Ćwiczenie 6

Managing Media
(Part I)

Author: Mariusz Fraś

Objectives of the exercise

The goal of the exercise is:

1. Learning the techniques of accessing and storing media content, especially Images.
2. Gaining basics of using Camera/CameraX.

- **The first part of the exercise** (Exercise – Part I) is an instruction to practice the basic techniques. – to prepare for self-reliant developing basic application.
- **The second part of the exercise** (Exercise – Part II) is to prepare and present or to complete on the next class according to teacher's orders.
- Details are given by the teacher.

1 Basics of developing media applications

1.1 Basic classes and components for media

The most basic classes and interfaces that support implementing media application are:

Support for access media files:

- `MediaStore` – the component supporting access to Media Provider content (see Content Provider below) – contains definitions for the supported URIs (do media resources) and columns, and many others.
- `ContentProvider` – provides encapsulated data from various sources with common model - as one or more tables,
  Media Provider provides such MIME type content as **image/***, **audio/***, and **video/***.
- `ContentResolver` – the client of Content Provider – queries Content Provider and returns Cursor that contains table of "media records" (the info to support access media files),
- `Cursor` – a set of tuples (records) and columns – provides mechanisms for operating on the query result (navigating the result elements)
- `FileProvider` –,

Components of direct use of media:

- `MediaPlayer` – the primary API class for playing sound and video,
- `MediaRecorder` – the class for recording audio and video,
- `VideoView` – a view object (widget) that loads an plays videos from various sources,
- `MediaController` – a view object containing controls for a `MediaPlayer` (and `VideoView`),
- `AudioManager` – the class for managing audio sources and audio output (provides audio control and constants),
- `AudioAttributes` – a class to encapsulate a collection of attributes describing information about an audio stream

System component for camera support:
To record media it is possible to use existing application (usually delivered with the system) by launching it with proper intent and receiving results. For video such app is **system Camera application**.

- `Camera / Camera2 / CameraX` – the classes used to set image capture settings, start/stop preview, snap pictures, and retrieve frames for encoding for video – a client for the Camera service, which manages the actual camera hardware. Camera2 package is newest, more featured, and recommended to use version. CameraX is based on Camera2.

To use given device feature it is necessary to specify proper permissions and optionally uses-features or check the existence of given hardware support in device.

1.2 Permissions and supported features

The following basic permissions may be necessary to build media application:

- `android.permission.CAMERA` – to be able to access the camera device,
- `android.permission.RECORD_AUDIO` – to record audio stream,
- `android.permission.WRITE_EXTERNAL_STORAGE` – allows an app to write to external storage (not needed for API>28),

- `android.permission.`**`READ_EXTERNAL_STORAGE`** – allows an app to read external storage (for API<33),
- `android.permission.`**`READ_MEDIA_IMAGES`** – allows an app to read images from storage (for API>=33),
- `android.permission.`**`READ_MEDIA_VIDEO`** – allows an app to read video from storage (for API>=33),
- `android.permission.`**`INTERNET`** – for accessing network media streams (if used),
- `android.permission.`**`WAKE_LOCK`** – to avoiding dimming screen (if necessary).

If the app tags images with GPS location information the **`ACCESS_FINE_LOCATION`** permission is also necessary. The required permissions change according to system API version. Especially read/write permissions.

The following features may be necessary to declare in manifest (or check at runtime).

- android.hardware.**microphone**
- android.hardware.**camera**

In case the app can be run on devices without support of any of above features (e.g. to do other functions) the support for these can be verified at runtime in the code:

```
PackageManager pManager = this.getPackageManager();

if(!pManager.hasSystemFeature(PackageManager.SOME_FEATURE))
  ...        //action when no support
else   ...   //processing when feature supported
```

The media uses-features to check (instead *SOME_FEATURE* above) are:

- `PackageManager.`**`FEATURE_MICROPHONE`**
- `PackageManager.`**`FEATURE_CAMERA`**

1.3 Managing media-related permissions

When using such sensitive permissions as recording audio or video, or storing files in shared storage the activity should check and (if not granted) request for such permissions first !.

Media-related permissions are managed as any risky android permissions:

A. Checking given permission is made with use of **`checkSelfPermission(…)`** method:

```
if (ContextCompat.checkSelfPermission(context,
   Manifest.permission.REQUIRED_PERMISSIONS) == PackageManager.PERMISSION_GRANTED)
   ...//action when permission is granted – continue the operation
else
   ... //action when permission is not granted, e.g. request for permission
```

Here: `REQUIRED_PERMISSIONS` is one or list of mentioned above media-related permissions. It can be built as a list of **Manifest.permission.*SOME_PERMISION*** permisions. E.g.:

```
REQUIRED_PERMISSIONS = listOf (Manifest.permission.CAMERA,
                               Manifest.permission.RECORD_AUDIO)
```

B. request for permission is made with use of **`requestPermission(…)`** method:

```
ActivityCompat.requestPermissions(activity: Activity, permissions: Array<String>,
                                  requestCode: int)
```

C. The results of the request (i.e. did the user granted permission or not) should be handled in **`onRequestPermissionsResult(…)`** method:

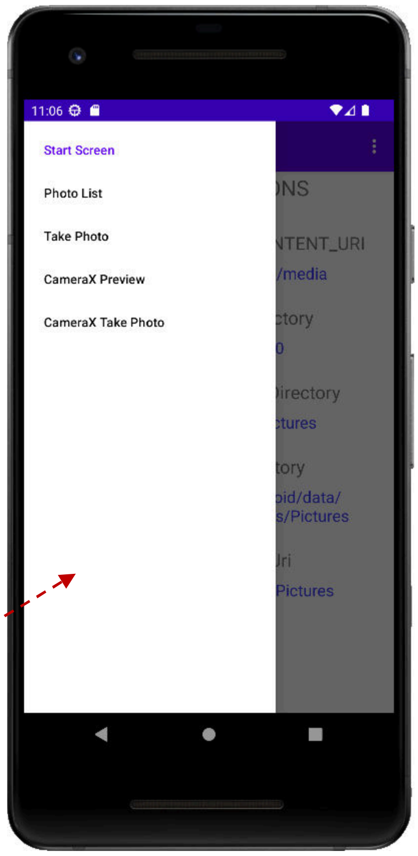
```
public void onRequestPermissionsResult(reqCode: Int, permissions: Array<String>,
                                       grantResults: IntArray) {
  super.onRequestPermissionsResult(requestCode, permissions, grantResults);
  when (reqCode) {
    MY_FIRST_REQEST_CODE ->
      requestResult1:B = (grantResults[0] == PackageManager.PERMISSION_GRANTED);
    MY_SECOND_REQEST_CODE ->
      requestResult2 = (grantResults[1] == PackageManager.PERMISSION_GRANTED);
    ...
  }
  ... [here: perform proper action due to received results (requestResults booleans)]
}
```

2 Working with images

***Note**: For the compactness and simplicity, the implemented here application will not cover all aspects and techniques for considered programming features. Especially to serve correctly recently changed system features, same necessary extensions are omitted. Anyway basic techniques are presented and only some extensions considering access restrictions should be added.*
The app focuses on images mainly (anyway other media (e.g. video) are used the same way).

- Implement application with navigation and fragments. Fragments will be used to present different mechanism of managing media. Here:
  - **Start Fragment** – to show some basic information about device storage / media locations/paths in application start screen.
  - **Photo List Fragment** – to show content of media storage.
  - **Take Photo Fragment** – to present mechanism for taking picture with use of system Camera.

  These in part II of the exercise:
  - **CameraX Preview Fragment** – to present mechanism of using CameraX library (Preview Use Case).
  - **CameraX Take Photo Fragment** – to present mechanism of using CameraX library (Capture Image Use Case).



Navigation drawer for running fragments

- In the app manifest, set desired feature and permissions:

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

<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
                 android:maxSdkVersion="28" />
```

- In the activity (in onCreate(…) method) implement requesting necessary permissions:

```
private val REQUIRED_PERMISSIONS = mutableListOf (
                                   Manifest.permission.CAMERA,
                                   Manifest.permission.RECORD_AUDIO,
                                   Manifest.permission.READ_EXTERNAL_STORAGE)
  .apply {
    if (Build.VERSION.SDK_INT <= Build.VERSION_CODES.P) {
      add(Manifest.permission.WRITE_EXTERNAL_STORAGE)
    }
  }.toTypedArray()
if (!allPermissionsGranted()) {
  ActivityCompat.requestPermissions(this, REQUIRED_PERMISSIONS, REQUEST_CODE)
}
```

where `allPermissionsGranted()` is a method:

```
fun allPermissionsGranted() = REQUIRED_PERMISSIONS.all {
  ContextCompat.checkSelfPermission(baseContext, it) ==
                                          PackageManager.PERMISSION_GRANTED
}
```

and the listener for result is:

```
override fun onRequestPermissionsResult(requestCode: Int,
                                        permissions: Array<String>,
                                        grantResults: IntArray) {

  super.onRequestPermissionsResult(requestCode, permissions, grantResults)
  if (requestCode == REQUEST_CODE) {
    if (!allPermissionsGranted()) {
      Toast.makeText(this,"Permissions not granted.", Toast.LENGTH_SHORT).show()
      finish()
} } }
```

where `REQUEST_CODE` is the unique request code.

Just finish the app if permissions are not granted.

2.1 Managing access to the storage

2.1.1 Application-specific storage

App-specific storage by default is available only for application.

To get app-specific internal storage location (absolute path) the **`getFilesDir()`** method of application context can be called:

```
val dir : File = requireContext().getFilesDir();
```

To get app-specific primary external storage location the **`getExternalFilesDir(String type)`** method of application context can be called:

```
val dir = requireContext().getExternalFilesDir(Environment.DIRECTORY_PICTURES)
```

Other example types are:

– `Environment.DIRECTORY_MOVIES`
– `Environment.DIRECTORY_MUSIC`
  etc.

2.1.2 Shared storage

Shared media are available for multiple applications.

To access shared media up to Android 9 (API 28) the proper uses-permission must be specified:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
                 android:maxVersion="28" />
```

Since Android 10 (API 29) to access files that other apps have created the app must be granted the **READ_EXTERNAL_STORAGE** permission. In recent versions the rules are still changing (including new philosophy of so called ***Scoped Storage*** (not considered here).

To access media content the media content provider **`MediaStore`** that delivers Uris for media can be used. The **`MediaStore.Images`**, **`MediaStore.Audio`** and **`MediaStore.Video`** support management of pictures, audio and video with use of **content://** style Uris for shared media:

– MediaStore.Images.Media.***EXTERNAL_CONTENT_URI*** – the **content://** style URI for the "primary" external storage.
– MediaStore. Images.Media.***INTERNAL_CONTENT_URI*** – the URI for the internal storage.

  And so on for audio and video.

- In the **Start Fragment** add text views and display basic information of storage locations:

```
private var basePhotoUri = MediaStore.Images.Media.EXTERNAL_CONTENT_URI

val dir2 = Environment.getExternalStorageDirectory()
val dir3 = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES)
val dir4 = requireContext().getExternalFilesDir(Environment.DIRECTORY_PICTURES)

viewBinding.tv1path.text = basePhotoUri.scheme + ":/" +
                           MediaStore.Images.Media.EXTERNAL_CONTENT_URI.path
viewBinding.tv2path.text = dir2.absolutePath
viewBinding.tv3path.text = dir3.absolutePath
viewBinding.tv4path.text = dir4?.absolutePath ?: "nothing"

dir4?.let {
  val theuri = FileProvider.getUriForFile(requireContext(),
                                         "${BuildConfig.APPLICATION_ID}.provider",it)
  viewBinding.tv5path.text = theuri.scheme + ":/" + theuri.path
 }
```

  For the last directive to use `FileProvider.getUriForFile` method for newer versions of android (API >= 29) you must **specify available Uris in app resouces** – see next section.

2.1.3 File Provider

FileProvider is a special class that supports secure access and sharing of app files with use of content:// schema Uri. Among the others it permits to get content Uri for File object with use of **`getUriForFile(context: Context, authority: String, file: File): Uri!`** method.

The FileProvider must be specified in application manifest, in **<provider> node** nested in manifest's **<application> node**. Among many attributes usually the following should be set:

```
android:name="com.example.AFileProvider"
android:authorities="domain_of_authority.fileprovider"
android:grantUriPermissions="true"
```

and if file provider does not need to be public:

```
android:exported="false"
```

com.example.AFileProvider is an app subclass of FileProvider or FileProvider class may be used itself. *domain_of_authority* is usually set to app package name.

A FileProvider can only generate/return a content Uri for file paths specified in **<paths> meta-data element** – in the project XML resource with **<paths>** main node. This resource (xml file) must be pointed in **meta-data**, in **<provider>** node in app manifest.

A FileProvider can only generate/return a content URI for files in directories that you specify beforehand. To specify a directory, specify its storage area and path in XML, using child elements of the <paths>. A few (very limited number) child elements are allowed, that correspond to a few storage directories. Some of them are:

- **<external-files-path>** – that corresponds to location returned by **getExternalFilesDir** method.
- **<external-path>** – that corresponds to **getExternalStorageDirectory** method.

- Enter (define) in application AndroidManifest.xml file the following file provider section:

```
<manifest ...
  <application
    ...
    <provider
      android:name="androidx.core.content.FileProvider"
      android:authorities="${applicationId}.provider"
      android:exported="false"
      android:grantUriPermissions="true" >
    <meta-data
      android:name="android.support.FILE_PROVIDER_PATHS"
      android:resource="@xml/file_paths">
    </meta-data>
  </provider>

  </application>
</manifest>
```

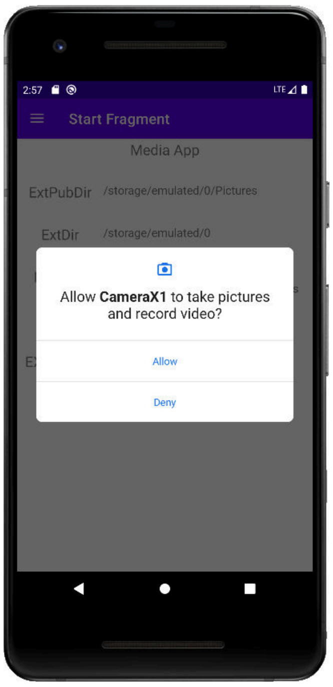
  The package name for *authorities* attribute is usually followed with "provider" (or likewise) part.

- Add to the projekt xml resource file named **file_paths** with the content:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <external-files-path name="my_app_images" path="." />
    <external-path name="my-external-images" path="Pictures/" />
</paths>
```

  The names may be of its own.

- Run the application and check the operation. The effect should be similar to the following:


Start Fragment

Media App

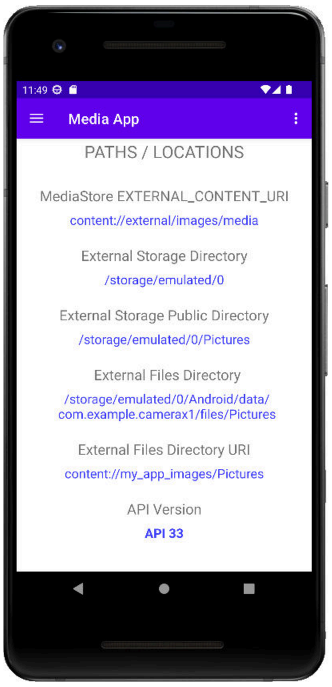
ExtPubDir /storage/emulated/0/Pictures

ExtDir /storage/emulated/0

Allow **CameraX1** to take pictures and record video?

Allow

Deny


Media App

PATHS / LOCATIONS

MediaStore EXTERNAL_CONTENT_URI
content://external/images/media

External Storage Directory
/storage/emulated/0

External Storage Public Directory
/storage/emulated/0/Pictures

External Files Directory
/storage/emulated/0/Android/data/
com.example.camerax1/files/Pictures

External Files Directory URI
content://my_app_images/Pictures

API Version
API 33

2.2 Presenting media content list.

***Note**: For the compactness and simplicity, in the implemented application pictures are almost not processed in any way – e.g. resizing or control or orientation. Also, as mentioned before, some extensions to work identically on all API versions of Android are omitted.*

Note2: Attention! In this app not all possible errors are handled – e.g. corrupted files, or files with 0 bytes size. In such a case be careful and remove such files with system apps otherwise this test app may hung/kick off.

2.2.1 Presenting the content with use of content provider and content Uri

To get "pointer" to media content (files) in shared storage the following steps must be done:

– Get proper **Uri** for given content type.
– Get a **content resolver**.
– Make a query with use of Uri (that specify given media collection) that returns a **Cursor** object.
– Use Cursor (**`moveToFirst()`**, **`moveToNext()`**, etc. methods) to get given media source/file info with use of **`getColumnIndex(…)`** method – especially media item (file) identifier.

An example code of above steps for images in external shared storage is the following:

```
var uri: Uri = MediaStore.Images.Media.EXTERNAL_CONTENT_URI
val contentResolver :ContentResolver = requireContext().contentResolver
val cursor = contentResolver.query(uri, null, null, null, null);
if (cursor == null) {
  // Error (e.g. no such volume)
} else if (!cursor.moveToFirst()) {
  // no media in specified store
} else {
  val idColumn = cursor.getColumnIndex(MediaStore.Images.Media._ID)
  val nameColumn = cursor.getColumnIndex(MediaStore.Images.Media.DISPLAY_NAME)
  do {
    var thisId = cursor.getLong(idColumn)
    var thisName = cursor.getString(nameColumn)
```

```
        // ...process entry...
        // e.g. add data to the list: sharedList?.add(DataItem(thisName,thisUri))
      } while (cursor.moveToNext());
    }
```

To access given media file you need the content Uri for that file composed of media collection Uri completed with unique media item identifier (row Id of media provider data). To include such data in the list you must use **ContentUris** class and **withAppendId** method::

```
var thisContentUri = ContentUris.withAppendedId(uri, thisId)
```

- Implement fragment (here named **PhotoList**) with RecyclerView and adapter. Use data repository with data list as a source data for adapter.
- Define class for data about media files – e.g.: similar to:

```
class DataItem {
  var name : String = "Empty name"
  var uripath : String = " Empty uri"
  var path : String = " Empty path"
  var curi : Uri? = null  //Content Uri
  //other possible constructors
  constructor(name:String, uripath:String, path: String, curi: Uri) : this()  {
    this.name = name
    this.uripath = uripath
    this.path = path
    this.curi = curi
  }
}
```

- Define repository in standard way. Include in repository two lists (for displaying pictures from shared and app-specific external storage). Pass to the repository app context for future use:

```
class DataRepo {
  lateinit var uri: Uri
  ...
  fun getSharedList() : MutableList<DataItem>? {...}
  fun getAppList() : MutableList<DataItem>? {...}
  companion object{
    private var INSTANCE: DataRepo? = null
    private lateinit var ctx: Context
    var sharedStoreList: MutableList<DataItem>? = null
    var appStoreList: MutableList<DataItem>? = null
    fun getinstance(ctx: Context): DataRepo {
      if (INSTANCE == null){
        INSTANCE = DataRepo()
        sharedStoreList = mutableListOf<DataItem>()
        appStoreList = mutableListOf<DataItem>()
        this.ctx = ctx
      }
      return INSTANCE as DataRepo
    }
  }
}
```

- In the **getSharedList** method include the following”
  - set uri to `MediaStore.Images.Media.EXTERNAL_CONTENT_URI`

- clear the repository list
- make a query for files on Uri with Content Resolver
- in the loop get data about images from rows in returned Cursor and store them (add) in the repository list

It means implement the code as shown at the beginning of this section with additions:

- before the query call:

```
sharedStoreList?.clear()
```

- in the loop add proper data to the repository list:

```
do {
  var thisId = cursor.getLong(idColumn)
  var thisName = cursor.getString(nameColumn)
  var thisContentUri = ContentUris.withAppendedId(uri, thisId)
  var thisUriPath = thisContentUri.toString()
  sharedStoreList?.add(DataItem(thisName, thisUriPath,"No path yet",
                              thisContentUri))
} while (cursor.moveToNext());
```

*The **getAppList** method will be implemented in later step.*

- Define the adapter for the list built on RecyclerView (here in separate class) in typical way:

```
class PhotoListAdapter(val appContext: Context, val dList: MutableList<DataItem>) :
                           RecyclerView.Adapter<PhotoListAdapter.MyViewHolder>()
{
  ...
}
```

- pass app context and list from repository as parameters
- define list item view layout with texts (file name, uri,…) and image
- in **onBindViewHolder** method set proper data in view holder e.g.:

```
vHolder.tv1.text = dList[position].name
vHolder.tv2.text = dList[position].uripath
vHolder.tv3.text = dList[position].curi?.path     //temporary this data
if (Build.VERSION.SDK_INT > Build.VERSION_CODES.P) {
  dList[position].curi?.let {
    vHolder.img.setImageBitmap(appContext.contentResolver.loadThumbnail(it,
                                                         Size(72,72), null))
  }
}
else
  vHolder.img.setImageBitmap(getBitmapFromUri(appContext, dList[position].curi))
```

For API >=29 we use convenient method for loading thumbnails **ContentResolver.loadThumbnail(…) : Bitmap**. For older versions we use here own method with direct image loading – see below

- define in the adapter method to directly load image using input stream and **BitmapFactory** class:

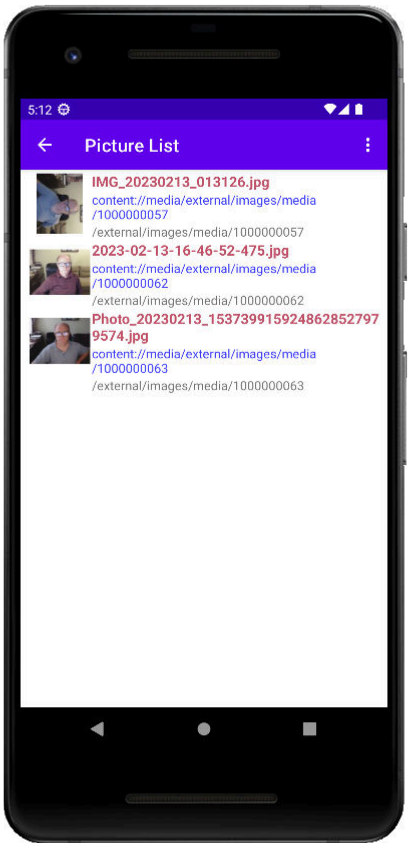
```
fun getBitmapFromUri(mContext: Context, uri: Uri?): Bitmap? {
  var bitmap: Bitmap? = null
  try {
    val image_stream: InputStream
```

```
        try {
          image_stream = uri?.let {
            mContext.getContentResolver().openInputStream(it)
          }!!
          bitmap = BitmapFactory.decodeStream(image_stream)
        } catch (e: FileNotFoundException) {
          e.printStackTrace()
        }
      } catch (e: Exception) {
        e.printStackTrace()
      }
      return bitmap
    }
```

- In the **PhotoList** fragment set adapter in standard way:

```
val dataRepo = DataRepo.getinstance(requireContext())
val adapter = dataRepo.getSharedList()?.let { PhotoListAdapter(requireContext(),it) }
  if (adapter == null) {
    Toast.makeText(requireContext(),"Invalid Data", Toast.LENGTH_LONG).show()
    requireActivity().onBackPressed()
  }
recView.adapter = adapter
```

- Take some photos with system camera and run the application.
  The effect should be similar to the following:

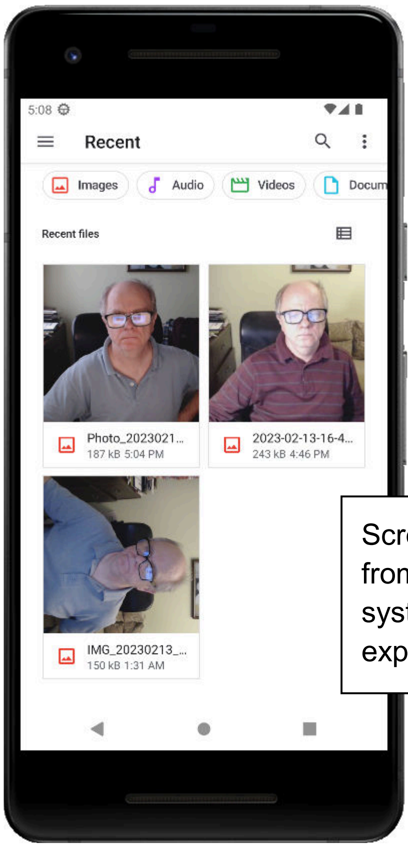

5:12

Picture List

IMG_20230213_013126.jpg
content://media/external/images/media
/1000000057
/external/images/media/1000000057

2023-02-13-16-46-52-475.jpg
content://media/external/images/media
/1000000062
/external/images/media/1000000062

Photo_20230213_15373991592486285279
79574.jpg
content://media/external/images/media
/1000000063
/external/images/media/1000000063



Screen from system file explorer

2.3 Using Camera with contracts/intents

***Note**: In recent APIs access control to storage changes frequently and to act upon this demands more extensive/complex code. To simplify exercise here implemented functions work (was tested) on latest API 33 emulator and partly on API 29 emulator. This apply to storing photos to shared external storage and app-specific storage (latest works on both).*

Taking a photo (picture captured with device camera) or a photo preview is possible with use of system Camera application (activity). It can be done with use of Result API, i.e. defining launchers with registered listeners for results and launching proper components (activities). To use camera can be used among the others the following contracts:

– **`ActivityResultContracts.TakePicturePreview()`** – to take a photo preview (low quality image)
– **`ActivityResultContracts.TakePicture()`** – to take higher quality picture.

The use of launcher is typical only with differences in returned results.

It is also possible to use `ActivityResultContracts.StartActivityForResult()` contract and take a picture in old fashion wat with appropriately defined Intent, here not presented.

- Define **TakePhoto** Fragment with the following elements:
  – include in the layout Image widget and buttons to run system camera to take (capture) a picture – see the screenshots below,
  – define in the fragment launcher and listener for results for taking picture preview,
  – define in the fragment launcher and listener for results for taking picture,
  – in the button's OnClickListeners launch/execute contracts accordingly.
- Define and run launcher (and listener for results) for picture preview:

```
val photoPreviewLauncher =
 registerForActivityResult(ActivityResultContracts.TakePicturePreview()) {
    result: Bitmap? ->
  if (result != null) {
    Toast.makeText(requireContext(),"PREVIEW RECEIVED",Toast.LENGTH_LONG).show()
    viewBinding.imgView.setImageBitmap(result)
  }
  else {
    Toast.makeText(requireContext(),"PREVIEW NOT RECEIVED",Toast.LENGTH_LONG).show()
  }
}
```

Here:
– `viewBinding.imgView` is a reference to Image widget in layout (`imgView` is an Image Id).
– the returned result is a picture preview **bitmap**.

- In the button's OnClickListener execute contract in **try … catch…** block:

```
photoPreviewLauncher.launch()
```

- Define and run launcher (and listener for results) to take a picture

```
val photoLauncher=registerForActivityResult(ActivityResultContracts.TakePicture()) {
    result : Boolean ->
  if (result) {
    // consume result  - see later remarks
    Toast.makeText(requireContext(),"Photo TAKEN",Toast.LENGTH_LONG).show()
  }
  else
    // make some action - warning
    Toast.makeText(requireContext(), "Photo NOT taken!", Toast.LENGTH_LONG).show()
    lastFile.delete()
  }
}
```

Here:

– The returned result is only Boolean value of success/failure – in case of further processing we must save essential data (e.g. Uri for file) on its own,
– **`lastFile`** is w reference (variable of type File) to the file where image is to be stored,
– it should be deleted when we cancel capturing to avoid empty file in storage.

- In the button's OnClickListener execute contract:

```
lastFileUri = getNewFileUri()
try {
  photoLauncher.launch(lastFileUri)
} catch (e: ActivityNotFoundException) {
  Toast.makeText(requireContext(),"CAMERA DOESN'T WORK!", Toast.LENGTH_LONG).show()
}
```

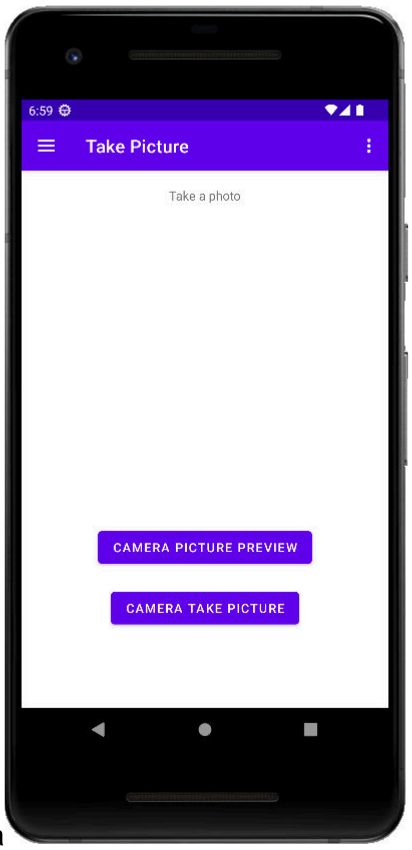
Here:

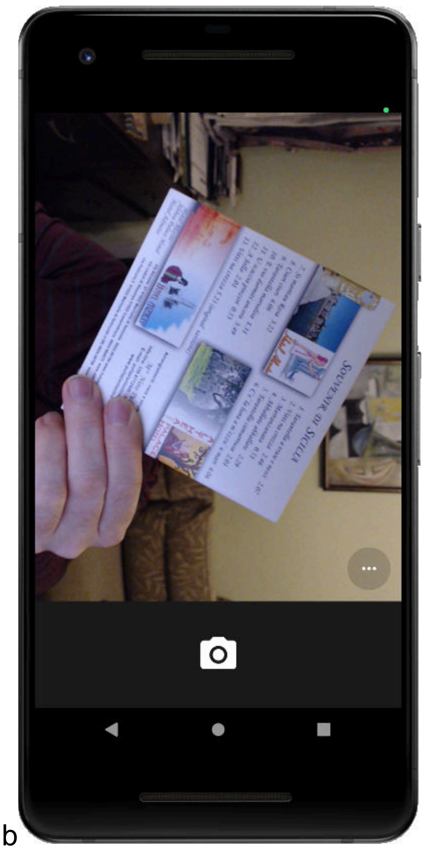
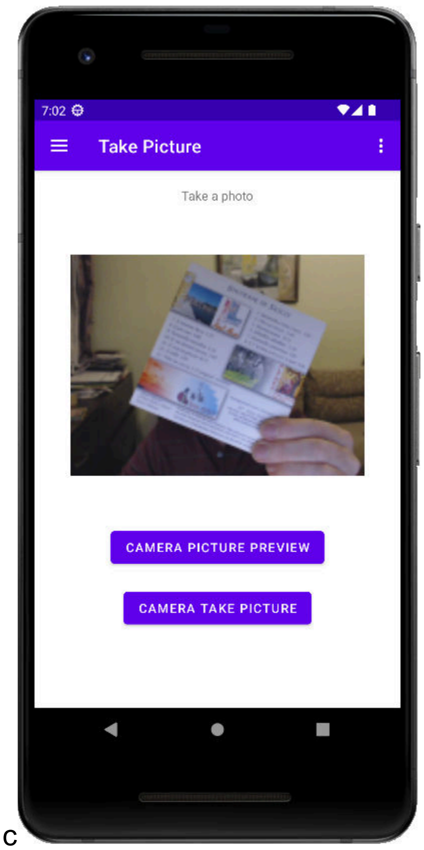
– the URI for file must be passed to launcher.
– The URI is get from separate method **getNewFileUri** on the basis of provided **media collection URI** and with use of **FileProvider**:

```
private fun getNewFileUri(): Uri {
  val tStamp: String = SimpleDateFormat("yyyyMMdd_HHmmss").format(Date())
  val dir = requireContext().getExternalStoragePublicDirectory(
                                                   Environment.DIRECTORY_PICTURES)
  val tmpFile = File.createTempFile("Photo_" + "${tStamp}", ".jpg", dir)
  lastFile = tmpFile  //save File for future use

  return FileProvider.getUriForFile(requireContext(),
                                    "${BuildConfig.APPLICATION_ID}.provider",
                                    tmpFile)
}
```

- Run the application and check the operation. The effect should be similar to the following:



a

b


c

On devices with older API (e.g. API 29) taking picture to external shared storage may crush due to some changes of control access to content via content URIs (as implemented above for newest Android version). To avoid this, the code should be more complex.

2.4 Storing media in app-specific storage.

To support operation on devices with older API it is good idea to use app-specific storage for private pictures. The procedure is the same with one difference – the URI passed TakePicture launcher which should point to app-specific directory.

After storing picture in private directory (or wherever else, e.g. in cache directory) it may be copied to shared storage using different means.

- Extend the application to support storing images in app-specific directory by:
  - Adding to the repository variable indicating which storage to use and support for second list of “private” images.
  - Adding in app menu selection of which storage to use.
  - Extending **TakePhoto** fragment with using target storage on the basis on storage selection.
  - Extending **PhootList** fragment to display list of images get from selected storage.

  The app will use (store and display) both storage – collection of images in shared storage and app-specific storage.
- Add in the data repository (**DataRepo** class with two lists – **sharedStoreList** and **appStoreList**) variable **`photo_storage`** (initialy shared storage) and methods to set and check which storage to use:

```
class DataRepo {
    ...
    val SHARED_S = 1
    val PRIVATE_S = 2
    var photo_storage = SHARED_S
    fun setStorage(storage:Int) : Boolean {
      if (storage != SHARED_S && storage != PRIVATE_S)
          return false
      else {
          photo_storage = storage
      }
      return true
    }
    fun getStorage(): Int {
      return photo_storage
    }
  ...
  }
```

- Add in the data repository **getAppList** method that returns the list with data abot images in app-specific external storage:

```
fun getAppList() : MutableList<DataItem>? {
  val dir : File? = ctx.getExternalFilesDir(Environment.DIRECTORY_PICTURES)
  dir?.listFiles()
  appStoreList?.clear()
  if (dir?.isDirectory() == true) {
    var fileList = dir.listFiles()
    if (fileList != null) {
      for (value in fileList) {
        var fileName = value.name
        if (fileName.endsWith(".jpg") || fileName.endsWith(".jpeg") ||
            fileName.endsWith(".png") || fileName.endsWith(".gif")) {
          val tmpUri = FileProvider.getUriForFile(ctx,
                                 "${BuildConfig.APPLICATION_ID}.provider", value)
          appStoreList?.add(DataItem(fileName, value.toURI().path,
                                  value.absolutePath, tmpUri))
        }
      }
    }
  }
  return appStoreList
}
```

Here:

– **FILE.listFiles()** method is used to get ArrayList of files in directory,

– separate **appStoreList** is used for “private” files/images,

– **ctx** is passed to repository app context as previously.

- Implement Overflow menu with items to select one of mentioned storage to use – define xml menu resource file, add in main activity **onCreateOptionsMenu** method, and in **onOptionsItemSelected** call

```
DataRepo.getinstance(this).setStorage(SHARED_S)
```

or

```
DataRepo.getinstance(this).setStorage(PRIVATE_S)
```

In dependence of user selection.

- Modify in **PhotoList** fragment instantiation of adapter (**PhotoListAdapter**) on the basis of storage selection:

```
...
var adapter : PhotoListAdapter? = null
when(dataRepo.getStorage()) {
  SHARED_S -> {
    adapter = dataRepo.getSharedList()?.let {
      PhotoListAdapter(requireContext(),it) }
  }
  PRIVATE_S-> {
    adapter = dataRepo.getAppList()?.let {
      PhotoListAdapter(requireContext(),it) }
  }
}
...
```

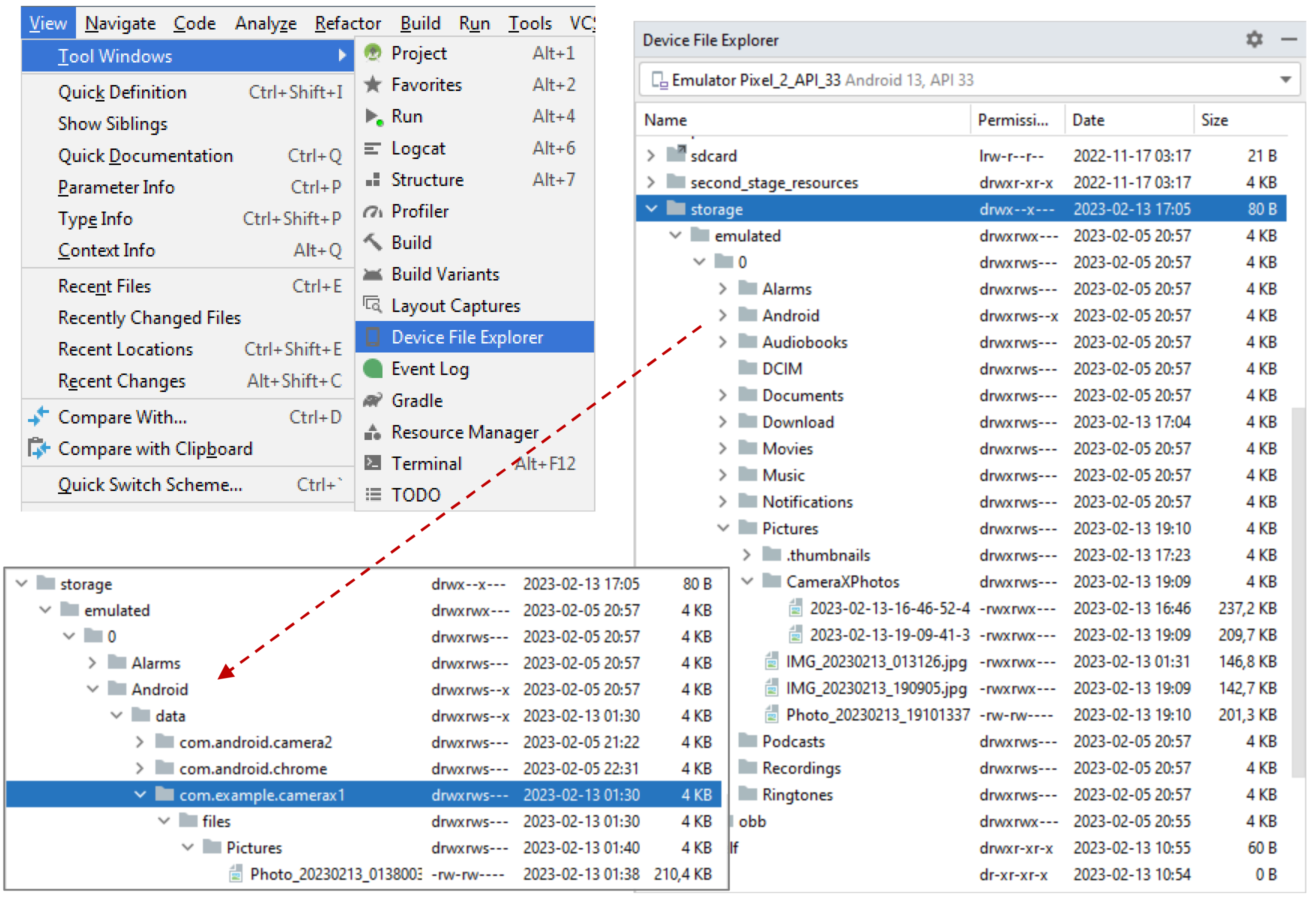
- Modify in **TakePhoto** fragment returning proper URI by **getNewFileUri()** method according to selected by user storage:

```
private fun getNewFileUri(): Uri {
  ...
  when(dataRepo.getStorage()) {
    SHARED_S -> dir =
        Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES)
    PRIVATE_S ->  dir =
        requireContext().getExternalFilesDir(Environment.DIRECTORY_PICTURES)!!
    else -> return MediaStore.Images.Media.EXTERNAL_CONTENT_URI
  }
  ...
}
```

where **dir** is as previously defined.

- Run the application and check the operation.

***Hint**: You can verify (and also download) files in emulated device.*
To do this just: open Device File Explorer from menu: View→Tools Window →Device File Explorer:



Name	Permissi...	Date	Size
sdcard	lrw-r--r--	2022-11-17 03:17	21 B
second_stage_resources	drwxr-xr-x	2022-11-17 03:17	4 KB
storage	drwx--x---	2023-02-13 17:05	80 B
emulated	drwxrwx---	2023-02-05 20:57	4 KB
0	drwxrws---	2023-02-05 20:57	4 KB
Alarms	drwxrws---	2023-02-05 20:57	4 KB
Android	drwxrws--x	2023-02-05 20:57	4 KB
Audiobooks	drwxrws---	2023-02-05 20:57	4 KB
DCIM	drwxrws---	2023-02-05 20:57	4 KB
Documents	drwxrws---	2023-02-05 20:57	4 KB
Download	drwxrws---	2023-02-13 17:04	4 KB
Movies	drwxrws---	2023-02-05 20:57	4 KB
Music	drwxrws---	2023-02-05 20:57	4 KB
Notifications	drwxrws---	2023-02-05 20:57	4 KB
Pictures	drwxrws---	2023-02-13 19:10	4 KB
.thumbnails	drwxrws---	2023-02-13 17:23	4 KB
CameraXPhotos	drwxrws---	2023-02-13 19:09	4 KB
2023-02-13-16-46-52-4	-rwxrwx---	2023-02-13 16:46	237,2 KB
2023-02-13-19-09-41-3	-rwxrwx---	2023-02-13 19:09	209,7 KB
IMG_20230213_013126.jpg	-rwxrwx---	2023-02-13 01:31	146,8 KB
IMG_20230213_190905.jpg	-rwxrwx---	2023-02-13 19:09	142,7 KB
Photo_20230213_19101337	-rw-rw----	2023-02-13 19:10	201,3 KB
Podcasts	drwxrws---	2023-02-05 20:57	4 KB
Recordings	drwxrws---	2023-02-05 20:57	4 KB
Ringtones	drwxrws---	2023-02-05 20:57	4 KB
obb	drwxrwx---	2023-02-05 20:55	4 KB
lf	drwxr-xr-x	2023-02-13 10:55	60 B
	dr-xr-xr-x	2023-02-13 10:54	0 B

storage	drwx--x---	2023-02-13 17:05	80 B
emulated	drwxrwx---	2023-02-05 20:57	4 KB
0	drwxrws---	2023-02-05 20:57	4 KB
Alarms	drwxrws---	2023-02-05 20:57	4 KB
Android	drwxrws--x	2023-02-05 20:57	4 KB
data	drwxrws--x	2023-02-13 01:30	4 KB
com.android.camera2	drwxrws---	2023-02-05 21:22	4 KB
com.android.chrome	drwxrws---	2023-02-05 22:31	4 KB
com.example.camerax1	drwxrws---	2023-02-13 01:30	4 KB
files	drwxrws---	2023-02-13 01:30	4 KB
Pictures	drwxrws---	2023-02-13 01:40	4 KB
Photo_20230213_013800	-rw-rw----	2023-02-13 01:38	210,4 KB

3 Working with CameraX

This part is not presented in this exercise