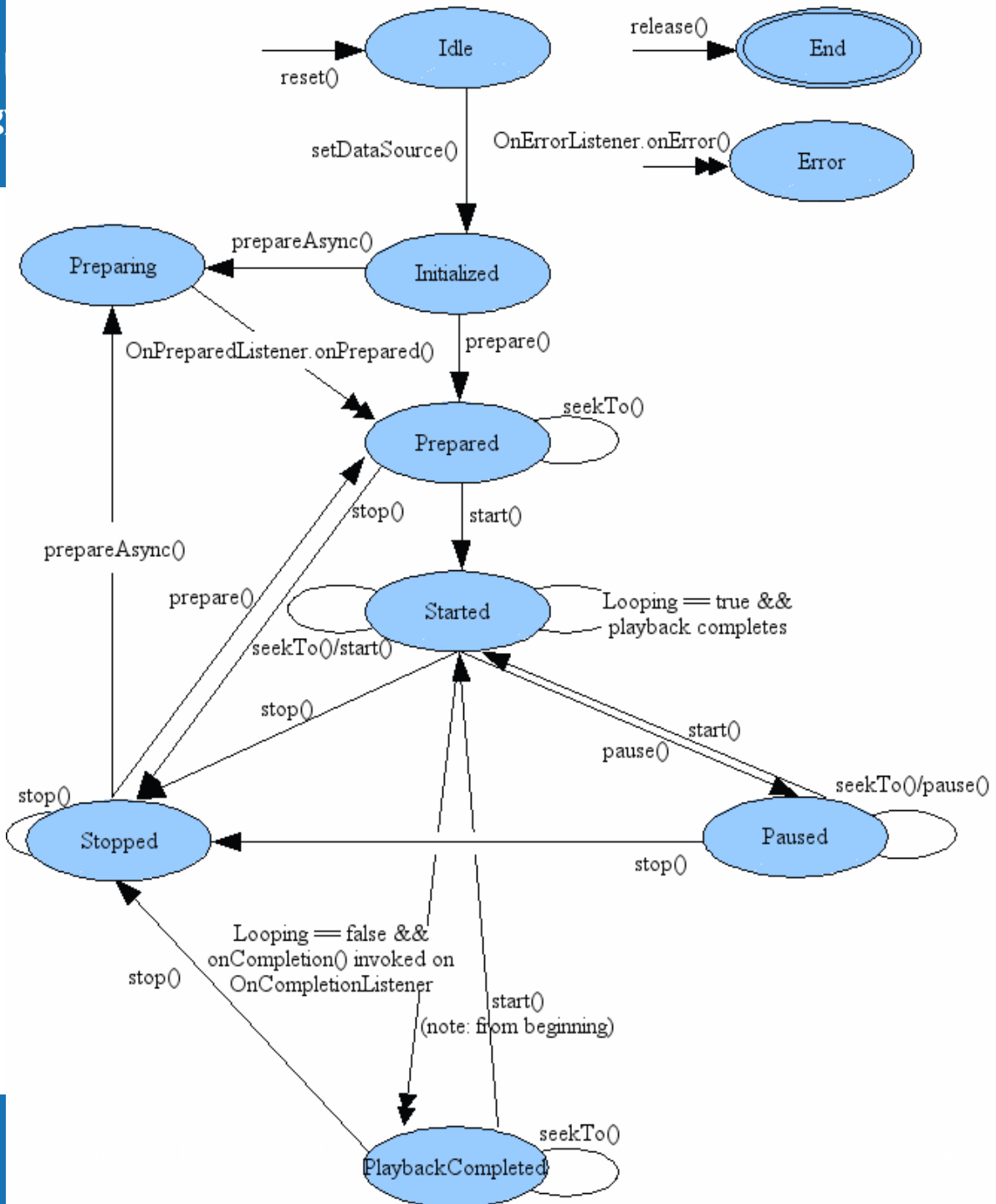
- In manifest the appropriate declarations has to be set to allow use of related features.
- Internet Permission
  - If you are using MediaPlayer to stream network-based content, your application must request network access.
  - <uses-permission android:name="android.permission.INTERNET" />
- Wake Lock Permission
  - To keep the screen from dimming or the processor from sleeping, or uses the MediaPlayer.setScreenOnWhilePlaying() or MediaPlayer.setWakeMode() methods.
  - <uses-permission android:name="android.permission.WAKE_LOCK" />

android.media.MediaPlayer

- To play Audio/Video streams on the mobile device
- Source can be either
  - Local resource
  - Internal URI (E.g. using the Content Resolver)
  - External URL

MediaPlayer

- State diagram
- Note that there are a
  - prepared state
  - preparing state
- That is the phase when the player is buffering, …



`MediaPlayer` functions

- Important ones:
  - `create(…)`
    - To initialize a instance of MediaPlayer
  - `prepare(), prepareAsync()`
    - To prepare the playing (buffering)
      - Synchronized (blocks the main thread)
      - Preparing on a separate thread
  - `release()`
    - To free up the locked resource
  - `start() stop() pause() seekto(int milli)`
    - To control the player
  - `reset()`
    - To reset the MP, it can be initialized again

`MediaPlayer` functions

- Additional important functions
  - `setVolume (float leftVolume, float rightVolume)`
    - Stereo volume
  - `setLooping(boolean)`
    - Set true to repeat the playback
  - `setDisplay (SurfaceHolder sh)`
    - In case of video, to set the View where the video will appear
  - `setDataSource (…)`
    - To specify the data source to be played
  - `setAudioStreamType (int streamtype)`
    - The pre-defined types can be found as final variables of the AudioManager

`MediaPlayer` functions

- Listeners
  - `setOnBufferingUpdateListener(MediaPlayer.OnBufferingUpdateListener listener)`
  - `setOnCompletionListener(MediaPlayer.OnCompletionListener listener)`
  - `setOnErrorListener(MediaPlayer.OnErrorListener listener)`
  - `setOnInfoListener(MediaPlayer.OnInfoListener listener)`
  - `setOnPreparedListener(MediaPlayer.OnPreparedListener listener)`
  - `setOnSeekCompleteListener(MediaPlayer.OnSeekCompleteListener listener)`
  - `setOnVideoSizeChangedListener(MediaPlayer.OnVideoSizeChangedListener listener)`

`MediaPlayer` functions

- Additional query functions
  - `getCurrentPosition()`
  - `getDuration()`
  - `getVideoHeight()`
  - `getVideoWidth()`
  - `isLooping()`
  - `isPlaying()`

Playing local resource

```
var mediaPlayer: MediaPlayer? =
    MediaPlayer.create(context, R.raw.sound_file_1)
mediaPlayer?.start()
    // a create() meghívja a prepare()-t

…

mediaPlayer?.release()
mediaPlayer = null
```

Playing a local URI

```
val myUri: Uri = .... // initialize Uri here
val mediaPlayer: MediaPlayer? = MediaPlayer().apply {
    setAudioStreamType(AudioManager.STREAM_MUSIC)
    setDataSource(applicationContext, myUri)
    prepare()
    start()
}

…

mediaPlayer?.release()
mediaPlayer = null
```

Play remote URL

```
val url = "http://........." // your URL here
val mediaPlayer: MediaPlayer? = MediaPlayer().apply {
    setAudioStreamType(AudioManager.STREAM_MUSIC)
    setDataSource(url)
    prepare() // might take long! (for buffering, etc)
    start()
}

…

mediaPlayer?.release()
mediaPlayer = null
```

Asynchronous call

```
private const val ACTION_PLAY: String = "com.example.action.PLAY"
class MyService: Service(), MediaPlayer.OnPreparedListener {
  private var mMediaPlayer: MediaPlayer? = null
  override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
      val action: String = intent.action
      when(action) {
          ACTION_PLAY -> {
              mMediaPlayer = ... // initialize it here
              mMediaPlayer?.apply {
                  setOnPreparedListener(this@MyService)
                  prepareAsync()
              }
          }
      }
  }
  override fun onPrepared(mediaPlayer: MediaPlayer) {
      mediaPlayer.start()
  }
}
```

Further possibilities

- WIFI Lock
  - To ensure continuous download
- PowerManagement Lock
  - To ensure continuous playback in background
- Audio focus
  - Managing the access to the audio devices
  - Ringing the phone (in case of a call) has the priority
- Service in foreground
- Using media providers
- http://developer.android.com/guide/topics/media/index.html

WifiLock

- If you are streaming media over the network and you are using Wi-Fi, you probably want to hold a WifiLock:

```
val wifiManager = getSystemService(Context.WIFI_SERVICE) as WifiManager
val wifiLock: WifiManager.WifiLock =
    wifiManager.createWifiLock(WifiManager.WIFI_MODE_FULL, "mylock")

wifiLock.acquire()
…
wifiLock.release()
```

PowerManager

- When designing applications that play media in the background, the device may go to sleep while your service is running.
- If your service is playing or streaming music, you want to prevent the system from interfering with your playback.

```
mediaPlayer = MediaPlayer().apply {
 setWakeMode(applicationContext, PowerManager.PARTIAL_WAKE_LOCK)
}
```

ExoPlayer

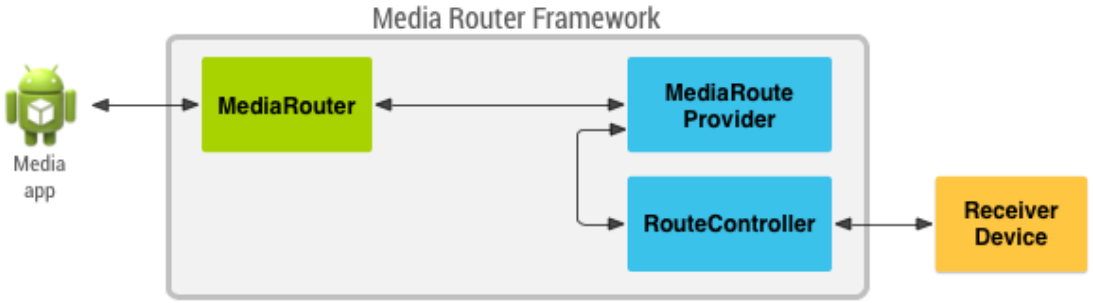
- ExoPlayer is an open source project that is not part of the Android framework and is distributed separately from the Android SDK.
- ExoPlayer's standard audio and video components are built on Android's MediaCodec API, which was released in Android 4.1 (API level 16).
- Because ExoPlayer is a library, you can easily take advantage of new features as they become available by updating your app.
- ExoPlayer supports features like Dynamic adaptive streaming over HTTP (DASH), SmoothStreaming and Common Encryption, which are not supported by MediaPlayer.

MediaRouter

- Users connect their televisions, home theater systems, and music players with wireless technologies.
  - They want to be able to play content from Android apps that devices.
  - The Android media router APIs are designed to enable media display and playback on remote receiver devices using a common user interface.
  - App developers that implement a MediaRouter interface can then connect to the framework and play content to devices that participate in the media router framework.

MediaRouter

- Concept



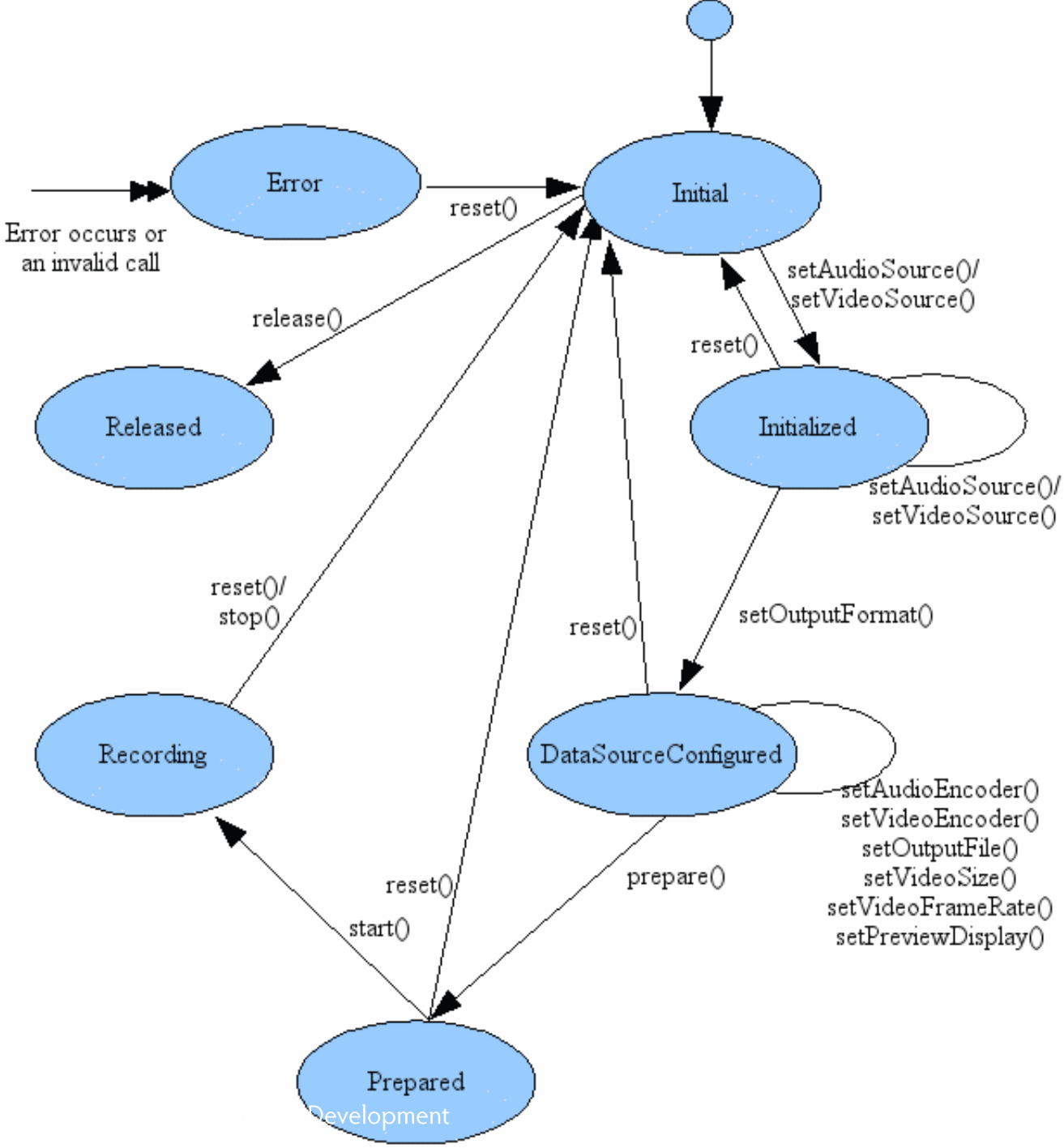
- MediaRouter API
  - A media app uses the MediaRouter API to discover available remote playback devices and to route audio and video to them.
- MediaRouteProvider API
  - The MediaRouteProvider API defines the capabilities of a remote playback device and makes it visible to apps that use a MediaRouter to search for alternative media paths.

android.media.MediaRecorder

- MediaRecorder can be used to recorded audio or video
  - Audio source is the microphone
  - Video source can be one of the cameras
- The logic is like the MediaPlayer

States

- Specify
  - Source
  - Format
  - Size
  - File
  - Preview surface



MediaRecorder – example

```
MediaRecorder recorder = new MediaRecorder()
 recorder.setAudioSource(MediaRecorder.AudioSource.MIC)
 recorder.setOutputFormat(
       MediaRecorder.OutputFormat.THREE_GPP)
 recorder.setAudioEncoder(
       MediaRecorder.AudioEncoder.AMR_NB)
 recorder.setOutputFile(PATH_NAME)
 recorder.prepare()
 recorder.start()   // Recording

 ...
 recorder.stop()
 recorder.reset()   // Can be reinitialized
 recorder.release() // End of work
```

`MediaRecorder` functions

- Most important
  - `setAudioSource(), setVideoSource()`
    - At least one of them must be called
    - The possibilities are defined as final variables in inner classes
  - `prepare()`
    - Prepare the MR to record
  - `release()`
    - Release the resources
  - `start() stop()`
    - Controls
  - `reset()`
    - Reset the MR to uninitialized state

`MediaRecorder` functions

- Important audio functions
  - `setAudioChannels (int numChannels)`
  - `setAudioEncoder(int audio_encoder)`
  - `setAudioEncodingBitRate(int bitRate)`
  - `setAudioSamplingRate(int samplingRate)`
- Important video functions
  - `setCamera(Camera c)`
  - `setCaptureRate(double fps)`
    - Capturing frame rate
  - `setPreviewDisplay(Surface sv)`
  - `setProfile(CamcorderProfile profile)`
  - `setVideoEncoder(int video_encoder)`
  - `setVideoEncodingBitRate(int bitRate)`
  - `setVideoFrameRate(int rate)`
    - Frame rate of the saved file
  - `setVideoSize(int width, int height)`

`MediaRecorder` functions

- Important video functions
  - Camcorder contains several predefined profiles
  - Examples

	Lower quality	Higher quality
Video codec	H.264 Baseline Profile	H.264 Baseline Profile
Video resolution	176 x 144 px	480 x 360 px
Video frame rate	12 fps	30 fps
Video bitrate	56 Kbps	500 Kbps
Audio codec	AAC-LC	AAC-LC
Audio channels	1 (mono)	2 (stereo)
Audio bitrate	24 Kbps	128 Kbps

  - Others: `QUALITY_1080P`, `QUALITY_2160P`, `QUALITY_480P`, `QUALITY_720P`, `QUALITY_CIF`, `QUALITY_HIGH`, `QUALITY_LOW`, `QUALITY_QCIF`, `QUALITY_QVGA`
  - Most of them can be used in time-elapse videos.

MediaRecorder

- Additional functions
  - `setOutputFormat(int output_format)`
  - `setOutputFile(String path)`
  - `setOutputFile(FileDescriptor fd)`
  - `setMaxFileSize(long max_filesize_bytes)`
    - The maximal file size in bytes
    - Zero means infinite
  - `setMaxDuration(int max_duration_ms)`
    - Zero and negative values means infinite
  - `setAuxiliaryOutputFile (String path)`
  - `setAuxiliaryOutputFile (FileDescriptor fd)`
    - Additional, low resolutions file

MediaRecorder

- Listeners
  - `setOnInfoListener(MediaRecorder.OnInfoListener listener)`
  - `setOnErrorListener(MediaRecorder.OnErrorListener l)`
- Other
  - `getMaxAmplitude()`
  - `setOrientationHint(int degrees)`
    - Speficty the orientation for playback
    - 0, 90, 180, 270

Example

```
class AudioRecordTest : AppCompatActivity() {

    private var fileName: String = ""

    private var recordButton: RecordButton? = null
    private var recorder: MediaRecorder? = null

    private var playButton: PlayButton? = null
    private var player: MediaPlayer? = null
    private fun onRecord(start: Boolean) = if (start) {
        startRecording()
    } else {
        stopRecording()
    }

    private fun onPlay(start: Boolean) = if (start) {
        startPlaying()
    } else {
        stopPlaying()
    }
```

Example

```
private fun startPlaying() {
    player = MediaPlayer().apply {
        try {
            setDataSource(fileName)
            prepare()
            start()
        } catch (e: IOException) {
            Log.e(LOG_TAG, "prepare() failed")
        }
    }
}

private fun stopPlaying() {
    player?.release()
    player = null
}
```

Example

```
private fun startRecording() {
    recorder = MediaRecorder().apply {
        setAudioSource(MediaRecorder.AudioSource.MIC)
        setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP)
        setOutputFile(fileName)
        setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB)

        try {
            prepare()
        } catch (e: IOException) {
            Log.e(LOG_TAG, "prepare() failed")
        }

        start()
    }
}

private fun stopRecording() {
    recorder?.apply {
        stop()
        release()
    }
    recorder = null
}
```

Measuring performance

- As of Android 8.0, the `getMetrics()` method is available for some media classes.
  - It returns a `PersistableBundle` object containing configuration and performance information, expressed as a map of attributes and values.
  - The `getMetrics()` method is defined for these media classes:
    - `MediaPlayer.getMetrics()`
    - `MediaRecorder.getMetrics()`
    - `MediaCodec.getMetrics()`
    - `MediaExtractor.getMetrics()`
  - Metrics are collected separately for each instance and persist for the lifetime of the instance.
    - If no metrics are available the method returns null.
    - The actual metrics returned depend on the class.

Direct access to Camera

Consider

- Before enabling your application to use cameras on Android devices, you should consider a few questions about how your app intends to use this hardware feature.
  - **Camera Requirement**
    - Is the use of a camera so important to your application that you do not want your application installed on a device that does not have a camera?
  - **Quick Picture or Customized Camera**
    - How will your application use the camera?
    - Are you just interested in snapping a quick picture or video clip, or will your application provide a new way to use cameras?
    - For a getting a quick snap or clip, consider using the existing apps, otherwise you can build a customized camera application

Consider

- Before enabling your application to use cameras on Android devices, you should consider a few questions about how your app intends to use this hardware feature.
  - **Foreground Services Requirement**
    - When does your app interact with the camera?
    - On Android 9 (API level 28) and later, apps running in the background cannot access the camera. Therefore, you should use the camera either when your app is in the foreground or as part of a foreground service.
  - **Storage**
    - Are the images or videos your application generates intended to be only visible to your application or shared so that other applications such as Gallery
    - or other media and social apps can use them?
    - Do you want the pictures and videos to be available even if your application is uninstalled?

Asking for pictures

Simplest way – using builtin apps

Using Intents

```
private void dispatchTakePictureIntent(int actionCode) {
    Intent takePictureIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivityForResult(takePictureIntent, actionCode);
}
```

Check the existence of the activity

```
public static boolean isIntentAvailable(Context context, String action) {
    final PackageManager packageManager = context.getPackageManager();
    final Intent intent = new Intent(action);
    List<ResolveInfo> list =
     packageManager.queryIntentActivities(intent, PackageManager.MATCH_DEFAULT_ONLY);
    return list.size() > 0;
}
```

Using Intents

- The resulting image is in a Bitmap
  - The call of `startActivityForResult(…)` returns an Intent
  - In this Intent
    - `Bundle extras = intent.getExtras();`
    - `mImageBitmap = (Bitmap) extras.get("data");`
- Saving the picture
  - ```
    takePictureIntent.putExtra(
        MediaStore.EXTRA_OUTPUT, Uri.fromFile(f));
    ```
    ```

Adding the picture to the gallery

```
private void galleryAddPic() {
    Intent mediaScanIntent = new
              Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
    File f = new File(mCurrentPhotoPath);
    Uri contentUri = Uri.fromFile(f);
    mediaScanIntent.setData(contentUri);
    this.sendBroadcast(mediaScanIntent);
}
```

Camera API

Simpler way – deprecated since API Level 21

Camera API

- The built-in camera can be used to record still images and videos as well
  - Restrictions
    - Format
    - Physical (optical) properties
      - Miracles are not possible
    - Sensor properties
- Easy to use API
- To access the camera the required permissions have to be declared
  - ```
    <uses-permission
    android:name="android.permission.CAMERA" />
    <uses-feature android:name="android.hardware.camera" />
    <uses-feature
    android:name="android.hardware.camera.autofocus" />
    ```
    ```

Classes

- `android.hardware.Camera`
  - Most important class, it is used to acces still images and preview
  - It is a client to camera services
  - Important to turn on and off in order to save battery power
  - It is not a thread safe class
    - Thus single thread access is required
    - Callback functions are used to retrieve information
- Several inner class are defined
  - For callback interfaces
  - Parameters and properties

Camera – Taking picture

1. Instantiate: `open(int)`
2. Retrieve parameters: `P = getParameters()`
3. Change if necessary: `setParameters(P)`
4. Set if necessary: `setDisplayOrientation(int)`
5. Provide a `SurfaceHolder` to show the previews: `setPreviewDisplay(SFH)` – IMPORTANT
6. `startPreview()` call – MANDATORY
7. Take a picture: `takePicture(…)`, image arrives with callback
8. Restart preview to the new picture
9. …
10. `stopPreview()`
11. `release()` – it is required to call in `onPause()`

Camera

- There may be several camera built in the device
  - The `open()` function returns with a primary camera of the background of the device
  - The `open(int)` function returns the specified camera
  - Properties
    - `getNumberOfCameras()`
    - `getCameraInfo(int, Camera.CameraInfo)`
- Paramteres of the opened camera
  - `getParameters(), setParameters()`
- Further calls
  - `lock()` – to have exclusive access
  - `unlock()` – to release the lock
  - `reconnect()`
  - `release()` – final call

Camera

- Preview
  - Set the view
    - `setPreviewDisplay(SurfaceHolder holder)`
    - The `Surface` is a graphical object where the camera image can be drawn on
      - Abstract
    - An instance can be retrieved by using the `SurfaceView` class
  - Texture
    - `setPreviewTexture(SurfaceTexture surfaceTexture)`
    - For example transformation can be defined
  - To start and stop
    - `startPreview()`
    - `stopPreview()`

Camera

- `PreviewCallback`
  - An interface which have to be implemented to receive the preview frames
  - `setPreviewCallback(Camera.PreviewCallback cb)`
  - `setPreviewCallbackWithBuffer(Camera.PreviewCallback cb)`
    - `addCallbackBuffer(byte[] callbackBuffer)`
  - `setOneShotPreviewCallback(Camera.PreviewCallback)`
    - For the next frame
- Only one callback implementation can be set
  - To unset previous one, set null callback

Camera

- Automatic focus
  - `autoFocus(Camera.AutoFocusCallback cb)`
    - To start the automatic focusing procedure
    - The callback is an interface which have to be implemented
    - It is being called when the procedure is finished (successful and unsuccessful as well)
  - `cancelAutoFocus()`
- `ErrorCallback`
  - `setErrorCallback(Camera.ErrorCallback cb)`
    - Notification about different errors

Information

- `Camera.CameraInfo`
  - Information about the location and orientation of the camera
- `Camera.CameraSize`
  - Possible images size
- `Camera.CameraParameters`
  - Effects, Focusing method, Flashlights, 50/60 Hz filtering
  - Preview FPS, Implemented camera profiles (Interior, Exterior, etc …)
  - White balance, EV, Image format, Image size
  - Image quality, Zoom information, EXIF data (GPS, Timestamp)
  - Orientation of the image (Portrait/Landscape)

Taking picture

- `takePicture(Camera.ShutterCallback shutter, Camera.PictureCallback raw, Camera.PictureCallback postview, Camera.PictureCallback jpeg)`
  - The procedure is done asynchronously
  - Shutter
    - Starting the shutter – e.g. a sound can be played
  - Raw
    - If the memory amount is sufficient, the raw data is accessible
  - Postview
    - The fully processed image can be accessed – not on all hardware
  - Jpeg
    - Final image in JPG, with EXIF data
  - All callback functions has byte[] as parameters

Record movie – Using Camera

1. Instantiate: `open(int)`
2. Retrieve camera parameters: `P = getParameters()`
3. Change if it is necessary: `setParameters(P)`
4. If necessary: `setDisplayOrientation(int)`
5. Provide a `SurfaceHolder` to show the previews: `setPreviewDisplay(SFH)` – IMPORTANT
6. `startPreview()` call – MANDATORY
7. `unlock()`
8. `setCamera(Camera)`
9. `reconnect()`
10. ….
11. `stopPreview()`
12. `release()` – it is required to call in `onPause()`

Camera 2 API

More sophisticated way

What is wrong with `Camera` API

- It can be used in three different, simple mode
  - Preview
  - Taking picture
  - Capture video
- New function cannot be implemented easily
  - Burst mode
- Frames cannot be accessed
- No RAW format support
- Small amount of metadata can be added
- Not to easy to set properties
  - Using the `Camera.Parameters` many things can be set, but there us no guarantee that the settings will take effect

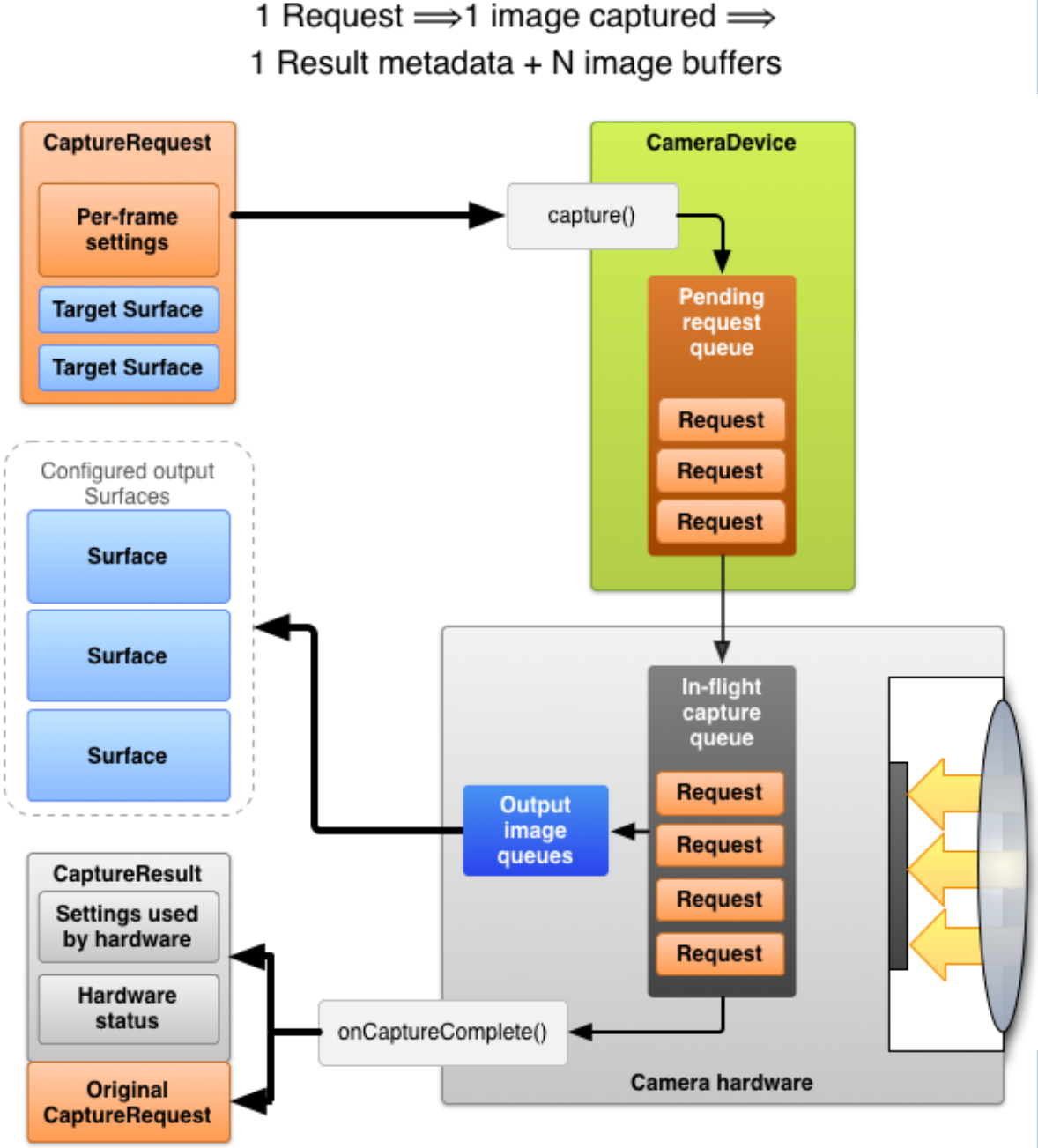
Possibilities

- Processing raw images
- Several calculations and transformations can be done on the device
  - Better range of dynamics
- Using Camera2 API
  - Further sensor data can be used
  - Multicore CPU and CPU can be used for calculations
  - BLE/NFC communications
  - Context aware process
- Access to
  - Sensors
  - Flashlight
  - Optical lens
  - Technical data for each captured frames
- HDR, Panorama image, Multi focused images

Concept

- This package models a camera device as a pipeline
  - Takes in input requests for capturing a single frame
  - Captures the single image per the request
  - Outputs one capture result metadata packet
    - plus a set of output image buffers for the request.
- The requests are processed in-order, and multiple requests can be in flight at once.
  - Since the camera device is a pipeline with multiple stages, having multiple requests in flight is required to maintain full framerate on most Android devices.

Concept

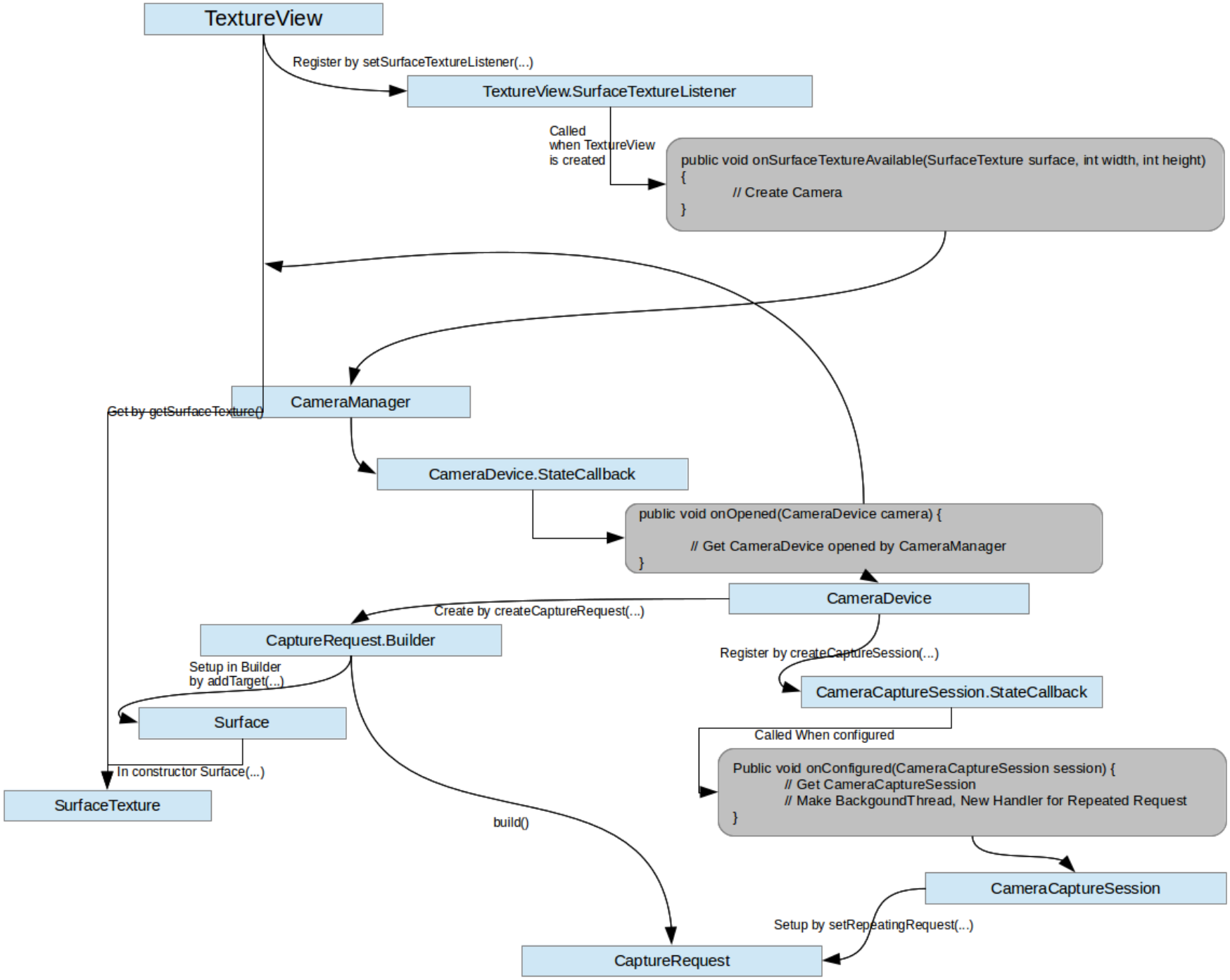

Camera2 API Core Operation Model
1 Request ⟹1 image captured ⟹
1 Result metadata + N image buffers
CaptureRequest
Per-frame settings
Target Surface
Target Surface
CameraDevice
capture()
Pending request queue
Request
Request
Request
Configured output Surfaces
Surface
Surface
Surface
In-flight capture queue
Request
Request
Request
Request
Output image queues
CaptureResult
Settings used by hardware
Hardware status
Original CaptureRequest
onCaptureComplete()
Camera hardware

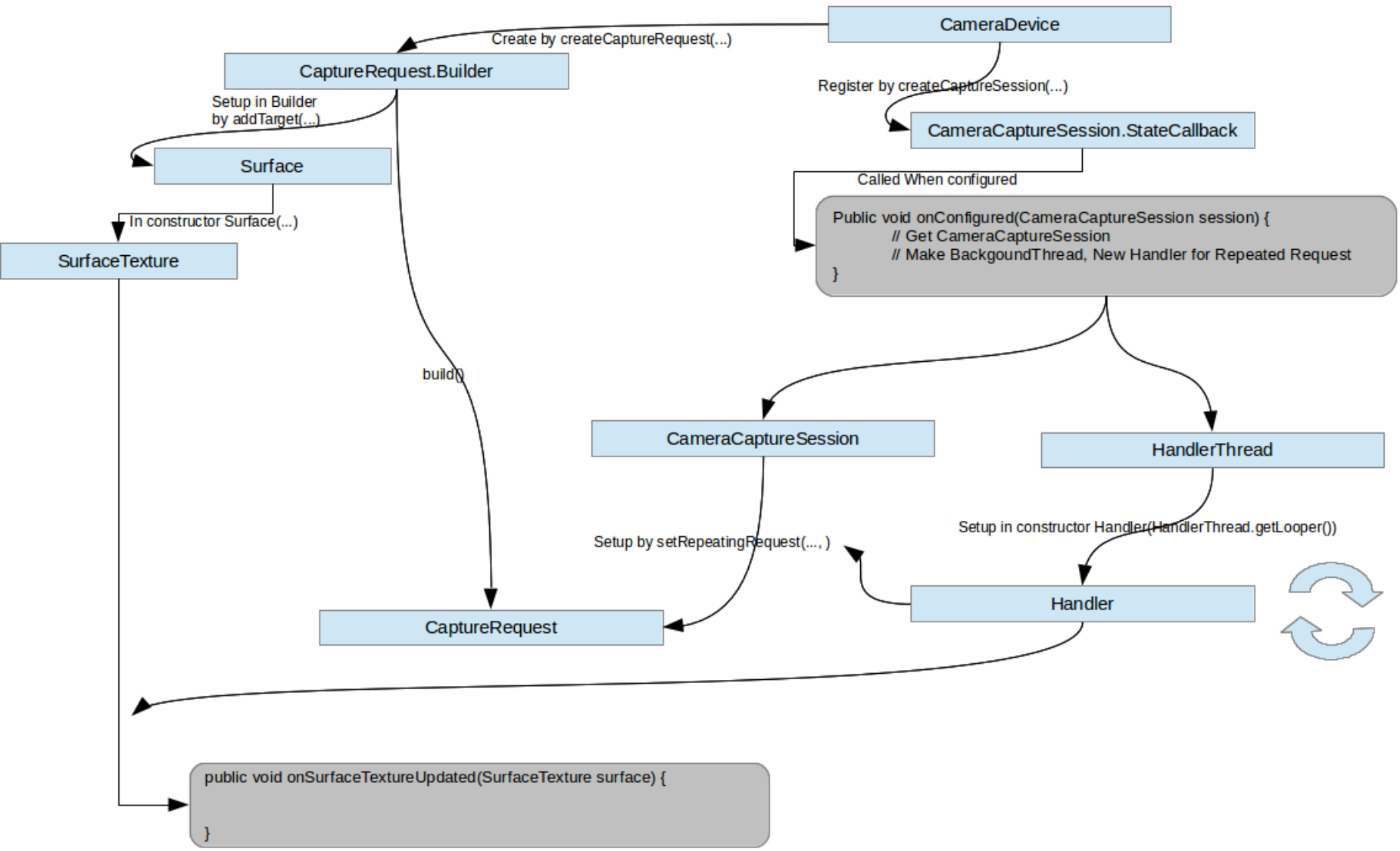
Camera2 API

- Two packages
  - android.hardware.camera2
  - android.hardware.camera2.params
- Important classes
  - CameraManager
    - Query the number of cameras
    - ```
      CameraManager manager=(CameraManager)getSystemService(Context.CAMERA_SERVICE);
      String [] camids = manager.getCameraIdList();
      ```
  - CameraCharacteristics
    - Parameters of a selected camera
      - ```
        CameraCharacteristics characteristics =
        manager.getCameraCharacteristics(camid);
        ```
    - The instance an immutable set of key-value pairs, representing the capabilities of the camera
    - INFO_SUPPORTED_HARDWARE_LEVEL
      - LEGACY, LIMITED, FULL
    - LENS_FACING
      - FRONT, BACK, EXTERNAL
    - JPEG_AVAILABLE_THUMBNAIL_SIZES

Camera2 API

- Important classes
  - `CameraDevice`
    - Represents an actual device, which can be used in later tasks
    - Different templates can be used for different tasks
      - Preview
      - Record
      - Still_capture
      - Video_snapshot
      - Zero_shutter_lag
  - `CameraCaptureSession`
    - A session must be created to take a picture or re-process existing pictures
    - The output surfaces and parameters must be set
    - CameraCaptureSession.CaptureCallback instance have to be sent in order to retrieve information about the stages of the process
    - A Handler have to be set to execute background tasks
    - The coders, etc. can be defined using the session


TextureView
Register by setSurfaceTextureListener(...)
TextureView.SurfaceTextureListener
Called
when TextureView
is created
public void onSurfaceTextureAvailable(SurfaceTexture surface, int width, int height)
{
// Create Camera
}
CameraManager
Get by getSurfaceTexture()
CameraDevice.StateCallback
public void onOpened(CameraDevice camera) {
// Get CameraDevice opened by CameraManager
}
CameraDevice
Create by createCaptureRequest(...)
CaptureRequest.Builder
Register by createCaptureSession(...)
Setup in Builder
by addTarget(...)
CameraCaptureSession.StateCallback
Surface
Called When configured
In constructor Surface(...)
Public void onConfigured(CameraCaptureSession session) {
// Get CameraCaptureSession
// Make BackgoundThread, New Handler for Repeated Request
}
SurfaceTexture
build()
CameraCaptureSession
Setup by setRepeatingRequest(...)
CaptureRequest


CameraDevice
Create by createCaptureRequest(...)
CaptureRequest.Builder
Register by createCaptureSession(...)
CameraCaptureSession.StateCallback
Setup in Builder
by addTarget(...)
Surface
Called When configured
In constructor Surface(...)
Public void onConfigured(CameraCaptureSession session) {
// Get CameraCaptureSession
// Make BackgoundThread, New Handler for Repeated Request
}
SurfaceTexture
build()
CameraCaptureSession
HandlerThread
Setup in constructor Handler(HandlerThread.getLooper())
Setup by setRepeatingRequest(...,)
Handler
CaptureRequest
public void onSurfaceTextureUpdated(SurfaceTexture surface) {
}

Demo

- Builtin demo
  - Android Studio
- Video tutorial
  - https://www.youtube.com/watch?v=Xtp3tH27OFs

Homework

- You will need to create a working audio player application.
  - Can open local, and remote URL-s
    - Local files on the devices
    - Files on a web server such as:
      - http://mad.itk.ppke.hu/android/tada.wav
  - Can run and play in the background
  - Can be controlled (stopped) by a notification
  - The app should have a user-friendly design

Hardware and ML

Next week