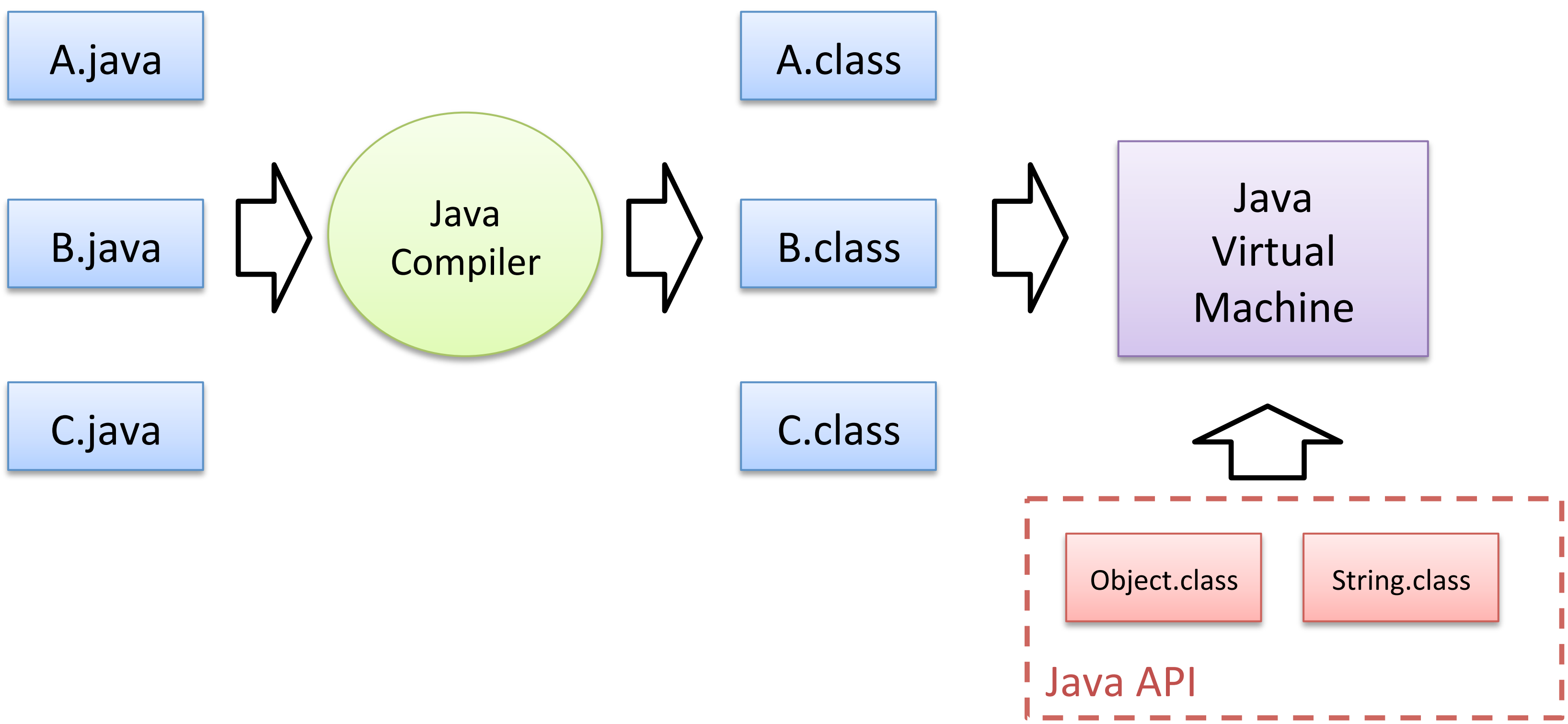
JNI
The Java Native Interface

Outline

- Java Architecture
- The Java Native Interface (JNI)
- The Native Development Kit for Android

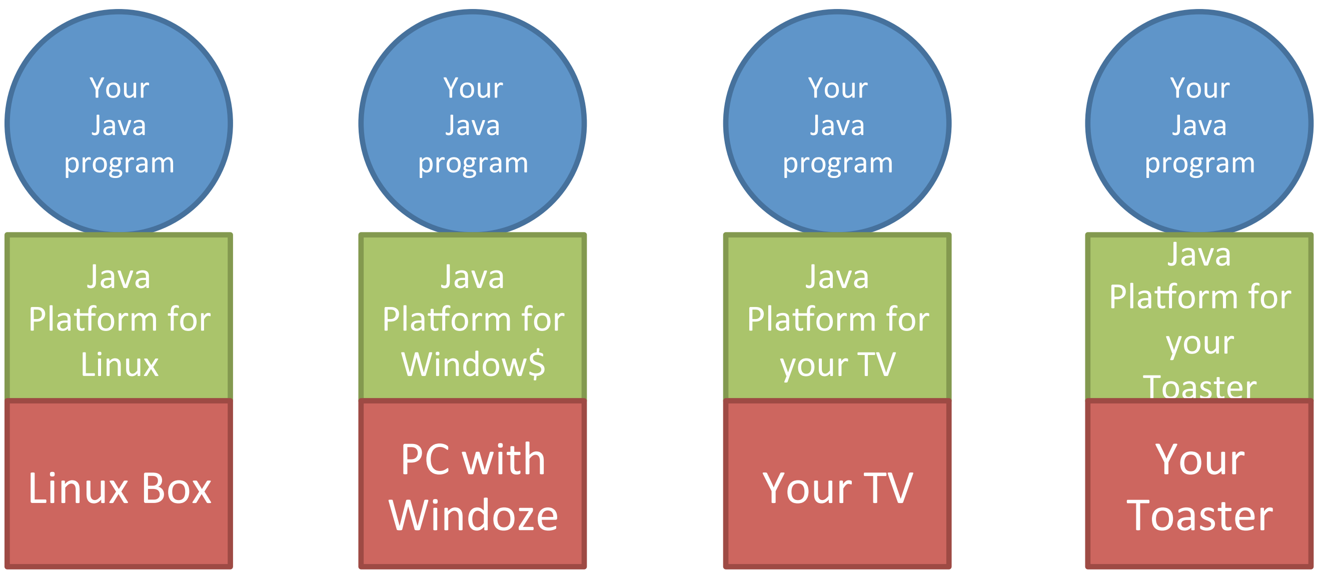
Back to the basics – Java Architecture

- Java is an interpreted language
- The code is not directly compiled into machine-language instructions
- The code is compiled into .class file that are run on the virtual machine
- Access to the system resources using the .class of the Java API



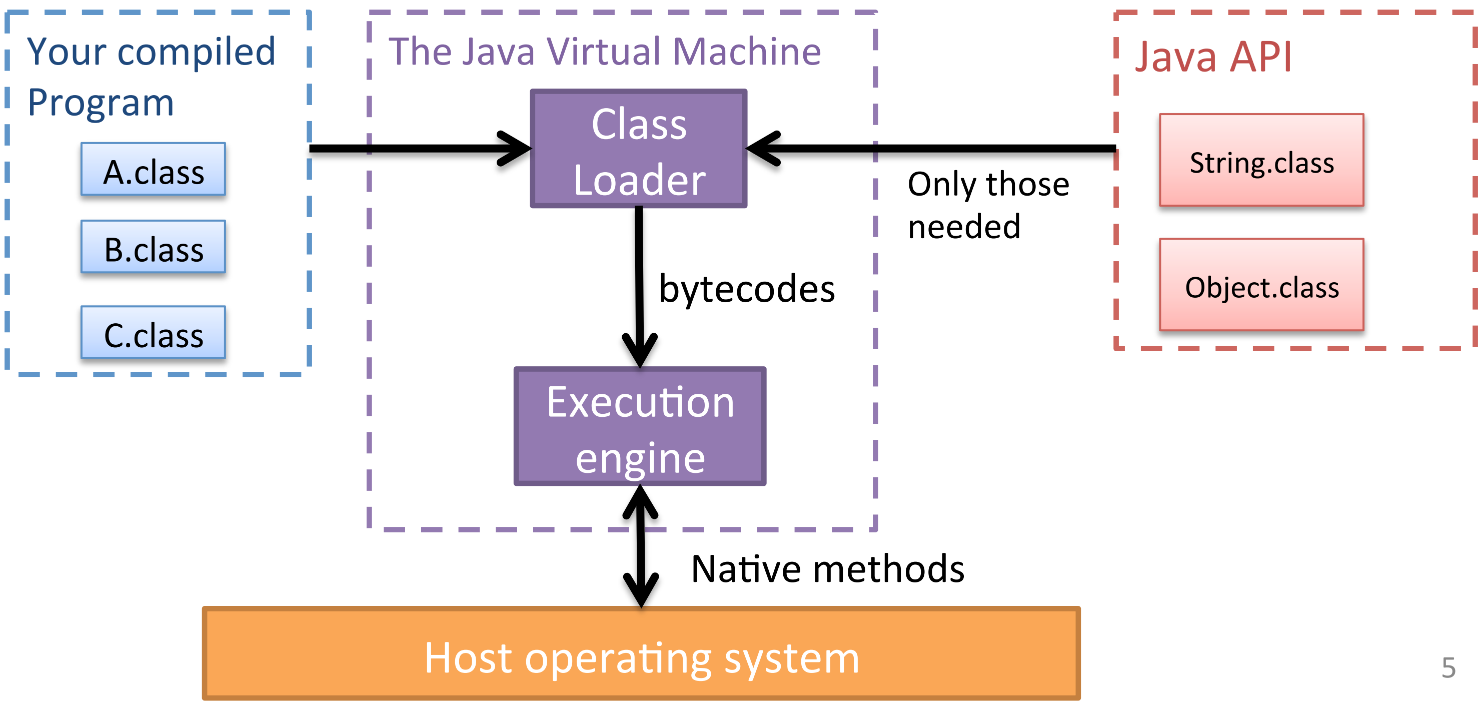
Back to the basics – Java Architecture

- Java Virtual Machine + Java API = **Java Platform** for which they are compiled
- The same java program can run on any device as long as they have an implementation of the Java platform



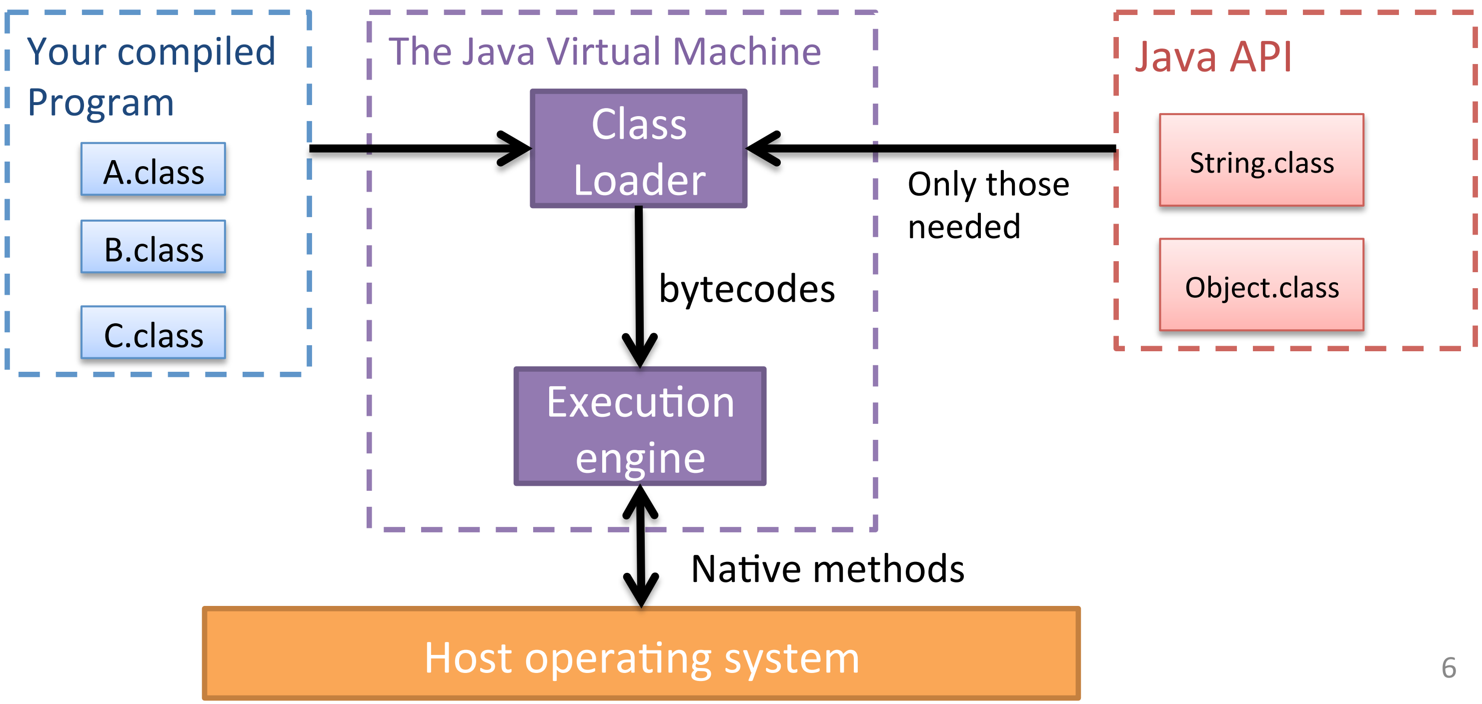
Back to the basics – Java Virtual machine

- **Abstract computer**: the specification define features that every JVM must have
- Implementation can be SW or HW
- It loads the .class files (class loader) and execute the bytecode (execution engine)



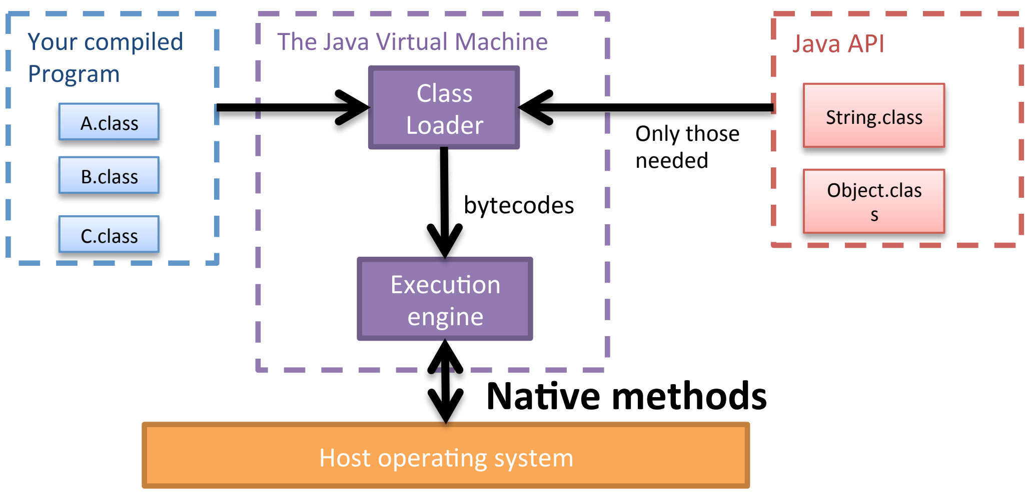
Back to the basics – Java Virtual machine

- **Class loader**: it loads the classes and their bytecodes
- **Execution engine**: it executes the bytecodes
  - Interpretation: direct translation in machine language
  - Just-in-time (JIT): translation and caching for later use
- Overheads…



Native Methods

- Java program interacts with the host by invoking *native methods*.
- Java method
  - written in the Java language, compiled to bytecodes, and stored in class files
  - platform independent
- Native method
  - written in some other language, such as C, C++, or assembly, and compiled to the native machine code of a particular processor.
  - Stored in a dynamically linked library whose exact form is platform specific.
  - JVM loads the dynamic library and invokes the method.



Native Methods

- Native methods can give direct access to the resources of the underlying operating system.
- The program then becomes platform specific
  - dynamic libraries containing the native methods are platform specific
  - Also specific to a particular implementation of the Java Platform and its native interface implementation (JNI)

2 choices:

- platform independent applications accessing system resources only through the Java API.
- platform-specific Java programs that calls native methods to exploit full potential

Java Native Interface

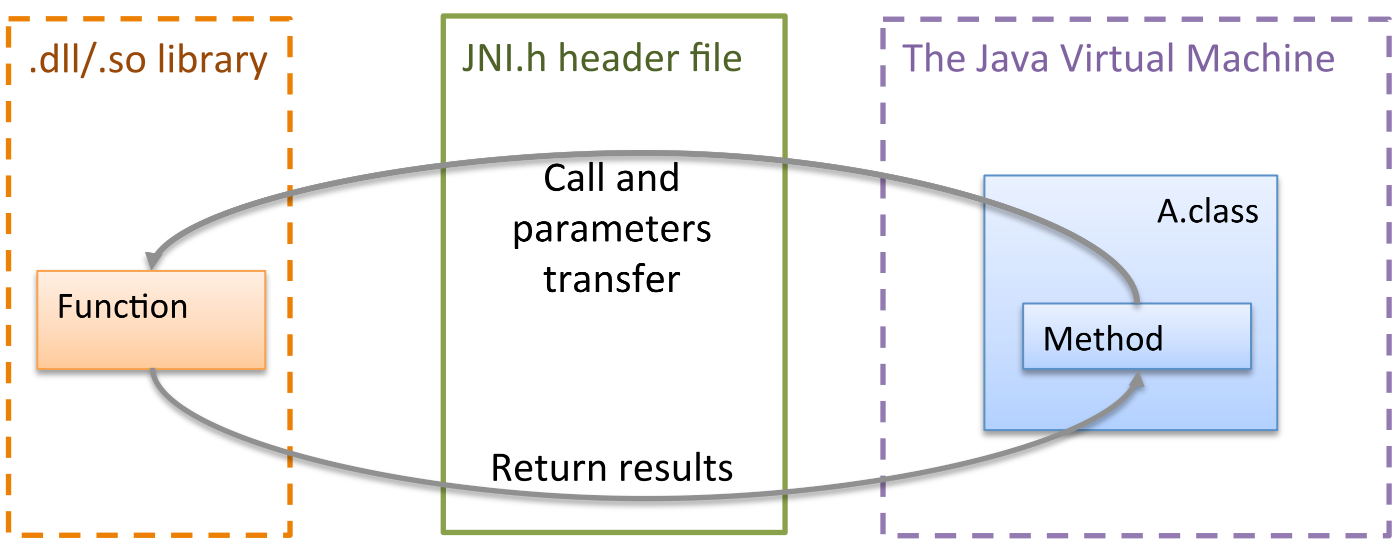
- Java's greatest advantages for cross-platform capability.
- On the other hand the local machine instructions cannot be utilized to achieve the full performance potential of the machine.
- **Java Native Interface**, a Java platform to interact with the machine on the local level.
- It can be employed to allow the use of legacy code and more interaction with the hardware for efficient performance..
  - Reuse already written libraries
  - Use machine specific features, eg GPU, special instruction sets

Implication of using JNI

- Java apps are portable
  - Runs on multiple platforms
  - The native component will not run on multiple platforms
  - Recompile the native code for the new platform
- Java is type-safe and secure
  - C/C++ are not
  - Misbehaving native code can affect the whole application
  - Security checks when invoking native code
  - Extra care when writing apps that use JNI
- Native methods in few classes
  - Clean isolation between native code and Java app

Java Native Interface

- JNI is a Java platform defined as the Java standard to interact with the code on the local platform.
- Local method stored in the form of library files.
  - Windows .dll or Unix/Linux .so file format.
- JNI is an adapter, completing mapping between Java and C/C++ types.
- **jni.h** file to complete the mapping between the two.



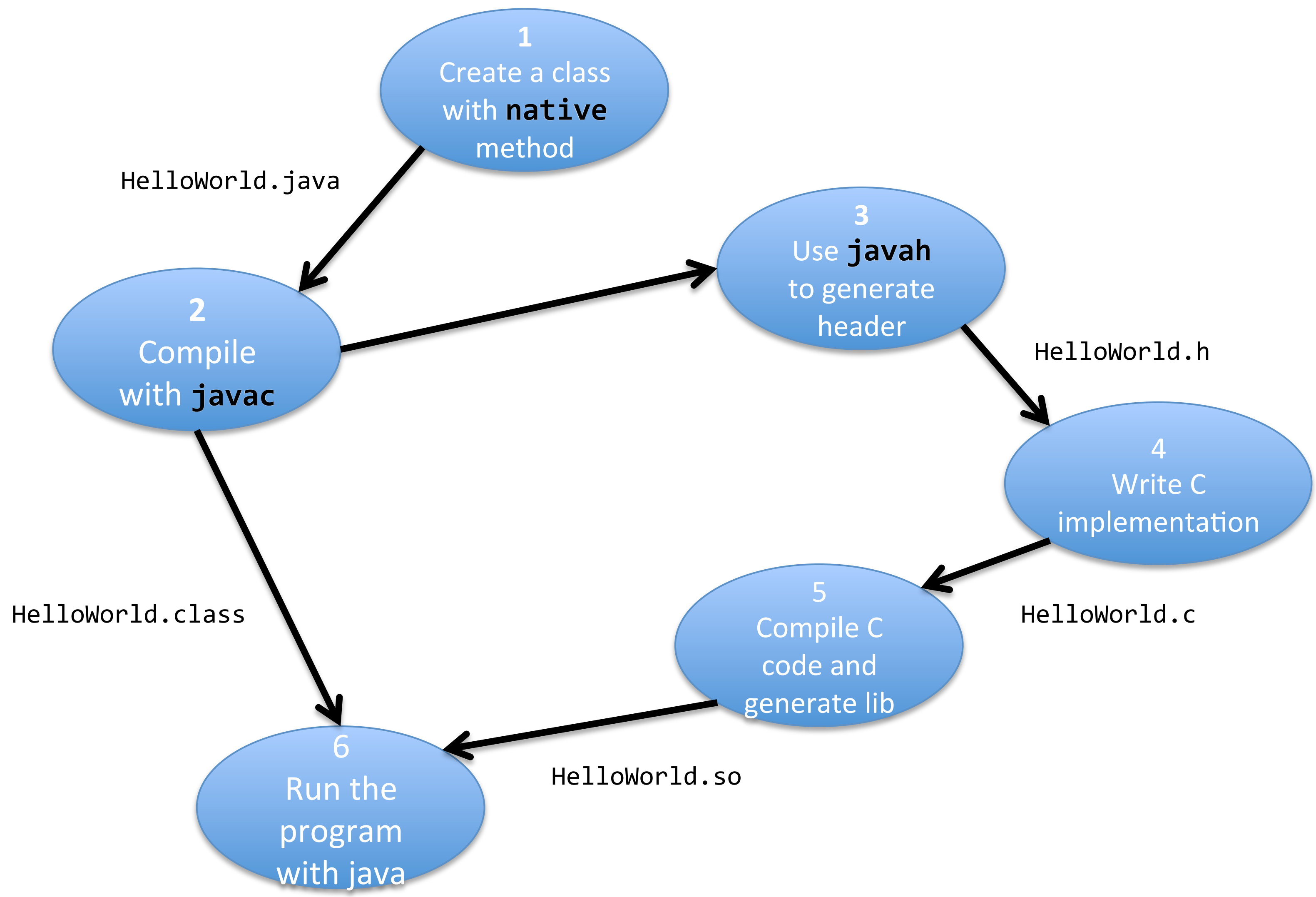
Declaring native methods in Java

```
public class MyClass
{

    // Methods as usual
    ...

    // Declare a native methods
    private native void firstJniMethod();
    private native void secondJniMethod();
    ...


    static
    {
      System.loadLibrary("myLibrary"); // Load native library at runtime
                                      // myLibrary.dll (Windows) or
                                        // myLibrary.so (Unixes)
    }
```

No java implementation

The name of the library

JNI workflow



Hello JNI

1
Create a class with **native** method

```
public class HelloJNI
{
    // Declare a java method sayHelloJava() that says hello world
    private void sayHelloJava()
    {
        System.out.println("Hello World from Java!");
    }

    // Declare a native method sayHelloJNI() that says hello world in JNI
    private native void sayHelloJNI();

    static
    {
      System.loadLibrary("helloJni"); // Load native library at runtime
    }

    public static void main(String[] args)
    {
        HelloJNI mHello = new HelloJNI();
        mHello.sayHelloJNI();  // invoke the native method
        mHello.sayHelloJava(); // invoke the Java method
    }
}
```

Compile the java part

2
Compile
with **`javac`**

- As usual:

```
javac HelloJNI.java
```

- And as usual it creates the `HelloJNI.class`

Generate the native header file

3
Use `javah`
to generate
header

- Using javah tool on the java class

```
javah -jni HelloJNI.java
```

- This generates a C header file containing the prototype of the function that has to implement HelloJNI.sayHelloJNI()

HelloJNI.h

3
Use **javah** to generate header

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloJNI */

#ifndef _Included_HelloJNI
#define _Included_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloJNI
 * Method:    sayHelloJNI
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloJNI_sayHelloJNI(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Our sayHelloJNI method in .java does not have parameters, here 2 default parameters are added. For now let's discard them...

Write the C implementation

4
Write C
implementation

- Now just write the implementation of the generated function
- Create a new C source file with the implementation following the prototype generated in HelloJNI.h

HelloJNI.c

```
#include <jni.h>
#include <stdio.h>
#include "HelloJNI.h"

JNIEXPORT void JNICALL Java_HelloJNI_sayHelloJNI(JNIEnv *e, jobject o)
{
   printf("Hello World from JNI!\n");
   return;
}
```

Write the C implementation

4
Write C
implementation

```c
#include <jni.h>
#include <stdio.h>
#include "HelloJNI.h"

JNIEXPORT void JNICALL Java_HelloJNI_sayHelloJNI(JNIEnv *e, jobject o)
{
    printf("Hello World from JNI!\n");
    return;
}
```

- `jni.h` for the JNI functions
- `stdio.h` required by printf (as usual!)
- `HelloJNI.h` the header generated by javah that includes the prototype for sayHelloJNI

Generate the native library

5
Compile C code and generate lib

- Compile the C code and generate the library
- On linux:

```
gcc -shared HelloJNI.c -I/usr/lib/jvm/java-6-openjdk/include -o libhelloJni.so
```

Build a shared library (not an executable...)

The path to the JNI interface of you java SDK. It may changes with the machine...

The name of the output library, always in the format libname.so

```
static
{
  System.loadLibrary("helloJni");
}
```

The output library of this step is the one that is loaded into the java file

Hello world!

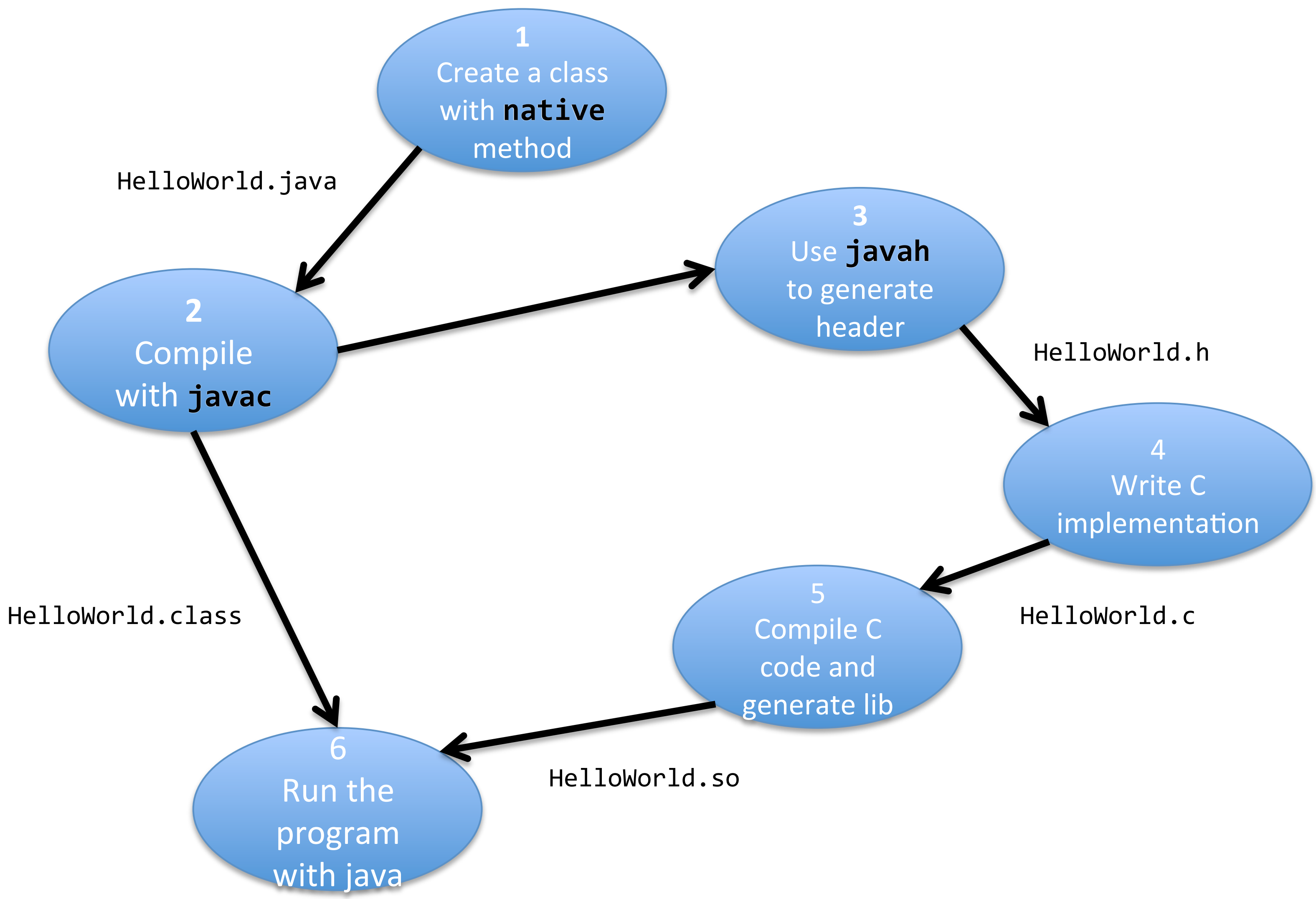
6
Run the program with java

- Now we can finally run the java program

```
java HelloJNI
```

```
> java HelloJNI
Hello World from JNI!
Hello World from Java!
```

JNI workflow

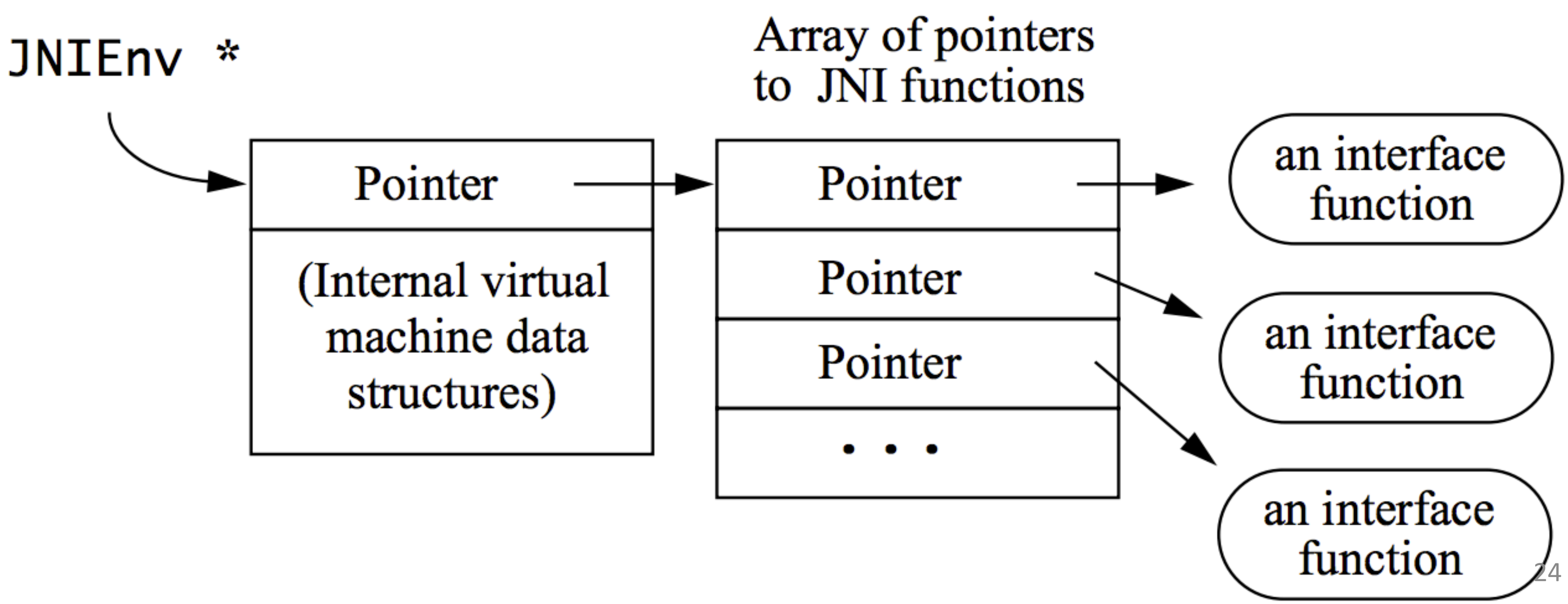


More on the prototype

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloJNI */

#ifndef _Included_HelloJNI
#define _Included_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloJNI
 * Method:    sayHelloJNI
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloJNI_sayHelloJNI(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

JNIEnv interface pointer

- Passed into each native method call as the first argument
- Valid only in the current thread (cannot be used by other threads)
- Points to a location that contains a pointer to a function table
- Each entry in the table points to a JNI function
- Native methods access data structures in the Java VM through JNI functions



More on the prototype

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloJNI */

#ifndef _Included_HelloJNI
#define _Included_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloJNI
 * Method:    sayHelloJNI
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloJNI_sayHelloJNI(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Second Argument

- Depends on the type of method

Instance methods

- can be called only on a class instance
  - i.e. `object1.method();`
- In a native method it's a reference to the object on which the method is invoked (`this` in C++)
  - e.g. `jobject thisObject`

public native void sayHelloJNI()

```
Java_HelloJNI_sayHelloJNI(JNIEnv *, jobject);
```

Static methods

- can be called directly from a static context
  - i.e. `Class.method();`
- In a native method it's a reference reference to the class in which the method is defined
  - e.g. `jclass thisClass`

public **static** native void sayHelloJNI()

```
Java_HelloJNI_sayHelloJNI(JNIEnv *, jclass);
```

Mapping of types

- JNI defines a set of C/C++ types corresponding to Java types
- Java types
  - Primitive types: int, float, char
  - Reference types: classes, instances (objects), arrays
- The two types are treated differently by JNI
- int -> jint (32 bit integer)
- float -> jfloat (32 bit floating point number)

Mapping of types – Primitive Types

Java Type	JNI Type	C/C++ Type	Size
boolean	jboolean	unsigned char	unsigned 8 bits
byte	jbyte	char	signed 8 bits
char	jchar	unsigned short	unsigned 16 bits
short	jshort	short	signed 16 bits
int	jint	int	signed 32 bits
long	jlong	long long	signed 64 bits
float	jfloat	float	32 bits
double	jdouble	double	64 bits

Reference Types – Objects

- Objects are passed as opaque references
  - i.e. C pointers to `struct` whose implementation is hidden to the programmer
- C pointer to internal data structures in the Java VM
- Objects accessed using JNI functions (`JNIEnv` interface pointer)
- e.g. `GetStringUTFChars()` function for accessing the contents of a string

Reference Types – Objects

Java Type	Native Type
java.lang.Class	jclass
java.lang. String	jstring
java.lang.Throwable	jthrowable
other objects	jobject
java.lang.Object[]	jobjectArray
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
short[]	jshortArray
int[]	jintArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
other arrays	jarray

String Types

- String is a reference type in JNI (jstring)
- Cannot be used directly as native C strings
  - Need to convert the Java string references into C strings and back
  - No function to modify the contents of a Java string (immutable objects)
- JNI supports UTF-8 and UTF-16/Unicode encoded strings
  - UTF-8 compatible with 7-bit ASCII
  - UTF-8 strings terminated with '\0' char
  - UTF-16/Unicode - 16 bits, not zero-terminated
  - Two sets of functions
  - Jstring is represented in Unicode in the VM

Example

```java
public class HelloJNI
{
    // Declare a native method sayHelloJNI()
    private native void sayHelloJNI(String text);
...
```

```c
#include <jni.h>
#include <stdio.h>
#include "HelloJNI.h"

JNIEXPORT void JNICALL Java_HelloJNI_sayHelloJNI(JNIEnv *e, jobject o, jstring
text)
{
    const jbyte *str;   // jbyte corresponds to char
    str = (*env)->GetStringUTFChars(env, prompt, NULL);
    if(str != NULL)
        printf(%s");
    return;
}
```

Another One – Babylonian method

- Square root finding iterative algorithm (Newton method)
- Given a number S find its square root
- Solve for

$$f(x) = x^2 - S = 0$$

- Chose an arbitrary x_0, then iteratively:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - S}{2x_n}$$

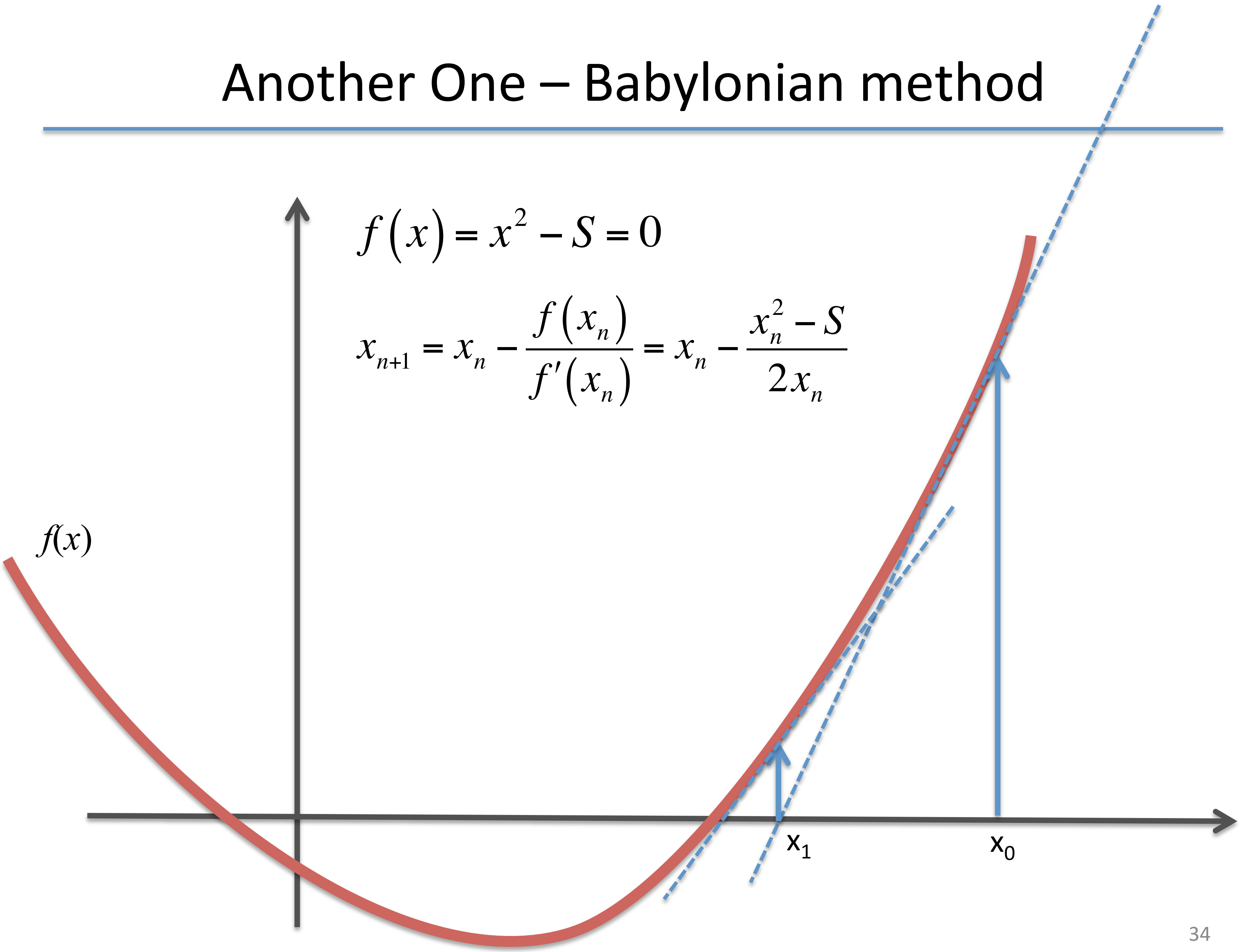
- until convergence

$$\left| x_{n+1} - x_n \right| < \varepsilon$$

Another One – Babylonian method

$$f\left(x\right) = x^2 - S = 0$$

$$x_{n+1} = x_n - \frac{f\left(x_n\right)}{f'\left(x_n\right)} = x_n - \frac{x_n^2 - S}{2x_n}$$



Another One – Babylonian method

```java
public class SqrtDemo {

    public static final double EPSILON = 0.05d;
    // the native method
    public static native double sqrtJNI(double d, double eps);
    // the java method
    public static double sqrtJava(double d, double eps) {
        double x0 = 10.0, x1 = d, diff;
        do {
            x1 = x0 - (((x0 * x0) - d) / (x0 * 2));
            diff = x1 - x0;
            x0 = x1;
        } while (Math.abs(diff) > eps);
        return x1;
    }

    public static void main(String[] args)
    {
        SqrtDemo mDemo = new SqrtDemo();
        mDemo.sqrtJNI( 89.6, SqrtDemo.EPSILON );  // the native method
        mDemo.sqrtJava( 89.6, SqrtDemo.EPSILON ); // the Java method
    }
}
```

Another One – Babylonian method

```
#include <stdio.h>
#include <stdlib.h>

#include "foo_ndkdemo_SqrtDemo.h"

JNIEXPORT jdouble JNICALL Java_foo_ndkdemo_SqrtDemo_sqrtJNI(
    JNIEnv *env, jclass clazz, jdouble d, jdouble eps) {

    jdouble x0 = 10.0, x1 = d, diff;
    do {
        x1 = x0 - (((x0 * x0) - d) / (x0 * 2));
        diff = x1 - x0;
        x0 = x1;
    } while (labs(diff) > eps);
    return x1;
}
```

Back to Android

- Android provides 2 toolkits
- Android **SDK** for developing native Java applications
  - What we have seen so far
- Android **NDK** (native development kit) for C/C++ crosscompilation
  - Optional, depending on the application
  - Download separately https://developer.android.com/tools/sdk/ndk/index.html
  - It comes with
  - A toolchain for building application (javah, gcc, ndk-build)
  - (lots of) Libraries and header files

Using the NDK

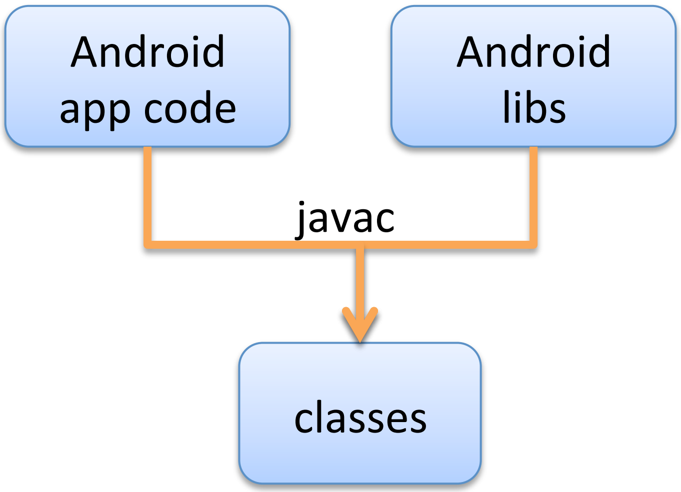
PROS

- Performance improvement
  - Accessing inner HW
- Legacy code
  - Reuse code already developed

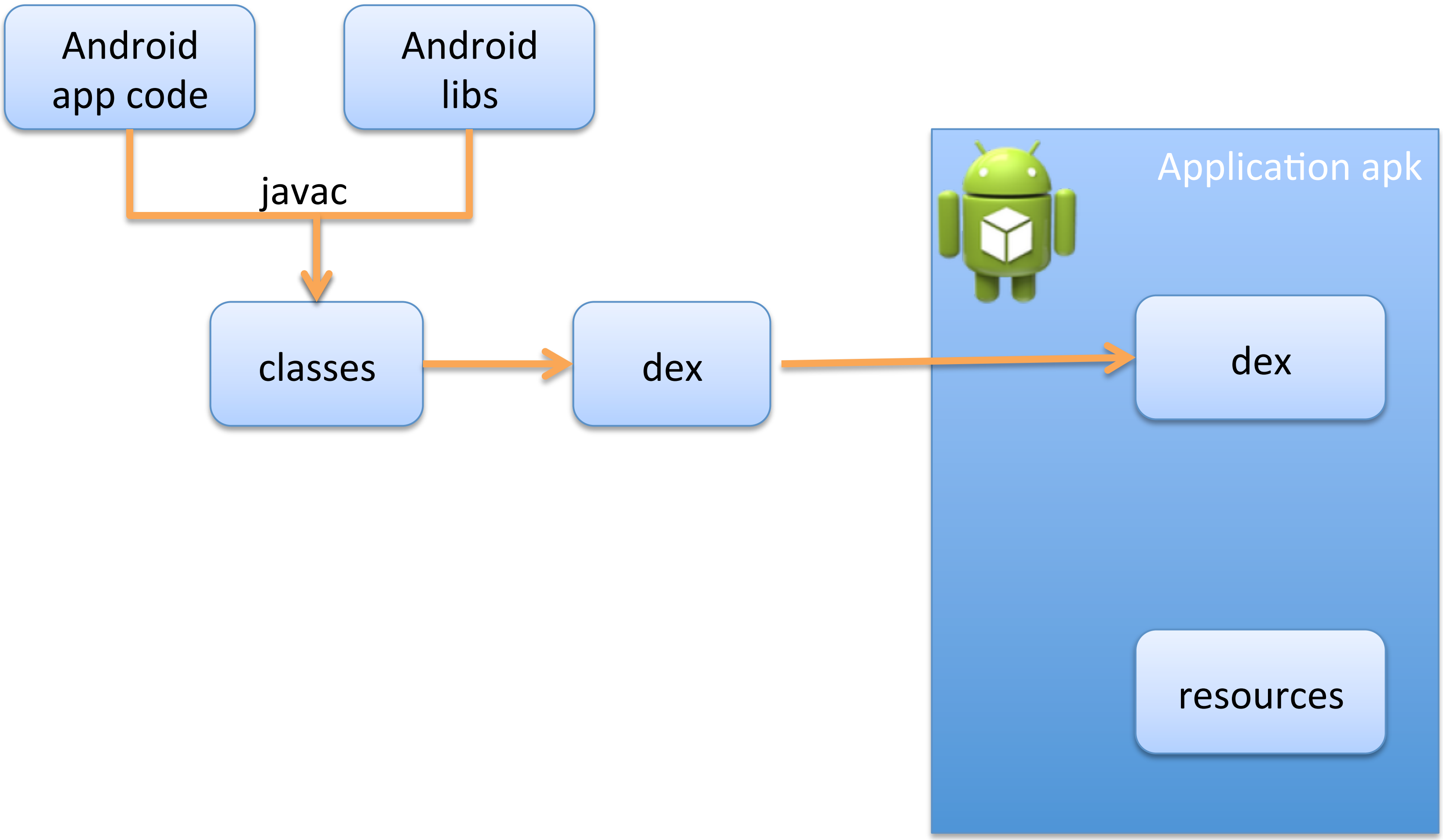
CONS

- Program complexity ++
- Compatibility not guaranteed
- Debugging difficulty ++
- Less flexibility
- Overheads when calling native function
  - Trade-off between increased performances and overhead

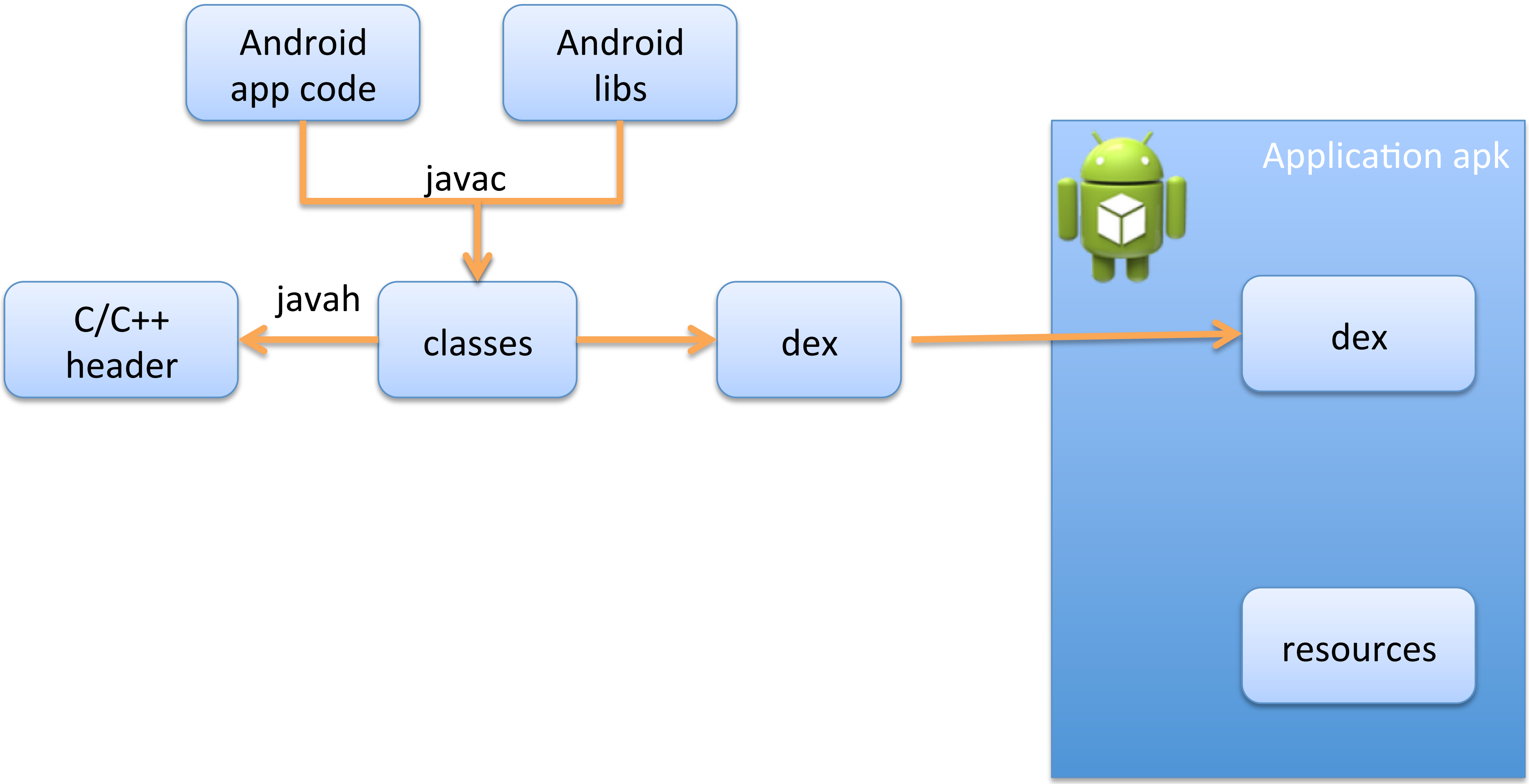
The workflow



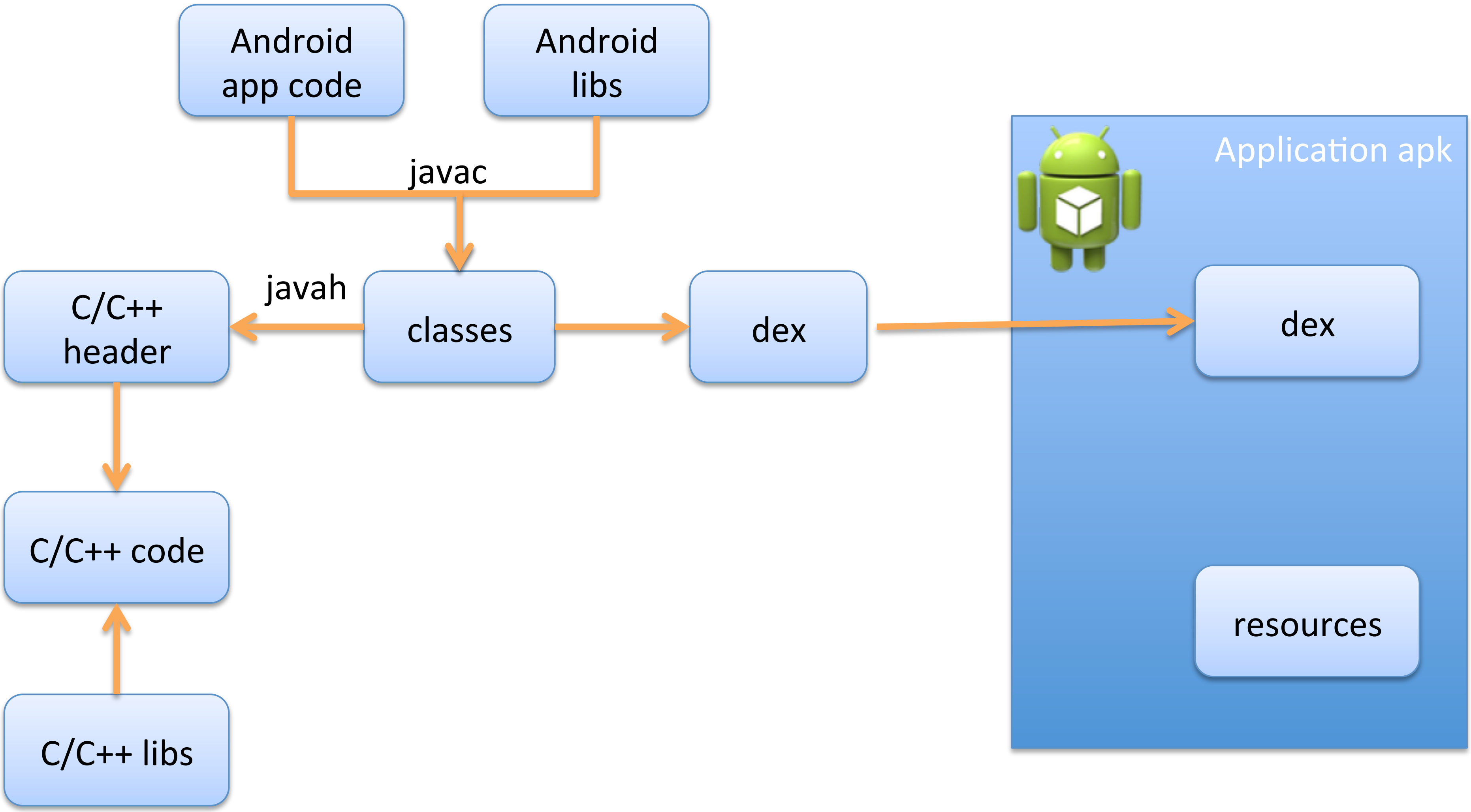
The workflow



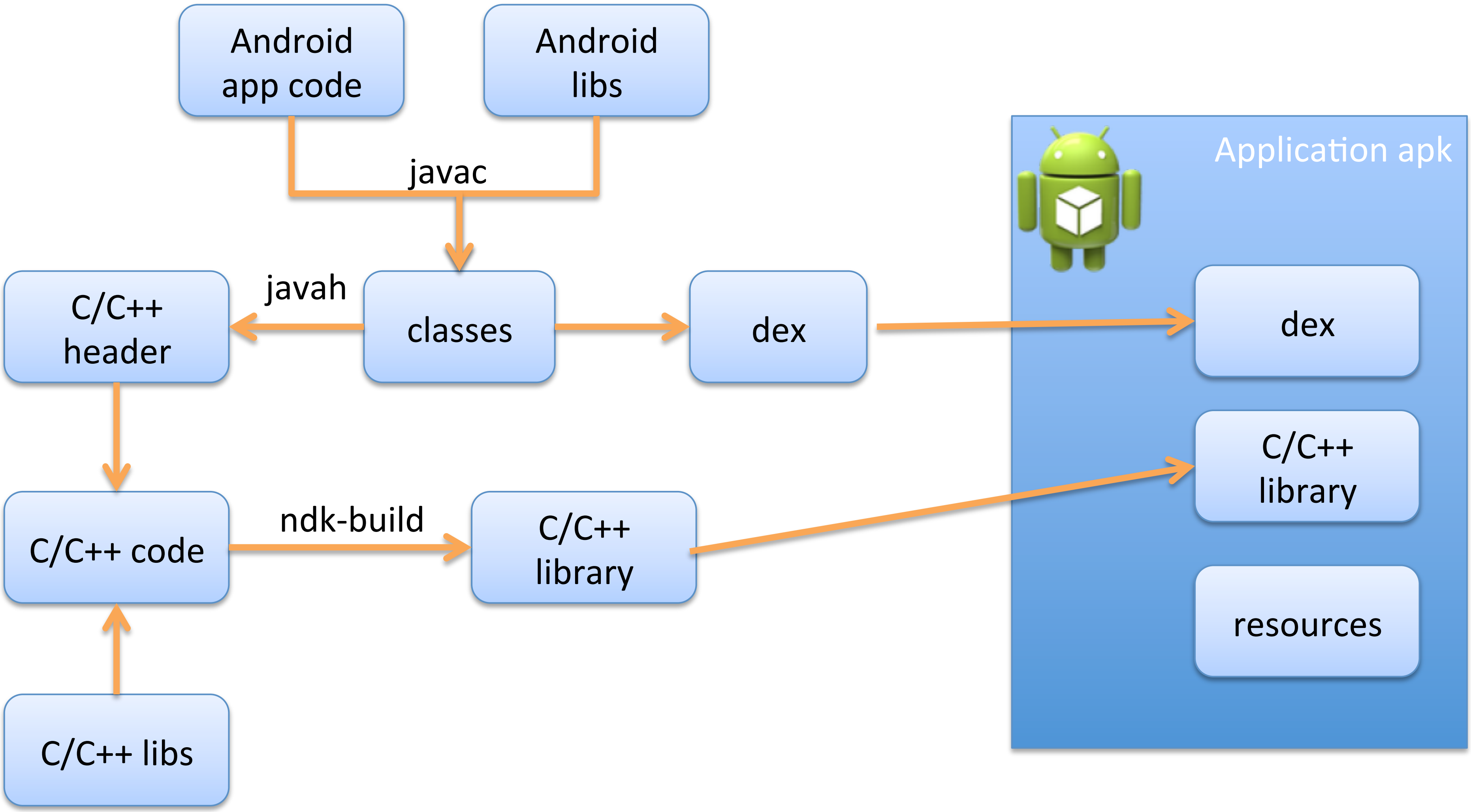
The workflow



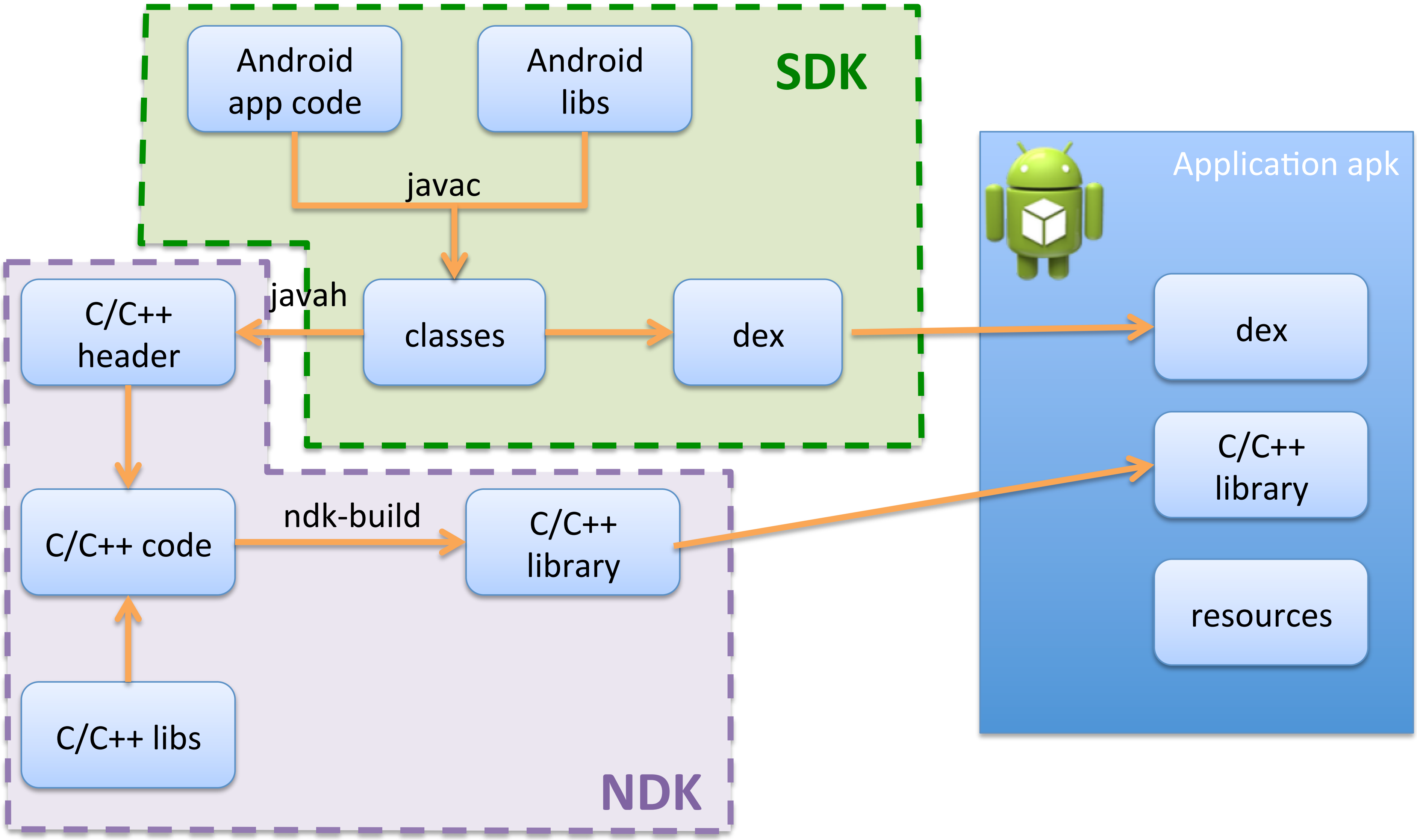
The workflow



The workflow



The workflow



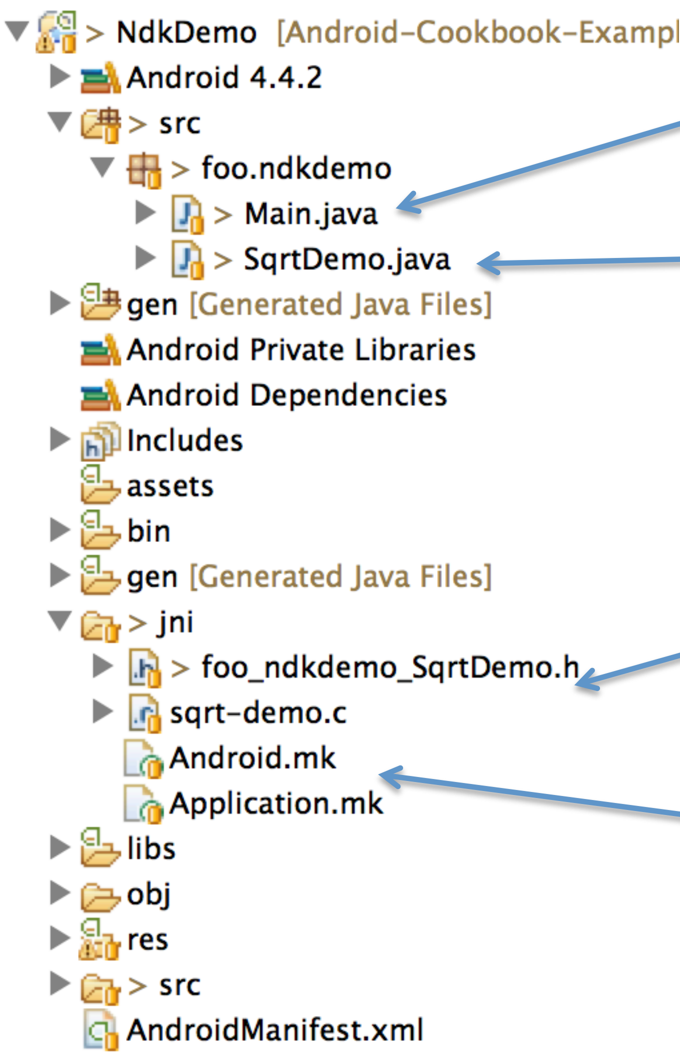
Application Binary Interface (ABI)

- The code generated by the NDK is specific to a HW platform
- ABI defines for which systems the code is compatible
  - i.e. what kind of libraries and implementation are used on the host
- ABI defines the
  - CPU instruction set to use
  - Format of executable files
  - The way the memory is accessed
  - Alignment and size for data type
- **armeabi** – machines with ARM cpu supporting ARMv5 instruction set
- **armeabi-v7a** – ARM cpus with extended instruction set, floating point hw and processor
- **X86** – usually for Intel Atom cpus

Back to sqrtDemo

- We can build an Android application around the SqrtDemo class

SqrtDemo on Android



Activity class with the UI

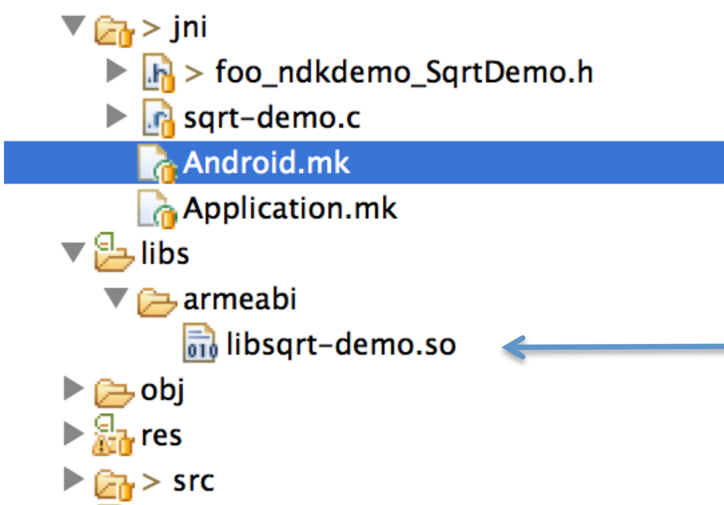
The class we already implemented

The jni part

The makefile for compiling

The makefile

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE     := sqrt-demo
LOCAL_SRC_FILES := sqrt-demo.c

include $(BUILD_SHARED_LIBRARY)
```

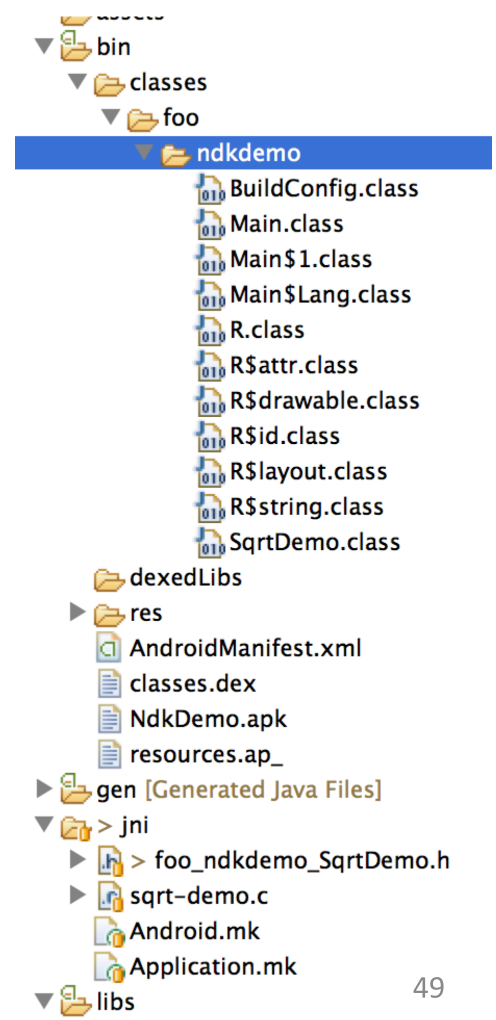


The .so library generated at compilation

Using javah in Android

```
javah -jni -classpath path the.class.name
```

- `-classpath` : the path to the .class file,
  - in this case `../bin/classes`
- the.class.name : the class including the package
  - In this case `foo.ndkdemo.SqrtDemo`



- This generates foo_ndkdemo_SqrtDemo.h
- [optional] Using –o `headerName.h` set another name for the header

References

- The Java™ Native Interface - Programmer's Guide and Specification