Cpp North 2025

Bridging C++ and Java with Qt JNI

An Android Application for On-Device Landmine Detection

Oleksandr Kunichik

Bridging C++ and Java with Qt JNI

An Android Application for On-Device Landmine Detection

Oleksandr Kunichik

The view from my balcony

About me:

- Oleksandr Kunichik, PhD
- Software Developer at PokerStars
- Founder, CollectiveSight.ai

ao669517126@gmail.com

linkedin.com/in/kunichik

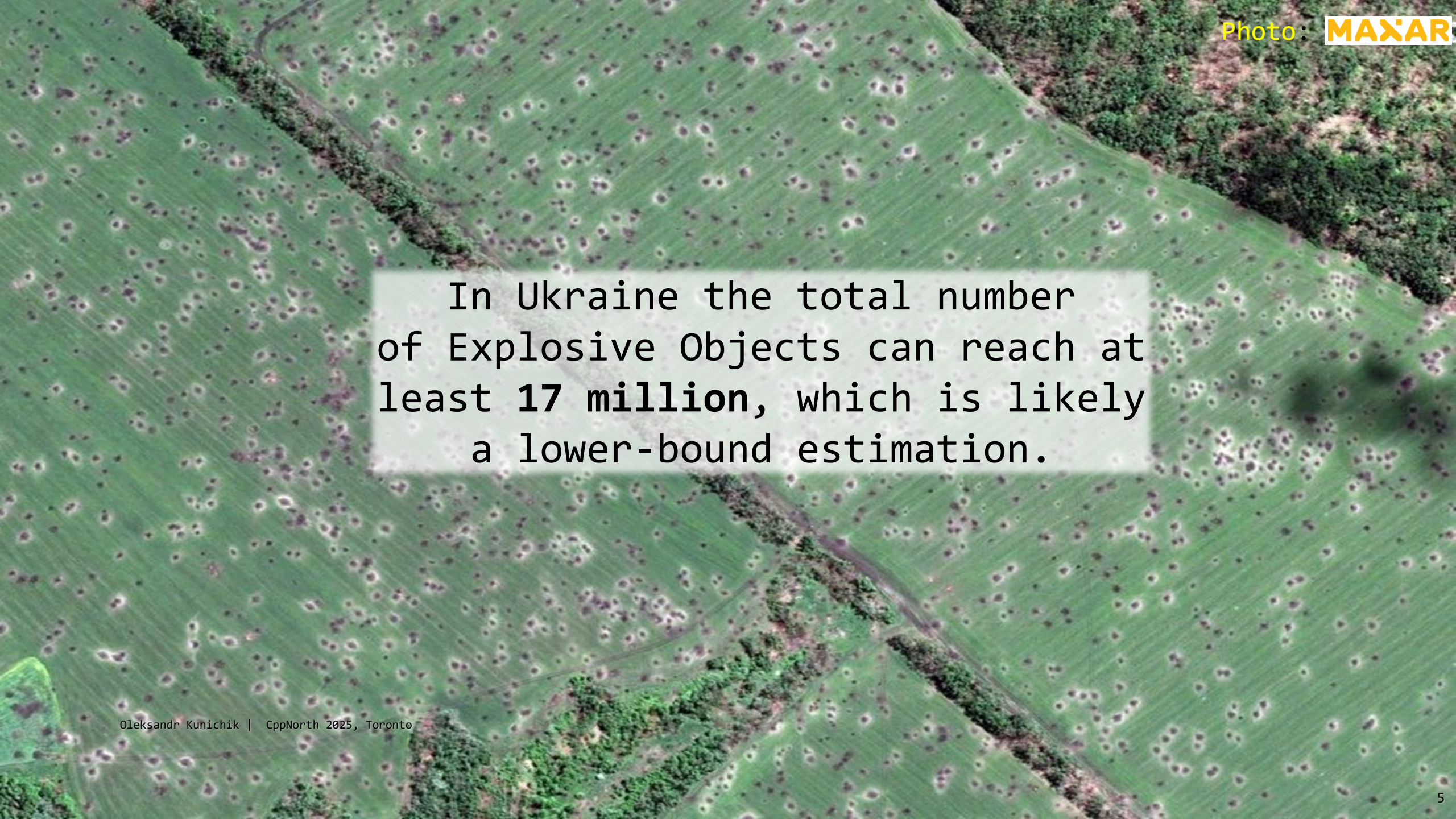
github.com/kunichik/conferences/cppnorth2025

Taras Shevchenko
National University
of Kyiv

Supervised by
Prof. Vasyl Tereshchenko

Why This Project Matters

"Technology should solve real problems"



In Ukraine the total number of Explosive Objects can reach at least **17 million**, which is likely a lower-bound estimation.

What we built

The models for landmine detection

What we needed

The data from deminers

Nobody will give you data

You must offer something

Therefore, we offered:

- Mine detection app

(using cloud detection)

Therefore, we offered:

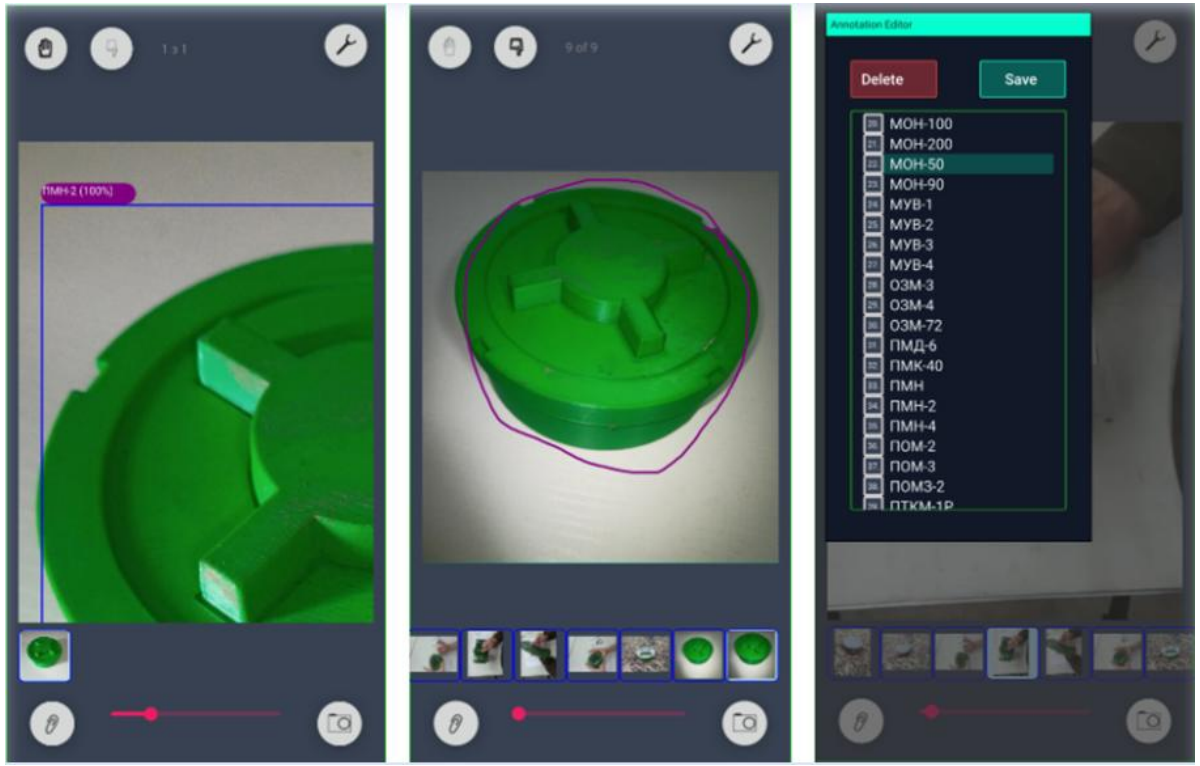
- Works on Linux,

Windows, Android, iOS*

*iOS is not tested yet

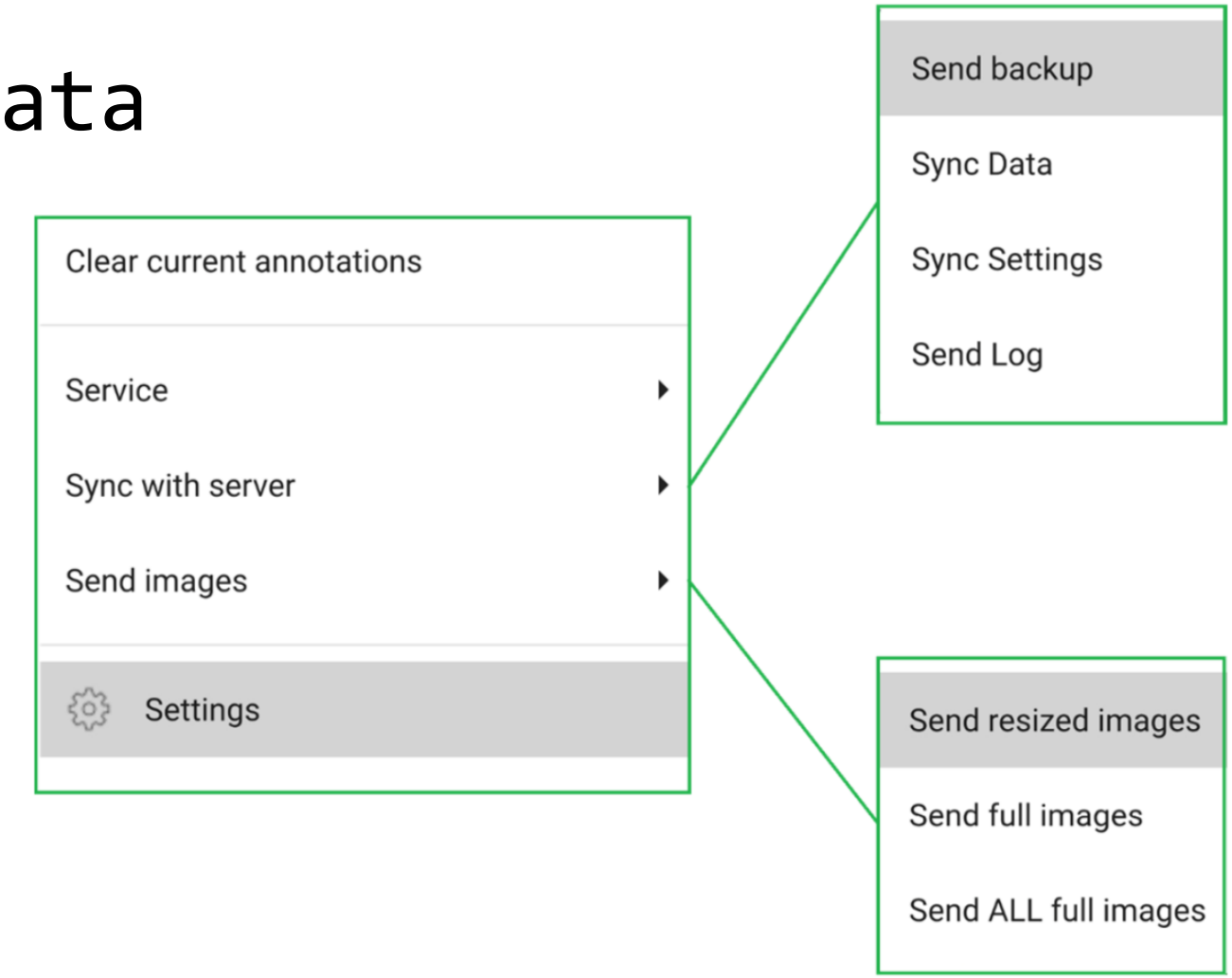
Therefore, we offered:

- Ability to edit detected mines



Therefore, we offered:

- Option to send data



We needed offline inference

On Android as it is the

most popular mobile OS*

*74% according to https://gs.statcounter.com/os-market-share/mobile/worldwide

The Stack

[QML UI*] ← QQuickView, UI bindings

```
ApplicationWindow {
    width: 400
    height: 400
    visible: true

    Button {
        id: button
        text: "A Special Button"
        background: Rectangle {
            implicitWidth: 100
            implicitHeight: 40
            color: button.down ? "#d6d6d6" : "#f6f6f6"
            border.color: "#26282a"
            border.width: 1
            radius: 4
        }
    }
}
```

*QML is a Java-script like declarative language https://doc.qt.io/qt-6/qmlapplications.html

The Stack

[QML UI] ← QQuickView, UI bindings

[C++ Logic] ← Image capture, processing

The Stack

[QML UI] ← QQuickView, UI bindings

[C++ Logic] ← Image capture, processing

[QtConcurrent] ← Async processing

The Stack

[QML UI] ← QQuickView, UI bindings

[C++ Logic] ← Image capture, processing

[QtConcurrent] ← Async processing

[Qt JNI Bridge] ← Calls to Java

[Java Wrapper] ← ONNX Runtime integration

[“ONNX[1] Runtime[2]”] ← Actual inference engine

[1] Open Neural Network Exchange (ONNX) – https://onnx.ai/

[2] ONNX Runtime – https://onnxruntime.ai/

The Challenge

```
// What we wanted:
QImage image = camera.capture();
auto detections = onnxRuntime.detect(image);
// Simple, right?
```

Java Native Interface (JNI)

- Core Android tech since API level 1
- Enables Java ⇔ C++ interaction
- We use it via QJniObject (Qt)

Android NDK (Native Development Kit)

- Toolchain for building .so libraries
- Allows native C/C++ code on Android
- Needed if you want C++ recognition

Android Integration Challenges

- ONNX Runtime C++
- Camera access
- Scoped storage
- JNI complexity

The Architecture

C++ Application (Qt)

↓ ???

AI inference

(Java/Android Layer)

Why Java ONNX Runtime?

Initial attempt: Pure C++ ONNX

3 Weeks Later...

Problem	Impact
🔥 **NDK Build Hell**	Days lost fighting cmake configurations
📱 **Device Compatibility**	Worked on Pixel, crashed on Samsung
📦 **APK Bloat**	+50MB for native libraries
⏰ **Time Crunch**	MVP deadline approaching fast

Why Java ONNX Runtime?

Initial attempt: Pure C++ ONNX

The Pragmatic Choice

	C++ ONNX	Java ONNX
Time to Working Solution	3+ weeks	30 minutes
Performance	200ms	230ms
Risk	High	Low
For MVP	❌	✅

ONNX Runtime – C++ vs Java

Feature	ONNX C++ (NDK)	ONNX Java (our setup)
Android Support	✅ But complex build	✅ Works out of the box
GPU Acceleration	⚠️ Not trivial	✅ With NNAPI delegate
Tensor Conversion	Manual (C++)	Managed (Java)
Integration Speed	Slow (days)	Fast (hours)

Qt’s JNI Wrapper or JNI

- QJniObject and QJniEnvironment
- Safer than raw JNIEnv*
- Easier type conversion

Tradeoff: Simplicity vs Purity

Option	Pros	Cons
C++ & ONNX Runtime C++	Single-language	NDK pain, memory bugs
C++ & JNI & ONNX Runtime Java	Stable API, easy access	JNI complexity
C++ & Qt JNI & ONNX Runtime Java	Reuse logic, clear layers	Some overhead

What Makes This Hard?

- 2MB+ images crossing JNI boundary
- Performance target: 800ms-1.3s
- Memory constraints on devices
- Thread safety complexities
- Error handling across languages

JNI Basics

```
// Calling Java from C++
QJniObject
```

JNI Basics

```
// Qt 6+ JNI approach:
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
    "activity",
    "()Landroid/app/Activity;");
```

JNI Basics

Full Java class name

(slashes instead of dots)

```
// Qt 6+ JNI approach:
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
    "activity",
    "()Landroid/app/Activity;");
```

JNI Basics

Cpp North

Method name

```
// Qt 6+ JNI approach:
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
    "activity",
    "()Landroid/app/Activity;");
```

JNI Basics

Method signature:

- `()` – no parameters
- `L…Activity;` – returns an object

```
// Qt 6+ JNI approach:
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
     "activity",
    "()Landroid/app/Activity;");
```

JNI Basics

```
// Qt 6+ JNI approach:
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
     "activity",
    "()Landroid/app/Activity;");
```

JNI Basics

```
// Qt 6+ JNI approach:
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
    "activity",
    "()Landroid/app/Activity;");
```

```
QJniObject context = activity.callObjectMethod(
    "getApplicationContext",
    "()Landroid/content/Context;");
```

JNI Basics

```
// Get current Activity (for UI-related calls)
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
    "activity",
    "()Landroid/app/Activity;");

// Application Context (for system services)
QJniObject context = activity.callObjectMethod(
    "getApplicationContext",
    "()Landroid/content/Context;");
```

JNI Basics

```
// Qt 6+ JNI approach:
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
    "activity",
    "()Landroid/app/Activity;");

QJniObject context = activity.callObjectMethod(
    "getApplicationContext",
    "()Landroid/content/Context;");
```

```
jint result = QJniObject::callStaticMethod<jint>(
    "ca/mineeye/InternetConnectionChecker",// Java class name
    "checkConnection",                     // Method name
    "(Landroid/content/Context;)I", // Method signature
    context.object<jobject>() // Passing context as jobject
);
```

JNI Basics

```
// Qt 6+ JNI approach:
QJniObject activity = QJniObject::callStaticObjectMethod(
    "org/qtproject/qt/android/QtNative",
    "activity",
    "()Landroid/app/Activity;");

QJniObject context = activity.callObjectMethod(
    "getApplicationContext",
    "()Landroid/content/Context;");


jint result = QJniObject::callStaticMethod<jint>(
    "ca/mineeye/InternetConnectionChecker",
    "checkConnection",
    "(Landroid/content/Context;)I",
    context.object<jobject>() // Passing context as jobject
);
```

Qt Solution: Template Unification

```
// Static methods
callStaticObjectMethod(…)
callStaticMethod<jint>(…)
callStaticMethod<jbool>(…)

// Instance methods
obj.callObjectMethod(…)
obj.callMethod<jint>(…)
obj.callMethod<jbool>(…)
```

Plain JNI – Without Any Wrapper ?

```cpp
// env is a valid JNIEnv*
// context is jobject (we'll get it below)

jobject getAndroidContext(JNIEnv* env) {
    // Step 1: Get QtNative class
    jclass qtNativeClass = env->FindClass("org/qtproject/qt/android/QtNative");
    if (!qtNativeClass) {
        __android_log_print(ANDROID_LOG_ERROR, "JNI", "Failed to find QtNative class");
        return nullptr;
    }

    // Step 2: Get static method ID for activity()
    jmethodID activityMethod = env->GetStaticMethodID(qtNativeClass, "activity", "()Landroid/app/Activity;");
    if (!activityMethod) {
        __android_log_print(ANDROID_LOG_ERROR, "JNI", "Failed to get method ID for activity()");
        return nullptr;
    }

    // Step 3: Call QtNative.activity() to get Activity instance
    jobject activity = env->CallStaticObjectMethod(qtNativeClass, activityMethod);
    if (!activity) {
        __android_log_print(ANDROID_LOG_ERROR, "JNI", "Failed to get Activity object");
        return nullptr;
    }

    // Step 4: Get Activity class to call getApplicationContext()
    jclass activityClass = env->GetObjectClass(activity);
    jmethodID getContextMethod = env->GetMethodID(activityClass, "getApplicationContext", "()Landroid/content/Context;");
    if (!getContextMethod) {
```

Never Write Signatures by Hand!

```
// Your Java method
public String processImage(byte[] data) { return ""; }
public float[] detect(String path, int threshold) { return null; }
```

javap Magic

```
# Run javap on your compiled class
javap -s TestJNI
```

```
// Your Java method
public String processImage(byte[] data);
    descriptor: ([B)Ljava/lang/String;    ← COPY THIS!
public float[] detect(String path, int threshold);
    descriptor: (Ljava/lang/String;I)[F   ← AND THIS!
```

Step 1 - QML Camera Capture

```
// QML - Start capture
onClicked: function() {
    captureSession.imageCapture.captureToFile("");
    // Empty string = let camera choose location
}
```

Step 1 - QML Camera Capture

```
// Signal when saved
onImageSaved: (id, path) => {
    console.log("Image saved to:", path)
    // Path: /…/cache/IMG_20231215_143022.jpg
    onQmlCapturedImageSavedSignal(path)
}
```

Step 2 - Process in C++

```
// C++ - Process and create permanent copy
QString DataController::saveCopyImage(const QString& path) {
    QImage image(QQmlFile::urlToLocalFileOrQrc(path));

    // Optional: resize for AI processing
    if (needsResize) {
        image = resizeImage(image, 640, 640);
    }

    // Save to Gallery (saveImageToGallery)
    // ...
}
```

Step 3 - Save to Gallery

```
// C++ - Prepare for JNI
QByteArray imageData;
QBuffer buffer(&imageData);
image.save(&buffer, "JPEG");

jbyteArray byteArray = env->NewByteArray(imageData.size());
env->SetByteArrayRegion(byteArray, 0, imageData.size(),
                        reinterpret_cast<const jbyte *>(imageData.constData()));

QJniObject javaReturnString = QJniObject::callStaticMethod<jstring>(
    "ca/mineeye/ImageSaver",
    "saveImageToGallery",
    "(Landroid/content/Context;[BLjava/lang/String;ZII)Ljava/lang/String;",
    context.object<jobject>(),
    byteArray, jTitle.object<jstring>(),
    jResized, jImageWidth, jImageHeight);

env->DeleteLocalRef(jData);
```

Step 4 - Java MediaStore

```
// Java - Save to gallery
ContentValues values = new ContentValues();
values.put(MediaStore.Images.Media.DISPLAY_NAME, title);
values.put(MediaStore.Images.Media.MIME_TYPE, "image/jpeg");
//...
Uri uri = context.getContentResolver()
    .insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values);

try (OutputStream out = context.getContentResolver()
                              .openOutputStream(uri)) {
    out.write(imageData);
}
return uri.toString();
```

Step 4 - Java MediaStore

/storage/emulated/0/Android/data/ca.mine/files/Pictures/image_0001.jpg

↓

file:///storage/emulated/0/Pictures/Mineeye/20250713_221525.jpg

Step 5 - Cleanup

```
// Back in QML - cleanup temp file
function onQmlCapturedImageSavedSignal(path) {
    // Process image
    var newImage = dataController.saveCopyOfImageFromQml(path);

    // Remove temporary file
    console.log("Removing temp file:", path);
    qmlRemoveImage(path);

    // Continue with the gallery copy
    qmlProcessFirstImage(newImage);
}
```

Step 5 - Cleanup

```
Q_INVOKABLE void scheduleRemoveImage(const QString &imgPath) {
    // Run the removeImage function in a background thread
    auto v = QtConcurrent::run([this, imagePath]() {
        this->removeImage(imagePath);
    });
}
```

Send Image to Java

```
// Back in QML - cleanup temp file
function onQmlCapturedImageSavedSignal(path) {
    // Process image
    var newImage = dataController.saveCopyOfImageFromQml(path)

    // Remove temporary file
    console.log("Removing temp file:", path);
    qmlRemoveImage(path);

    // Continue with the gallery copy
    qmlProcessFirstImage(newImage);
}
```

Java Side - Decode the Image

```
// From PNG bytes to Android Bitmap
Bitmap bitmap = BitmapFactory.decodeByteArray(
    imageData, 0, imageData.length
);

// Original: 1920x1080 (2MP)
// Model needs: 640x640
bitmap = Bitmap.createScaledBitmap(
    bitmap, 640, 640, true
);
```

The Pixel Format Conversion

```
// Android stores pixels as ARGB in one integer:
// [Alpha][Red][Green][Blue] = 0xAARRGGBB

int pixel = intValues[i];
float red   = ((pixel >> 16) & 0xFF) / 255.0f;
float green = ((pixel >> 8)  & 0xFF) / 255.0f;
float blue  = ((pixel)       & 0xFF) / 255.0f;

// But wait... there's more!
```

The Pixel Format Conversion

```
// Android bitmap: Height x Width x Channels (HWC)
// pixels[y][x] = [R, G, B]

// ONNX wants: Channels x Height x Width (CHW)
// tensor[channel][y][x]

// So we reorganize 640x640x3 = 1,228,800 values:
for (int i = 0; i < pixels.length; i++) {
    floatValues[i] = red[i];                  // Reds first
    floatValues[i + 640*640] = green[i];  // All greens
    floatValues[i + 2*640*640] = blue[i]; // All blues
}
```

The Pixel Format (Height/Width/Channels)

```
[
  [ [R,G,B], [R,G,B], [R,G,B] ],   ← 1st row (y = 0)
  [ [R,G,B], [R,G,B], [R,G,B] ],   ← 2nd row (y = 1)
  [ [R,G,B], [R,G,B], [R,G,B] ]    ← 3rd row (y = 2)
]
```

The Pixel Format (Channels/Height/Width)

Cpp North

```
Channel R
(All reds):
[
  [R, R, R],
  [R, R, R],
  [R, R, R]
]
```

```
Channel G
(All greens):
[
  [G, G, G],
  [G, G, G],
  [G, G, G]
]
```

```
Channel B
(All blues):
[
  [B, B, B],
  [B, B, B],
  [B, B, B]
]
```

Add Batch Dimension

```
// Model expects 4D tensor: [batch, channels, height, width]
// We have: [channels, height, width]
// Solution: Add batch dimension of 1

float[] batchedValues = new float[1 * 3 * 640 * 640];
System.arraycopy(floatValues, 0, batchedValues, 0, length);

// Create ONNX tensor
OnnxTensor tensor = OnnxTensor.createTensor(
    environment,
    FloatBuffer.wrap(batchedValues),
    new long[]{1, 3, 640, 640}
); // Ready for inference! Total preprocessing: ~80ms
```

Wait, but how many copies have we made?

The Full Journey of One Image

1. QML Camera Capture → …/myapp/files/Pictures/0001.jpg
2. C++ Crop image → same location
3. Java Save Cropped Image → /Pictures/Mineeye/20231215_143022.jpg
4. C++ Load → QImage from app storage
5. C++ → JNI → byte[] for inference
6. HWC → CHW transformation
7. Add batch dimension

Plus, the result copies

Copy #1: Java ONNX → detected objects

Copy #2: Results → JSON → C++ → QML

And what about inference?

```
OrtSession.Result output = session.run(
    Collections.singletonMap(
        session.getInputNames().iterator().next(),
        tensor)
);
```

Have to transpose (j,i swap)

```
// Original ONNX output: [1, N, M]
// Actually means:
// 1 = batch size
// N = attributes per detection (6-7 values)
// M = number of detections

// After transpose: [M, N]
// M = number of detections (rows)
// N = attributes per detection (columns)
```

Results Processing

```
// Shape: [1, N, M], transpose to [M,N].
float[][][] array3d = (float[][][]) onnxValue.getValue();
float[][] transformedOutput = new float[M][N];

for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        transformedOutput[i][j] = array3d[0][j][i];
    }
}
```

Extract Detections

```
for (int i = 0; i < transformedOutput.length; i++) {
    float[] outputRow = transformedOutput[i];
    MaxClassScore maxClassScore = getMaxClassScore(outputRow);

    if (maxClassScore.score >= threshold) {
        // Apply scaling to bounding box coordinates
        float x = outputRow[0] * xScale;
        float y = outputRow[1] * yScale;
        ...
```

The Architecture We Shipped

```
1  QString processNewPhoto(QString path) {
2      path = cropImage(path);
3
4      QString galleryPath = saveToGallery(path);
5
6      QImage image(galleryPath);
7      QByteArray bytes = imageToBytes(image);
8
9      auto detections = detectViaJNI(bytes);
10
11     QFile::remove(path);
12
13     return galleryPath;
14 }
```

Lessons Learned

✗ Don’t forget to delete arrays, temporary images

✗ Can't avoid MediaStore on Android 10+

✗ Multiple copies are inevitable

✓ Embrace the platform constraints

✓ Ship first, optimize later

✓ Document why it's complex

"Perfect is the enemy of good" - Voltaire

Let's recap the problems we faced

What is a local reference?

C++ code	JNI Layer	Java Heap
array (invalid)	Local Ref Table [ref -> object]	byte[] object [actual data]

↓

```
env->DeleteLocalRef(array);
```

↓

C++ code	JNI Layer	Java Heap
array (invalid)	Local Ref Table [empty]	byte[] object [still exists!]

Why the byteArray still exists?

```
QJniObject::callStaticMethod<jstring>(
    "ca/mysite/ImageSaver",
    "saveImageToGallery",
    "...[B...",  // [B means byte[]
    byteArray    // Pass an array to Java
);
```

```
env->DeleteLocalRef(byteArray);
```

```
public class ImageSaver {
    private static byte[] lastImage;

    public static String saveImageToGallery(Context ctx, byte[] data, ...) {
        // Java code received the array and stores a reference
        lastImage = data; // Now Java holds a reference!

        new Thread(() -> {      // Asynchronous save operation
            saveToFile(data);   // Uses the array later
        }).start();

        return "OK";
    }
}
```

PushLocalFrame/PopLocalFrame

```
// Main frame: 512 slots available
for (int i = 0; i < 1000; i++) {
    // Create a new local frame with 20 slots
    if (env->PushLocalFrame(20) < 0) { return; } // Out of memory

    // Create multiple references within this frame
    jbyteArray imageData = env->NewByteArray(1024 * 1024);
    jstring fileName = env->NewStringUTF("image.jpg");

    // Process the image
    jobject result = env->CallStaticObjectMethod( ...
        imageData,
        fileName
    );

    // All local references created in this frame are automatically freed!
    env->PopLocalFrame(NULL); // No need for individual DeleteLocalRef calls
}
```

PushLocalFrame/PopLocalFrame

```
// Main frame: 512 slots available
for (int i = 0; i < 1000; i++) {
    // Create a new local frame with 20 slots
    if (env->PushLocalFrame(20) < 0) { return; } // Out of memory

     // Create multiple references within this frame
     jbyteArray imageData = env->NewByteArray(1024 * 1024);
     jstring fileName = env->NewStringUTF("image.jpg");

    // Process the image
    jobject result = env->CallStaticObjectMethod( ...
        imageData,
        fileName
    );

    // All local references created in this frame are automatically freed!
    env->PopLocalFrame(NULL); // No need for individual DeleteLocalRef calls
}
```

Memory Management

```
for (int i = 0; i < 1000; i++) {
    jbyteArray array = env->NewByteArray(size);
    // Process array...
}
```

Memory Management

```
for (int i = 0; i < 1000; i++) {
    jbyteArray array = env->NewByteArray(size);
    // Process array...
}
```

Memory Management

```
for (int i = 0; i < 1000; i++) {
    jbyteArray array = env->NewByteArray(size);
    // Process array...
}
```

Reference Table (512 slots)
Loop iteration 1: [ref1]
Loop iteration 2: [ref1][ref2]
...

Memory Management

```
for (int i = 0; i < 1000; i++) {
    jbyteArray array = env->NewByteArray(size);
    // Process array...
}
```

Reference Table (512 slots)

```
Loop iteration 1: [ref1]
Loop iteration 2: [ref1][ref2]
...
Loop iteration 512: [ref1]...[ref512]
Loop iteration 513: CRASH!
```

Memory Management

```
// Problem – JNI local reference table overflow
for (int i = 0; i < 1000; i++) {
    jbyteArray array = env->NewByteArray(size);
    // Process array...
    // Local ref table fills up → CRASH!
}
```

Memory Management

```
// Solution – Delete JNI local reference in loops
for (int i = 0; i < 1000; i++) {
    jbyteArray array = env->NewByteArray(size);
    // Process array...
    env->DeleteLocalRef(array); // Free table slot
}
```

Memory Management

```
jbyteArray array = env->NewByteArray(imageData.size());
// ... use array
```

Memory Management

```
// Memory leak
jbyteArray array = env->NewByteArray(imageData.size());
// ... use array
// ... Missing: DeleteLocalRef
```

Memory Management

```
// Memory leak fixed
jbyteArray array = env->NewByteArray(imageData.size());
// ... use array
env->DeleteLocalRef(array);     // Critical!
```

Similar issue:
stackoverflow.com/questions/8574241/jni-newbytearray-memory-leak

Other tips:
developer.android.com/training/articles/perf-jni

Memory Management

```
// RAII wrapper for jbyteArray with
DeleteLocalRef
class JniByteArrayWrapper {
private:
    jbyteArray array;
    QJniEnvironment env;

public:
    explicit JniByteArrayWrapper(jsize size) {
        array = env->NewByteArray(size);
    }

    // Dtor calls DeleteLocalRef automatically
    ~JniByteArrayWrapper() {
        if (array != nullptr) {
            env->DeleteLocalRef(array);
        }
    }
//...
};
```

Memory Management

```
{
    // Removed after going out of the scope
    JniByteArrayWrapper byteArray(imageData.size());

    // Copying the data
    QJniEnvironment env;
    env->SetByteArrayRegion(byteArray, 0, imageData.size(),
        reinterpret_cast<const jbyte*>(imageData.constData()));

    // Call Java method
    QJniObject result = QJniObject::callStaticMethod<jstring>(
        "com/myapp/ImageProcessor", "processImage",
        "([B)Ljava/lang/String;",
        byteArray.get()
    );
}
```

JNI Memory Rules

- Always delete local refs in loops
- Use Push/PopLocalFrame operations
- Use RAII wrapper
- Don't assume GC will save you

JNI and threads – core problem

```
// main thread
{
    JniByteArrayWrapper arr(data.size());
    // ...`
    // QtConcurrent thread - CRASH!
    QJniObject::callStaticMethod<void>(
        "org/example/Helper", "save", "([B)V",
        arr.get()
    );
}
```

Solution – run in lambda

```
QtConcurrent::run([=]() {
    JniByteArrayWrapper arr(data.size());

    QJniObject::callStaticMethod<void>(
        "org/example/Helper", "save", "([B)V",
        arr.get()
    );
    // arr automatically cleaned up when lambda ends
});
```

Cache across threads

```
// Qt Core pattern - global static cache
typedef QHash<QString, jclass> JClassHash;
Q_GLOBAL_STATIC(JClassHash, cachedClasses)
Q_GLOBAL_STATIC(QReadWriteLock, cachedClassesLock)

// Inside callStaticMethod - check cache first
jclass clazz = nullptr;
{
    QReadLocker locker(cachedClassesLock);
    clazz = cachedClasses->value(className, nullptr);
}

if (!clazz) {
    clazz = env.FindClass(className);
    clazz = static_cast<jclass>(env->NewGlobalRef(clazz));

    QWriteLocker locker(cachedClassesLock);
    cachedClasses->insert(className, clazz);
}
```

Cache across threads

```cpp
// Qt Core pattern - global static cache
typedef QHash<QString, jclass> JClassHash;
Q_GLOBAL_STATIC(JClassHash, cachedClasses)
Q_GLOBAL_STATIC(QReadWriteLock, cachedClassesLock)

// Inside callStaticMethod - check cache first
jclass clazz = nullptr;
{
    QReadLocker locker(cachedClassesLock);
    clazz = cachedClasses->value(className, nullptr);
}

if (!clazz) {
    clazz = env.FindClass(className);
    clazz = static_cast<jclass>(env->NewGlobalRef(clazz));

    QWriteLocker locker(cachedClassesLock);
    cachedClasses->insert(className, clazz);
}
```

Cache across threads

```
// Qt Core pattern - global static cache
typedef QHash<QString, jclass> JClassHash;
Q_GLOBAL_STATIC(JClassHash, cachedClasses)
Q_GLOBAL_STATIC(QReadWriteLock, cachedClassesLock)

// Inside callStaticMethod - check cache first
jclass clazz = nullptr;
{
    QReadLocker locker(cachedClassesLock);
    clazz = cachedClasses->value(className, nullptr);
}

if (!clazz) {
    clazz = env->FindClass(className);
    clazz = static_cast<jclass>(env->NewGlobalRef(clazz));

    QWriteLocker locker(cachedClassesLock);
    cachedClasses->insert(className, clazz);
}
```

A local ref is created here

A global ref is created here

Global references for sharing across threads

```
// Convert local reference to global
QJniEnvironment env;
QJniObject localCtx = env->NewByteArray(imageData.size());
jobject globalCtx = env->NewGlobalRef(localCtx.object<jobject>());

// Now safe to use in any thread
// When done (e.g., in destructor):
env->DeleteGlobalRef(globalCtx);
```

https://stackoverflow.com/questions/30721635/must-i-deletelocalref-an-object-i-have-called-newglobalref-on

Check what has been returned

```
// Qt JNI returns default values on exception!
auto result = QJniObject::callStaticMethod<jint>(
    "com/example/Class", "method", "()I"
);
// result = 0 on exception, no crash!

// Always check:
if (QJniEnvironment::checkAndClearExceptions()) {
    qWarning() << "JNI exception occurred!";
    return; // handle error
}
```

Check what has been returned

```
QJniEnvironment env;
jbyteArray arr = env->NewByteArray(data.size());
if (!arr) { // check if nullptr
    qWarning() << "Failed to allocate byte array";
    return;
}
```

Error Handling Strategy

```
// Strategic checking - not after every call
QJniEnvironment env;
jbyteArray data = env->NewByteArray(size);
// Skip check - NewByteArray rarely fails

// Group related operations
auto result = QJniObject::callStaticMethod<jobject>(
    "com/myapp/Processor", "analyze", "([B)Ljava/lang/String;", data
);

// Check once after the risky operation
if (!result.isValid() || env.checkAndClearExceptions()) {
    qWarning() << "Processing failed";
    return;
}

env->DeleteLocalRef(data);
// No check needed - cleanup rarely fails
```

Threading Performance Problem

```
// Naive approach - new thread for each task
for (auto& imageData : images) {
    std::thread([=]() {
        QJniEnvironment env; // AttachCurrentThread happens here!

        jbyteArray arr = env->NewByteArray(imageData.size());
        QJniObject::callStaticMethod<void>(
            "...", "process", "([B)V", arr);
        env->DeleteLocalRef(arr);

        // DetachCurrentThread happens in destructor
    }).detach();
}
// Problem: 100 images = 100 attach/detach cycles!
```

Use ThreadPool for tasks

```
class JniTask : public QRunnable {
    QByteArray data;
public:
    void run() override {
        // Thread attaches on first task only!
        QJniEnvironment env;

        jbyteArray arr = env->NewByteArray(imageData.size());
        QJniObject::callStaticMethod<void>(
            "...", "process", "([B)V", arr);
        env->DeleteLocalRef(arr);
    }
};
// Submit tasks - threads are reused
for (auto& data : dataList) {
    QThreadPool::globalInstance()->start(new JniTask(data));
}
```

Delegate threading to Java

```cpp
// C++ side - single JNI call
QJniObject::callStaticMethod<void>(
    "org/example/BatchProcessor", "runAsync", "([Ljava/lang/String;)V", paths
);


// Java side - handles threading
public static void runAsync(String[] paths) {
    ExecutorService executor = Executors.newFixedThreadPool(4);

    for (String path : paths) {
        executor.submit(() -> {
            processFile(path);
            // Careful with callbacks to C++ here!
        });
    }
}
```

Camera Without Permissions

```
void takePicture() {
    // No permission check - will crash!
    QJniObject::callStaticMethod<void>(
        "org/example/Camera", "capture", "()V"
    );
}

// Android logcat:
// java.lang.SecurityException: Permission Denial:
// camera requires android.permission.CAMERA
```

Android Permissions

```
void takePicture() {
    auto permission = QtAndroidPrivate::checkPermission(
        "android.permission.CAMERA").result();

    if (permission == QtAndroidPrivate::Denied) {
        permission = QtAndroidPrivate::requestPermission(
            "android.permission.CAMERA").result();
    }

    if (permission == QtAndroidPrivate::Authorized) {
        // NOW it's safe to call camera
        QJniObject::callStaticMethod<void>(
            "org/example/Camera", "capture", "()V"
        );
    }
}
```

Android Permissions

```
bool PermissionsManager::checkAndRequestCameraPermission()
{
    bool granted = true;
    auto cameraPermission = QtAndroidPrivate::checkPermission(\
        QString("android.permission.CAMERA")).result();
    if (cameraPermission == QtAndroidPrivate::Denied)
    {
        cameraPermission = QtAndroidPrivate::requestPermission(
            QString("android.permission.CAMERA")).result();
        if (cameraPermission == QtAndroidPrivate::Denied) {
            granted = false;
        }
    }
    return granted;
}
```

Where Can We Write Files?

```
// Where to save captured images?
QString savePath = ???;  // What path works on Android?

// These all fail:
savePath = "/sdcard/DCIM/";           // ❌ Permission denied
savePath = "/storage/Pictures/";       // ❌ Doesn't exist
savePath = QDir::homePath() + "/pics"; // ❌ Not accessible

// We need the RIGHT path for THIS device!
```

Android Scoped Storage

```
// C++ side - ask Java for the correct path
QString savePath = QJniObject::callStaticObjectMethod(
    "org/example/StorageHelper",
    "getPicturesDirectoryPath",
    "(Landroid/content/Context;)Ljava/lang/String;",
    context
).toString();
// Returns: /storage/emulated/0/Android/data/com.myapp/files/Pictures/

// Java helper:
public static String getPicturesDirectoryPath(Context context) {
    if (Environment.MEDIA_MOUNTED.equals(
            Environment.getExternalStorageState())) {
        return context.getExternalFilesDir(
            Environment.DIRECTORY_PICTURES).getAbsolutePath();
    } else {
        return context.getFilesDir().getAbsolutePath();
    }
}
```

Performance Optimization

Initial: 4 seconds per image

→ Bitmap scaling: -400ms

→ QtConcurrent usage -1500ms

Final: 2.1s per image

Performance Optimization

- ONNX tensor reuse

- Batch processing

Beyond Landmines

The same architecture works for:

- Quality control in manufacturing
- Medical image analysis
- Infrastructure inspection
- Agricultural monitoring

Key Takeaways

1. Qt JNI bridges C++ and Android effectively
2. Data marshalling is critical for performance
3. QtConcurrent simplifies threading
4. Real-world AI needs hybrid architectures

Code and Resources

Sample Code and These Slides:

github.com/kunichik/conferences/cppnorth2025

Documentation:

doc.qt.io/qt-6/qjniobject.html

Deep dive: Qt's JNI internals

JNI Method Signatures

Rules:
"(parameters)returnType"

Examples:

```
()V                    → void method()
(I)Z                   → boolean method(int)
(Ljava/lang/String;)V  → void method(String)
([B)Ljava/lang/String;    → String method(byte[])
```

Common types:

Z=boolean, B=byte, I=int, J=long, F=float, D=double

[=array, L...;=object

Data Type Mapping

C++ → JNI → Java

---

int	→ jint	→ int
bool	→ jboolean	→ boolean
std::string	→ jstring	→ String
std::vector<uint8_t>	→ jbyteArray	→ byte[]

Qt specifics:

QString → jstring (via toUtf8())

QByteArray → jbyteArray (via data())

QImage → jbyteArray (via JPG encoding)

Raw JNI: Function Explosion

```
// Static methods
CallStaticObjectMethod(...)    // for Java objects
CallStaticIntMethod(...)       // for jint primitives
CallStaticBooleanMethod(...)   // for jbool primitives

// Instance methods
CallObjectMethod(...)          // for Java objects
CallIntMethod(...)             // for jint primitives
CallBooleanMethod(...)         // for jbool primitives
```

Under the Hood: Compile-Time Macros

```
#define MAKE_CALLER(Type, Method) \
template <typename T> \
struct Caller<T, std::enable_if_t<sameTypeForJni<T, Type>>> \
{ \
    static constexpr void callMethodForType(JNIEnv *env, T &res, \
        jobject obj, jmethodID id, va_list args) \
    { \
        res = T(env->Call##Method##MethodV(obj, id, args)); \
    } \
//...
}
```

Type-to-Signature Mapping

```
template <typename T>
struct Traits {
    static constexpr auto signature() {
        if constexpr (std::is_same_v<T, int>) {
            return CTString("I");
        } else if constexpr (std::is_same_v<T, jbooleanArray>) {
            return CTString("[Z");
        } else if constexpr (std::is_same_v<T, jstring>) {
            return CTString("Ljava/lang/String;");
        }
        // ... more type mappings
    }
};
```

Type Detection: Primitive Magic

```cpp
template<typename T>
static constexpr bool isPrimitiveType()
{
    return Traits<T>::signature().size() == 2;
}

template<typename T>
static constexpr bool isArrayType()
{
    constexpr auto signature = Traits<T>::signature();
    return signature.startsWith('[') && signature.size() > 2;
}

template<typename T>
static constexpr bool isObjectType()
{
    if constexpr (std::is_convertible_v<T, jobject>) {
        return true;
    } else {
        constexpr auto signature = Traits<T>::signature();
        return (signature.startsWith('L') && signature.endsWith(';')) || isArrayType<T>();
    }
}
```

```cpp
template <typename T>
struct Traits {
    static constexpr auto signature() {
        if constexpr (std::is_same_v<T, int>) {
            return CTString("I");
        } else if constexpr (std::is_same_v<T, jbooleanArray>) {
            return CTString("[Z");
        } else if constexpr (std::is_same_v<T, jstring>) {
            return CTString("Ljava/lang/String;");
        }
        // ... more type mappings
    }
};
```

Type Detection: Array Patterns

```
template<typename T>
static constexpr bool isPrimitiveType()
{
    return Traits<T>::signature().size() == 2;
}
```

```
template<typename T>
static constexpr bool isArrayType()
{
    constexpr auto signature = Traits<T>::signature();
    return signature.startsWith('[') && signature.size() > 2;
}
```

```
template<typename T>
static constexpr bool isObjectType()
{
    if constexpr (std::is_convertible_v<T, jobject>) {
        return true;
    } else {
        constexpr auto signature = Traits<T>::signature();
        return (signature.startsWith('L') && signature.endsWith(';')) || isArrayType<T>();
    }
}
```

```
template <typename T>
struct Traits {
    static constexpr auto signature() {
        if constexpr (std::is_same_v<T, int>) {
            return CTString("I");
        } else if constexpr (std::is_same_v<T, jbooleanArray>) {
            return CTString("[Z");
        } else if constexpr (std::is_same_v<T, jstring>) {
            return CTString("Ljava/lang/String;");
        }
        // ... more type mappings
    }
};
```

Type Detection: Object Signatures

```cpp
template<typename T>
static constexpr bool isPrimitiveType()
{
    return Traits<T>::signature().size() == 2;
}

template<typename T>
static constexpr bool isArrayType()
{
    constexpr auto signature = Traits<T>::signature();
     return signature.startsWith('[') && signature.size() > 2;
}

template<typename T>
static constexpr bool isObjectType()
{
    if constexpr (std::is_convertible_v<T, jobject>) {
        return true;
    } else {
        constexpr auto signature = Traits<T>::signature();
        return (signature.startsWith('L') && signature.endsWith(';')) || isArrayType<T>();
    }
}
```

```cpp
template <typename T>
struct Traits {
    static constexpr auto signature() {
        if constexpr (std::is_same_v<T, int>) {
            return CTString("I");
        } else if constexpr (std::is_same_v<T, jbooleanArray>) {
            return CTString("[Z");
        } else if constexpr (std::is_same_v<T, jstring>) {
            return CTString("Ljava/lang/String;");
        }
        // ... more type mappings
    }
};
```

Special thanks for my team at PokerStars for recensions:

Maria Arkelatyan
Constantin Fishkin
Mikhail Grosman
Ury Glozshtein
Timur Sultanov
Aram Hambardzumyan
Michael Podolsky
Viktor Alkhimov
Ivan Kravets
Boris Tkatch

Let's Connect!

Oleksandr Kunichik

✉ ao669517126@gmail.com

🔗 linkedin.com/in/kunichik

Happy to discuss:

- Your JNI challenges
- Mobile AI applications
- Performance optimizations
- Similar detection problems