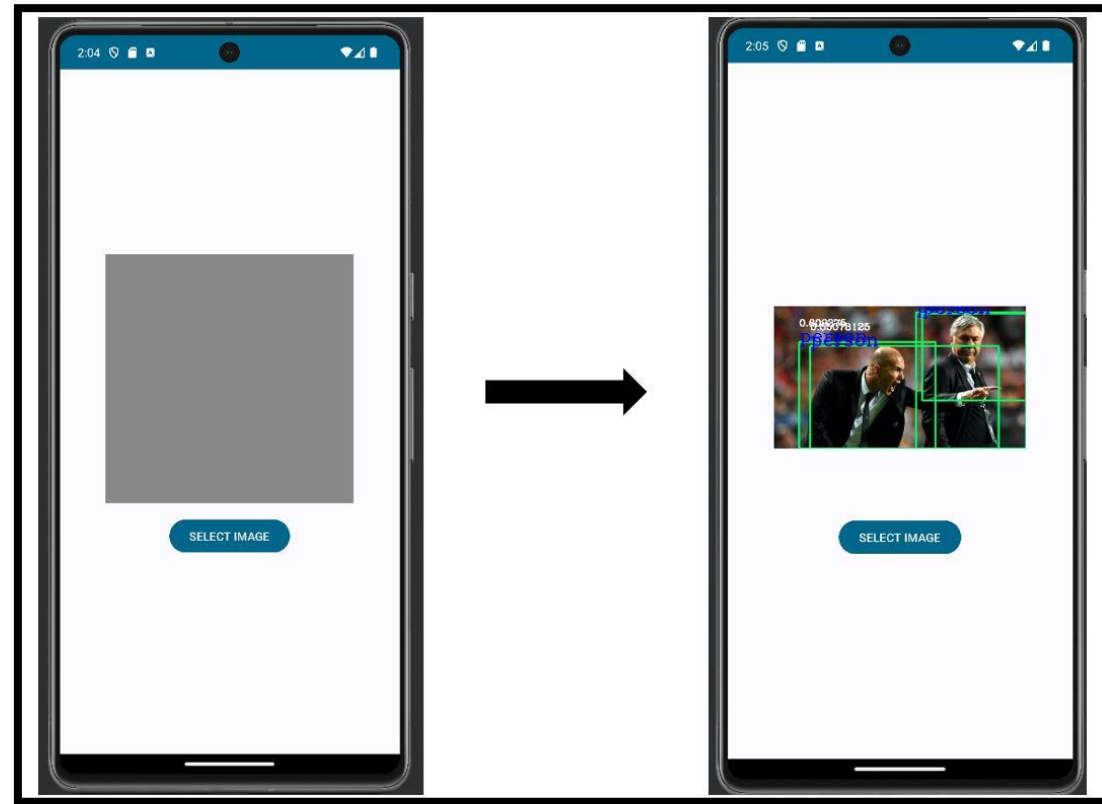


Kotlin&Android 勉強会

Androidで物体検出してみよう!

ソースコード



https://github.com/textcunma/object_detection_android_app

参考資料

- Androidでのオブジェクト検出（公式チュートリアル）
https://www.tensorflow.org/lite/android/tutorials/object_detection?hl=ja
- Androidアプリエンジニアがよく見るサイトとチュートリアル
<https://qiita.com/Nabe1216/items/792914ae6803c6ad3066>
- TensorFlowとCameraXでリアルタイム物体検知Androidアプリ
<https://qiita.com/ymshun/items/fc661a72d08f7f59cf41>

概要

Javaをより書きやすくしたプログラミング言語

Java

- バックエンドで採用する企業多い
- コードが長くなりがち

Kotlin

- Javaよりも短く簡潔にコード記述可能
- Javaと完全互換性
- Androidアプリ開発でよく利用



Jetpack Compose

Kotlinで宣言的にUIを記述するUIツールキット

従来

XMLによるUIの記述

- 複雑
- 複数の言語を使用する必要



- KotlinだけでUIが書ける
- Kotlinのコードは比較的シンプル

開発環境

開発環境

- OS → Windowsを想定
- 統合開発環境 → **android studio** を利用

ダウンロードリンク:

<https://developer.android.com/studio>



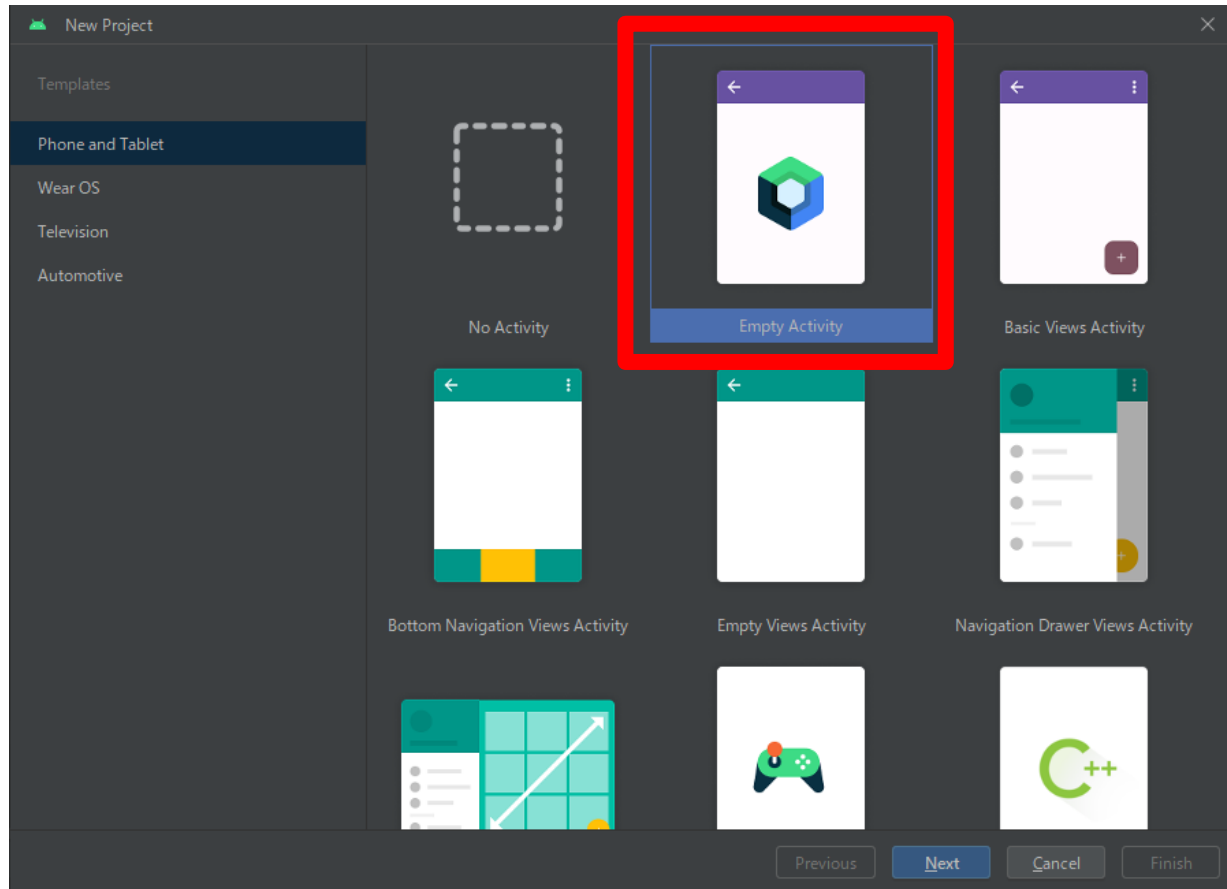
Hello World

プロジェクトの作成

「Empty Activity」を選択

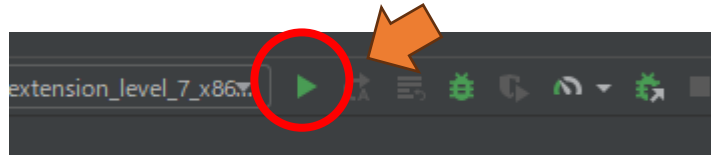


自動的にHello Worldが表示される
プロジェクトが作成



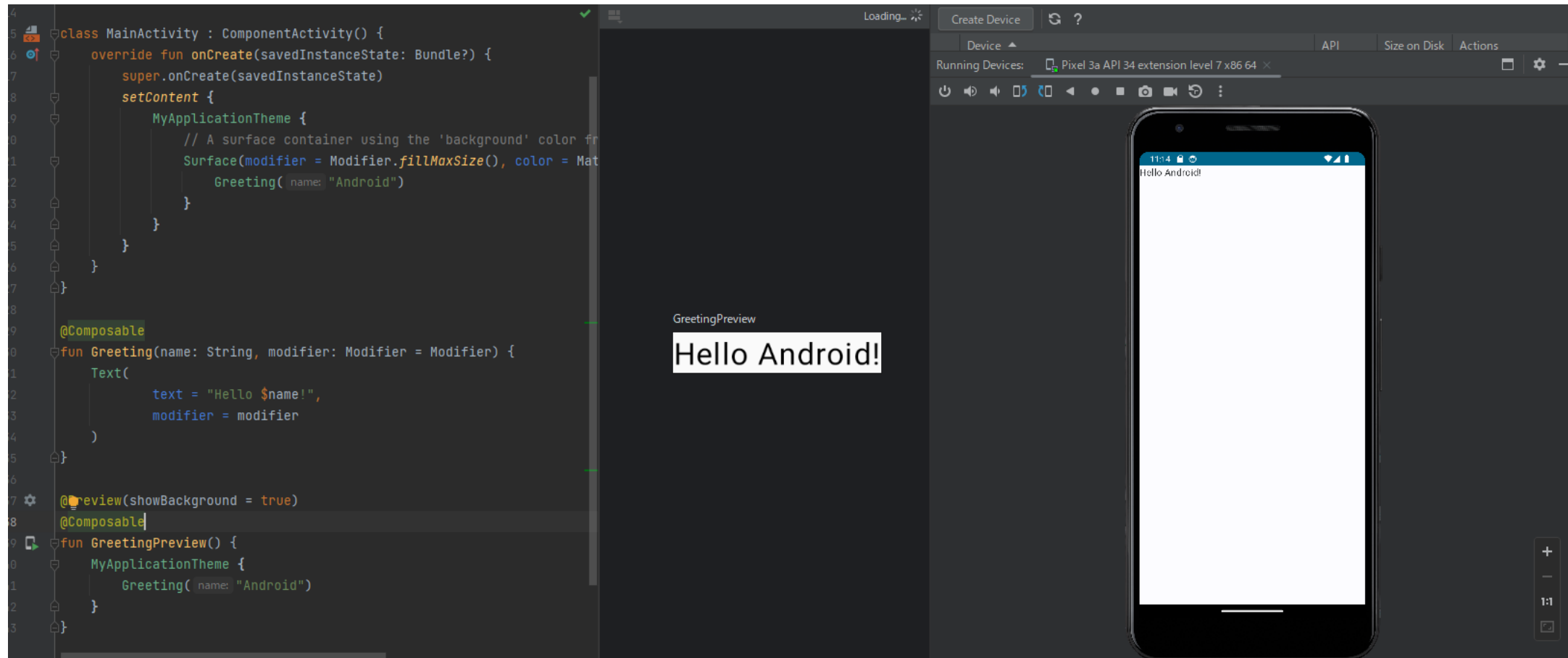
実行までのローディングに
結構時間かかります...

実行



実行ボタンを押す

結果

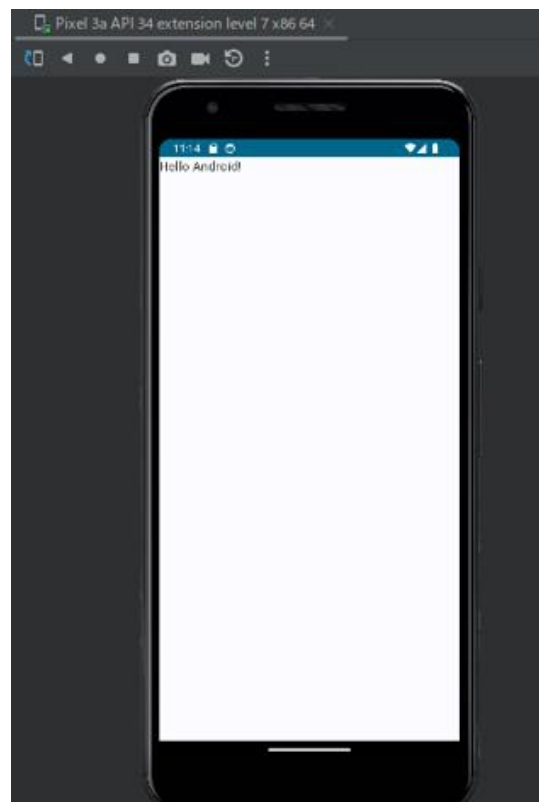


Android Emulator

仮想スマホを起動可能

結構容量食います…

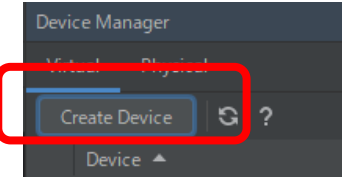
実際のスマホでどのように表示されるか理解



デフォルトでは「Pixel 3A」での表示

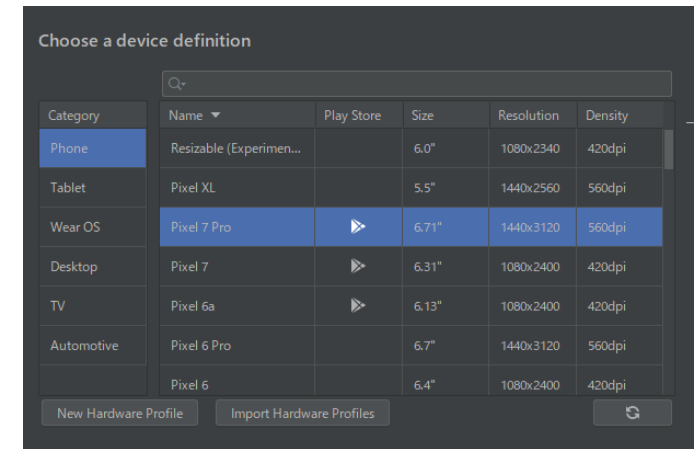
Android Emulator 仮想スマホの追加

1. Toolsタブ → Device Manager を選択



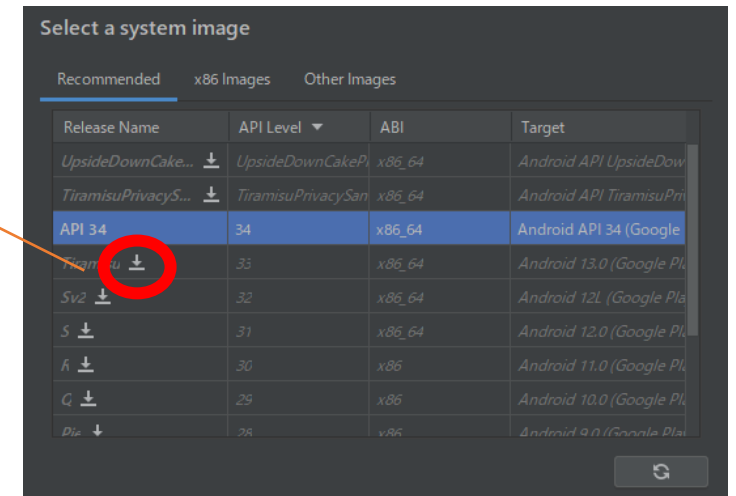
2. 「Create Device」を押す

3. デバイスを選択（画像の場合、Pixel 7 pro）

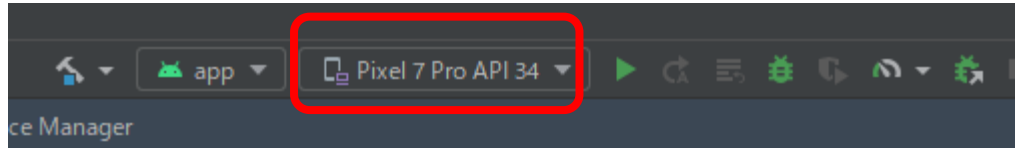


4. システムイメージをダウンロード（↓マークボタンを押す）

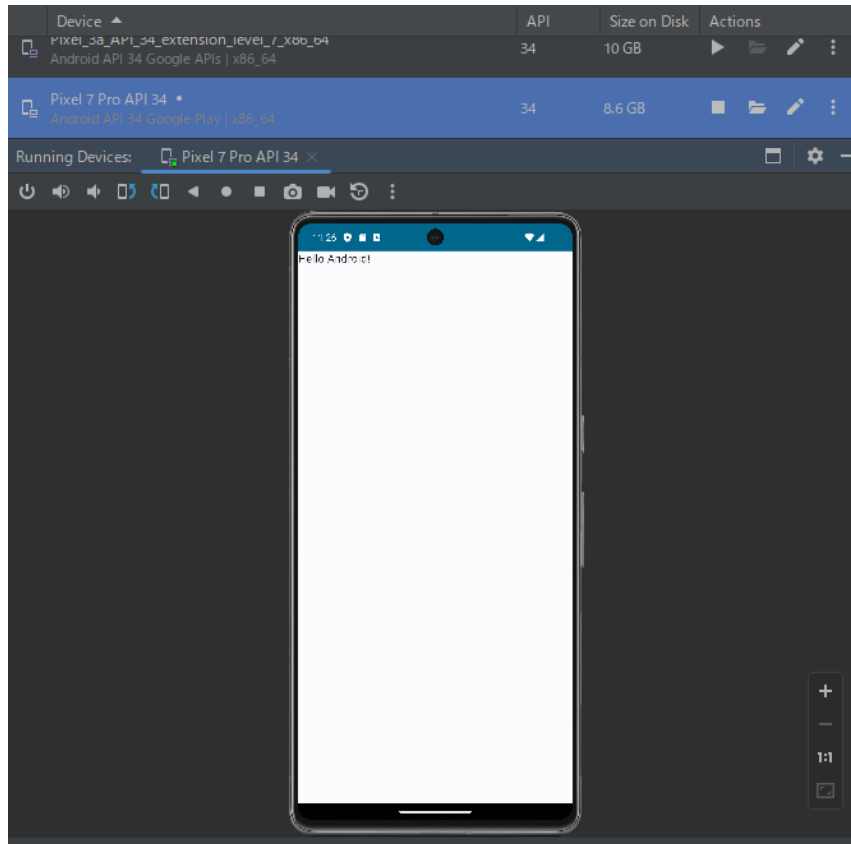
5. Finishボタンで完了



Android Emulator 追加した仮想スマホの実行



新たに追加した仮想スマホを選択して実行ボタン

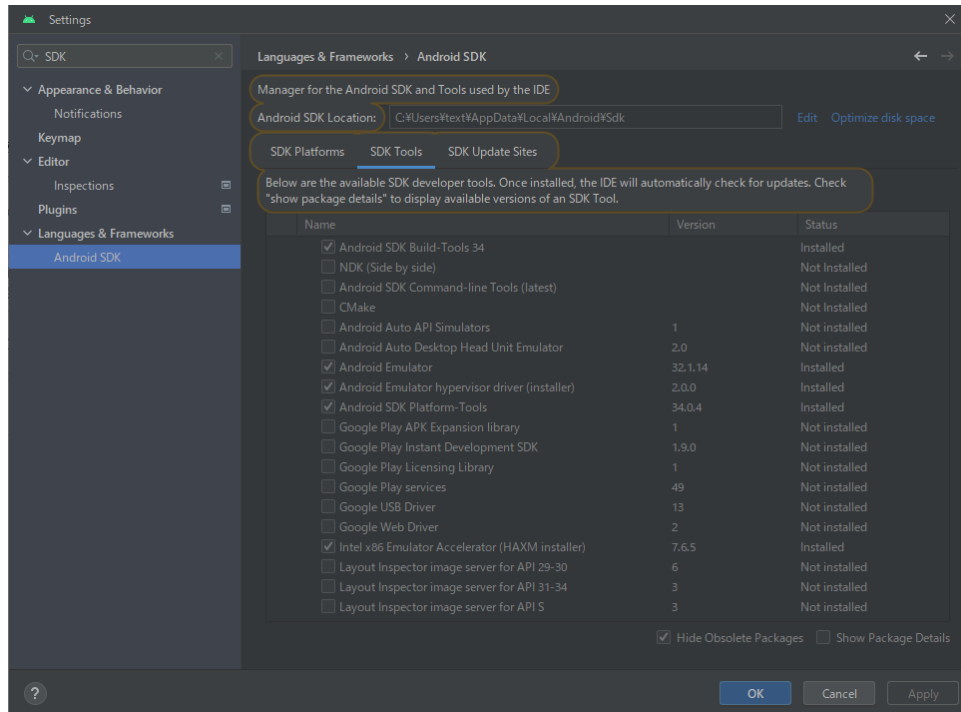


追加した仮想スマホで実行できた♪

実際のスマホで表示してみよう (Androidのみ)

準備として各設定が必要 (以下、サイト参考)

<https://developer.android.com/codelabs/basic-android-kotlin-compose-connect-device?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-1-pathway-2&hl=ja#2>



ただし、Android SDKの場所が変わっているので注意！

1. USB ケーブルを使用し、Android デバイスをパソコンに接続
2. PC側で実行ボタンを押すと、実機のスマホ画面に表示【完】

Jetpack Composeを学習

Jetpack Composeチュートリアル

下記、チュートリアルを読むだけで理解できるのでスライドでは省略

- テキストの配置
- 画像の挿入

<https://developer.android.com/courses/pathways/android-basics-compose-unit-1-pathway-3?hl=ja>

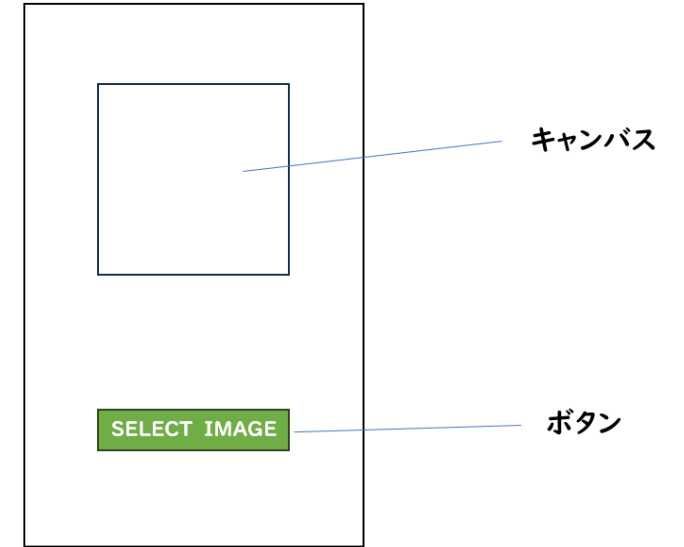
物体検出アプリを作成してみよう！

構想

機能

画像を入力すると、物体検出が行われ、結果画像が表示

- 画像選択ボタン
- キャンバス（画像表示用）



画像選択されたら自動的に実行し、結果をキャンバスに表示

利用

学習済みのTF Liteモデルを使用
前処理等にOpenCVを使用

TensorFlow Light (TF Lite) とは

深層学習モデルをエッジデバイスで動作可能にするフレームワーク

- 推論しか使わない

別のフレームワークで学習し、
学習済みモデルをTensorFlow Lightに変換

- Androidアプリを作る前提で使用する必要

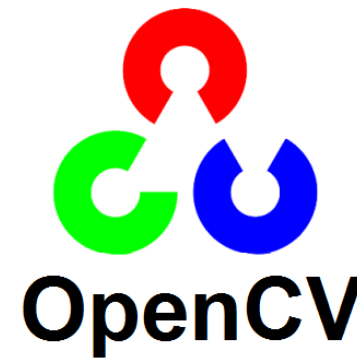
Kotlinの場合、API側がAndroidで使用するように設計



OpenCVとは

代表的な画像処理ライブラリ

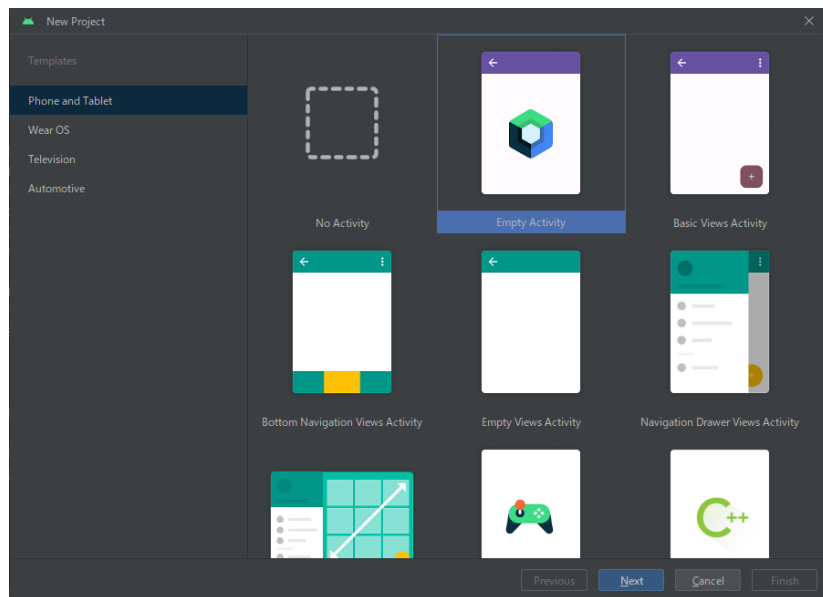
- Android版が提供
- ネット上に多くの資料があるため、使いやすい



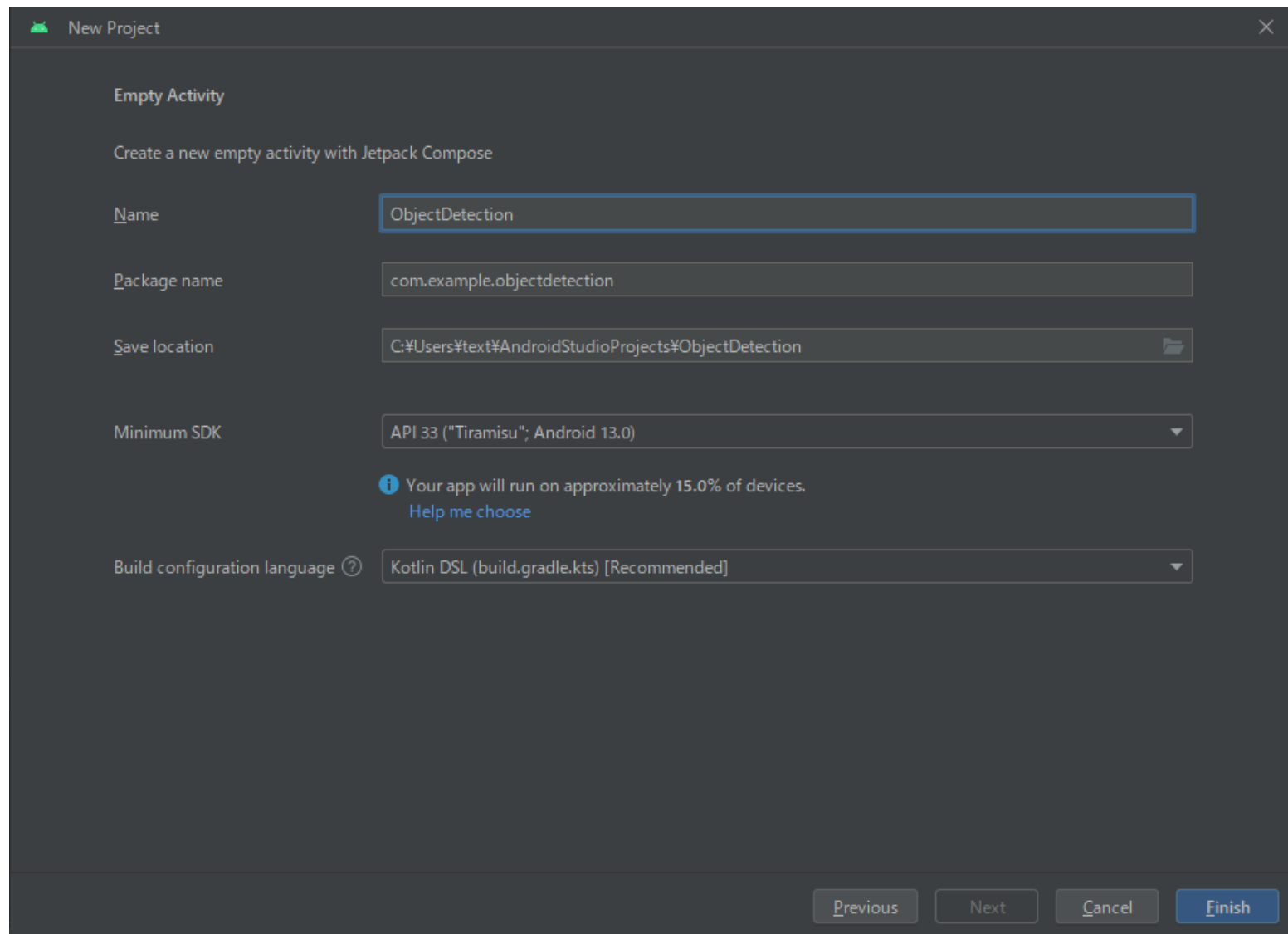
今回のアプリの場合、前処理や描画の際に使用

プロジェクト作成

プロジェクト選択



「Empty Activity」を選択



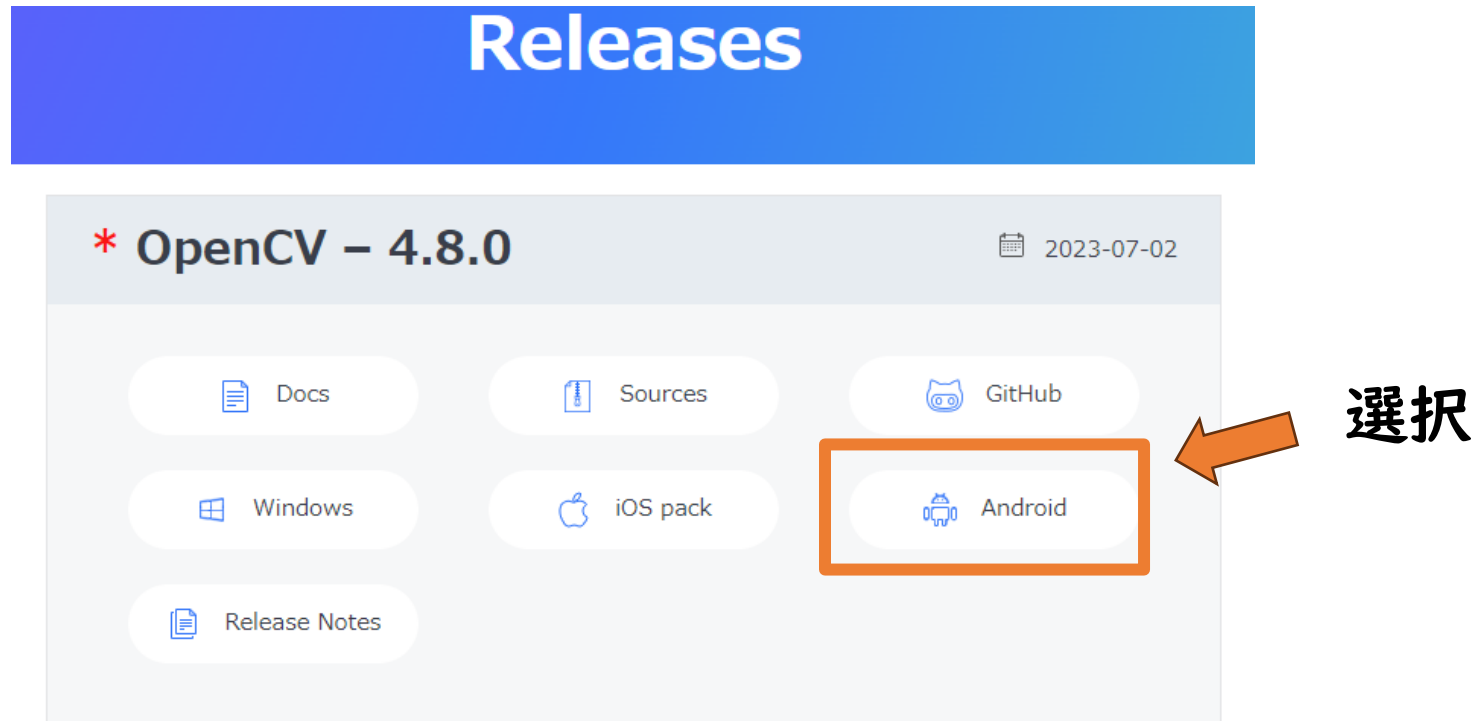
不要なコードの削除

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Android_YOLOTheme {  
                // A surface container using the 'background' color from the theme  
                Surface(  
                    color = MaterialTheme.colorScheme.background  
                ) {  
                    // 新規  
                }  
            }  
        }  
    }  
}
```

これらのコードだけを残して
それ以外を削除

OpenCVの準備

OpenCVをダウンロード

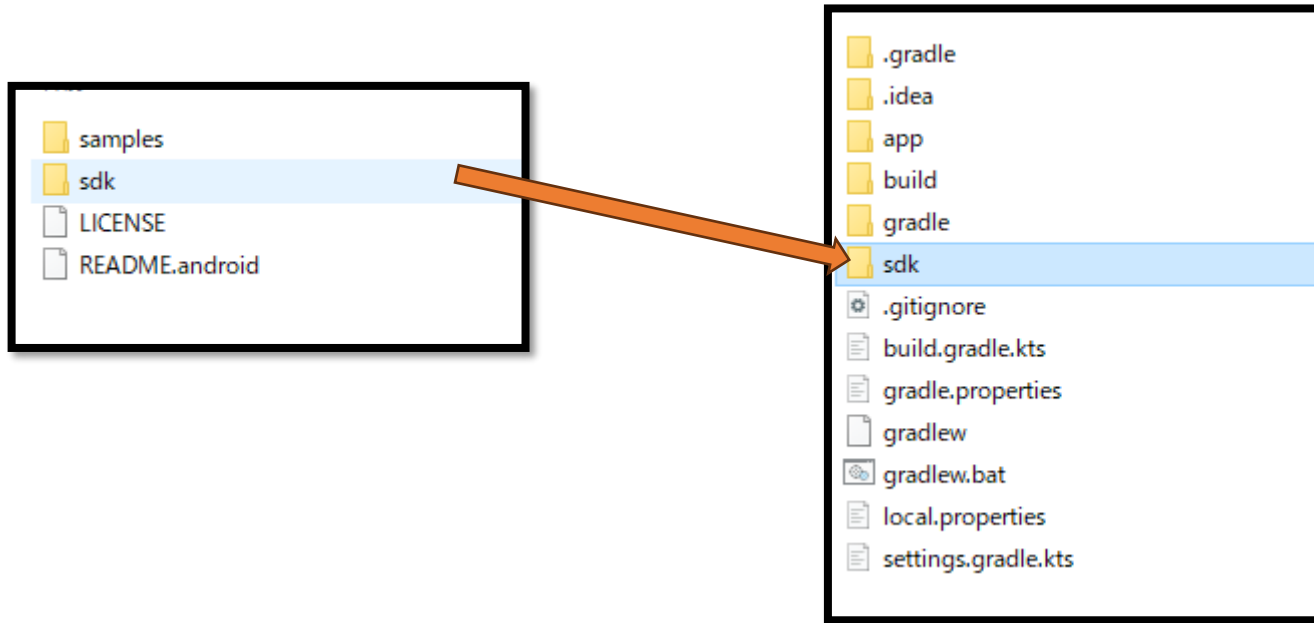


<https://opencv.org/releases/>

ダウンロード後、Cドライブ直下に設置（別に場所はどこでも良かったんですが…）

sdkフォルダをプロジェクト内に設置

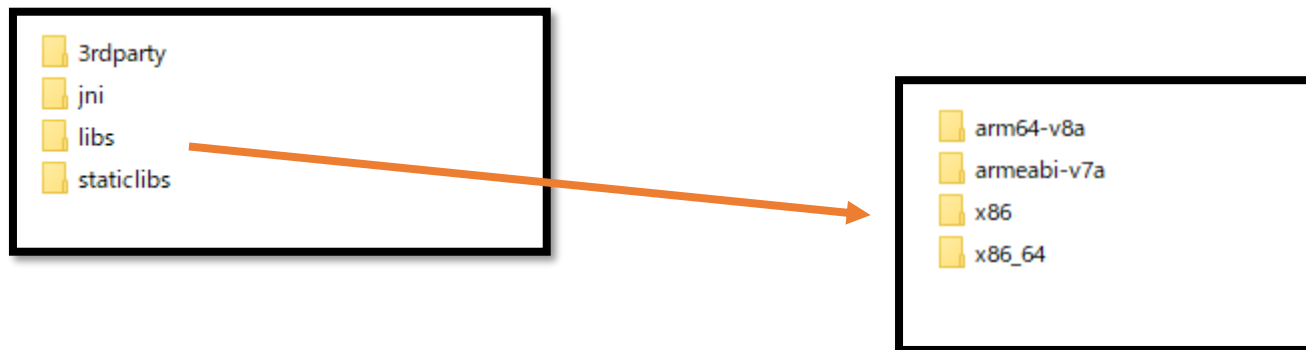
sdkフォルダをコピーして、プロジェクト内に設置



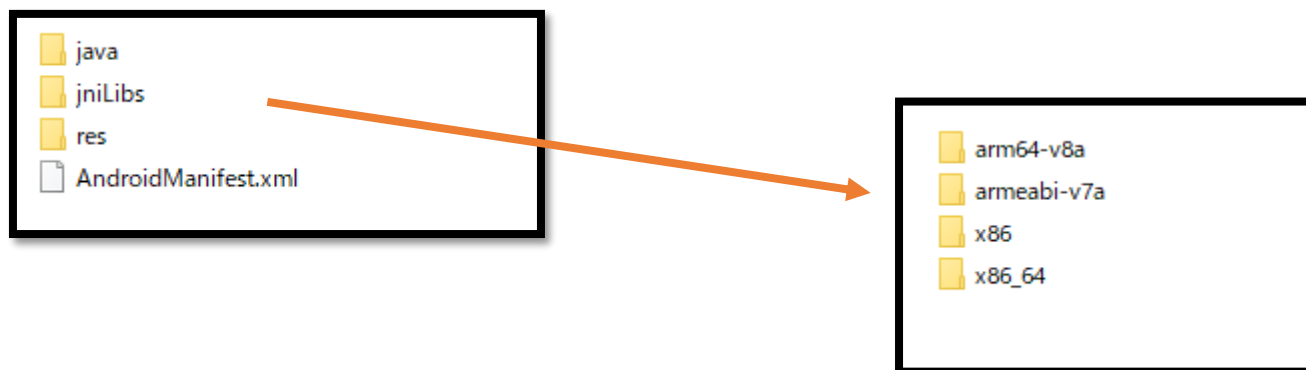
jniLibsフォルダの作成と設定

1. sdk/nativeフォルダ内の「libs」の中身をコピー

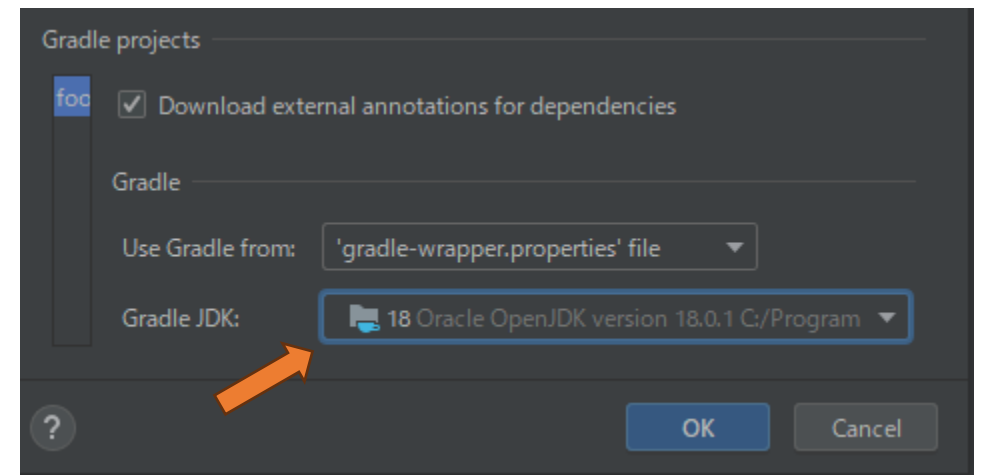
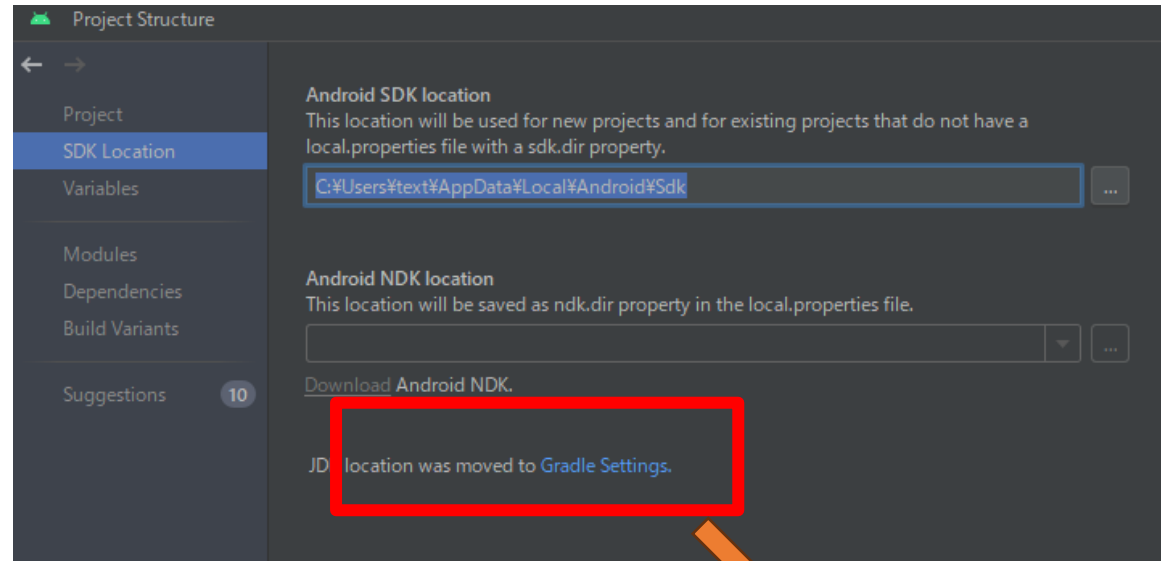
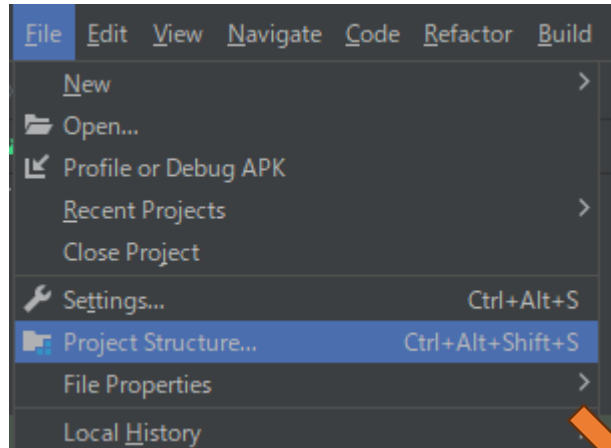
ネイティブ ライブラリを設定



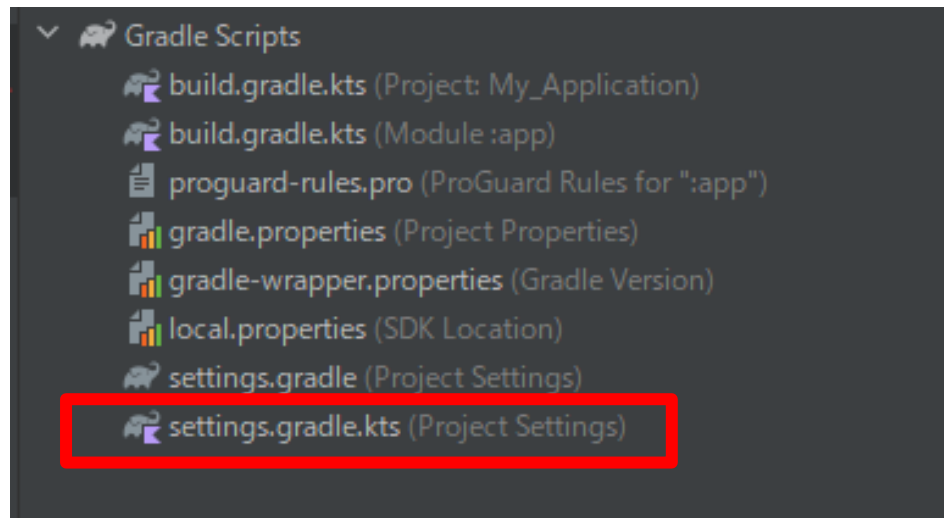
2. app/src/mainフォルダ内に『jniLibs』フォルダを作成して、ペースト



JDK 18に変更



setting.gradle.ktsを設定



```
rootProject.name = "test"
include( ...projectPaths: ":app")
include( ...projectPaths: ":sdk")
```



追加

sdkフォルダ内のbuild.gradle.kts を設定

Android {}内に以下を記述

```
compileOptions {  
    sourceCompatibility JavaVersion.VERSION_18  
    targetCompatibility JavaVersion.VERSION_18  
}  
  
buildFeatures {  
    aidl = true  
    buildConfig = true  
}  
  
java {  
    toolchain {  
        languageVersion = JavaLanguageVersion.of(18)  
    }  
}
```

```
android {  
  
    compileSdk = 33  
    namespace 'org.opencv'  
  
    defaultConfig {  
        minSdk = 33  
        targetSdk = 33  
    }  
}
```

変更

追加

```
android { this: BaseAppModuleExtension  
    namespace = "com.example.test"  
    compileSdk = 33  
  
    defaultConfig { this: ApplicationDefaultConfig  
        applicationId = "com.example.test"  
        minSdk = 33  
        targetSdk = 33  
        versionCode = 1  
        versionName = "1.0"  
        applicationId = "opencv.org"
```

追加

build.gradle.kts (app) を設定

```
dependencies { this: DependencyHandlerScope
    implementation(project(":sdk"))
    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:")
    implementation("androidx.activity:activity-compose:1.7.2")
```

追加

```
compileOptions { this: CompileOptions
    sourceCompatibility = JavaVersion.VERSION_18
    targetCompatibility = JavaVersion.VERSION_18
}
kotlinOptions { this: KotlinJvmOptions
    jvmTarget = "18"
}
```

defaultConfig {}内に以下を記述

```
sourceSets { this: NamedDomainObjectContainer<out AndroidSourceSet>
    getByName( name: "main") { this: AndroidSourceSet
        jniLibs.srcDirs("jniLibs")
    }
}

ndk { this: Ndk
    abiFilters += listOf("x86_64", "x86", "armeabi-v7a", "arm64-v8a")
}
```

```
android { this: BaseAppModuleExtension
    namespace = "com.example.test"
    compileSdk = 33

    defaultConfig { this: ApplicationDefaultConfig
        applicationId = "com.example.test"
        minSdk = 33
        targetSdk = 33
        versionCode = 1
        versionName = "1.0"
        applicationId = "opencv.org"
```

追加

TensorFlow Lightの準備

TF Liteを使えるようにする

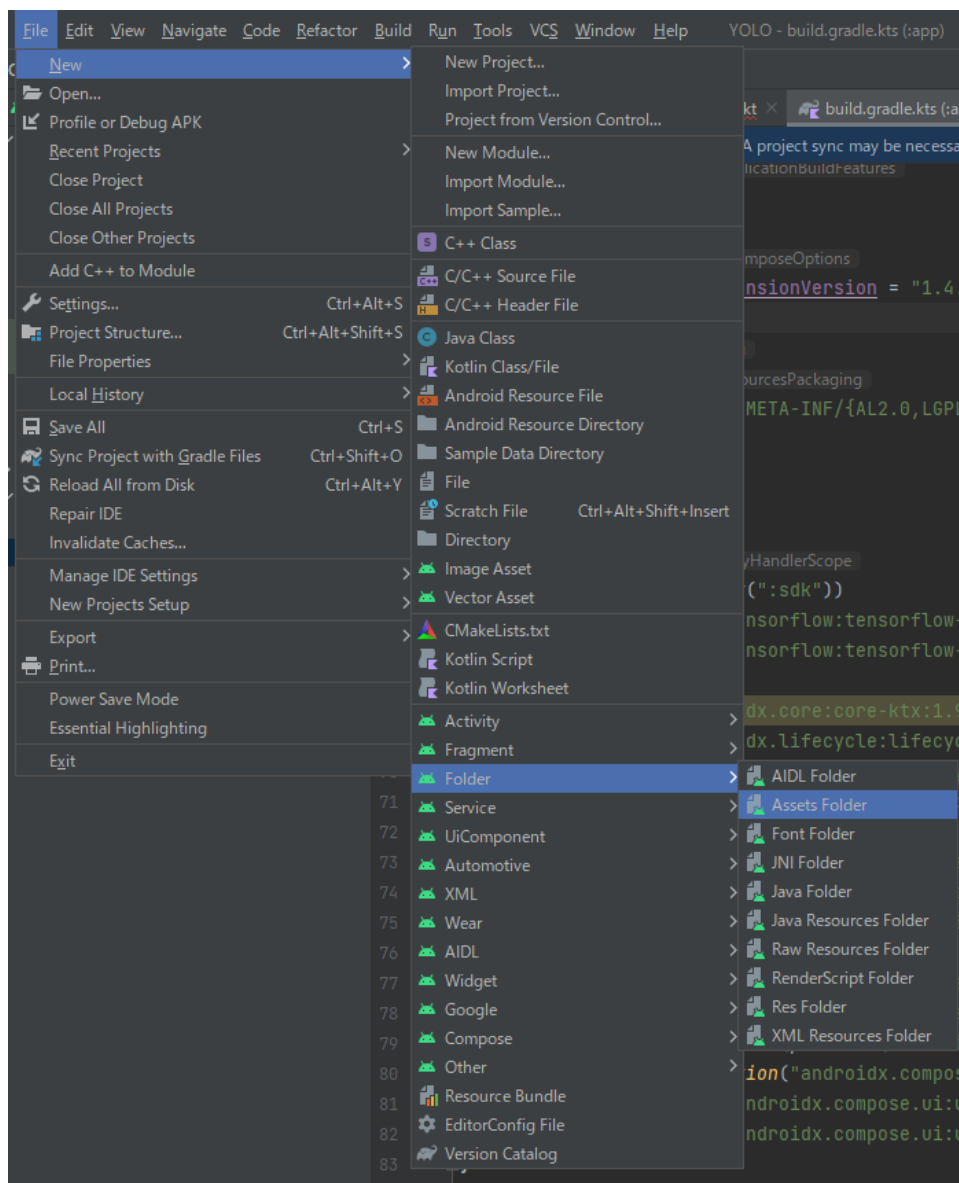
build.gradle.kts (app) を設定

```
dependencies { this: DependencyHandlerScope
    implementation(project(":sdk"))
    implementation("org.tensorflow:tensorflow-lite:2.13.0")
    implementation("org.tensorflow:tensorflow-lite-support:0.4.4")
```

追加

モデルとラベルの準備

Assetsフォルダを作成



File → New → Folder → Assets Folderを選択



app¥src¥main¥assets が自動的に作成される

ダウンロード

モデル、ラベルをダウンロードして、Assetsフォルダに設置



モデルURL

https://tfhub.dev/tensorflow/lite-model/ssd_mobilenet_v1/1/metadata/2

ラベルURL

https://raw.githubusercontent.com/SY-BETA/ObjectDetectionApp/master/app/src/main/assets/coco_dataset_labels.txt

そのままだとファイル名が長すぎるので、以下のように変更

 coco_dataset_labels.txt
 ssd_mobilenet_v1.tflite

UI構築

ボタンの作成

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            YOLOTheme {
                Surface(
                    color = MaterialTheme.colorScheme.background
                ) {
                    SelectedButton()
                }
            }
        }
    }
}

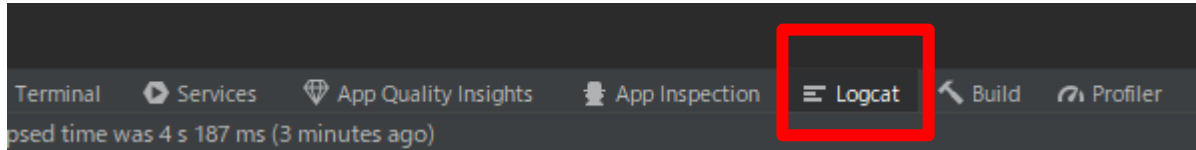
@Composable
fun SelectedButton() {
    Button(
        onClick = {
            Log.d( tag: "ButtonActivity", msg: "pushed")
        },
        modifier = Modifier.padding(16.dp)
    ){ this: RowScope
        Text(text = "SELECT IMAGE")
    }
}
```

ボタンを押すと、Logが表示



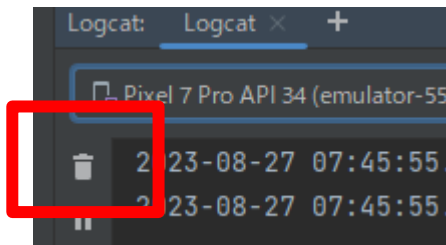
Logcatでログを試みる

Logcat



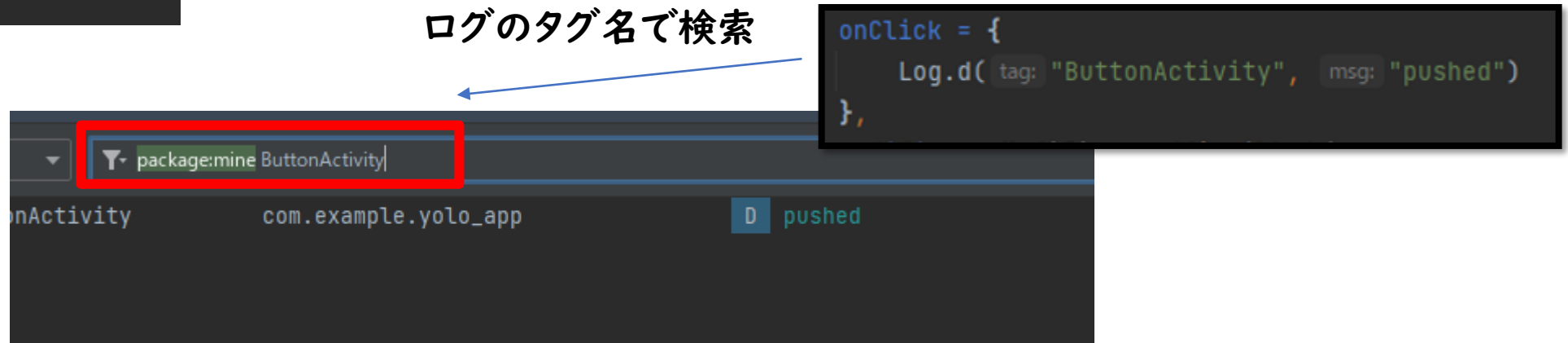
ログ削除

ログが多すぎて見辛い場合、ごみ箱ボタンで消去



ログ検索

ログのタグ名で検索



キャンバスの作成

```
const val DISPLAY_SIZE_DP = 300 // ディスプレイサイズ(dp)
```

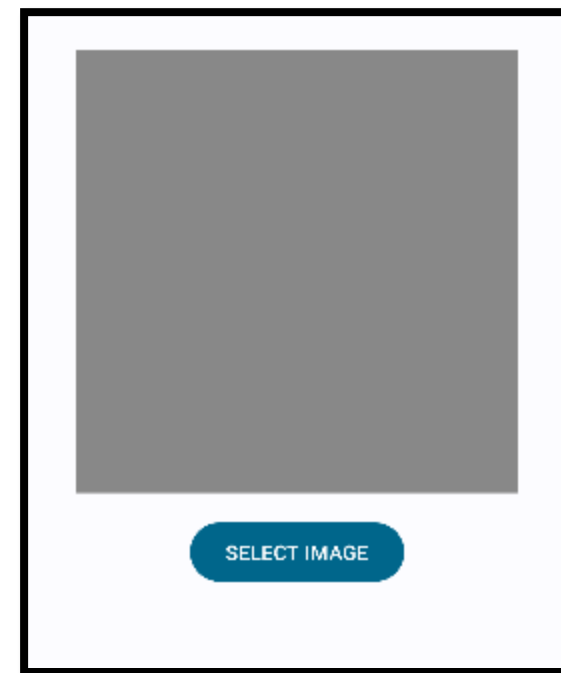
```
@Composable
fun SelectedButton() {
    Canvas(modifier = Modifier.size(DISPLAY_SIZE_DP.dp), onDraw = { this: DrawScope
        drawRect(color = Color.Gray)
    })
    Button(
        onClick = {
            Log.d(tag: "ButtonActivity", msg: "pushed")
        },
        modifier = Modifier.padding(16.dp)
    ){ this: RowScope
        Text(text = "SELECT IMAGE")
    }
}
```



ボタンの下に
灰色の矩形が描画

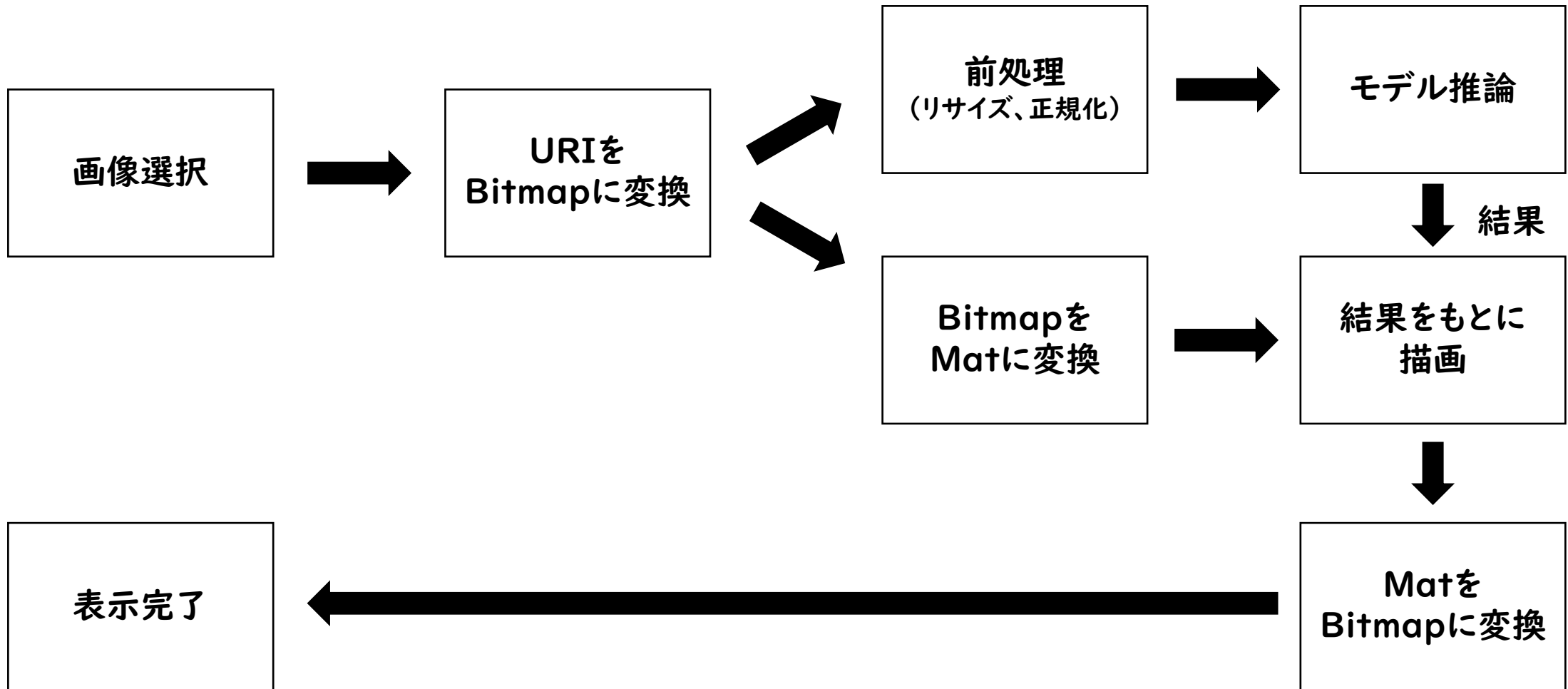
レイアウトを修正

```
@Composable
fun SelectedButton() {
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ){ this: ColumnScope
        Canvas(modifier = Modifier.size(DISPLAY_SIZE_DP.dp), onDraw = { this: DrawScope
            drawRect(color = Color.Gray)
        })
        Button(
            onClick = {
                Log.d( tag: "ButtonActivity", msg: "pushed")
            },
            modifier = Modifier.padding(16.dp)
        ){ this: RowScope
            Text(text = "SELECT IMAGE")
        }
    }
}
```

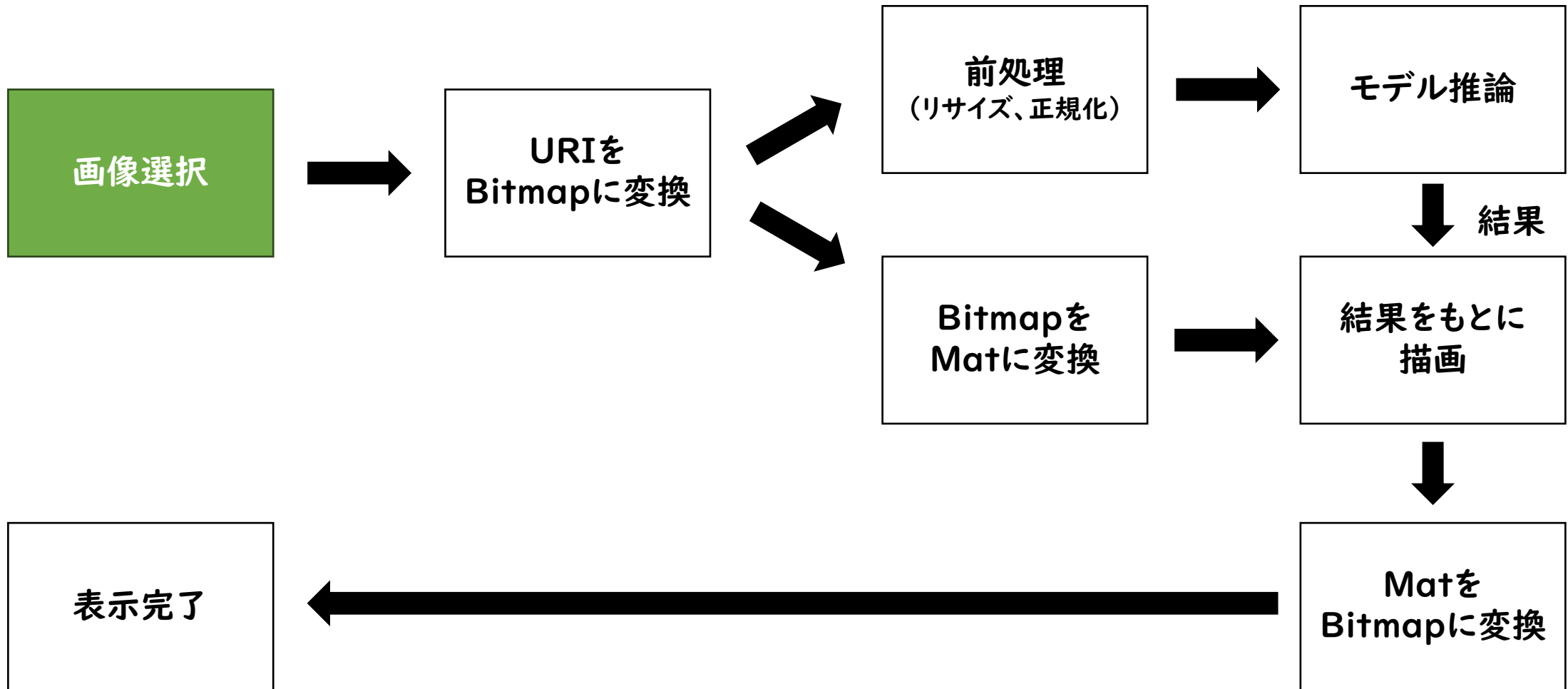


Columnでまとめ、中央ぞろえにする

処理全体の流れ



画像選択



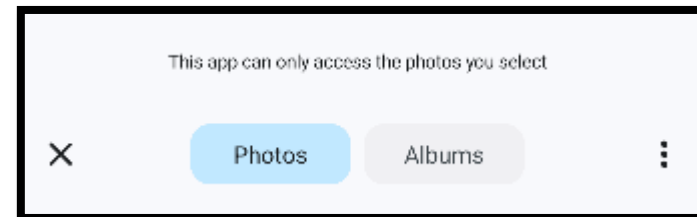
ボタンを押すと、画像ギャラリーを表示

```
@Composable
fun SelectedButton() {
    // 選択された画像のURIを保持
    var imageUrl by remember { mutableStateOf<Uri?>(value: null) }

    // ギャラリーから画像を選択するためのアクティビティ結果コントラクトを宣言
    val launcher = rememberLauncherForActivityResult(ActivityResultContracts.GetContent()) { uri: Uri? ->
        imageUrl = uri
    }

    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ){ this: ColumnScope
        Canvas(modifier = Modifier.size(DISPLAY_SIZE_DP.dp), onDraw = { this: DrawScope
            // わかりやすいように矩形を描画
            drawRect(
                color = Color.Magenta
            )
        })
        Button(
            onClick = {
                // 画像ギャラリーの表示 (選択画像URIはimageUrlに格納)
                launcher.launch(input: "image/*")
            },
            modifier = Modifier.padding(16.dp)
        ){ this: RowScope
            Text(text = "SELECT IMAGE")
        }
    }
}
```

※ URI = データ という認識でOK

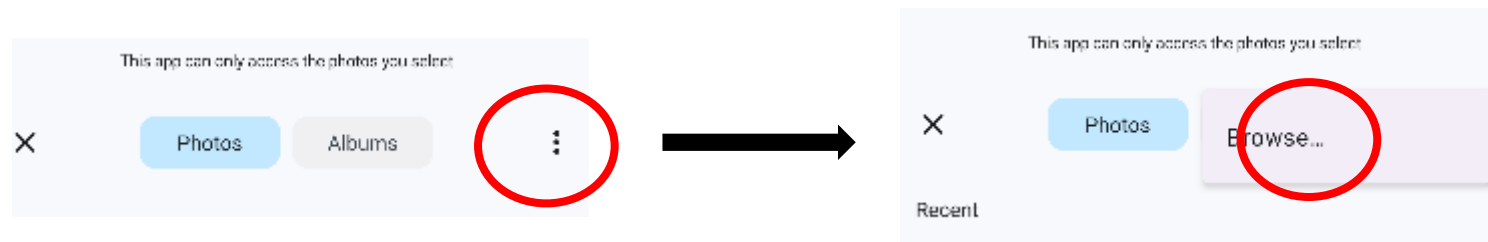


仮想デバイスに画像を追加

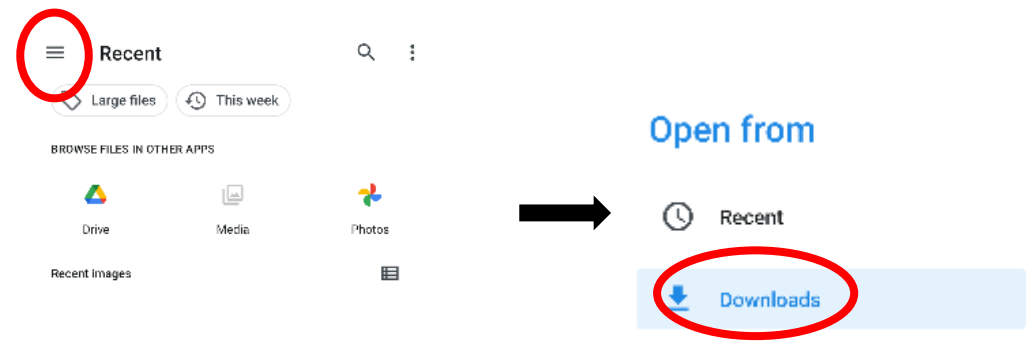
仮想デバイスには画像がない、追加する必要



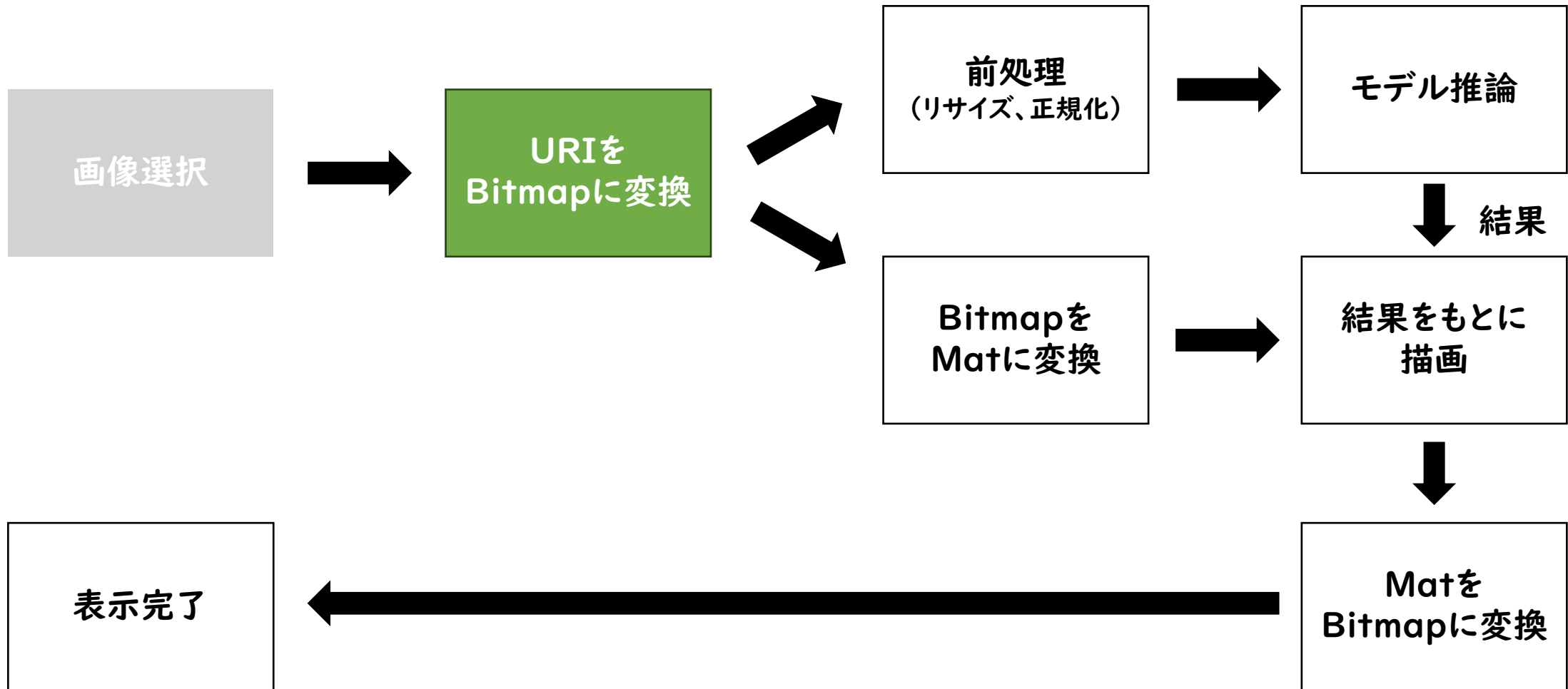
1. 仮想デバイスに画像をドラッグ&ドロップ
2. Browseボタンを選択



3. 左上のボタンから「Downloads」を選択、その中にドラッグ&ドロップした画像がある



URIをBitmapに変換



コンテキストを使用

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
  
    val context = applicationContext  
    setContent {  
        YOLOTheme {  
            Surface(  
                color = MaterialTheme.colorScheme.background  
            ) {  
                SelectedButton(context)  
            }  
        }  
    }  
}
```

Contextを作成

アプリケーション側がデータにアクセス
できるようにするため
(URIの操作に必要)

次のスライドで説明

URIをBitmapに変換

```
// Composableでない関数の場合, contextを渡す必要
private fun uri2bitmap (uri: Uri, context: Context): Bitmap {
    val source = ImageDecoder.createSource(context.contentResolver, uri)
    return ImageDecoder.decodeBitmap(source)
}
```

Composableな関数でない場合に
contextを使うなら引数として渡す必要



そのため、contextを渡した

```
SelectedButton(context)
```

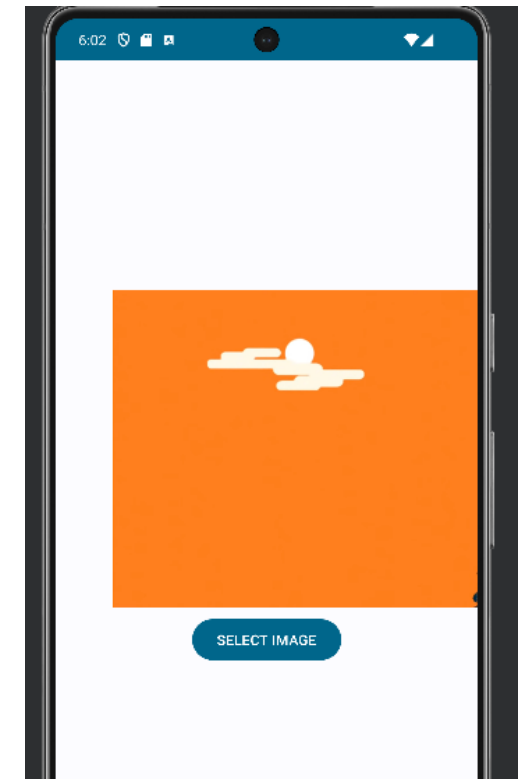
1. アプリケーション側がURIにアクセス
2. ソースを作成
3. ソースをデコードしてBitmapにする

ボタンが押されたら、URIをBitmapに変換

```
Canvas(modifier = Modifier.size(DISPLAY_SIZE_DP.dp), onDraw = { this: DrawScope  
    if (imageUri == null) {drawRect(color = Color.Gray)}  
    imageUri?.let { uri ->  
        val bitmap = uri2bitmap(uri, context)  
  
        // 一旦、画像を表示させてみる  
        val imageBitmap = bitmap.asImageBitmap()  
        drawImage(imageBitmap)  
    }  
})
```

imageUriに中身が入った場合の処理

キャンバスの範囲外まで描画される



画像をキャンバスサイズに合わせる

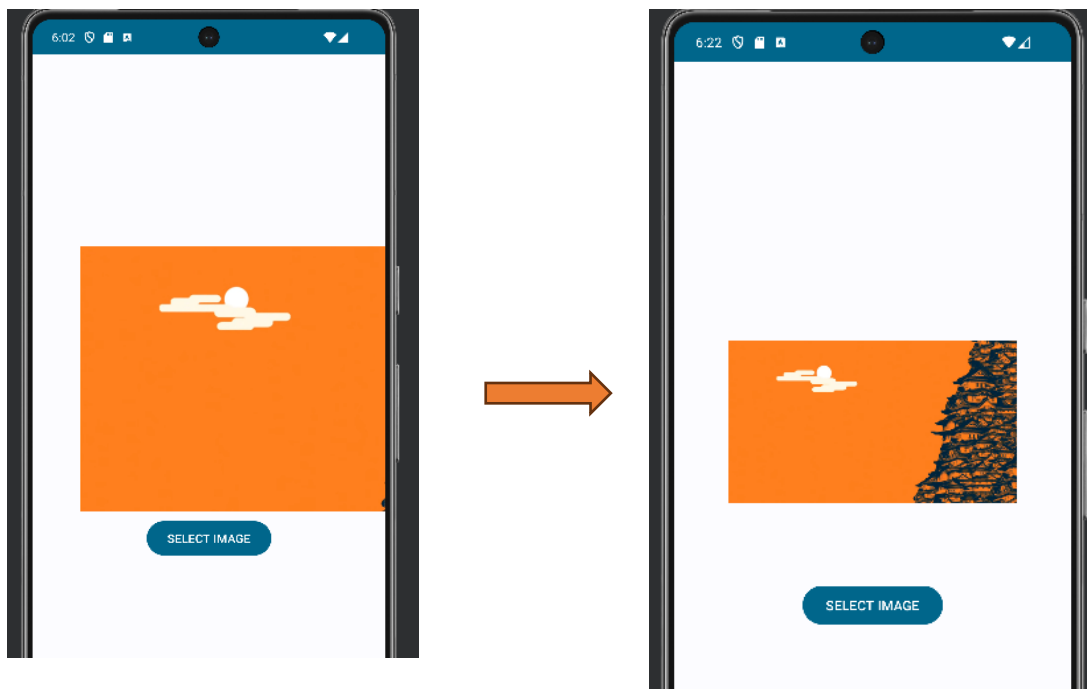
```
imageUri?.let { uri ->
    var bitmap = uri2bitmap(uri, context)
    bitmap = resizedBitmap(bitmap)    // リサイズ

    // 一旦、画像を表示させてみる
    val imageBitmap = bitmap.asImageBitmap()
    drawImage(imageBitmap, topLeft = calcCanvasCenter(size.height))
}
```

```
val context = applicationContext
val metrics = context.resources.displayMetrics
canvasPixel = DISPLAY_SIZE_DP * metrics.density    // dp -> pixel
```

```
// キャンバスサイズの幅に合わせてリサイズ
private fun resizedBitmap(bitmap: Bitmap): Bitmap{
    val w = bitmap.width.toFloat()
    val h = bitmap.height.toFloat()
    val ratio = canvasPixel / w
    resizedW = (w * ratio).toInt()
    resizedH = (h * ratio).toInt()
    return Bitmap.createScaledBitmap(bitmap, resizedW, resizedH, filter: true)
}

// キャンバスの中心に描画できるように位置を計算
private fun calcCanvasCenter(canvasH: Float): Offset{
    val top = (canvasH - resizedH) / 2
    return Offset(x = 0f, y = top)
}
```



物体検出処理の準備

モデルとラベルを読み込む準備

MainActivityクラス内で以下を記述

```
// Assetsフォルダ内にあるモデル、ラベルの名称
companion object {
    private const val MODEL_FILE_NAME = "ssd_mobilenet_v1.tflite"
    private const val LABEL_FILE_NAME = "coco_dataset_labels.txt"
}

// tfliteモデルを扱うためのラッパーを含んだinterpreter
private val interpreter: Interpreter by lazy {
    Interpreter(loadModel())
}

// モデルの正解ラベルリスト
private val labels: List<String> by lazy {
    loadLabels()
}
```

```
// AssetsフォルダからTF Liteモデルを読み込む
private fun loadModel(fileName: String = MODEL_FILE_NAME): ByteBuffer {
    lateinit var modelBuffer: ByteBuffer
    var file: AssetFileDescriptor? = null
    try {
        file = assets.openFd(fileName)
        val inputStream = FileInputStream(file.fileDescriptor)
        val fileChannel = inputStream.channel
        modelBuffer = fileChannel.map(FileChannel.MapMode.READ_ONLY, file.startOffset, file.declaredLength)
    } catch (e: Exception) {
        Toast.makeText(context: this, text: "モデルファイル読み込みエラー", Toast.LENGTH_SHORT).show()
        finish()
    } finally {
        file?.close()
    }
    return modelBuffer
}

// Assetsフォルダからラベルデータを取得
private fun loadLabels(fileName: String = LABEL_FILE_NAME): List<String> {
    var labels = listOf<String>()
    var inputStream: InputStream? = null
    try {
        inputStream = assets.open(fileName)
        val reader = BufferedReader(InputStreamReader(inputStream))
        labels = reader.readLines()
    } catch (e: Exception) {
        Toast.makeText(context: this, text: "txtファイル読み込みエラー", Toast.LENGTH_SHORT).show()
        finish()
    } finally {
        inputStream?.close()
    }
    return labels
}
```


Object Detectorクラスの作成

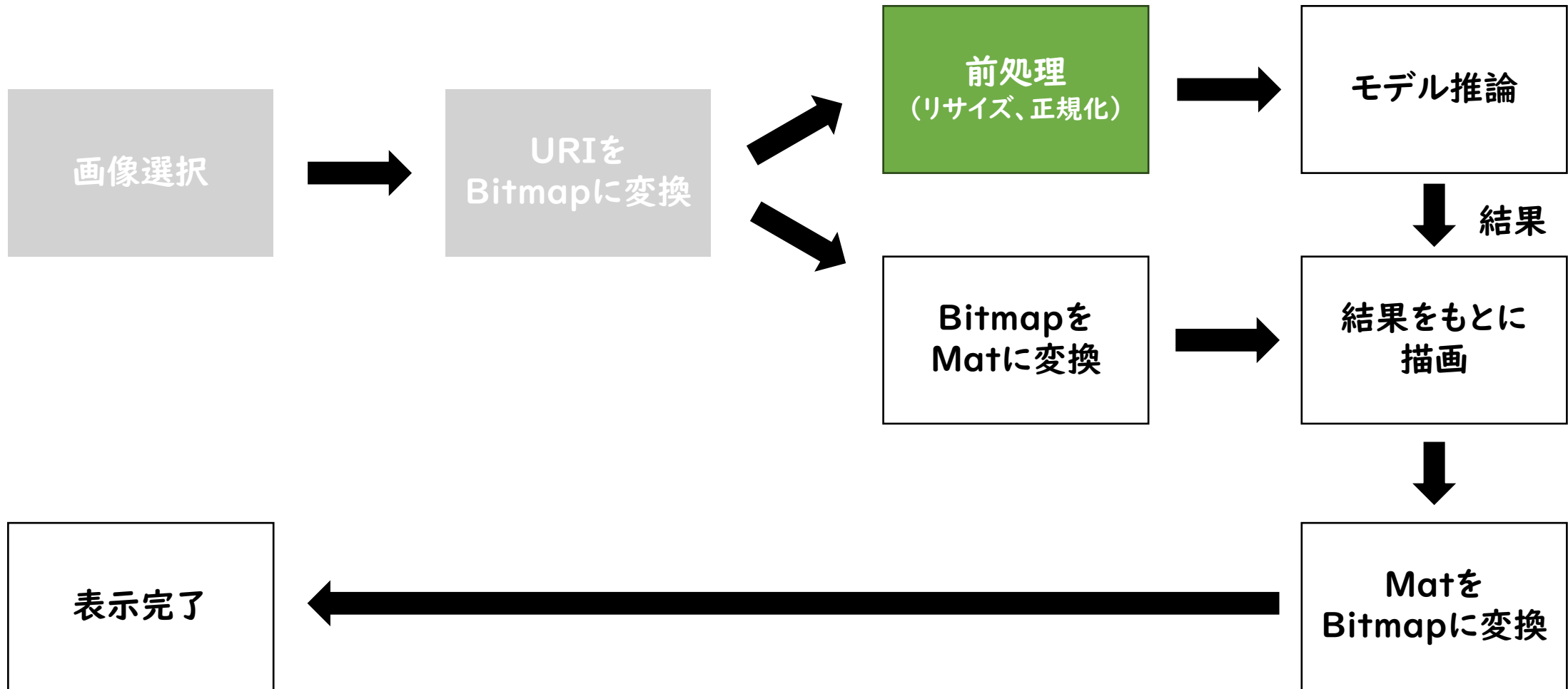
入力サイズや出力結果のデータの格納方法などを記述

```
companion object {  
    // モデルのinputとoutputサイズ  
    private const val IMG_SIZE_X = 300  
    private const val IMG_SIZE_Y = 300  
    private const val MAX_DETECTION_NUM = 10    // TFLite変換時に設定されているので変更不可  
  
    // 利用するTF Liteモデルは量子化済み | normalize関連は127.5fではなく以下の通り  
    private const val NORMALIZE_MEAN = 0f  
    private const val NORMALIZE_STD = 1f  
  
    // 検出結果のスコアしきい値  
    private const val SCORE_THRESHOLD = 0.5f  
}
```

```
// 検出結果クラス  
@data class DetectionObject(  
    val score: Float,  
    val label: String,  
    val boundingBox: Rect  
)
```

```
// バウンディングボックス [1:10:4], 10は物体検出数, 4は4隅の座標  
private val outputBoundingBoxes: Array<Array<FloatArray>> = arrayOf(  
    Array(MAX_DETECTION_NUM) { it: Int  
        FloatArray(size: 4)    // 4隅: [top, left, bottom, right]  
    }  
)  
  
// クラスラベルインデックス [1:10], 10は物体検出数  
private val outputLabels: Array<FloatArray> = arrayOf(  
    FloatArray(MAX_DETECTION_NUM)  
)  
  
// 各スコア [1:10], 10は物体検出数  
private val outputScores: Array<FloatArray> = arrayOf(  
    FloatArray(MAX_DETECTION_NUM)  
)  
  
// 物体検出数 = 10  
private val outputDetectionNum: FloatArray = FloatArray(size: 1)  
  
// 検出結果をmapでまとめる  
private val outputMap = mapOf(  
    0 to outputBoundingBoxes,    // バウンディングボックス  
    1 to outputLabels,           // ラベル  
    2 to outputScores,           // スコア  
    3 to outputDetectionNum       // 検出数  
)
```

前处理



前処理

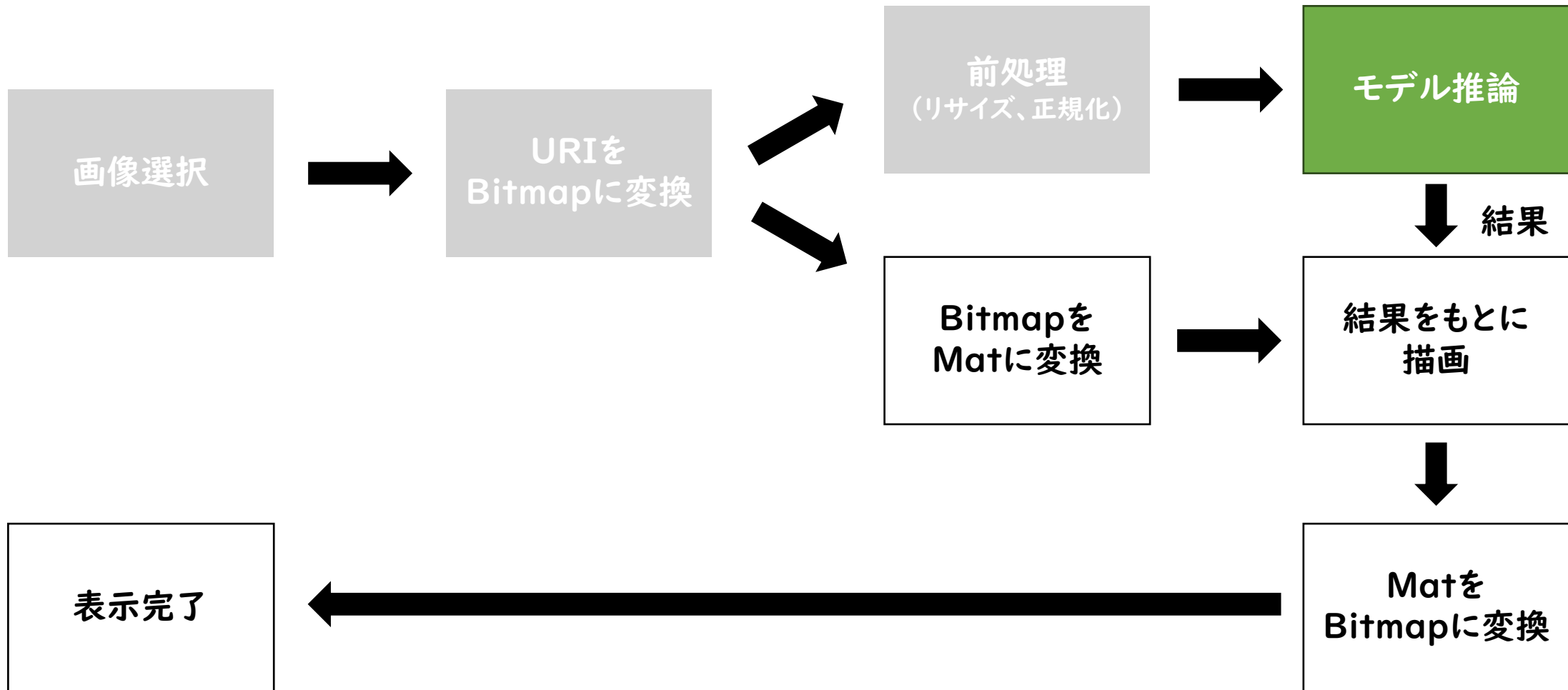
モデルに入力するためのサイズの変更、正規化を行う

```
// Bitmap -> TensorFlowBuffer
tfImageBuffer.load(targetBitmap)

// 前処理(画像サイズの変更、正規化)
val tensorImage = tfImageProcessor.process(tfImageBuffer)
println(tensorImage)
```

```
private val tfImageProcessor by lazy {
    ImageProcessor.Builder()
        .add(ResizeOp(IMG_SIZE_X, IMG_SIZE_Y, ResizeOp.ResizeMethod.BILINEAR)) // 画像サイズの変更
        .add(NormalizeOp(NORMALIZE_MEAN, NORMALIZE_STD)) // 正規化
        .build()
}
```

モデル推論



モデル推論

推論を実行して
各結果をまとめる

```
// 推論実行, outputMap内に格納
interpreter.runForMultipleInputsOutputs(arrayOf(tensorImage.buffer), outputMap)

// 推論結果(outputMap)内にある4つの情報を整形, リストにして返す
val detectedObjectList = arrayListOf<DetectionObject>()
loop@ for (i in 0 until < outputDetectionNum[0].toInt()) {
    val score = outputScores[0][i]
    val label = labels[outputLabels[0][i].toInt()]
    val boundingBox = Rect(
        (outputBoundingBoxes[0][i][1] * w).toInt(),
        (outputBoundingBoxes[0][i][0] * h).toInt(),
        (outputBoundingBoxes[0][i][3] * w).toInt(),
        (outputBoundingBoxes[0][i][2] * h).toInt()
    )

    // 閾値より大きければリストに追加, 検出結果はソートされているため閾値以下なら処理終了
    if (score >= SCORE_THRESHOLD) {
        detectedObjectList.add(
            DetectionObject(
                score = score,
                label = label,
                boundingBox = boundingBox
            )
        )
    } else {
        break@loop
    }
}

return detectedObjectList.take(n: 4) // 先頭4つ, つまり上位4つの要素を返す
```

MainActivity側からの処理

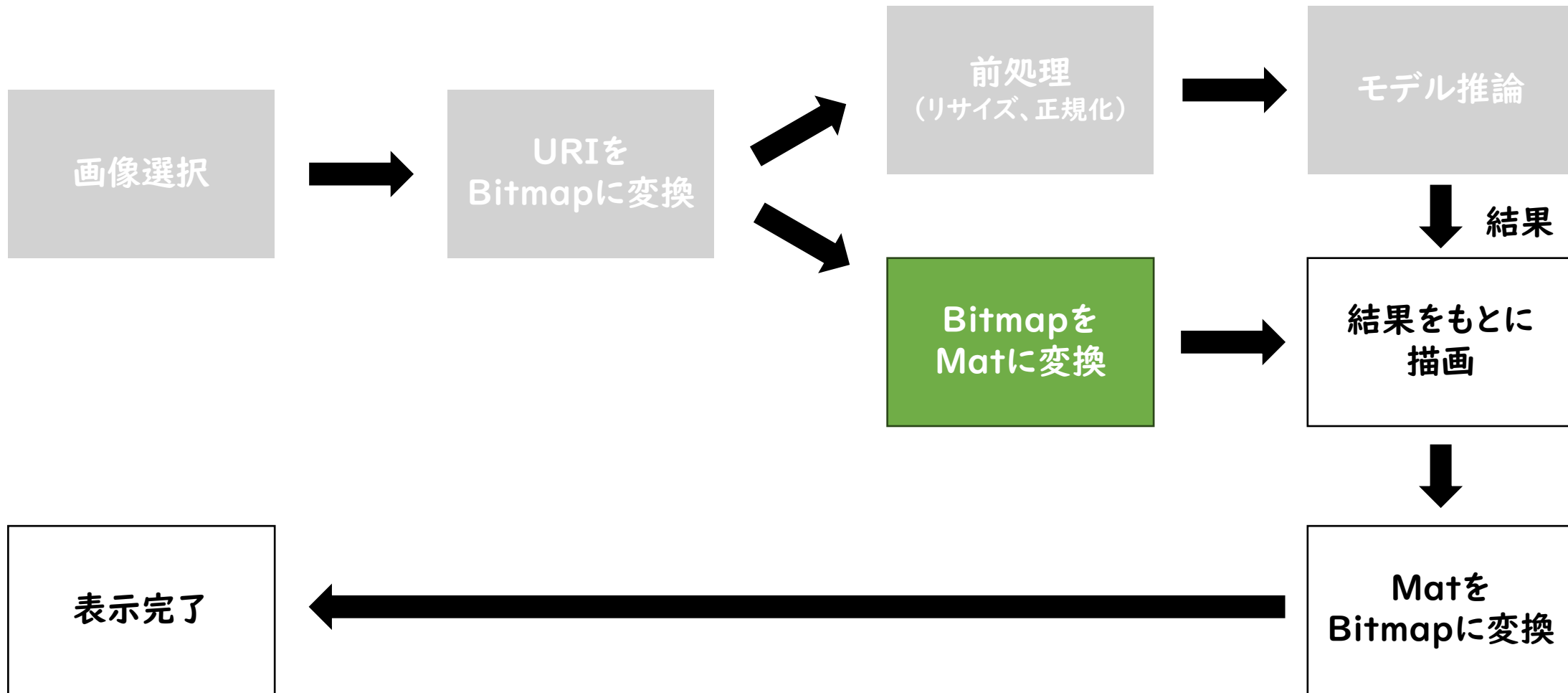
ObjectDetectorクラスを呼び出す

MainActivityクラスからObjectDetectorクラスを呼び出す

```
private fun detect(bitmap: Bitmap) : List<DetectionObject>{  
    val od = ObjectDetector(interpreter, labels)  
    return od.detect(bitmap)  
}
```

読み込んだモデルとリストを渡している

BitmapをMatに変換



OpenCVエラー対策

OpenCVのエラーに対応するために、2点修正

```
override fun onCreate(savedInstanceState: Bundle?) {  
    // OpenCV初期化  
    if (!OpenCVLoader.initDebug()) {  
        println("error")  
    }  
    super.onCreate(savedInstanceState)  
}
```

OpenCV初期化

```
// Composableでない関数の場合、contextを渡す必要  
private fun uri2bitmap (uri: Uri, context: Context): Bitmap {  
    val source = ImageDecoder.createSource(context.contentResolver, uri)  
    return ImageDecoder.decodeBitmap(source){ decoder, _, _ ->  
        decoder.isMutableRequired = true  
    }  
}
```

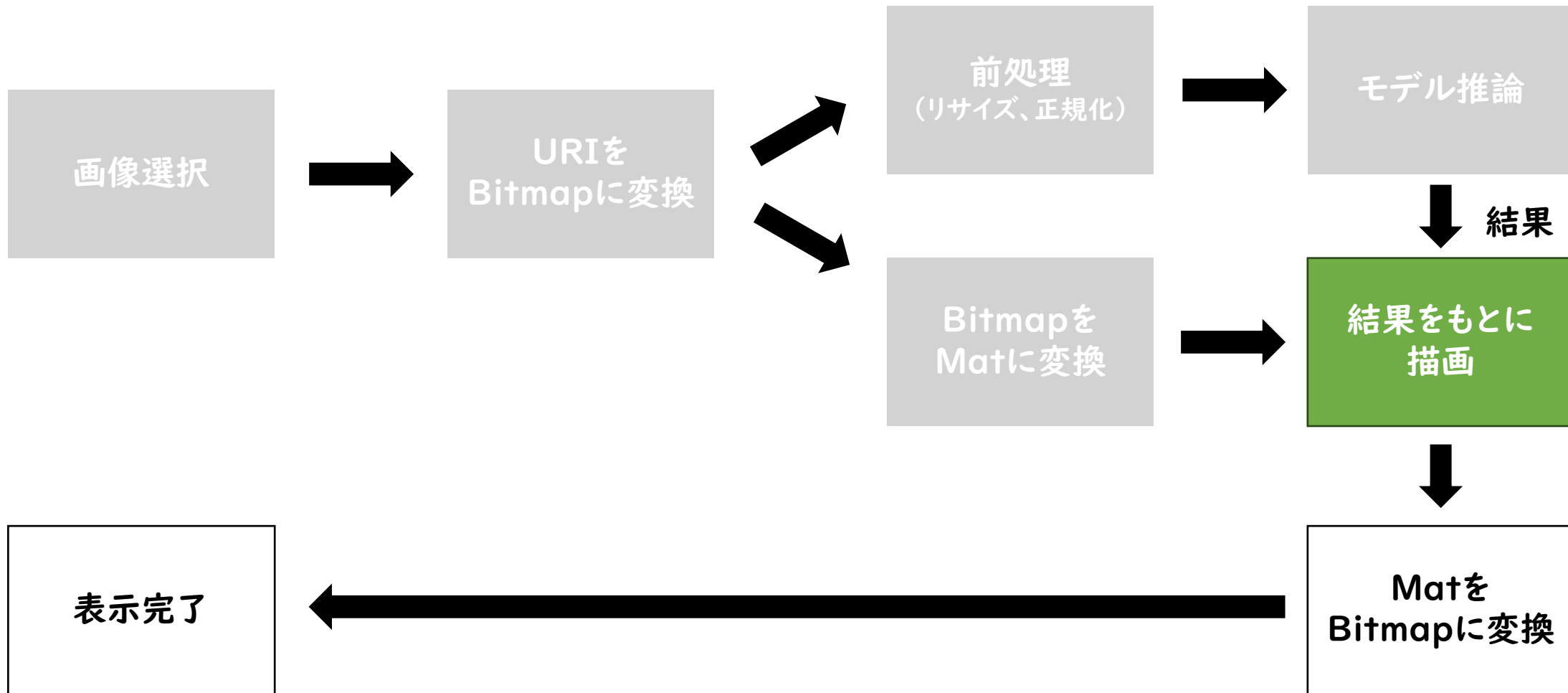
デフォルトで不変なため、変更可能にするよう変更

BitmapをMatに変換

```
// Bitmap -> Mat  
val drawMat = Mat()  
Utils.bitmapToMat(bitmap, drawMat)
```

OpenCV側が用意している関数で
変換可能

結果をもとに描画



描画

```
// 色空間の変更 (OpenCVの仕様です)
Imgproc.cvtColor(drawMat, drawMat, Imgproc.COLOR_BGR2RGB)

for (i in detectList.indices) {
    val element = detectList[i]
    val bboxCoord = element.boundingBox
    val label = element.label
    val score = element.score

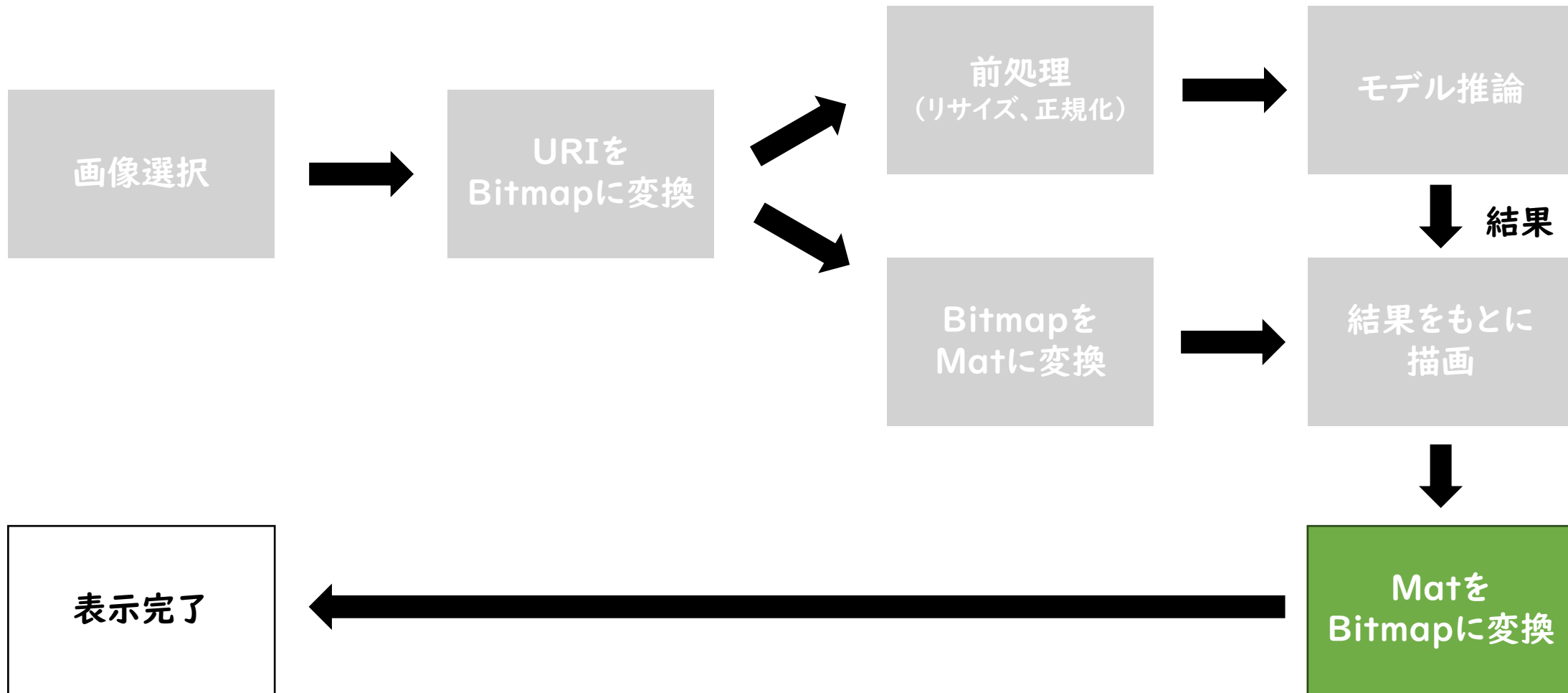
    // 矩形の描画
    Imgproc.rectangle(drawMat, bboxCoord, Scalar(v0: 100.0, v1: 255.0, v2: 0.0), thickness: 4)

    // ラベルの描画
    val labelCoord = Point(bboxCoord.x.toDouble(), bboxCoord.y.toDouble())
    Imgproc.putText(drawMat, label, labelCoord, fontFace: 3, fontScale: 1.5, Scalar(v0: 255.0, v1: 0.0, v2: 0.0), thickness: 2)

    // スコアの描画
    val pointCoord = Point(bboxCoord.x.toDouble(), y: bboxCoord.y.toDouble() - 40.0)
    Imgproc.putText(drawMat, score.toString(), pointCoord, fontFace: 3, fontScale: 0.8, Scalar(v0: 255.0, v1: 255.0, v2: 255.0), thickness: 2)
}

// 色空間を元に戻す
Imgproc.cvtColor(drawMat, drawMat, Imgproc.COLOR_RGB2BGR)
```


MatをBitmapに変換

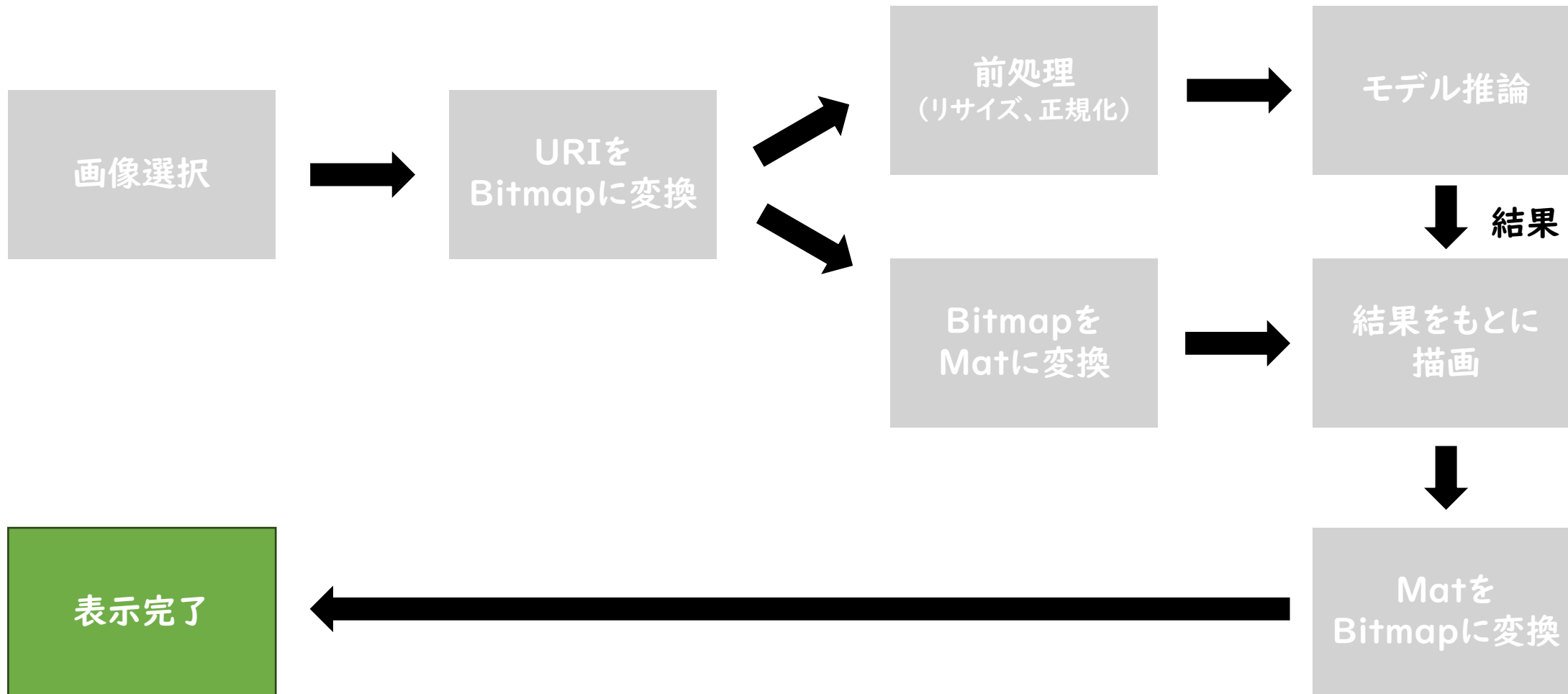


MatをBitmapに変換

画面に表示するにはBitmapでないといけないため

```
// Mat -> Bitmap  
Utils.matToBitmap(drawMat, bitmap)  
return bitmap
```

表示完了



表示

```
// uriをBitmapに変換
var bitmap = uri2bitmap(uri, context)

// 推論を実行
val detectedObjectList = detect(bitmap)

// 描画
bitmap = draw(detectedObjectList, bitmap)
```

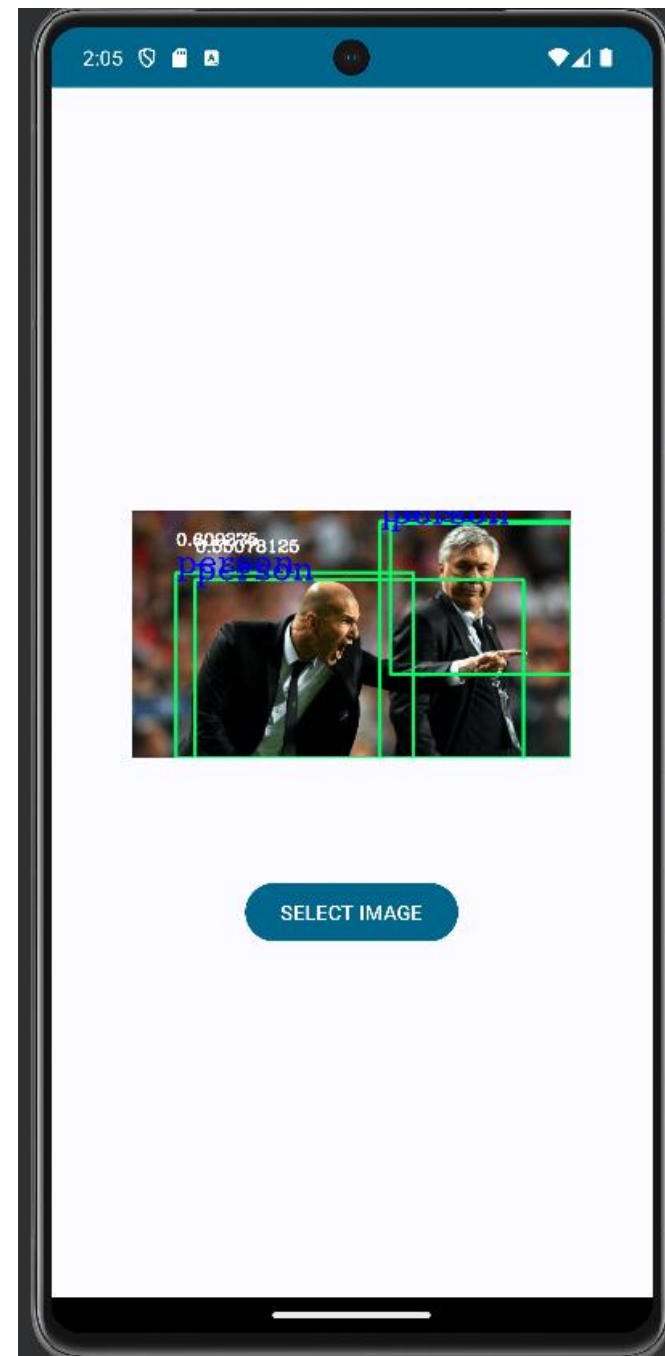
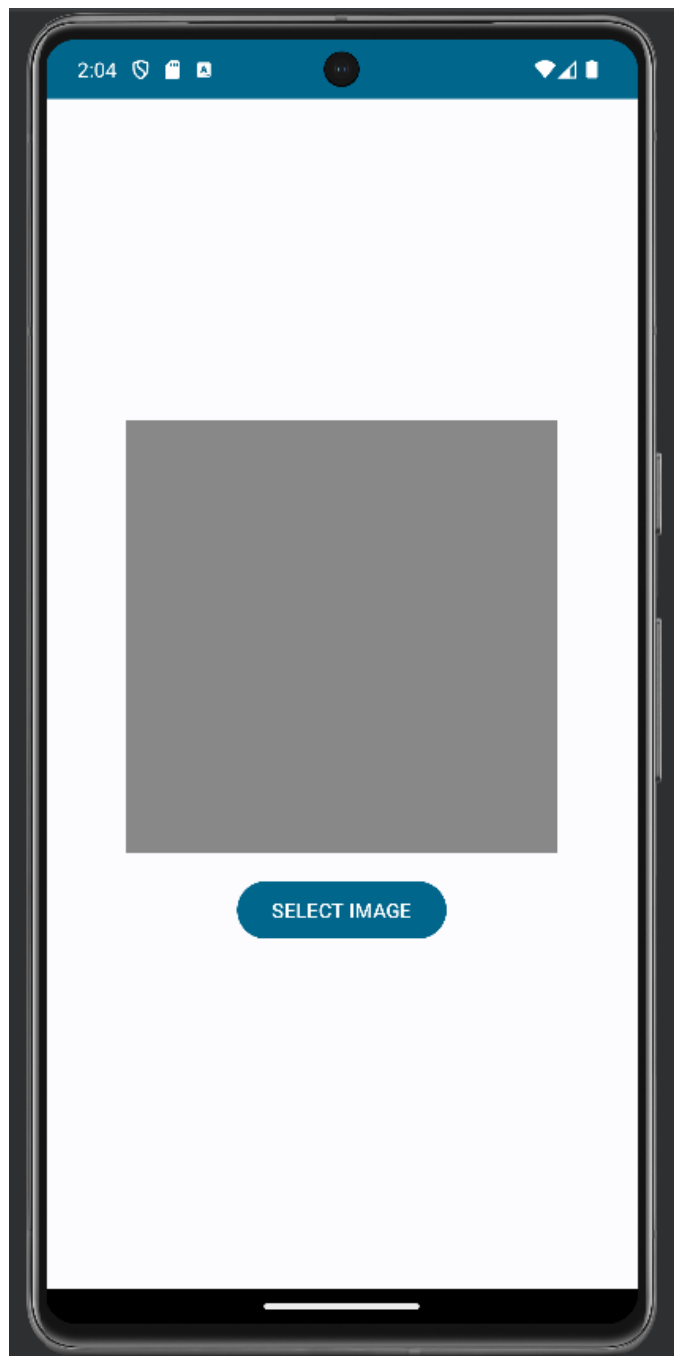
```
// リサイズ
bitmap = resizedBitmap(bitmap)

// 描画
val imageBitmap = bitmap.asImageBitmap()
drawImage(imageBitmap, topLeft = calcCanvasCenter(size.height))
```

キャンバスの中心に描画されるように
少しだけ処理

```
// キャンバスの中心に描画できるように位置を計算
private fun calcCanvasCenter(canvasH: Float): Offset{
    val top = (canvasH - resizedH) / 2
    return Offset(x = 0f, y = top)
}
```

結果



アプリをパッケージ化して
実機で動かしてみよう！

APKファイル

APKファイル

Android Application Package

- Androidスマートフォンにアプリをインストールするためのアーカイブファイル
- 配布の際に利用



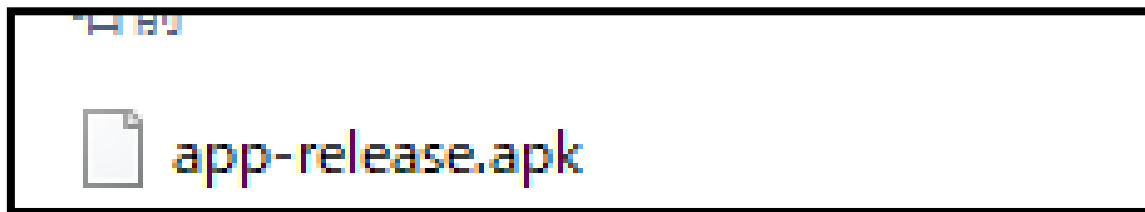
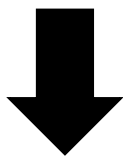
仮想デバイスや実機にUSBで接続しなくとも、アンドロイド上で動かせる！

APKファイルを作成

下記2つのサイトを見るとわかります!!

【AndroidStudio/Kotlin】Key store pathを作成する方法～APKファイルやaabを作る前にやっておくこと～
<https://ios-docs.dev/create-key-store-path/>

【Android Studio】APKファイルを作成する方法
<https://ios-docs.dev/create-apk/>

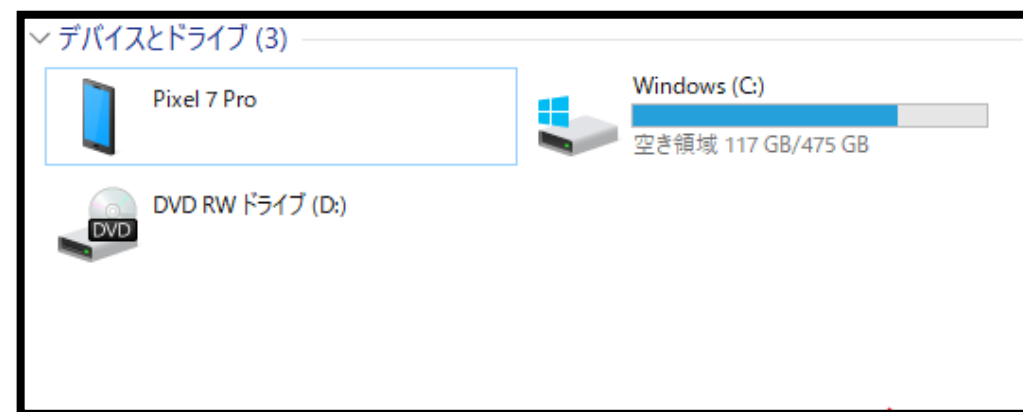


APKファイルを実機に転送①

方法 2: USB ケーブルを使ってファイルを移動する

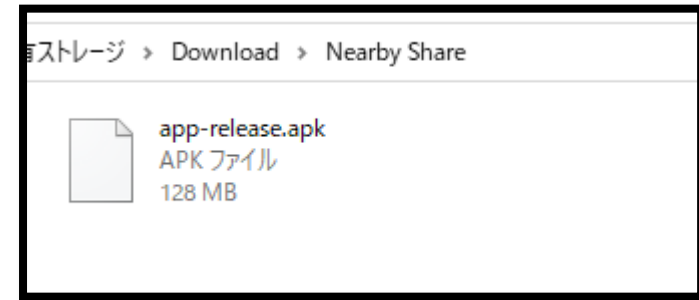
Windows の場合

1. デバイスのロックを解除します。
2. USB ケーブルを使って、デバイスをパソコンに接続します。
3. デバイスで [このデバイスを USB で充電中] 通知をタップします。
4. [USB の使用] で **[ファイル転送]** を選択します。



<https://support.google.com/android/answer/9064445?hl=ja>

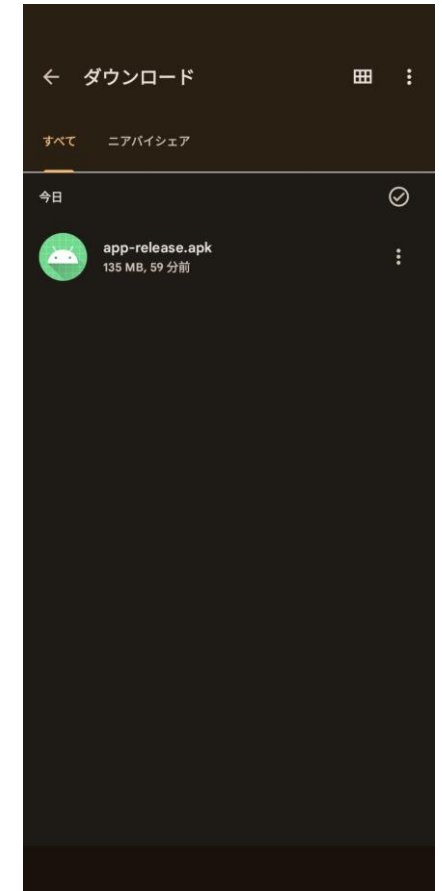
APKファイルを実機に転送②



ダウンロードフォルダにAPKファイルを設置

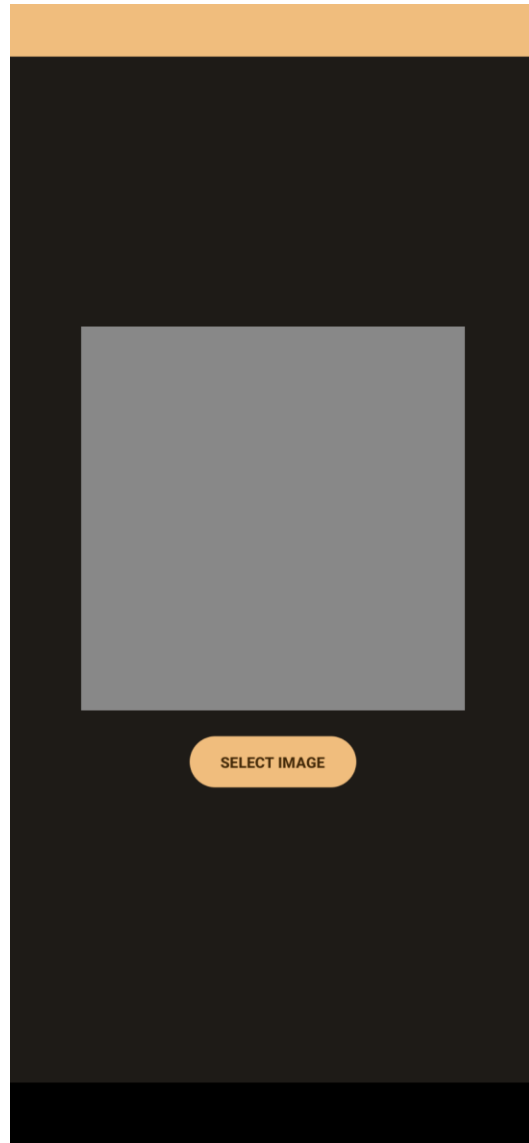
インストール

Android実機の「Files」アプリにある「ダウンロード」にAPKファイルが存在



APKファイルを押すとインストールが開始
(※不明なアプリのため、インストールできないとなるが、強制的にボタンを押せば可能)

実機で動いた



おしまい!