Extending your Qt Android application using JNI

Dev Days, 2014

Presented by BogDan Vatra

Material based on Qt 5.3, created on November 13, 2014

Extending your application using JNI

JNI Crash Course

What is JNI and why do you need it ?

- JNI is the Java Native Interface. It is needed to do calls to/from Java world from/to native (C/C++) world.
- It is impossible for Qt to implement all Android features, so to use them you'll need JNI to access them.

Missing Qt APIs

- listening for Android O.S. notifications:
  - sd card notifications: bad removal, eject, mounted, unmounted, etc.
  - network notifications: network up/down.
  - battery level and charging state.
  - etc.
- accessing Android O.S. features:
  - telephony (Initiate calls, MMS, SMS, etc.)
  - contacts
  - speech (TTS and Speech Recognizer)
  - system accounts
  - system preferences
  - NFC
  - USB
  - printing (WARNING: needs API-19+)
- create own Android Activities and Services

Use case 1: call a Java function from C/C++

Create the Java function

```java
// java file android/src/com/kdab/training/MyJavaClass.java
package com.kdab.training;

public class MyJavaClass
{
    // this method will be called from C/C++
    public static int fibonacci(int n)
    {
        if (n < 2)
            return n;
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

Demo android/JNIIntro

Call a Java method from C/C++, the Qt way

Let's see what **fibonacci** function call looks like using the Qt androidextras module.

```
# Changes to your .pro file
# ....
QT += androidextras
# ....
```

```
// C++ code
#include <QAndroidJniObject>
int fibonacci(int n)
{
    return QAndroidJniObject::callStaticMethod<jint>
                        ("com/kdab/training/MyJavaClass" // java class name
                        , "fibonacci" // method name
                        , "(I)I" // signature
                        , n);
}
```

Yes, that's all folks !

Use case 2: callback a C/C++ function from Java

Your job

- declare a native method in Java using **`native`** keyword (see slide 11)
- register native method in C/C++ (see slide 12, slide 15)
- do the actual call (see slide 11)

Declare and invoke the Java native methods

```java
// java file android/src/com/kdab/training/MyJavaClass.java
package com.kdab.training;

class MyJavaNatives
{
    // declare the native method
    public static native void sendFibonaciResult(int n);
}

public class MyJavaClass
{
    // this method will be called from C/C++
    public static int fibonacci(int n)
    {
        if (n < 2)
            return n;
        return fibonacci(n-1) + fibonacci(n-2);
    }

    public static void compute_fibonacci(int n)
    {
        // callback the native method with the computed result.
        MyJavaNatives.sendFibonaciResult(fibonacci(n));
    }
}
```

C/C++ register native methods

Registering functions using Java_Fully_Qualified_Class_Name_MethodName template.

```
// C++ code
#include <jni.h>

#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT void JNICALL
    Java_com_kdab_training_MyJavaNatives_sendFibonaciResult(JNIEnv */*env*/,
                                                            jobject /*obj*/,
                                                            jint n)
{
    qDebug() << "Computed fibonacci is:" << n;
}

#ifdef __cplusplus
}
#endif
```

Using Java_Fully_Qualified_Class_Name_MethodName.

Pro:

- **easier to declare**, you don't need to specify the function signature
- **easier to register**, you don't need to explicitly register it

Con:

- the function names are **huge**: `Java_com_kdab_training_MyJavaNatives_sendFibonaciResult`
- the library will export lots of functions
- **unsafer**, there is no way for the VM to check the function signature

Use JNIEnv::RegisterNatives to register native functions.

Step 4 call JNIEnv::RegisterNatives(java_class_ID, methods_vector, n_methods)

Step 3 find the ID of java class that declares these methods using JniEnv::FindClass

Step 2 create a vector with all C/C++ methods that you want to register

Step 1 get JNIEnv pointer by defining
JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* /*reserved*/).
You can define it (once per .so file) in any .cpp file you like

DEVELOPER DAYS2014

Use JNIEnv::RegisterNatives

```
// C++ code
#include <jni.h>

// define our native method
static void sendFibonaciResult(JNIEnv */*env*/, jobject /*obj*/, jint n)
{
    qDebug() << "Computed fibonacci is:" << n;
}

// step 2
// create a vector with all our JNINativeMethod(s)
static JNINativeMethod methods[] = {
     { "sendFibonaciResult", // const char* function name;
       "(I)V", // const char* function signature
       (void *)sendFibonaciResult // function pointer }
};
```

Use JNIEnv::RegisterNatives

```
// step 1
// this method is called automatically by Java after the .so file is loaded
JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* /*reserved*/)
{
    JNIEnv* env;
    // get the JNIEnv pointer.
    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) != JNI_OK)
        return JNI_ERR;

    // step 3
    // search for Java class which declares the native methods
    jclass javaClass = env->FindClass("com/kdab/training/MyJavaNatives");
    if (!javaClass)
        return JNI_ERR;

    // step 4
    // register our native methods
    if (env->RegisterNatives(javaClass, methods,
                            sizeof(methods) / sizeof(methods[0])) < 0) {
        return JNI_ERR;
    }
    return JNI_VERSION_1_6;
}
```

Use JNIEnv::RegisterNatives

Pro:

- the native function can have any name you want
- the library needs to export only one function (JNI_OnLoad).
- safer, because the VM checks the declared signature

Con:

- is slightly harder to use

SD Card Notifications Example

Game plan

- Plumbing
  - understanding Android files
  - how to use an external IDE to manage the Java part
    - import existing Android files
    - debug Java
- Architecture
  - how to extend the Java part of your Qt Application
  - how to do *safe* calls from Qt thread to Android UI thread
  - how to do *safe* calls from Android UI thread to Qt thread

Understanding Android files

Copy missing Android files

- copy all files from <qt_install_path>/src/android/java/ to <your_src>/android/ folder
- make sure your .pro file has ANDROID_PACKAGE_SOURCE_DIR property set

```
ANDROID_PACKAGE_SOURCE_DIR = $$PWD/android
```

Meaning of the Android files

- AndroidManifest.xml - is the same file copied by Qt Creator when you click on Create AndroidManifest.xml button.
- version.xml - is used by Qt Creator to update your existing Android files
- res/values/libs.xml - this file contains the information about the needed libs
- res/values(-*)/strings.xml - these files contain the (translated) strings needed by Java part
- src/org/kde/necessitas/ministro/ *.aidl - these files are used to connect to Ministro service
- src/org/qtproject/qt5/android/bindings/QtApplication.java - this class contains all the logic needed to forward all the activity calls to the activity delegate
- src/org/qtproject/qt5/android/bindings/QtActivity.java - this is the application activity class, it contains the logic to connect to Ministro or to unpack the bundled Qt libs. It also forwards all calls to the activity delegate.

Using an external IDE for Java part

Using an external IDE to extend the Java part

Sadly, the Java support in Qt Creator is close to zero, so we need to use an external IDE to easily extend the Java part. Android provides two powerful IDEs:

- Eclipse
- Android Studio (this is very good and stable even though it is labeled as beta).

Eclipse users

If you are not using ADT bundle, then make sure you have all ADT plugins installed in Eclipse.

#NOTES

Using an external IDE

The external IDE will be used to:

- import existing Android files
- create and edit Java files
- debug Java part

The external IDE will NOT be used to run the application, we still need QtCreator for that job!

This presentation will teach you only the very basic usage of both of them. If you want to learn more please check their own documentation.
Also it's up to you which one you choose to use.

#NOTES

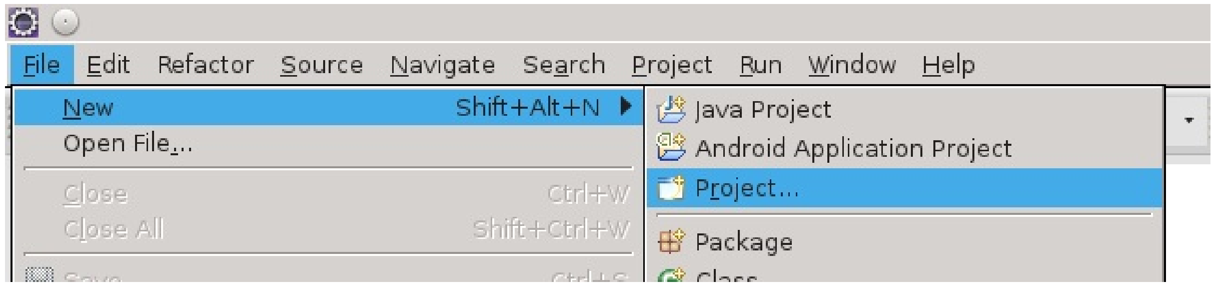
Import existing Android files in Eclipse

WARNING

Before you start to import the project make sure the Build Automatically feature is turned OFF!
Otherwise when you start to build your project you'll get lots of mysterious build problems.
Uncheck Project->Build Automatically

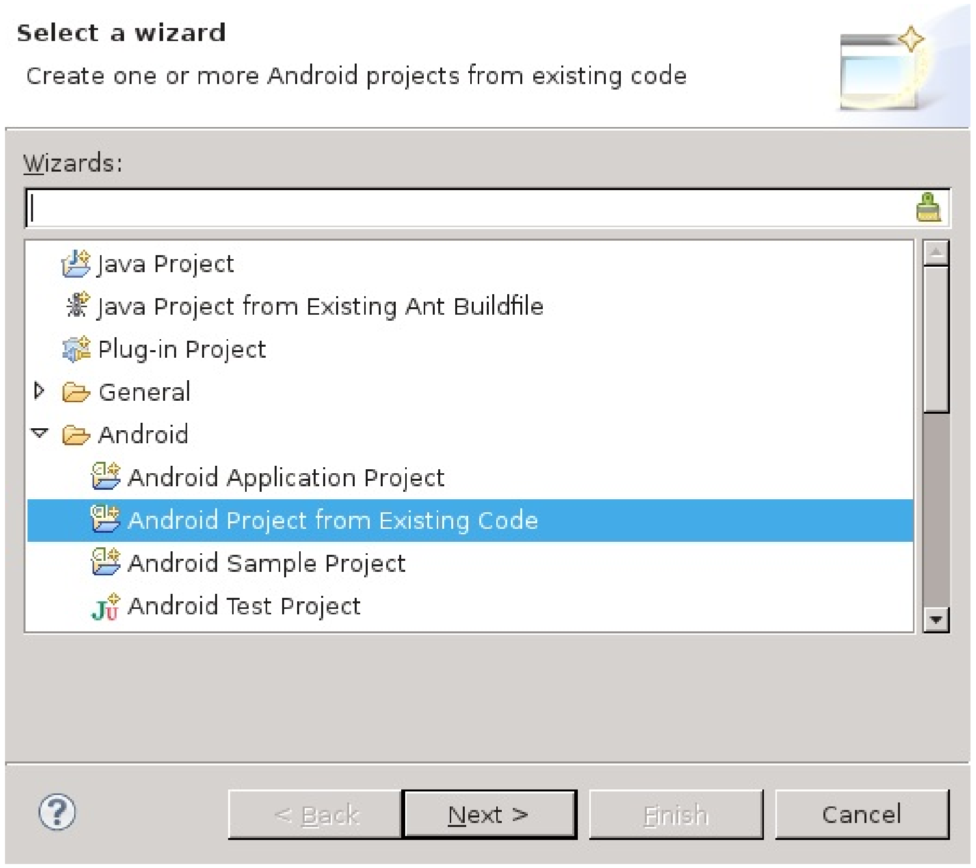
Importing in Eclipse

File->New->Project ...



Importing in Eclipse

Choose **Android Project from Existing Code** wizard



Select a wizard

Create one or more Android projects from existing code

Wizards:

- Java Project
- Java Project from Existing Ant Buildfile
- Plug-in Project
- General
- Android
  - Android Application Project
  - Android Project from Existing Code
  - Android Sample Project
  - Android Test Project

Importing in Eclipse

Select the root folder of your Android files (it should be your_qt_src/android) and press Finish button.

Import existing Android files in Android Studio

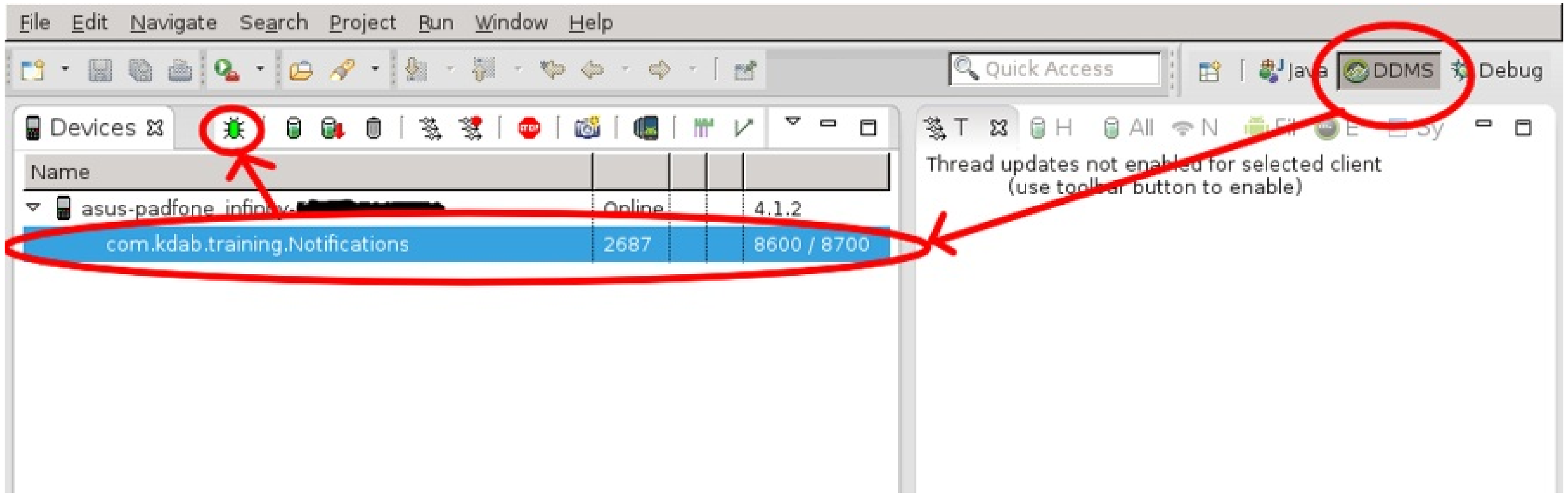
Importing in Android Studio

File->Import project … does all the magic, you just need to select the root folder of your Android files (it should be your_qt_src/android) and continue the wizard until it is finished.

Debugging Java part

Debug with Eclipse

- set the break points
- switch to DDMS perspective (**Window->Open perspective->DDMS**)
- start the application (using QtCreator)
- wait for application to start
- select the application
- click the debug icon



asus-padfone_infinity-[redacted] Online 4.1.2
com.kdab.training.Notifications 2687 8600 / 8700

Thread updates not enabled for selected client
(use toolbar button to enable)

Debug with Android Studio

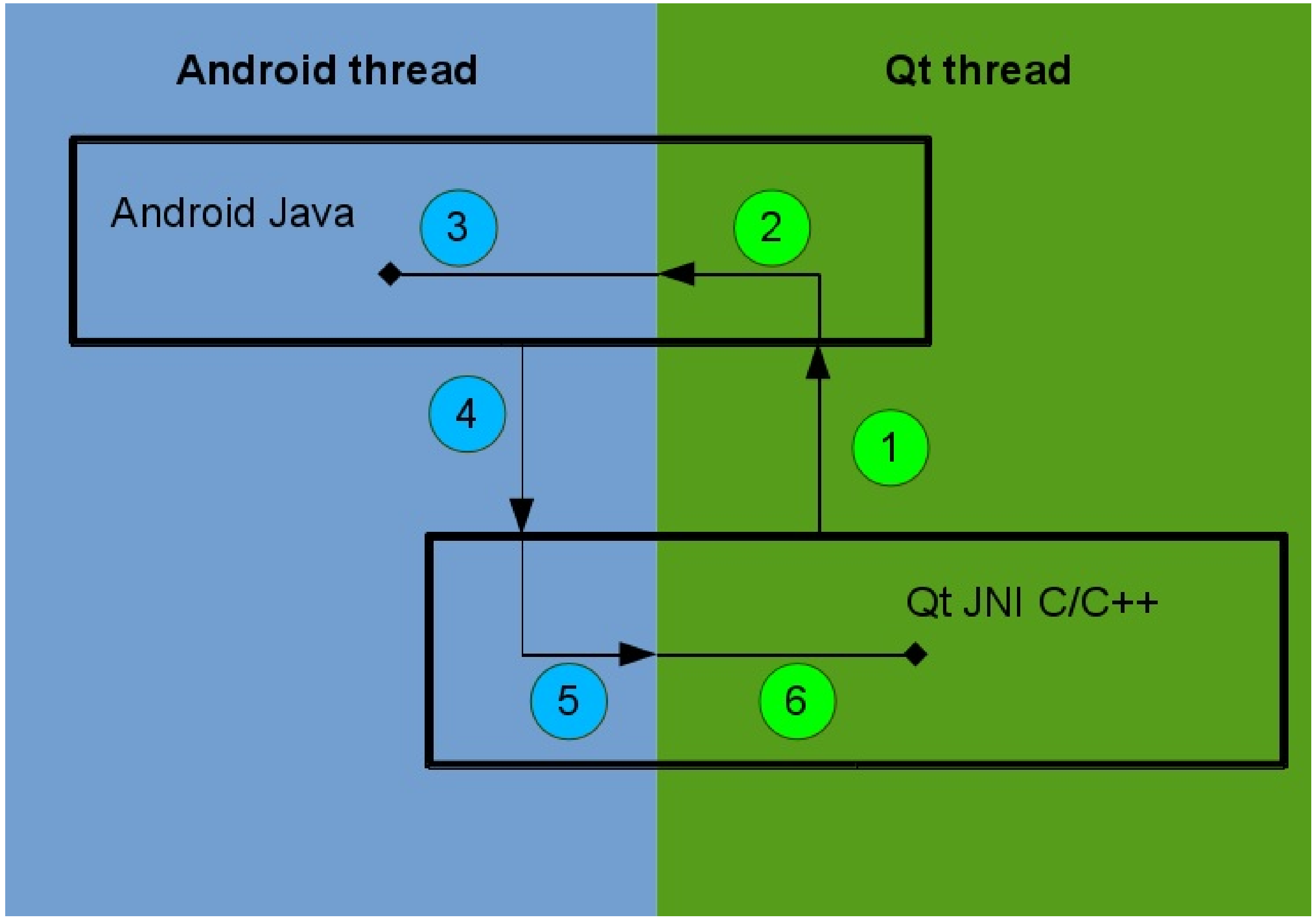
- set the break points
- attach debugger (Run->Attach debugger to Android process)
- start the application (using QtCreator)
- wait for application to start
- select the application
- click the OK button

Debugging tips and tricks

The problem comes when we need to start the debugging very early. To do that we are going to use the old fashioned sleep trick. This is how our custom *onCreate* function looks:

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    ....
}
```

Architecture diagram



Extending Java part

Extending, not changing

- You should not change any of the existing .java files, instead you should extend them
- There is no need to extend the QtApplication application class
- You should extend only the QtActivity class. The Activity object is very important, you'll need it to instantiate most of the Android classes

Demo android/SD_Card_Notifications

Extending QtActivity

```java
// src/com/kdab/training/MyActivity.java
package com.kdab.training;

import android.os.Bundle;
import org.qtproject.qt5.android.bindings.QtActivity;

public class MyActivity extends QtActivity
{
    // we'll need it in BroadcastReceiver
    public static MyActivity s_activity = null;

    // every time you override a method, always make sure you
    // then call super method as well
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        s_activity = this;
        super.onCreate(savedInstanceState);
    }

    @Override
    protected void onDestroy()
    {
        super.onDestroy();
        s_activity = null;
    }
}
```

Extending Java part

Change default activity to AndroidManifest.xml, from:

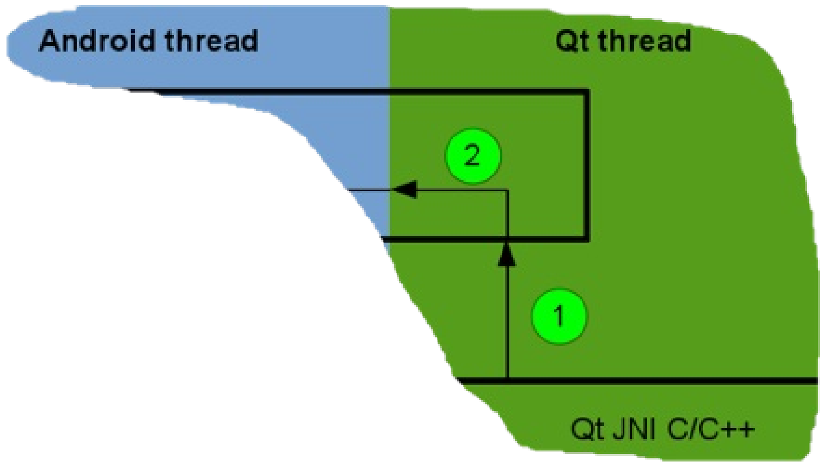
```
<activity ...
        android:name="org.qtproject.qt5.android.bindings.QtActivity"
        ... >;
```

To:

```
<activity ...
        android:name="com.kdab.training.MyActivity"
        ... >;
```

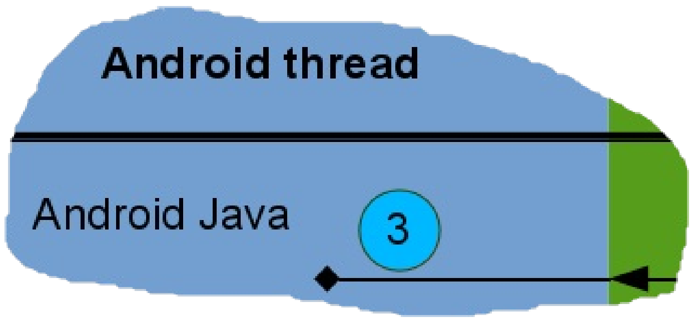
NativeFunctions.java part 1

```java
public class NativeFunctions {
    // declare the native functions
    // these functions will be called by the BroadcastReceiver object
    // when it receives a new notification
    public static native void onReceiveNativeMounted();
    public static native void onReceiveNativeUnmounted();

    // this static method is called by C/C++ to register the BroadcastReceiver. (step 1)
    public static void registerBroadcastReceiver() {
        if (MyActivity.s_activity != null) {
            // Qt is running on a different thread than Android. (step 2)
            // In order to register the receiver we need to execute it in the UI thread
            MyActivity.s_activity.runOnUiThread( new RegisterReceiverRunnable());
        }
    }
}
```



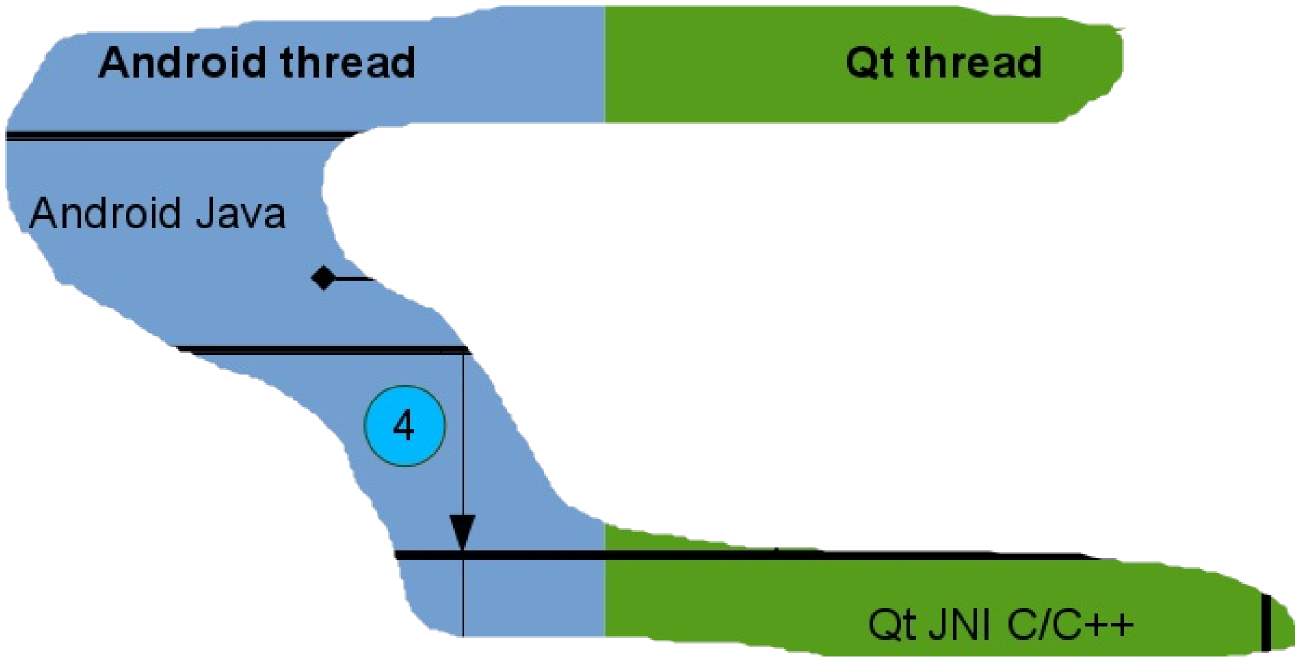
NativeFunctions.java part 2

```
class RegisterReceiverRunnable implements Runnable
{
    // Step 3
    // this method is called on Android Ui Thread
    @Override
    public void run() {
        IntentFilter filter = new IntentFilter();
        filter.addAction(Intent.ACTION_MEDIA_MOUNTED);
        filter.addAction(Intent.ACTION_MEDIA_UNMOUNTED);
        filter.addDataScheme("file");
        // this method must be called on Android Ui Thread
        MyActivity.s_activity.registerReceiver(new SDCardReceiver(), filter);
    }
}
```

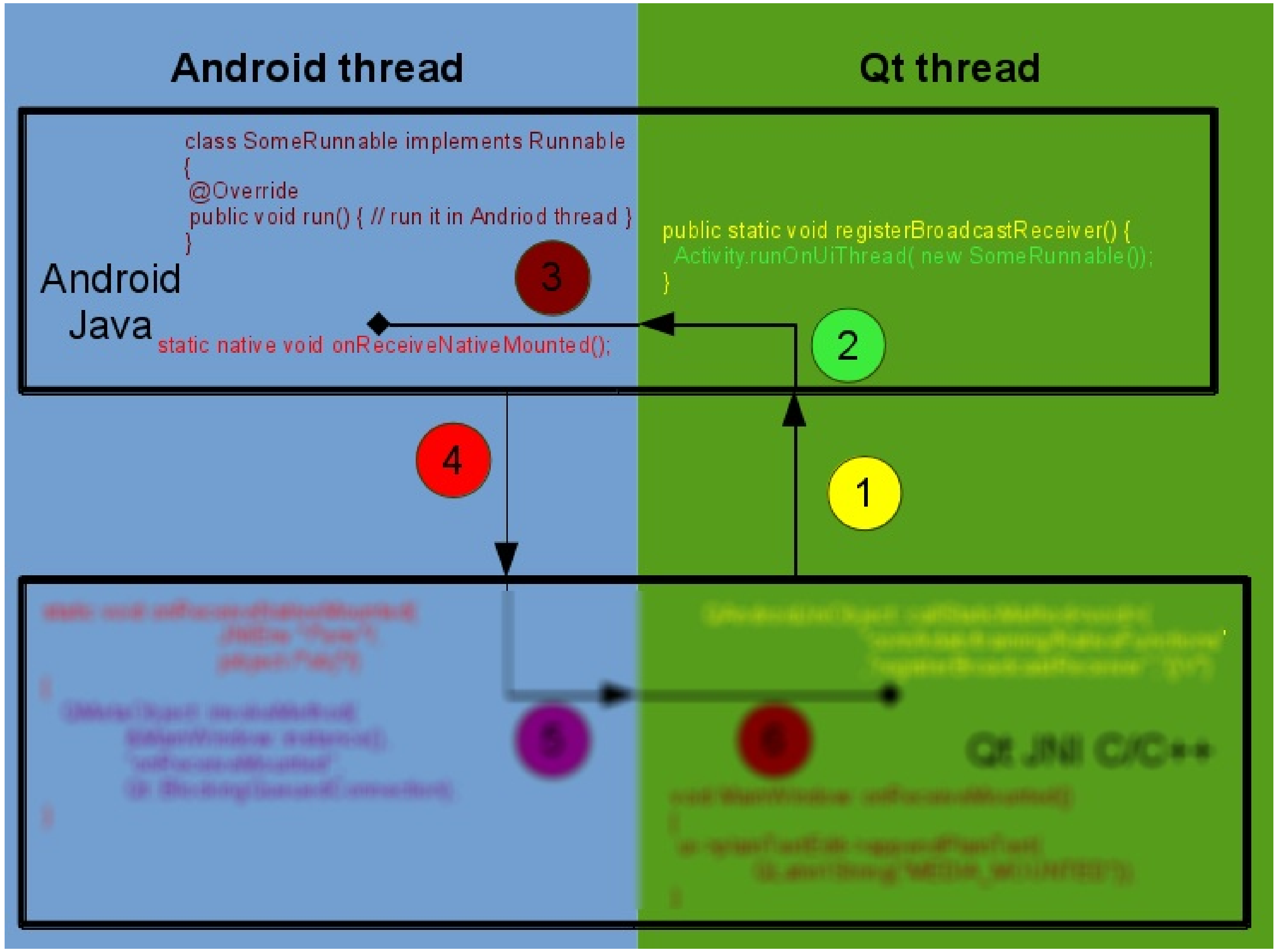


NativeFunctions.java part 3

```java
class SDCardReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Step 4
        // call the native method when it receives a new notification
        if (intent.getAction().equals(Intent.ACTION_MEDIA_MOUNTED))
            NativeFunctions.onReceiveNativeMounted();
        else if (intent.getAction().equals(Intent.ACTION_MEDIA_UNMOUNTED))
            NativeFunctions.onReceiveNativeUnmounted();
    }
}
```



Architecture diagram, Java part



```
class SomeRunnable implements Runnable
{
 @Override
 public void run() { // run it in Andriod thread }
}
```

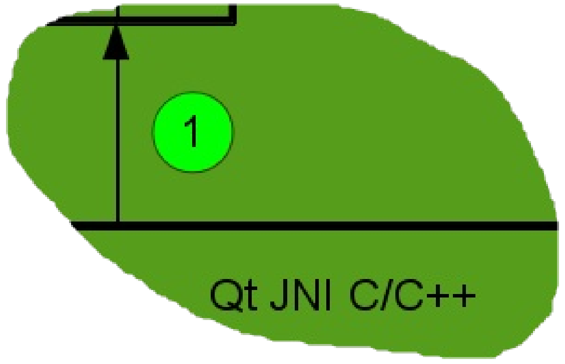
```
public static void registerBroadcastReceiver() {
  Activity.runOnUiThread( new SomeRunnable());
}
```

```
static native void onReceiveNativeMounted();
```

Extending C/C++ part

main.cpp

```cpp
#include "mainwindow.h"
#include <QApplication>
#include <QAndroidJniObject>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    // Step 1.
    // call registerBroadcastReceiver to register the broadcast receiver
    QAndroidJniObject::callStaticMethod<void>("com/kdab/training/NativeFunctions"
                                  , "registerBroadcastReceiver"
                                  , "()V");

    MainWindow::instance().show();
    return a.exec();
}
```

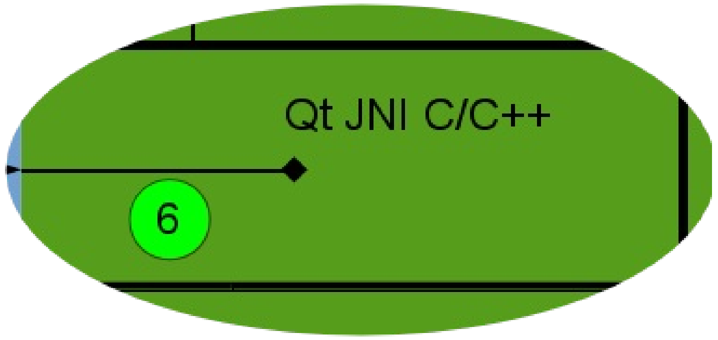


mainwindow.h

```cpp
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    static MainWindow &instance(QWidget *parent = 0);

public slots:
    void onReceiveMounted();
    void onReceiveUnmounted();

private:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
};
```

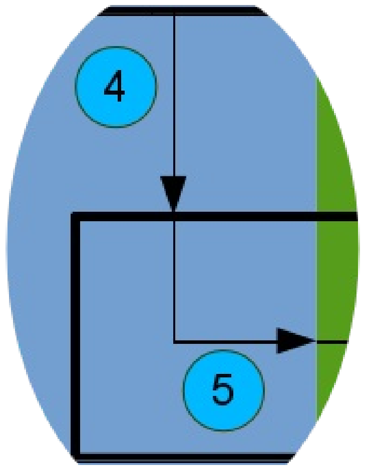
mainwindow.cpp

```
MainWindow &MainWindow::instance(QWidget *parent)
{
    static MainWindow mainWindow(parent);
    return mainWindow;
}

// Step 6
// Callback in Qt thread
void MainWindow::onReceiveMounted()
{
    ui->plainTextEdit->appendPlainText(QLatin1String("MEDIA_MOUNTED"));
}

void MainWindow::onReceiveUnmounted()
{
    ui->plainTextEdit->appendPlainText(QLatin1String("MEDIA_UNMOUNTED"));
}
```



native.cpp part 1

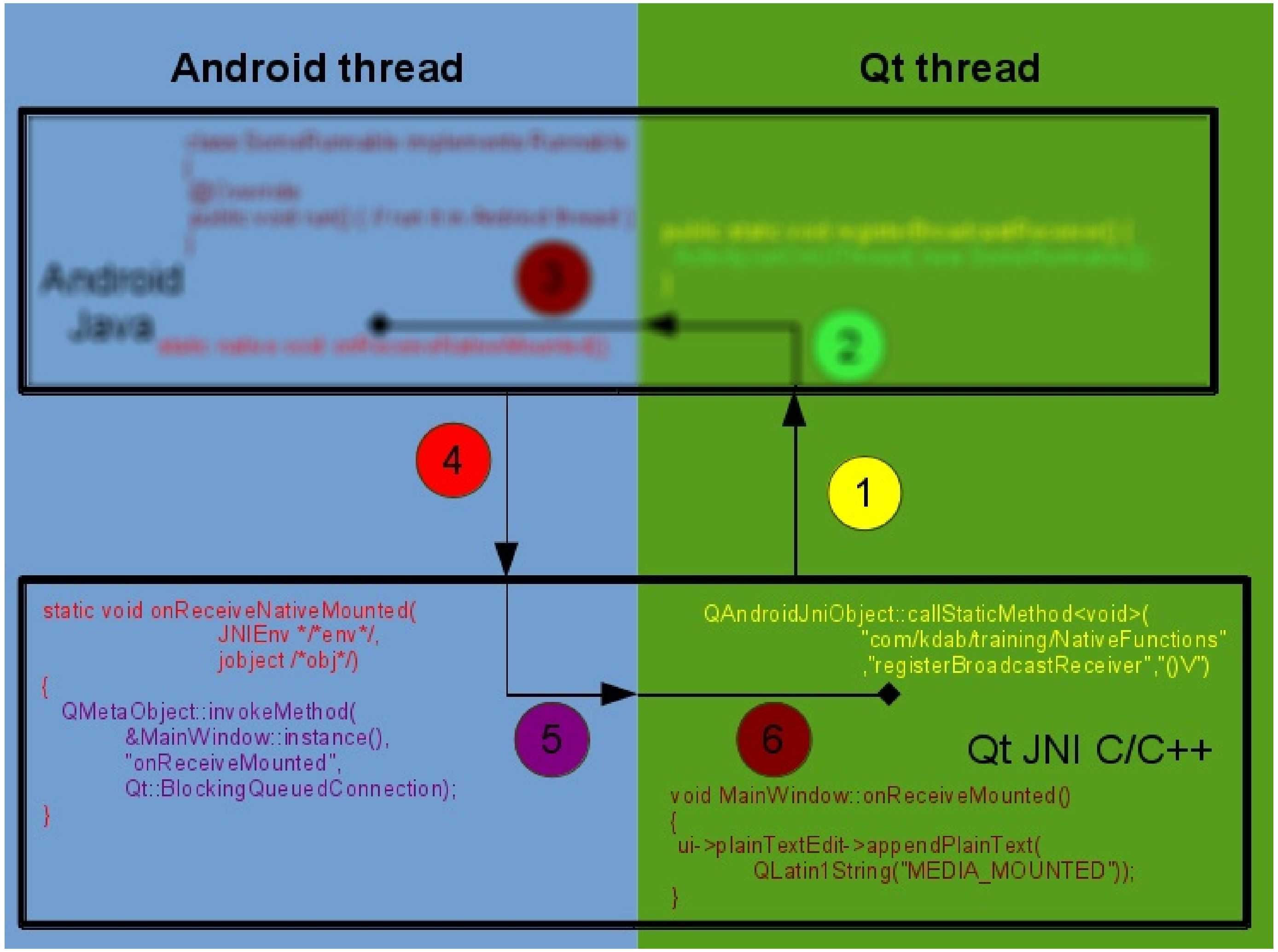
```
// Step 4. callback from java in Android thread
// define our native methods
static void onReceiveNativeMounted(JNIEnv */*env*/, jobject /*obj*/)
{
    // Step 5. Delegate the call to Qt thread.
    QMetaObject::invokeMethod(&MainWindow::instance(), "onReceiveMounted"
                            , Qt::BlockingQueuedConnection);
}

static void onReceiveNativeUnmounted(JNIEnv */*env*/, jobject /*obj*/)
{
    // Step 5. Delegate the call to Qt thread.
    QMetaObject::invokeMethod(&MainWindow::instance(), "onReceiveUnmounted"
                            , Qt::BlockingQueuedConnection);
}
```



native.cpp part 2

```cpp
//create a vector with all our JNINativeMethod(s)
static JNINativeMethod methods[] = {
    {"onReceiveNativeMounted", "()V", (void *)onReceiveNativeMounted},
    {"onReceiveNativeUnmounted", "()V", (void *)onReceiveNativeUnmounted},
};

// this method is called automatically by Java after the .so file is loaded
JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* /*reserved*/)
{
    JNIEnv* env; // get the JNIEnv pointer.
    if (vm->GetEnv(reinterpret_cast<void**>(&env), JNI_VERSION_1_6) != JNI_OK)
        return JNI_ERR;

    // search for Java class which declares the native methods
    jclass javaClass = env->FindClass("com/kdab/training/NativeFunctions");
    if (!javaClass)
        return JNI_ERR;

    // register our native methods
    if (env->RegisterNatives(javaClass, methods,
                        sizeof(methods) / sizeof(methods[0])) < 0) {
        return JNI_ERR;
    }
    return JNI_VERSION_1_6;
}
```

Architecture diagram, Qt JNI part

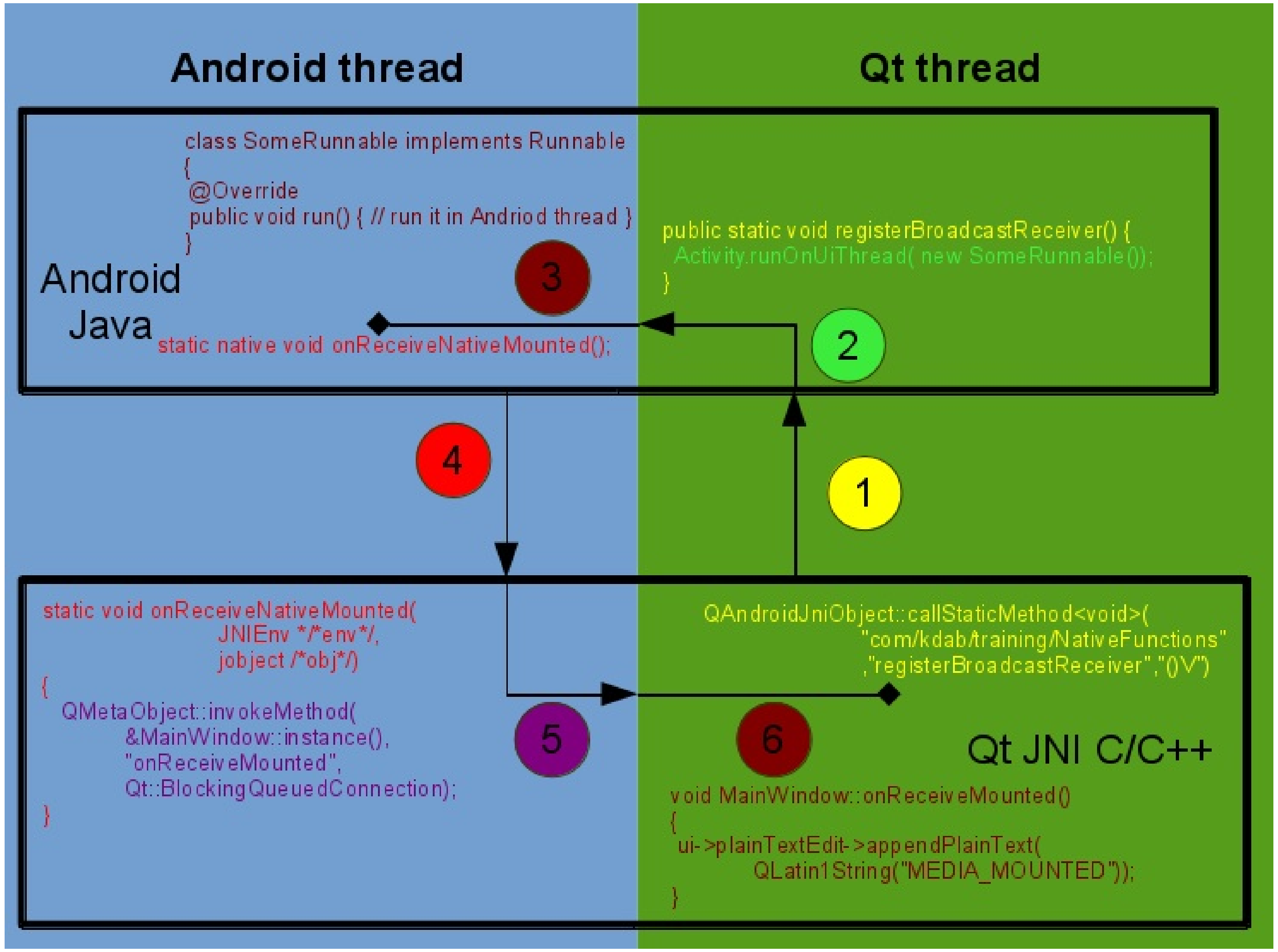


```
QAndroidJniObject::callStaticMethod<void>(
                    "com/kdab/training/NativeFunctions"
                    ,"registerBroadcastReceiver","()V")
```

```
static void onReceiveNativeMounted(
                        JNIEnv */*env*/,
                        jobject /*obj*/)
{
  QMetaObject::invokeMethod(
          &MainWindow::instance(),
          "onReceiveMounted",
          Qt::BlockingQueuedConnection);
}
```

```
void MainWindow::onReceiveMounted()
{
 ui->plainTextEdit->appendPlainText(
          QLatin1String("MEDIA_MOUNTED"));
}
```

Architecture diagram, summary



```
class SomeRunnable implements Runnable
{
 @Override
 public void run() { // run it in Andriod thread }
}
```

```
public static void registerBroadcastReceiver() {
  Activity.runOnUiThread( new SomeRunnable());
}
```

```
static native void onReceiveNativeMounted();
```

```
static void onReceiveNativeMounted(
                    JNIEnv */*env*/,
                    jobject /*obj*/)
{
  QMetaObject::invokeMethod(
          &MainWindow::instance(),
          "onReceiveMounted",
          Qt::BlockingQueuedConnection);
}
```

```
QAndroidJniObject::callStaticMethod<void>(
                    "com/kdab/training/NativeFunctions"
                    ,"registerBroadcastReceiver","()V")
```

```
void MainWindow::onReceiveMounted()
{
 ui->plainTextEdit->appendPlainText(
          QLatin1String("MEDIA_MOUNTED"));
}
```

Questions ?

Thank you for your time!

Contact us:

- http://www.kdab.com
- qtonandroid@kdab.com
- info@kdab.com
- training@kdab.com
- bogdan@kdab.com