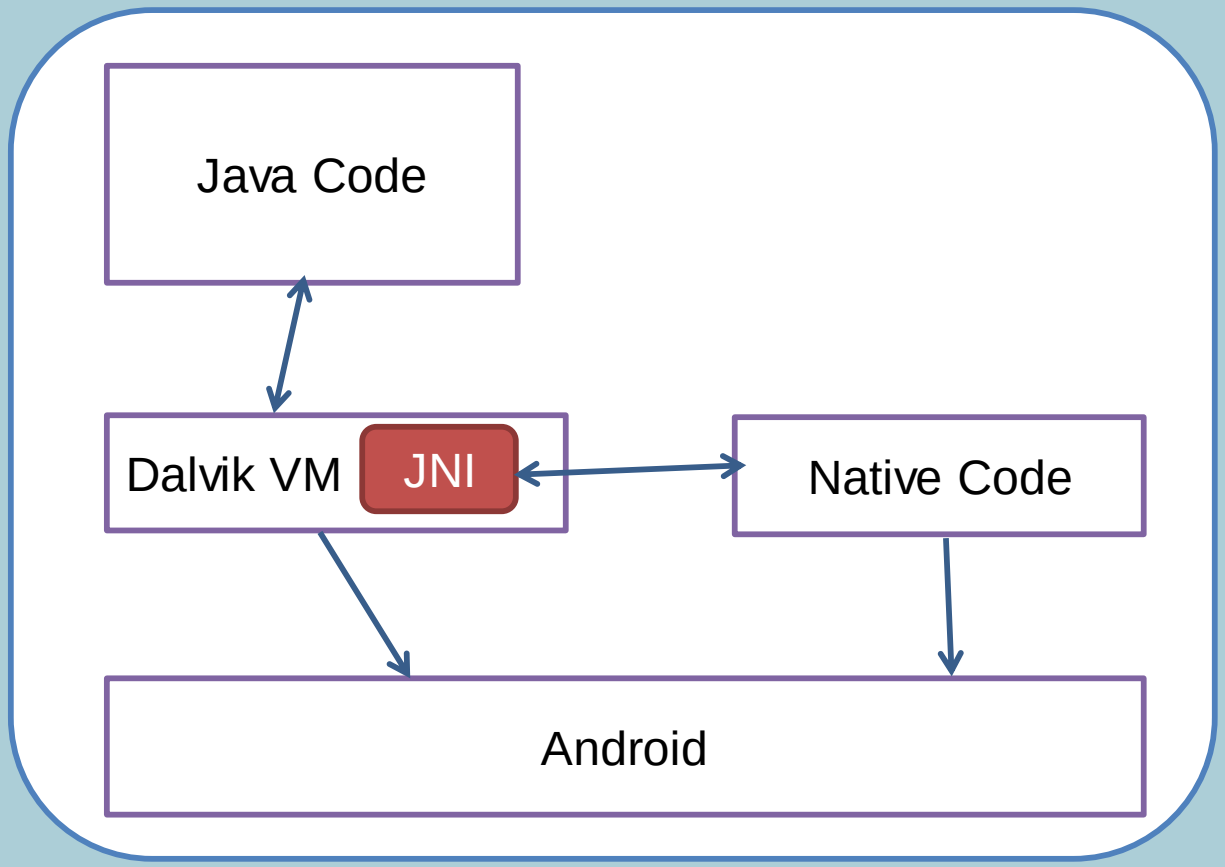
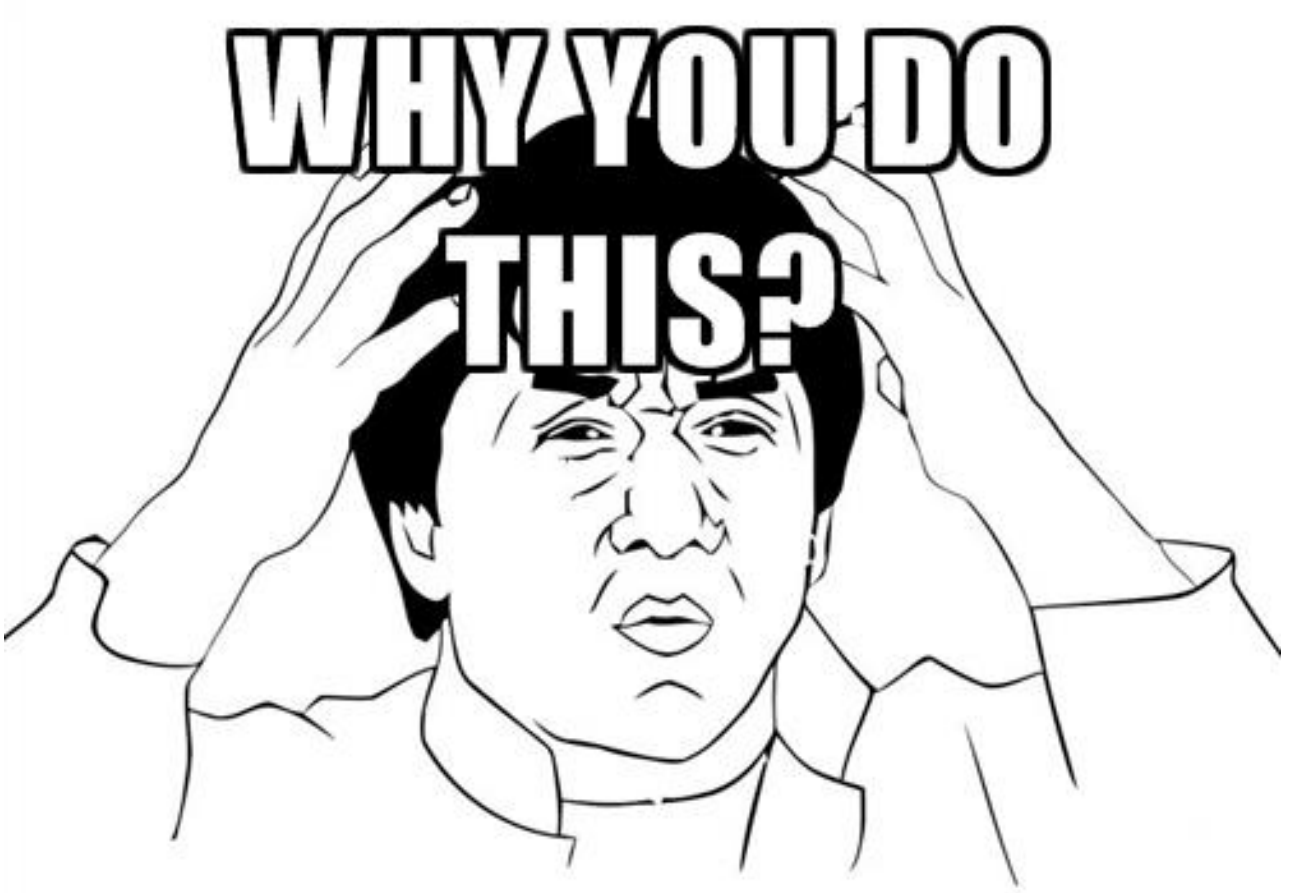
NDK *Intro with examples*

1) Why use it?
2) Gradle experimental plugin
3) Hello NDK example
4) Building NDK app
5) More examples
6) Performance
7) JNI and its Datatypes
8) Other NDK possibilities

Execution




WHY YOU DO
THIS?

Why?

1) Performance

- Native code is run on OS (not DVM)
- Access to CPU features (NEON – 16 ops)
- Critical paths even in assembly code

2)Reuse of existing .c/.cpp code

- Multimedia, games

Calling JNI == extra work!

Gradle experimantal plugin

#not4production

- for production apps use older tools
- Uses new component model (reduced configuration time)
- **Enables NDK support and build**
- Changes in build.gradle


Hello

Hello NDK

Three steps to call the native method:

1. **Load the native library**
   This is done by calling System.loadLibrary("*name*")
2. **Declare the method**
   We declare the method with a native keyword
3. **Invoke the method**
   We call the method just like any normal Java method.

- c/cpp files (declaration name pattern)
- make files (Android.mk, (optional) Application.mk)
- Java declaration (native keyword)
- run native method

javah – helps to generate native declarations based on java declaration


BUILD

NDK build

No need to write build files – just Android.mk and Application.mk

- Lib*.so will be generated automatically (under specified architecture subfolder)
- Can build for specific ABI (Application Binary Interface) or all

Link OS native libraries through gradle

- ldLibs.addAll(["android", "log"])

ABIs

Define how the Android application's machine code is supposed to interact with the system at runtime.

- the CPU instruction set
- endianness
- alignment of memory
- … basically defines a type of architecture.

Most common: **armeabi** and **armeabi-v7a**

- can detect CPU features and facilitate them by preparing different code


100
120
140
160
180
200
220
240
260
280
300

Performance

Execution of *sum of all squares of array containing 100 elements.*

Number	Native[ns]	Java[ns]	Ratio Native/Java
1	61000	66000	0,924242424
2	142000	84000	1,69047619
3	41000	66000	0,621212121
4	31000	61000	0,508196721
5	26000	75000	0,346666667
6	25000	63000	0,396825397
7	26000	56000	0,464285714
8	24000	114000	0,210526316
9	29000	59000	0,491525424
Average ratio			
0,496409075			

Extreme values need to be excluded. Caused by:

- Thread interuption and fight
- Different VM states

After exculsion the average ratio is considered.

Conclusion: in this scenario **native execution** is almost **2 times faster**.

After every time app needs to be relaunched. This provides more reliable results (prevent VM from making optimizations)

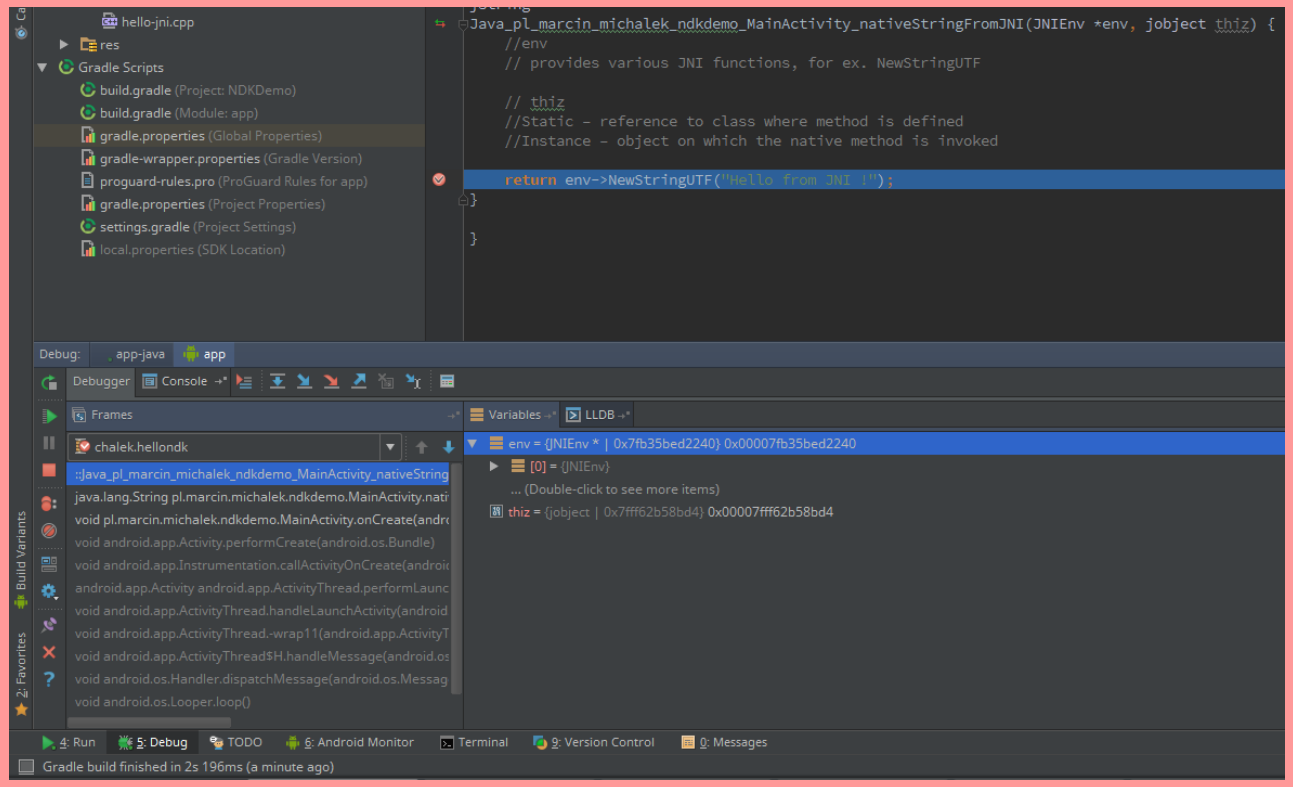
JNI environment methods

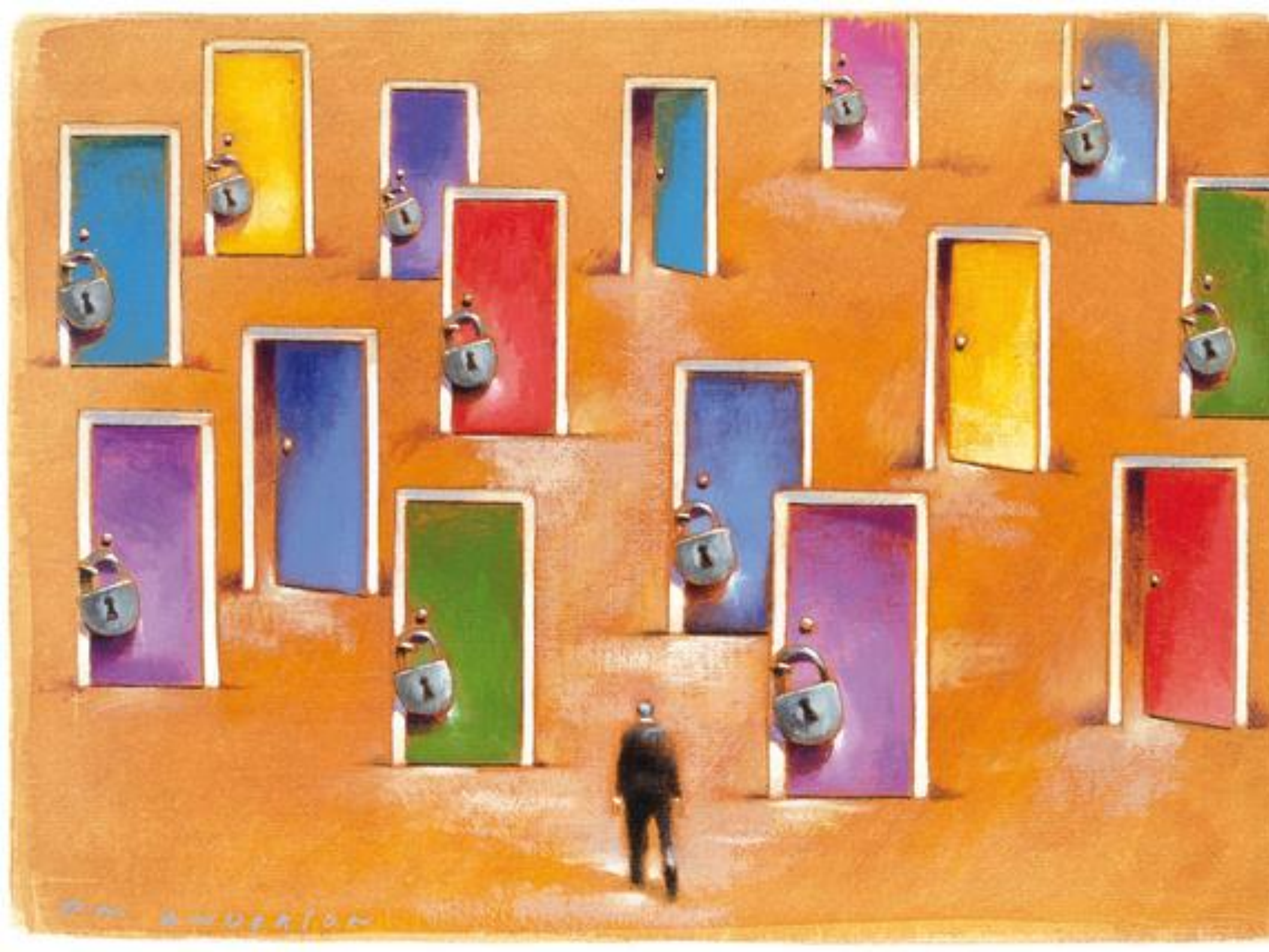
- manipulating strings (UTF-16 (Java string) to native array of UTF-8 characters)
- Managing references (Local, Global, Week)
- Manipulating classes, object and arrays
- Accessing Java statics and instance fields and methods from native code
- Integrating assembly code


BUG

Debugging & Error handling

- No error checking by default (performance) (CheckJNI flag – enables different versions of functions with error checking)
- Fastest debugging tool – androd Log (circular)
- Can set breakpoints after setting Hybrid debugger (*Edit run configurations → Debugger*) (attaching for ~1 min)





Other possibilities

- Native Activity ☺ - No java code required, *native_activity.h interface*
- Advanced graphics with **Open G**raphics **L**ibrary ES and Vulkan
- Multimedia frameworks **OpenMAX** AL
- Audio processing with **OpenSL** AS
- **jni graphics** library
- Fast compression with **Zlib**

NDK is **not useful in most of the cases**, **but can save life in specific cases** (performance increase, library reuse)

Thanks!

Sources

1. Feipeng Liu Android *Native Development Kit Cookbook*
2. Gradle Experimental plugin docs: https://sites.google.com/a/android.com/tools/tech-docs/new-build-system/gradle-experimental
3. Android Developer/NDK: https://developer.android.com/ndk/guides/index.html