These slides use a photo-laucher (activity-results-launcher) to take a picture rather than start-activity

Taking a Picture

based on Chapter 16 of
Android Programming: A Big Nerd Ranch Guide (4th edition)

BNR

Taking a Picture

- AndroidManifest.xml
- file.xml (resource)
- Crime
- CrimeRepository
- CrimeDetailViewModel
- CrimeDetailFragment
- PictureUtil

We will be using a menu item to take a picture, so the CrimeDetailFragment code will differ from BNR

Also, we will use an activity results launcher instead of a start activity call.

File Storage

`getFilesDir(): File` – returns a handle to the directory for private application files

`openFileInput(name: String): FileInputStream` – opens an existing file in the files directory for input

`openFileOutput(name: String, mode: Int): FileOutputStream` – opens a file in the files directory for output, possibly creating it

`getDir(name: String, mode: Int): File` – gets (and possibly creates) a subdirectory within the files directory

`fileList(…): Array<String>` – gets a list of filenames in the main files directory, such as for use with `openFileInput(String)`

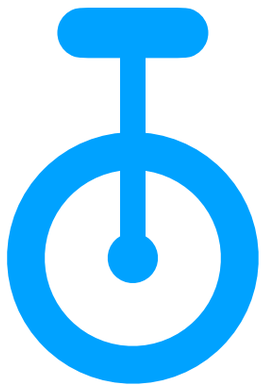
`getCacheDir(): File` – returns a handle to a directory you can use specifically for storing cache files; you should take care to keep this directory tidy and use as little space as possible

File Storage

Full-size pictures are **too large** to stick inside a SQLite database, much less an Intent

They need to live on your device's filesystem – in your **private storage**

But **only your application** can read or write to them



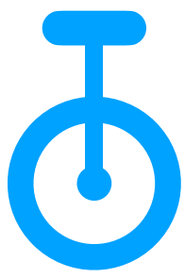
— BNR

If you need to share files with other apps, use a **ContentProvider**

File Provider

When all you need to do is **receive** a file from another application, implementing a ContentProvider is **overkill**

FileProvider takes care of everything except the configuration



Adding the FileProvider declaration

AndroidManifest.xml (after last Activity)

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

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

<application
    ...
    <activity android:name=".MainActivity">
        ...
    </activity>
    <provider
        android:name="androidx.core.content.FileProvider"
android:authorities="edu.vt.cs.cs5254.criminalintent.fileprovider"

        android:exported="false"
        android:grantUriPermissions="true">
        <meta-data
            android:name="android.support.FILE_PROVIDER_PATHS"
            android:resource="@xml/files"/>
    </provider>
</application>
```

files.xml resource

```
<paths>
    <files-path name="crime_photos" path="."/>
</paths>
```

```
@Entity
data class Crime(
        @PrimaryKey val id: UUID = UUID.randomUUID(),
        var title: String = "",
        var date: Date = Date(),
        var isSolved: Boolean = false
) {
        val photoFileName
                get() = "IMG_$id.jpg"
}
```

photoFileName does not include the path to the photo, but it is unique since it's based on the crime's ID.

```
class CrimeRepository private constructor(context: Context) {

    private val filesDir = context.applicationContext.filesDir

    fun getPhotoFile(crime: Crime): File = File(filesDir, crime.photoFileName)
```

add after val executor

```
class CrimeDetailViewModel() : ViewModel() {

    fun getPhotoFile(crime: Crime): File {
        return crimeRepository.getPhotoFile(crime)
    }
```

add at the end

```
class PictureUtils {

    companion object {

        fun isCameraAvailable(activity: Activity): Boolean {
            val packageManager: PackageManager = activity.packageManager
            return packageManager.hasSystemFeature(PackageManager.FEATURE_CAMERA_ANY)
        }

        fun getScaledBitmap(path: String, destWidth: Int, destHeight: Int): Bitmap {
            // Read in the dimensions of the image on disk
            var options = BitmapFactory.Options()
            options.inJustDecodeBounds = true
            BitmapFactory.decodeFile(path, options)
            val srcWidth = options.outWidth.toFloat()
            val srcHeight = options.outHeight.toFloat()
            // Figure out how much to scale down by
            var inSampleSize = 1
            if (srcHeight > destHeight || srcWidth > destWidth) {
                val heightScale = srcHeight / destHeight
                val widthScale = srcWidth / destWidth
                val sampleScale = if (heightScale > widthScale) {
                    heightScale
                } else {
                    widthScale }
                inSampleSize = Math.round(sampleScale)
            }
            options = BitmapFactory.Options()
            options.inSampleSize = inSampleSize
            // Read in and create final bitmap
            return BitmapFactory.decodeFile(path, options)
        }
    }
}
```

```
class CrimeDetailFragment : Fragment(), DatePickerFragment.Callbacks {

    private lateinit var crime: Crime
    private lateinit var photoFile: File
    private lateinit var photoUri: Uri
    private lateinit var photoLauncher: ActivityResultLauncher<Uri>
```

photoFile and photUri are initialized in onViewCreated when Observer is notified of update to crime

Ensure the updateUI is only called *after* these are initialized

photoLauncher is initialized in onCreate

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    super.onCreateOptionsMenu(menu, inflater)
    inflater.inflate(R.menu.fragment_crime_detail, menu)
    val cameraAvailable = CameraUtil.isCameraAvailable(requireActivity())
    val menuItem = menu.findItem(R.id.take_crime_photo)
    menuItem.apply {
        Log.d(TAG, "Camera available: $cameraAvailable")
        isEnabled = cameraAvailable
        isVisible = cameraAvailable
    }
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
R.id.take_crime_photo -> {
    val captureIntent = Intent(MediaStore.ACTION_IMAGE_CAPTURE).apply {
        putExtra(MediaStore.EXTRA_OUTPUT, photoUri)
    }
    requireActivity().packageManager
        .queryIntentActivities(captureIntent, PackageManager.MATCH_DEFAULT_ONLY)
        .forEach { cameraActivity ->
            requireActivity().grantUriPermission(
                cameraActivity.activityInfo.packageName,
                photoUri,
                Intent.FLAG_GRANT_WRITE_URI_PERMISSION
            )
        }
    photoLauncher.launch(photoUri)
    true
}

        else -> return super.onOptionsItemSelected(item)
    }
}
```

Response to camera menu button press

Grant file-writing permission for all apps that can take pictures

registerForActivityResult takes (1) an ActivityResultContract, and (2) an ActivityResultCallback, and it returns an ActivityResultLauncher

Here, the contract is for taking a photo (and saving it to the Uri)...

The callback updates the photo view...

and the launcher is used to launch the take-photo activity

```
    override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    crime = Crime()
    val crimeId: UUID = arguments?.getSerializable(ARG_CRIME_ID) as UUID
    Log.d(TAG, "Crime fragment created with ID $crimeId")
    vm.loadCrime(crimeId)
    setHasOptionsMenu(true)
    photoLauncher = registerForActivityResult(ActivityResultContracts.TakePicture()) {
        if (it) {
            updatePhotoView()
        }
        requireActivity().revokeUriPermission(photoUri, Intent.FLAG_GRANT_WRITE_URI_PERMISSION)
    }
}
```

Once the photo has been taken and the UI has been updated, don't allow other apps to write to the file.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    viewModel.crimeLiveData.observe(
        viewLifecycleOwner,
        Observer { crime ->
            crime?.let {
                this.crime = crime
                photoFile = viewModel.getPhotoFile(crime)
                photoUri = FileProvider.getUriForFile(requireActivity(),
                    "edu.vt.cs.cs5254.criminalintent.fileprovider",
                    photoFile)
                updateUI()
            }
        })
}
```

```
private fun updateUI() {
    binding.crimeTitle.setText(crime.title)
    binding.crimeDateButton.text = crime.date.toString()
    binding.crimeSolvedCheckbox.apply {
        isChecked = crime.isSolved
        jumpDrawablesToCurrentState()
    }
    updatePhotoView()
}

private fun updatePhotoView() {
    if (photoFile.exists()) {
        val bitmap = CameraUtil.getScaledBitmap(photoFile.path, 120, 120)
        binding.crimePhoto.setImageBitmap(bitmap)
    } else {
        binding.crimePhoto.setImageDrawable(null)
    }
}
```

updateUI calls updatePhotoView

updatePhotoView creates a bitmap with size 120x120 and puts it into the crime_photo view