

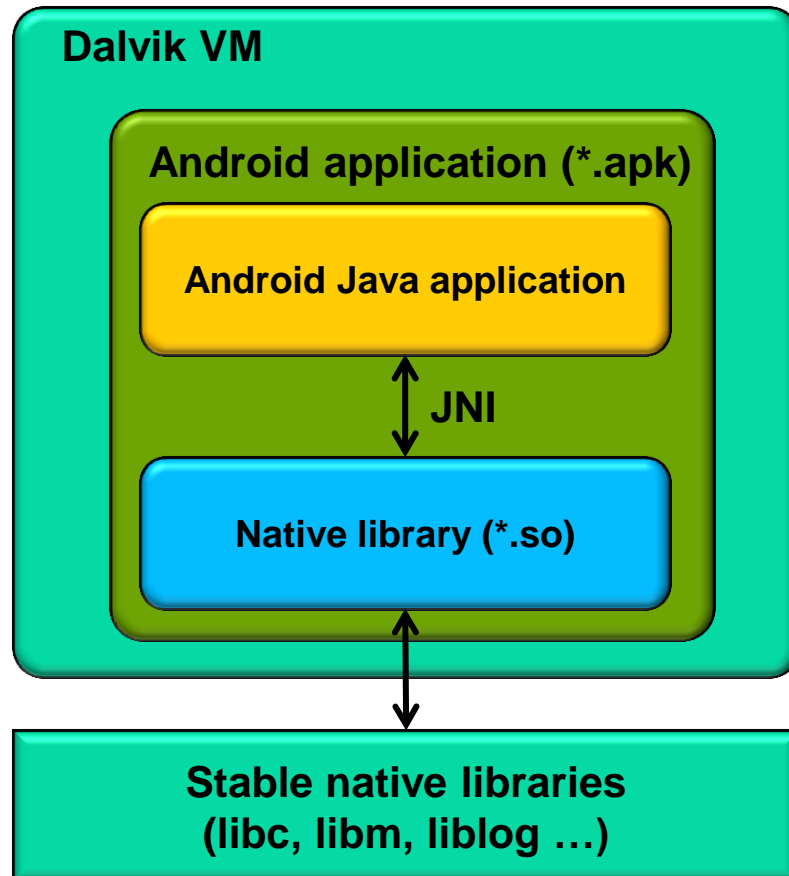
- **Contents**

1. What you can do with NDK
2. When to use native code
3. Stable APIs to use / available libraries
4. Build native applications with NDK
5. NDK contents and structure
6. NDK cross-compiler suite
7. Android EABI
8. NDK C++ support
9. JNI - Calling native functions from Java code
10. SDK project with native code
11. Native activity



1. What you can do with NDK

- Build **native libraries** that are callable from Android Java application code (JNI).
- **Build executables** (non-recommended use of NDK).
- Debug native program (with gdb).



Recommended use of native functions:

An Android Java application makes native calls through JNI.

Thus the entire application running in the VM is subject to the defined Android application lifecycle.

It is possible to run entirely native applications on Android. However, it is recommended to use a small Java wrapper for managing the lifecycle of the application (start, stop).

2. When to use native code

The power of Android lies in the rich Java application framework to be used by Android applications written in Java.

In special cases, however, it may be required to write native code that directly runs on the CPU without the Android VM interpreter.

NDK is a toolkit for writing and integrating native code with Java application code.

Native code characteristics for use in Android:

- Graphically and computationally intensive (e.g. complex algorithms)
- Few library dependencies (restricted to stable Android libraries provided by NDK)
- Little interaction between Java application code and native code (ideally, the Java application calls computationally intensive native functions and receives the result; there should not be frequent calls and callbacks between Java and native code)

Primary uses of NDK:

NDK should be used to build **native libraries** (shared objects) that are called by an Android application.

Entirely native applications without Java code are possible starting from Android 2.3 (Gingerbread) by using **NativeActivity**.

Non-recommended uses of NDK:

Custom native applications that run outside the VM.

3. Stable APIs to use / available libraries

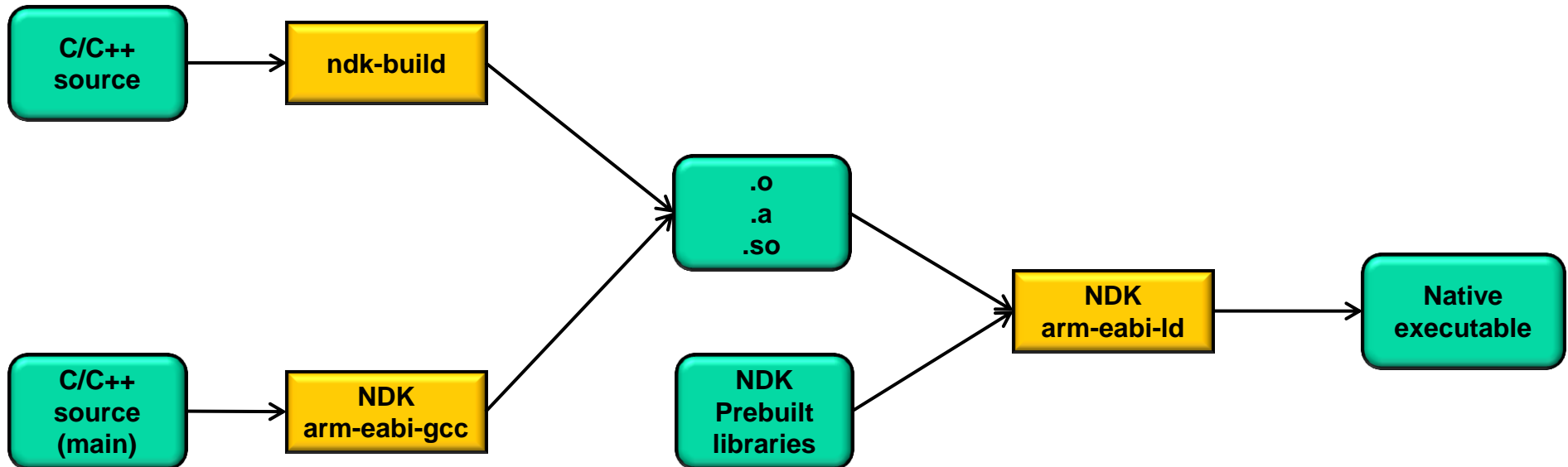
The Android NDK contains a small number of stable libraries that are guaranteed to be contained in successive Android versions.

It is recommended that native code only make use of these stable libraries. If native code uses non-stable libraries, the native application may break upon an Android update.

| | | android-3 | android-4 | android-5 android-6 android-7 | android-8 | android-9 | android-14 |
|--------------------|--|-------------|-------------|-------------------------------------|-------------|-------------|-------------|
| Library | Description | Android 1.5 | Android 1.6 | Android 2.0 | Android 2.2 | Android 2.3 | Android 4.0 |
| crtbegin_dynamic.o | Calls of global object ctors | Yes | Yes | Yes | Yes | Yes | Yes |
| crtbegin_so.o | Calls of global object ctors | Yes | Yes | Yes | Yes | Yes | Yes |
| crtbegin_static.o | Calls of global object ctors | Yes | Yes | Yes | Yes | Yes | Yes |
| crtend_android.o | Calls of global object dtors | Yes | Yes | Yes | Yes | Yes | Yes |
| crtend_so.o | Calls of global object dtors | Yes | Yes | Yes | Yes | Yes | Yes |
| libandroid.so | Functions for access to Java platform from native code | No | No | No | No | Yes | Yes |
| libc.so | Standard C library (bionic) | Yes | Yes | Yes | Yes | Yes | Yes |
| libdl.so | Dynamic linker library | Yes | Yes | Yes | Yes | Yes | Yes |
| libEGL.so | Interface library for low level graphics buffer access | No | No | No | No | Yes | Yes |
| libGLESv1_CM.so | Open GL graphics library | No | Yes | Yes | Yes | Yes | Yes |
| libGLESv2.so | Open GL graphics library | No | No | Yes | Yes | Yes | Yes |
| libjnigraphics.so | C-function-based library for graphics pixel access | No | No | No | Yes | Yes | Yes |
| liblog.so | Android logging library | Yes | Yes | Yes | Yes | Yes | Yes |
| libm.so | Math library | Yes | Yes | Yes | Yes | Yes | Yes |
| libOpenMAXAL.so | Audio and video streaming library | No | No | No | No | No | Yes |
| libOpenSLES.so | Audio streaming library | No | No | No | No | Yes | Yes |
| libstdc++.so | Minimal C++ library (no exceptions, no RTTI) | Yes | Yes | Yes | Yes | Yes | Yes |
| libthread_db.so | Thread debug support library. | Yes | Yes | Yes | Yes | Yes | Yes |

4. Build native applications with NDK

- The NDK build system is made for creating .a (static libs) and .so (shared libs).
- The shell script <NDK-base>/ndk-build creates the library output.
- With some minimal effort it is possible to create fully native applications:



















5. NDK contents and structure (1/2)

NDK installation simply requires unzipping it to a suitable location.

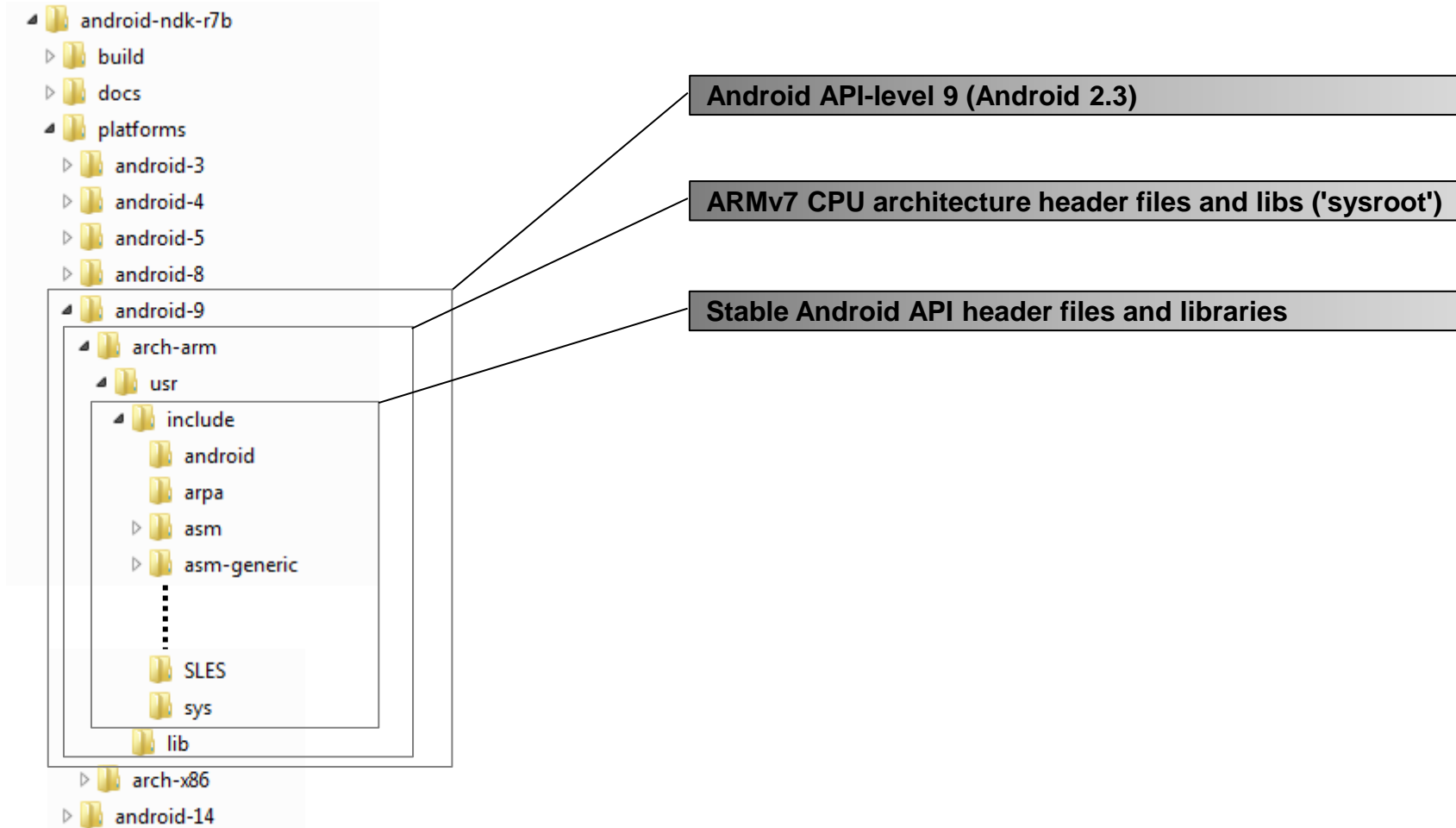
NDK contains a cross-toolchain for ARM and x86 based CPUs, header files and stable libraries.

NDK R7 structure:

| | | |
|--|-------|--|
|  build | ----- | Build scripts (makefiles, awk scripts etc.) |
|  docs | ----- | Documentation (HTML) |
|  platforms | ----- | Platforms (header files and stable libraries) |
|  prebuilt | ----- | Build executables (make, awk, sed, echo) |
|  samples | ----- | Samples (hello world, JNI example etc.) |
|  sources | ----- | Source files that can be linked to an application or library |
|  tests | ----- | Test scripts for automated tests of the NDK |
|  toolchains | ----- | ARM Linux and x86 toolchains (compiler, linker etc.) |
|  documentation.html | ----- | Documentation entry point |
|  GNUmakefile | ----- | Makefile for building NDK |
|  ndk-build | ----- | Build script for building a native application or library |
|  ndk-build.cmd | ----- | Experimental Windows native build script (working?) |
|  ndk-gdb | ----- | GDB debug start script |
|  ndk-stack.exe | ----- | Stack trace analysis tool |
|  README.TXT | ----- | Readme file |
|  RELEASE.TXT | ----- | NDK release identifier (contents for R7: r7d) |

5. NDK contents and structure (2/2)

The platforms sub-folder contains stable header files and libraries.



C++ headers and libraries are under `<NDK-base>/sources/cxx-stl`.

6. NDK cross-compiler suite (1/3)

Standard naming convention for cross-compilers:

<arch>-<vendor>-<os>-<abi>

Example:

`arm-linux-androideabi-c++.exe`

→ Architecture (CPU): ARM

→ Vendor: None

→ OS: Linux

→ ABI: Android EABI (see below)

NDK toolchains:

NDK contains GNU-based cross-compile tools for ARM7 and x86 CPUs.

The NDK toolchain can be used for:

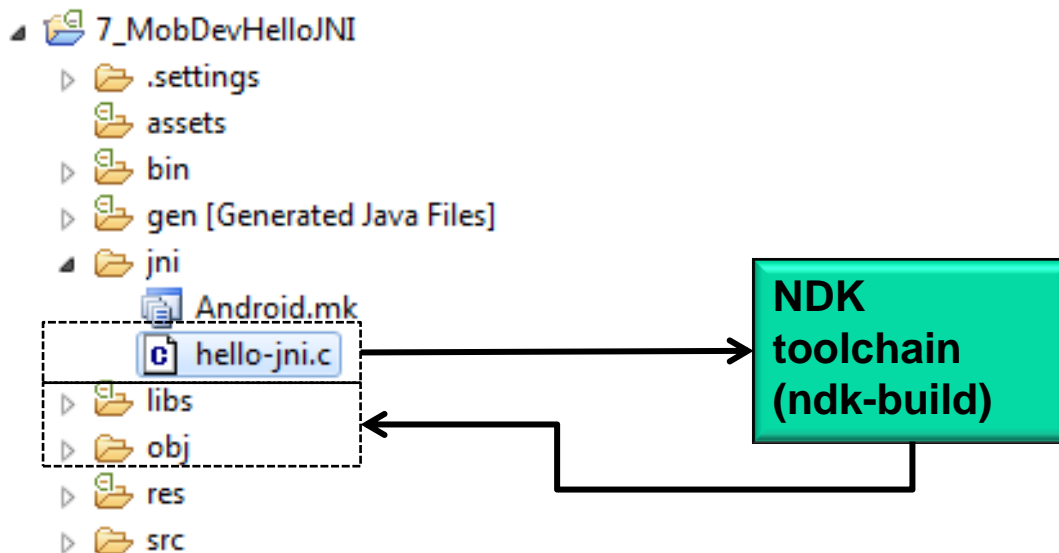
- a. NDK integrated toolchain for building shared libraries for use in an Android application
- b. Standalone toolchain that is invoked by a custom build

6. NDK cross-compiler suite (2/3)

a. NDK integrated toolchain:

Location: <NDK-base>/toolchains/arm-linux-androideabi-4.4.3/prebuilt/windows (likewise for x86 toolchain).

The NDK integrated toolchain uses the scripts, header files and library files that are part of the NDK installation.



6. NDK cross-compiler suite (3/3)

Standalone toolchain:

The NDK standalone toolchain is useful for situations where another build system, e.g. as part of an open source package, needs to invoke a cross-compiler for building.

In the standalone toolchain, everything that is needed for building (compilers etc., header files, library files) is contained in a single location.

How to create standalone-toolchain:

1. Start bash shell (on Windows start cygwin shell as administrator)

2. Run the make standalone toolchain command:

```
/cygdrive/c/install/Android-NDK/android-ndk-r7b/build/tools/make-  
standalone-toolchain.sh --platform=android-9 --install-  
dir=/cygdrive/c/temp/android-standalone-toolchain/
```

How to invoke the standalone-toolchain:

```
SET PATH=c:\temp\android-standalone-toolchain;%PATH%  
SET CC=arm-linux-androideabi-gcc.exe  
%CC% -o foo.o -c foo.c
```

7. Android EABI

What is an ABI?

ABI (Application Binary Interface) defines how an application interacts with the underlying system at run-time.

An ABI is a low-level interface definition that comprises the following:

- CPU instruction set to use
- Endianness of memory load and store operations
- Format of executable binaries (programs, libraries)
- Function call conventions (stack framing when functions are called, argument passing)
- Alignment of structs and struct fields, enums

The goal of an ABI is binary compatibility between executables (e.g. program calling a library function).

An EABI (Embedded ABI) defines an ABI for embedded targets.

Android EABI:

Android EABI is basically identical to the Linux (GNU) EABI with the difference of the C-library (bionic C-library instead of GNU C-library).

Android provides 3 EABIs:

- a. armeabi (ARMv5TE instruction set, thumb mode)
- b. armeabi-v7a (Thumb-2 instruction set extensions, hardware floating point support)
- c. x86 (IA-32 based instruction set)

For more details see [<NDK-base>/docs/CPU-ARCH-ABIS.html](http://android.googlesource.com/NDK-base/docs/CPU-ARCH-ABIS.html)

8. NDK C++ support

NDK provides some basic C++ runtime support through the default `/system/lib/libstdc++` library.

The following C++ features are not supported:

- C++ exceptions
- RTTI (Run-Time Type Information)
- Standard C++ library

C++ runtimes:

NDK provides different libraries (run-times) with different levels of C++ support:

| C++ Runtime | Library | C++ exceptions | RTTI | Standard C++ library |
|-------------|------------|----------------|------|----------------------|
| system | libstdc++ | No | No | No |
| gabi+ | libgabi++ | No | Yes | No |
| stlport | libstlport | No | Yes | Yes |
| gnustl | libgnustl | Yes | Yes | Yes |

Application files must all be linked against the same runtime library (mixing is not possible). The C++ runtime is specified in the (optional) `Application.mk` makefile.

Static versus shared libraries:

Shared libraries are the preferred mode of library use to conserve space (library not contained multiple times in different executables) and avoid problems with global library variables.

More details see CPLUSPLUS-SUPPORT.html.

9. JNI - Calling native functions from Java code

Java code:

Declaration of native function that is contained in a library.

```
public native String  stringFromJNI();
```

Native code:

```
jstring
```

```
Java_<path to Java package>_<Java-Class>_<function-name>(JNIEnv* env,  
                                                         jobject thiz)
```

```
{
```

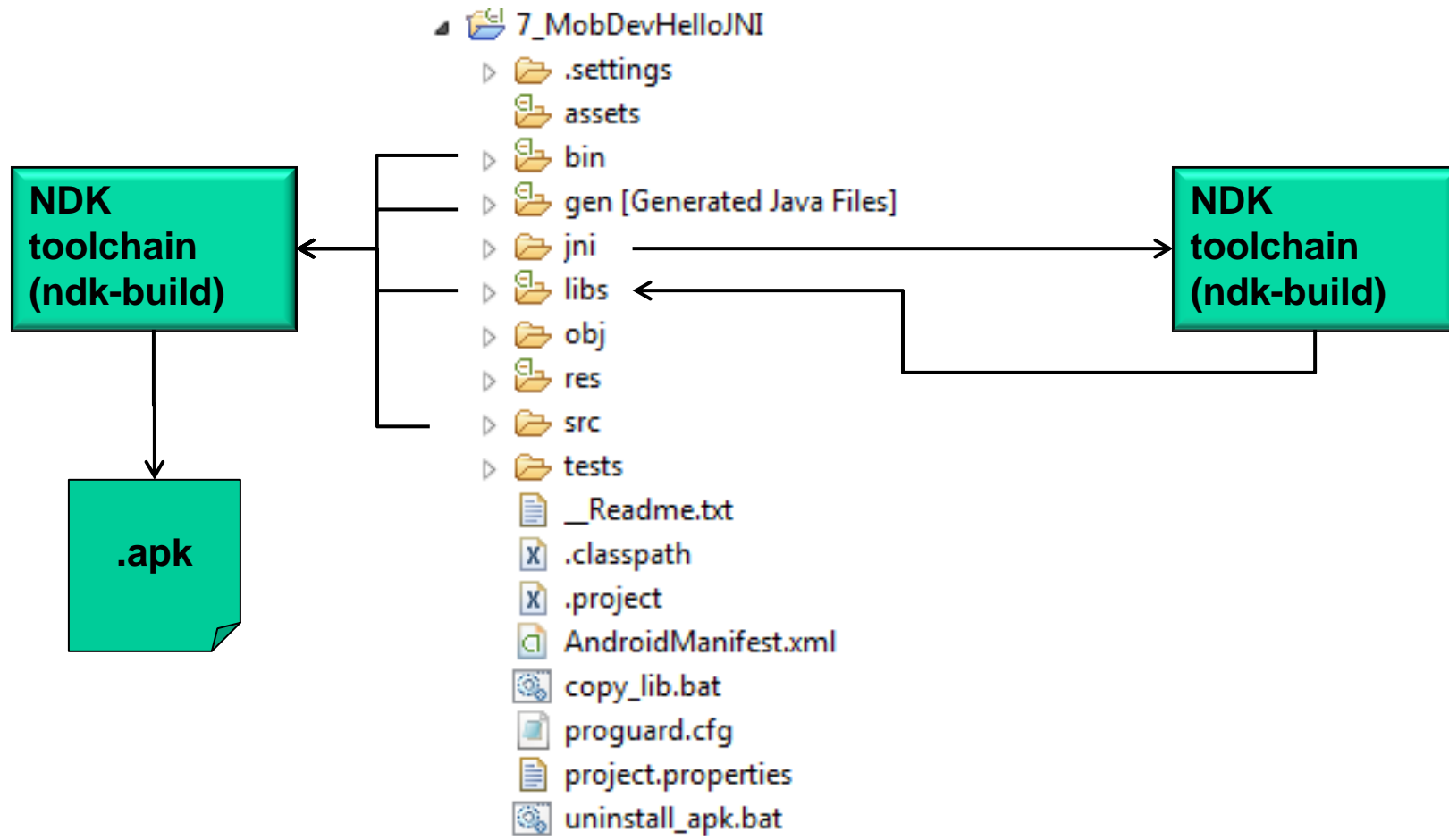
```
    ...
```

```
}
```

where **JNIEnv** identifies the JNI context of the calling VM and **jobject** is a reference to the calling Java object.

10. SDK project with native code

1. Build native sources to library with ndk-build
2. Compile Android Java sources with ADT plugin
3. Create Android application package (.apk) with ADT plugin



11. Native activity

Android provides to possibility to implement a completely **native activity**.

Possible use cases:

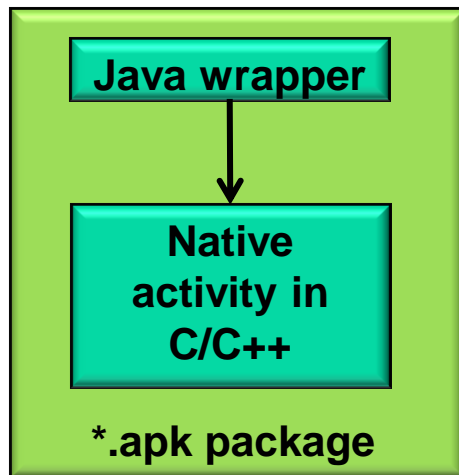
- a. Games (direct access from native code to graphics)
- b. Use of existing application code available in C++

→ Native activities are still running in the VM. Thus the lifecycle for normal Android application still applies.

→ Native activities can be started in 2 ways:

→ Small Java Wrapper starts native activity

→ Attribute HasCode=true in manifest



→ Native activity directly started

→ Attribute HasCode=false in manifest

