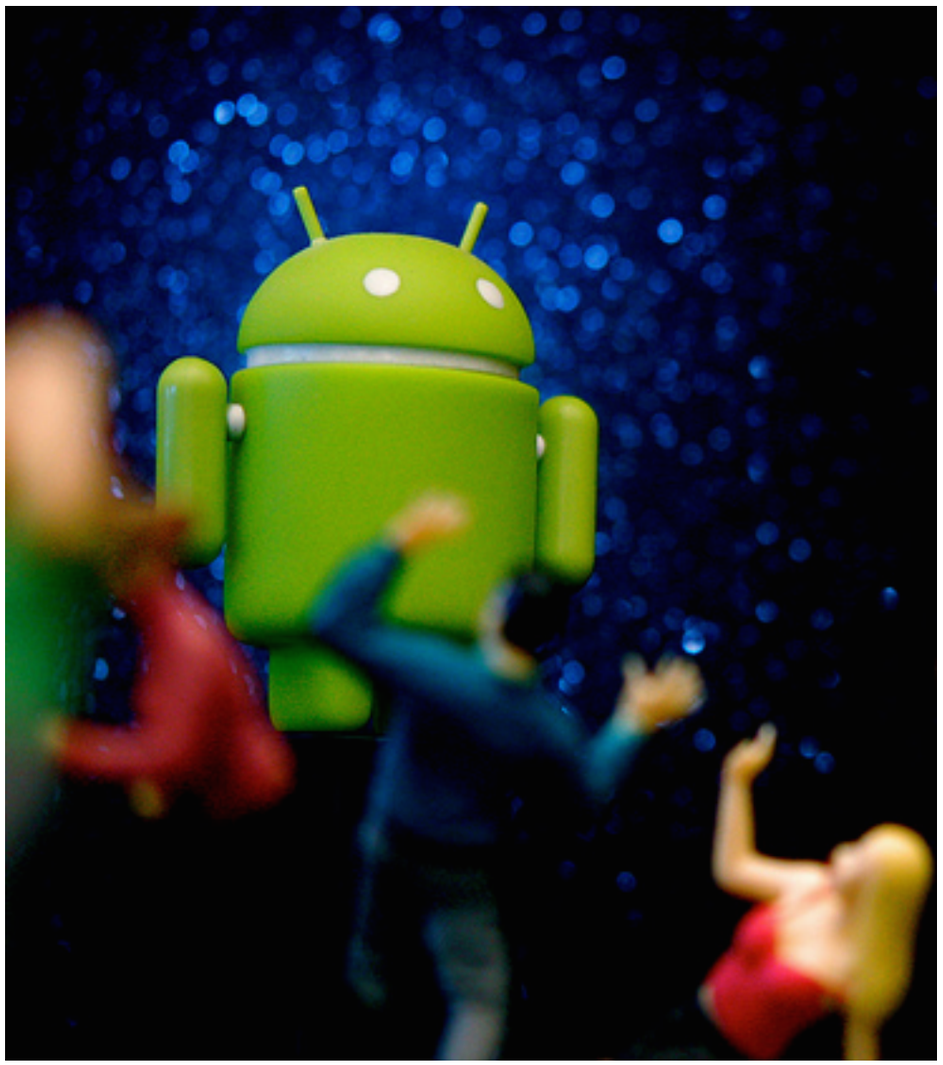


JD Hancock: Reign of The Android. http://bit.ly/1GN8vmg

Building High-Performance Android Applications in Java & C++

Kenneth Geisshirt
kg@realm.io

Realm Inc.
@realm
http://github.com/Realm/
http://realm.io/

Today's goal

- Android development is not just Java
- C and C++ can speed apps up - and reuse old legacy code
- Learn a few tricks when working with Java Native Interface

Swen-Peter Ekkebus: Goal! http://bit.ly/1x1suYm

Avinash Sharma: http://bit.ly/1aRorcH

Agenda

- About me and Realm
- Example: estimating π
- Java Native Interface
- Building
- Memory
- Java types, arrays, and strings
- Logging and profiling

Exploratorium. After Dark: Time http://bit.ly/1aSQxo0

About me

- Lives in Tårnby, a suburb of Copenhagen
- Education
  - M.Sc. in computer science and chemistry
  - Ph.D. in soft condensed matter
- Commodore 64 was my first home computer
- UNIX user, developer, and sysadmin since 1990
  - BSD UNIX (Tahoe), Solaris, Linux, Irix, OS X, …
- Tech writer and reviewer
  - Packt Pub, O'Reilly, Linux Magazine, Alt om DATA
- Currently working for Realm as developer

Leo Reynolds, https://www.flickr.com/photos/lwr/


Shane Doucette, https://www.flickr.com/photos/lwr/

About Realm

- Founded in 2011 (YC S11)
- Funded by Khosla, Scale, …
- Distributed work force
  - Copenhagen: 8 developers
  - San Francisco: 4 developers + marketing + management
  - Developers in Århus, Perth, Berlin, Tokyo, and Seoul

- New database written from scratch
  - Cross-platform core in C++
  - Full ACID
  - Fast, low footprint, easy to use
- Apache License 2.0

Java Native Interface (JNI)

- Use C or C++ in your apps → no JIT needed as always machine instructions

- Access to system calls (like `mmap()`)

- Almost no limits on memory usage

- OpenGL programming

Alan Bates. Good Times http://bit.ly/1BXX7FN

JNI disadvantages

Rocher: That_Sux http://bit.ly/1BdGNhJ

- C and C++ memory management → easy to crash your app
- And debugging across languages is hard

- Bionic C library isn't the GNU C library
- Doing a native call is expensive (in time)
- Cumbersome to use non-NDK libraries

Use cases

- AndEngine (2D Game Engine)

  http://www.andengine.org

- ImageMagick for Android

  https://github.com/paulasiimwe/Android-ImageMagick

- LevelDB for Android

  https://github.com/litl/android-leveldb

- Support Vector Machine for Android

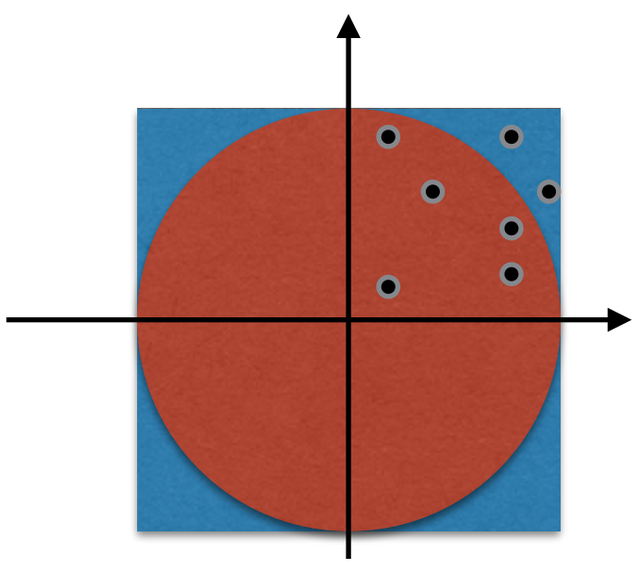
  https://github.com/cnbuff410/Libsvm-androidjni

Estimating π

Estimating π using dart

- Throw arrows at random
- $N_{hit}/N = A/4$ (quarter of a circle)
- $A = \pi \cdot r^2$ (area of a circle)
- $\pi = 4 \cdot N_{hit}/N$ (with $r = 1$)

Bogdan Suditu: Darts. http://bit.ly/18jCExj



```
for i = 1,..,N
    x = random(0, 1)
    y = random(0, 1)
    if (sqrt(x·x+y·y) < 1)
        Nhit = Nhit +1
π = 4·Nhit/N
```

Example: the app

Android app to estimate π

- pure Java implementation
- pure C++ implementation
- mixture of Java (iterations) and C++ (random number generator)
- https://github.com/kneth/AndroidCalcPi

Example: the classes

jeramiah.andrick. My random number generator http://bit.ly/1wbTbP2

Random number generator

- Constructor with seed
- Method `uniform()` to return random number between 0.0 and 1.0

π estimator

- Constructor with number of iteration
- Method `estimate()` to calculate the value of π

Quinn Dombrowski: Pi http://bit.ly/1Kz55cp

Organising your files

- Place C/C++ files in `app/src/main/jni/src`
- Add `ndk.dir` to `local.properties`
- Add an `ndk` section to your app's `build.gradle`

add to the `android` section (currently deprecated)

```
ndk {
    moduleName "calcPi"
    cFlags "-std=c++11 -fexceptions"
    stl "gnustl_shared"
}
```

Building and loading your native code

- Automatically build for all supported architectures (ARM, ARMv7, ARM64, MIPS, MIPS64, x86, x86-64)

- Android only installs the relevant architecture

- Load native code early (main activity's `onCreate`)

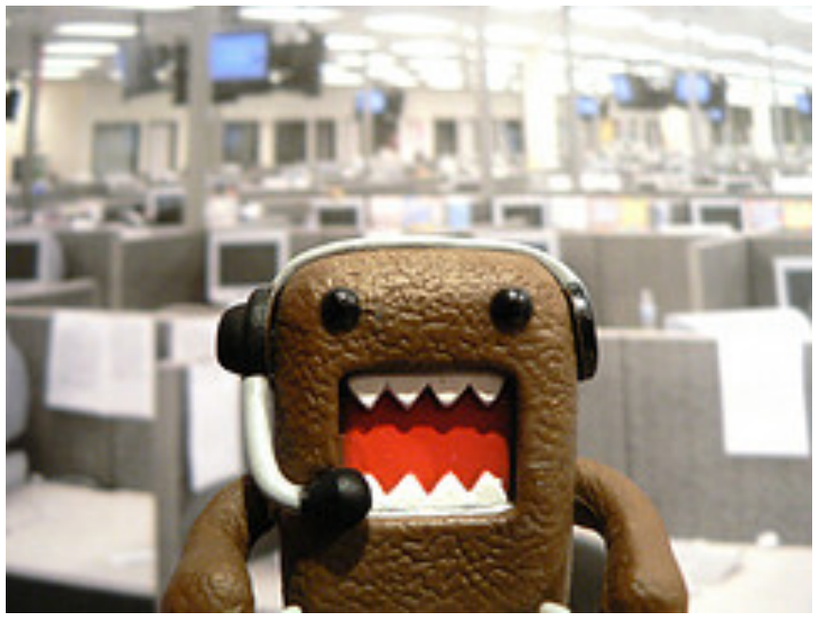
Thomas Hawk: Loading Zone http://bit.ly/1MoBk9S

```
System.loadLibrary("calcPi");
```

The module name (from `build.gradle`)

Calling a C++ function

- Use Java's `native` keyword in the signature
- Implement the function in C++
- Use a `long` to store the C++ pointer when you creates an object



lamont_cranston: call center http://bit.ly/1A4JlK5

Returns a pointer (a C++ object)

Operates on a C++ object

Tell the Java compiler it's a native function

```
native long nativeCreate(long iterations);
native double nativeEstimate(long nativePtr);
```

Generate header files

Paul: Generator Room Door http://bit.ly/1C3SSs7

- `.class` files contain information about native calls
- `javah` can extract signatures and generate C/C++ header files

Add path to jar files you depend on

Classes to extract from

```
!/bin/bash
# Setting up
CLASSDIR="$(pwd)/../app/build/intermediates/classes/debug"
JNIDIR="$(pwd)/src"

# Generate the headers
(cd "$CLASSDIR" && javah -jni -classpath "$CLASSDIR" -d "$JNIDIR"
net.zigzak.calcpi.NativePi net.zigzak.calcpi.NativeRNG)

# Remove "empty" header files (they have 13 lines)
wc -l "$JNIDIR"/*.h | grep " 13 " | awk '{print $2}' | xargs rm -f
```

Java types in C/C++

- Java types are mapped to C/C++ types
- Convert `Date` to `long` before call
- Pass pointers over as `long`/`jlong`

Java	C/C++
`long`	`jlong`
`String`	`jstring`
`long[]`	`jlongArray`
`Object`	`jObject`
`float`	`jfloat`

The pointer

Generated function name

cast to a proper C++ pointer

```
JNIEXPORT jdouble JNICALL Java_net_zigzak_calcpi_NativePi_nativeEstimate
  (JNIEnv *, jobject, jlong nativePtr)
{
    Pi *pi = reinterpret_cast<Pi *>(nativePtr);
    double estimate = pi->estimate();
    return estimate;
}
```

Define as a macro if used often:

```
#define P(x) reinterpret_cast<Pi *>(x)
```

Working with arrays

- Copy values from JVM memory to C/C++ memory

```
jsize arr_len = env->GetArrayLength(columnIndexes);
jlong *arr = env->GetLongArrayElements(columnIndexes, NULL);
// use the elements of arr
env->ReleaseLongArrayElements(columnIndexes, arr, 0);
```

- Remember to free C/C++ memory → avoid memory leaks

```
jArray = env->NewByteArray(jlen);
if (jArray)
  // Copy data to Byte[]
  env->SetByteArrayRegion(jArray, 0, jlen,
      reinterpret_cast<const jbyte*>(bufPtr));
free(bufPtr);
return jArray;
```

- Create a new Java array
- Copy C/C++ array to Java array

Strings as troublemakers

- Java strings are in modified UTF-8
  - null character encoded as 0xC0 0x80 (not valid UTF-8)
  - Many C/C++ libraries assume plain UTF-8

Create a new Java string

```
jchar stack_buf[stack_buf_size];
// adding characters to stack_bf
jchar* out_begin = stack_buf;
if (int_cast_with_overflow_detect(out_curr - out_begin, out_size))
    throw runtime_error("String size overflow");
return env->NewString(out_begin, out_size);
```

- Copy Java string to C/C++ array
- Remember to deallocate C array

```
jstring s;  // typical a JNI function parameter
jchar *c = e->GetStringChars(s, 0);
// use c
env->ReleaseStringChars(s, c);
```

C++ exceptions

- C++ code can throw exceptions
  - `new` can throw `bad_alloc`,etc.
- If uncaught the app crashes with hard-to-understand messages in the log
- Better solution:
  - catch and rethrow as Java exception

```
Build fingerprint: 'generic_x86/sdk_x86/generic_x86:4.4.2/KK/999428:eng/test-keys'
Revision: '0'
pid: 1890, tid: 1890, name: t.zigzak.calcpi  >>> net.zigzak.calcpi <<<
signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
eax 00000000  ebx 00000762  ecx 00000762  edx 00000006
esi 00000762  edi 0000000b
xcs 00000073  xds 0000007b  xes 0000007b  xfs 00000000  xss 0000007b
eip b7707c96  ebp b776cce0  esp bfa22090  flags 00200203
backtrace:
#00  pc 0003bc96  /system/lib/libc.so (tgkill+22)
#01  pc 00000005  <unknown>
stack:
bfa22050  00000001
bfa22054  b899c6d0  [heap]
bfa22058  00000015
bfa2205c  b76d9ef9  /system/lib/libc.so (pthread_mutex_unlock+25)
bfa22060  a73bfd19  /data/app-lib/net.zigzak.calcpi-1/libgnustl_shared.so
bfa22064  b7767fcc  /system/lib/libc.so
```

Throwing a Java exception

- You don't throw an exception
- You set the "exception" state in JVM
- Return from C/C++ function

followtheinstructrions: hand werpen http://bit.ly/1Bo3gZx

1 Get a reference to the exception

2 Set the "exception" state

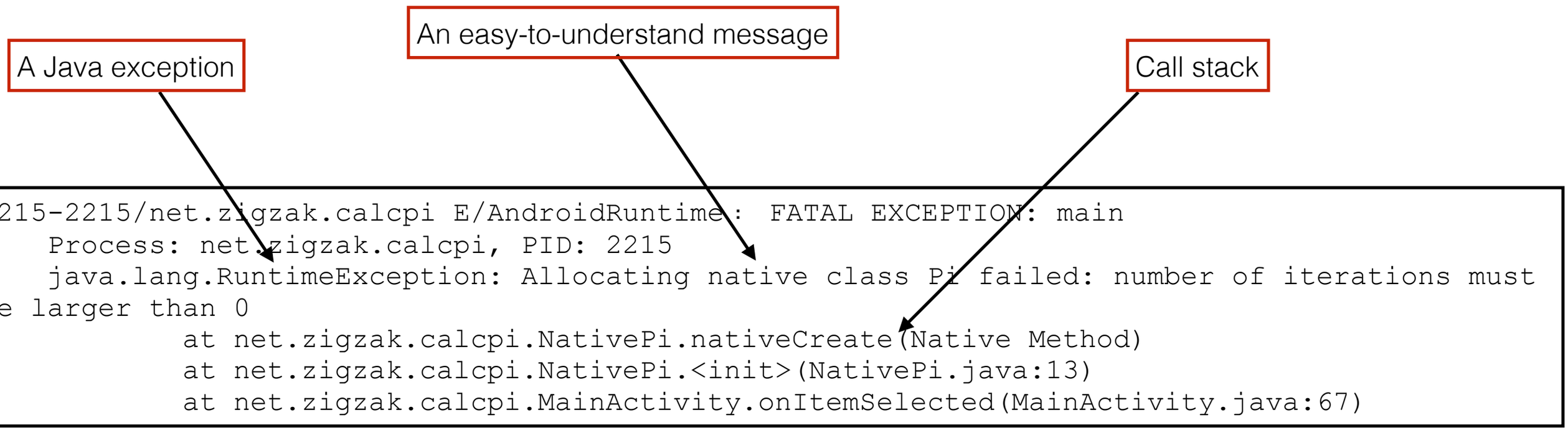
3 Clean up

```
try {
    Pi *pi = new Pi(static_cast<long>(iterations));
    return reinterpret_cast<jlong>(pi);
}
catch (std::exception& e) {
    jclass jExceptionClass =
       env->FindClass("java/lang/RuntimeException");
    std::ostringstream message;
    message << "Allocating native class Pi failed: " << e.what();
    env->ThrowNew(jExceptionClass, message.str().c_str());
    env->DeleteLocalRef(jExceptionClass);
}
return 0;
```

Throwing a Java exception

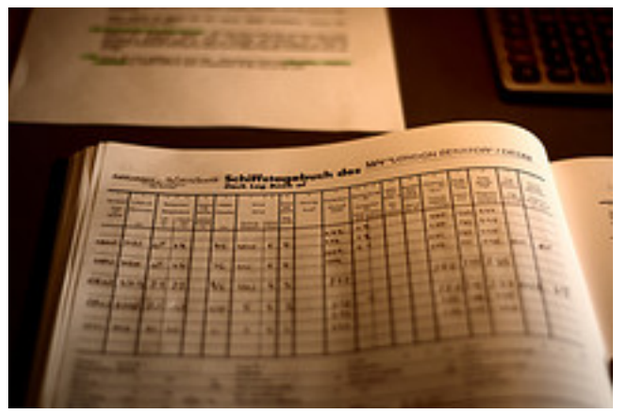
- You have a proper Java exception

- Catch it or die (almost gracefully)



Logging

- JNI provides access to the Android log
- `__android_log_print` is a simple variadic log function



vitelone: Deck Log Book http://bit.ly/1AXKZ0j

- Linking with `liblog` is required
- `size_t` depends on bit width → cast to `int64_t` before logging

Link flag

```
ndk {
    moduleName "calcPi"
    cFlags "-std=c++11 -fexceptions"
    ldLibs "log"
    stl "gnustl_shared" // for exceptions
}
```

Macro to simplify logging

```
#define LOG(...) __android_log_print(ANDROID_LOG_DEBUG, "calcPi", __VA_ARGS__);
```

Tracing JNI calls

- It is useful the trace calls
  - Define `ENTER`, `ENTER_PTR`, and `LEAVE` macros
    - Logging native pointer as `jlong` often produce negative numbers - don’t worry

```
1955-1955/net.zigzak.calcpi D/calcPi: Enter Java_net_zigzak_calcpi_NativePi_nativeCreate
1955-1955/net.zigzak.calcpi D/calcPi: iterations: 10000
1955-1955/net.zigzak.calcpi D/calcPi: Enter Java_net_zigzak_calcpi_NativePi_nativeEstimate
-1202665872
1955-1955/net.zigzak.calcpi D/calcPi: estimate = 3.128400e+00
```

Profiling and debugging

- Use `Debug.startMethodTrace()` to trace Java calls

- The Android NDK Profiler project is a port of GNU Profiler (gprof)[1]. Supports only ARM and ARMv7.

- Valgrind 3.10 supports Android[2]

- nVidia has a debugger for the Tegra platform[3]

```
adb ps
adb shell am profile start PID FILE
adb shell am profile stop
adb pull FILE .
traceview FILE
```

[1] https://code.google.com/p/android-ndk-profiler/

[2] http://valgrind.org/downloads/current.html

[3] https://developer.nvidia.com/nvidia-debug-manager-android-ndk

Compiler and linker options

- A few options can reduce the APK size[1]
  - Link Time Optimization (`-flto`)
  - Eliminate unused code (`-ffunction-sections -fdata-sections`) and unused ELF symbols (`-fvisibility=hidden`)
  - Optimize for space (-Os)
- Requires custom Gradle script 😞

[1]http://realm.io/news/reducing-apk-size-native-libraries/

References

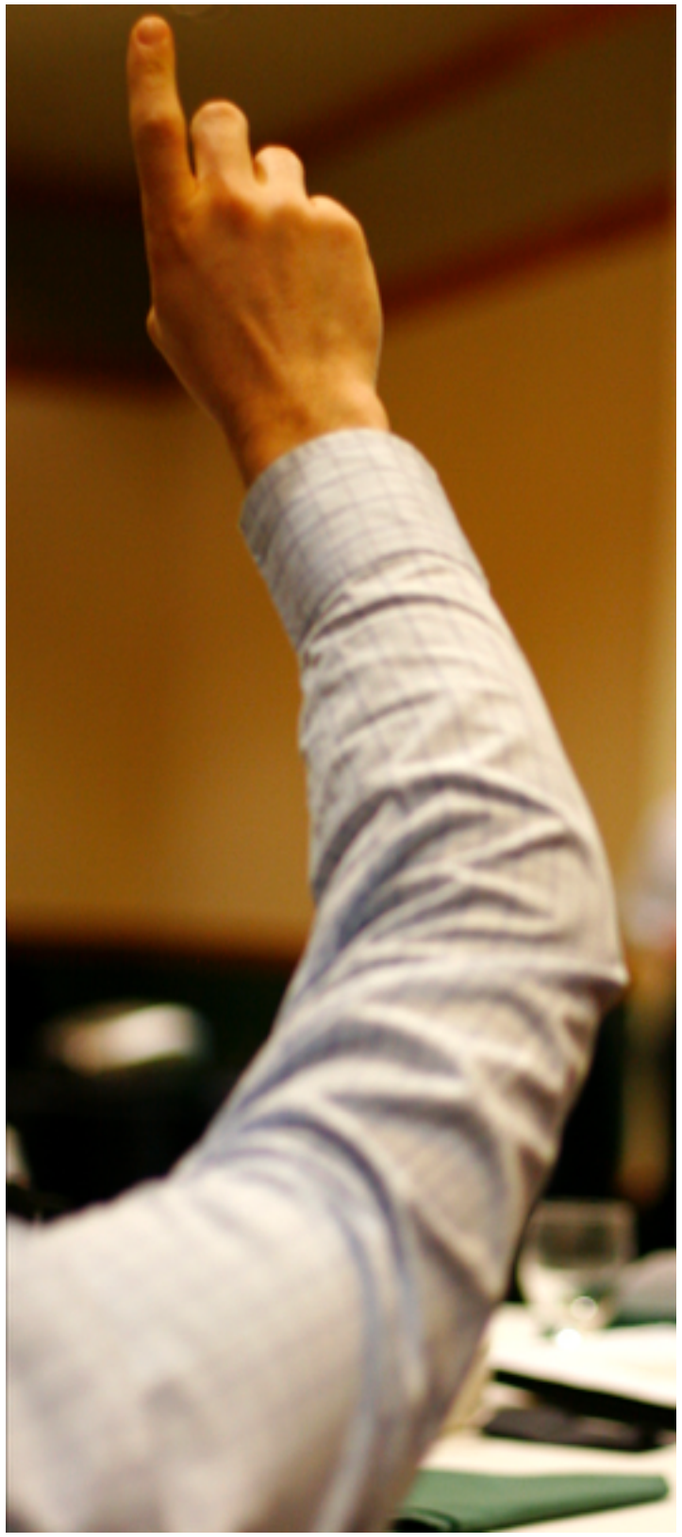
- *High Performance Android Apps* by Doug Sillars. O'Reilly Media, 2015 (not yet published).

- *Essential JNI* by Rob Gordon. Prentice Hall, 1998.

- *The Java Native Interface* by Sheng Liang. Addison-Wesley, 1999.

- *Android NDK* by Sylvian Rataboil. Packt Publishing, 2012.

Questions?



sean dreilinger: question from the audience http://bit.ly/1C1CVB7