INTRO TO JNI

JNI

JNI

- The Java Native Interface (JNI) is a framework that enables Java code running in a Java virtual machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly.
- Making it simple, it enables the interaction between java application and the native code (C/C++)

Why JNI?

- The need to handle some hardware
- Performance improvement for a very demanding process
- An existing library that we want to reuse instead of rewriting it in Java.
- Remember this point: *To achieve this, the JDK introduces a bridge (JNI) between the bytecode running in our JVM and the native code.*
- *Now , the bridge cited above is the JNI!! Also, what is the bytecode? Can we understand that quickly?*
- *Byte Code can be defined as an intermediate code generated by the compiler after the compilation of source code(JAVA Program). This intermediate code makes Java a platform-independent language.*

Native Methods / Static / Shared Libs

- Normally, when making a native executable program, we can choose to use static or shared libs:
    - Static libs – all library binaries will be included as part of our executable during the linking process. Thus, we won't need the libs anymore, but it'll increase the size of our executable file. (.lib/.a)
    - Shared libs – the final executable only has references to the libs, not the code itself. It requires that the environment in which we run our executable has access to all the files of the libs used by our program.
- The latter is what makes sense for JNI as we can't mix bytecode and natively compiled code into the same binary file.
- Therefore, our shared lib will keep the native code separately within its .dll/.so/.dylib file (depending on which Operating System we're using) instead of being part of our classes.

JNI Components

- Java Code – our classes. They will include at least one native method.
- Native Code – the actual logic of our native methods, usually coded in C or C++.
- JNI header file – this header file for C/C++ (include/jni.h into the JDK directory) includes all definitions of JNI elements that we may use into our native programs.
- C/C++ Compiler – we can choose between GCC, Clang, Visual Studio, or any other we like as far as it's able to generate a native shared library for our platform.

JNI Elements - Java

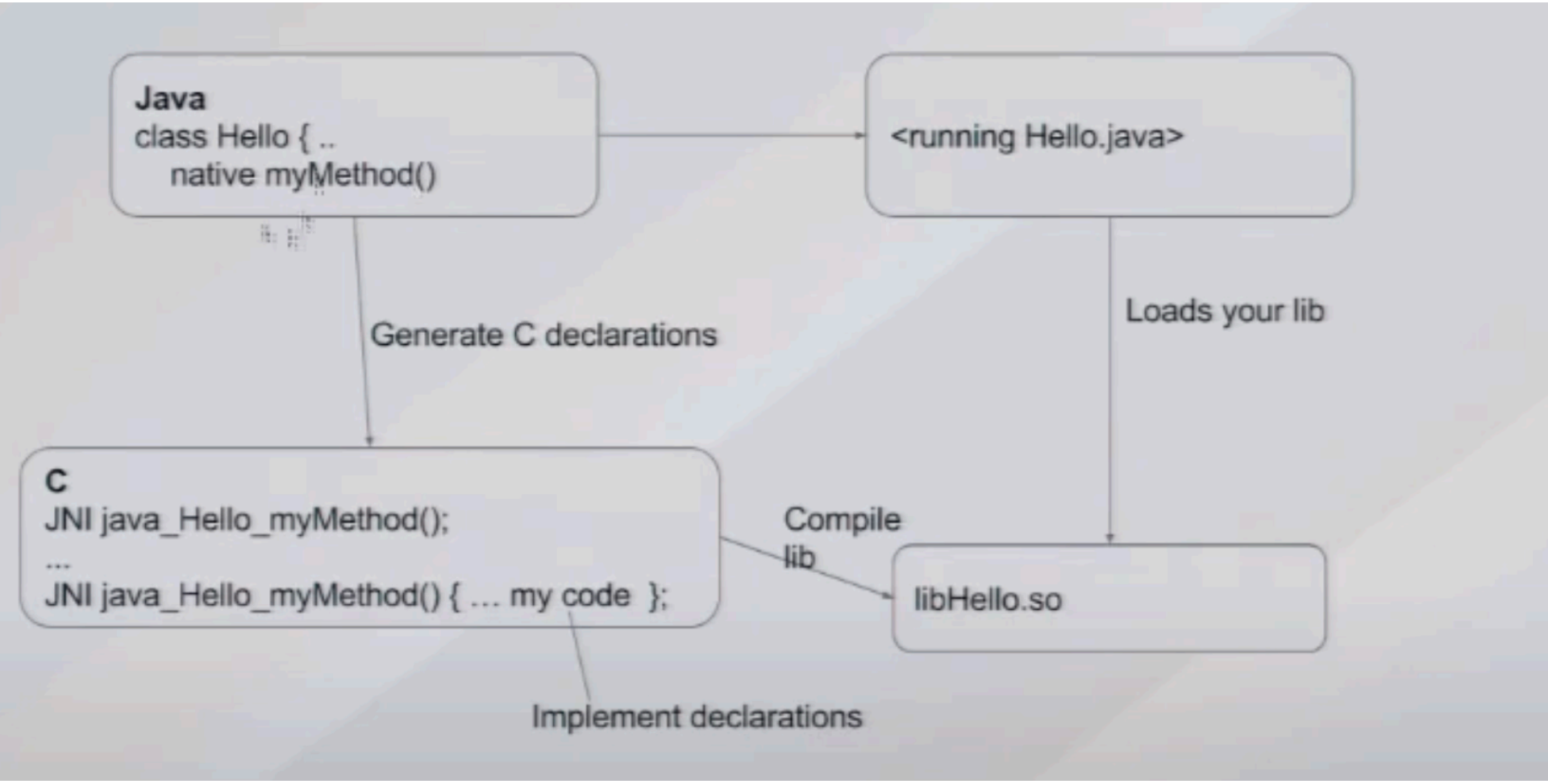
- "native" keyword in the function. (Already Explained)

- System.loadLibrary(String libname) / System.load(String AbsolutePath) – a static method that loads a shared library from the file system into memory and makes its exported functions available for our Java code.

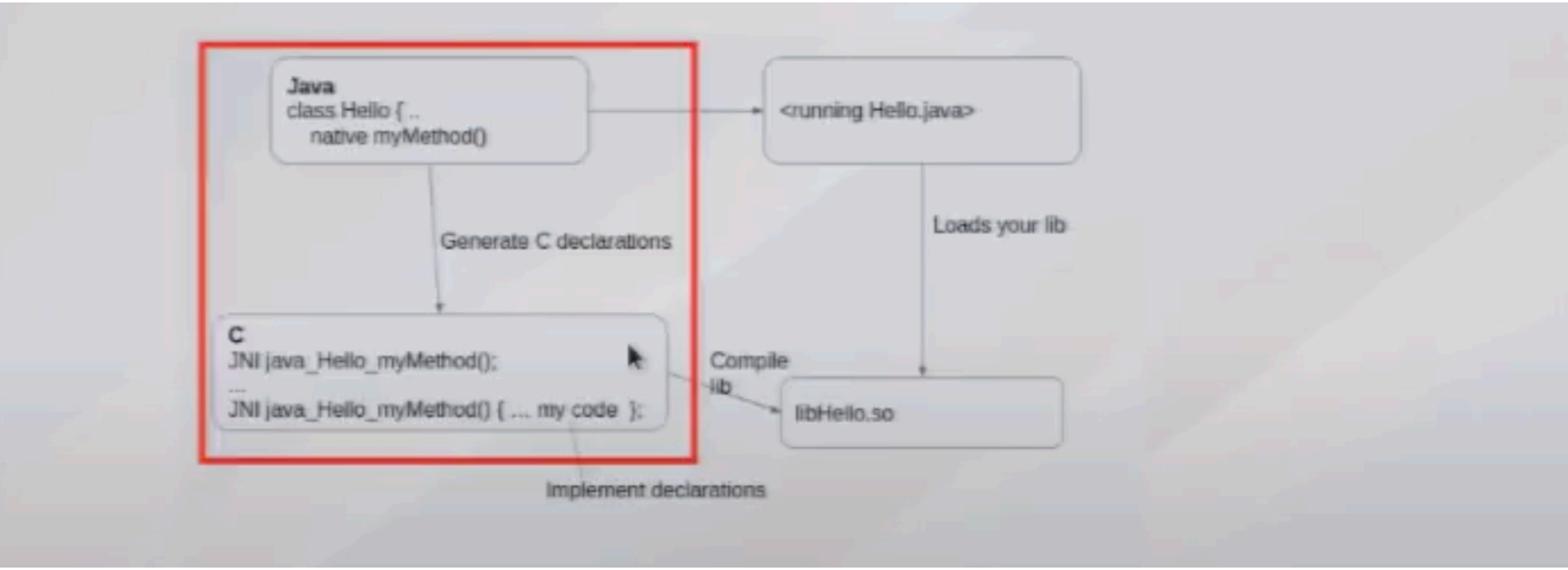
JNI Elements - C/C++

- JNIEXPORT- marks the function into the shared lib as exportable so it will be included in the function table, and thus JNI can find it
- JNICALL – combined with JNIEXPORT, it ensures that our methods are available for the JNI framework
- JNIEnv – a structure containing methods that we can use our native code to access Java elements
- JavaVM – a structure that lets us manipulate a running JVM (or even start a new one) adding threads to it, destroying it, etc…
- All of these are defined in *jni.h* header file to be included in C/C++ code.

JNI


Java
class Hello { ..
native myMethod()
<running Hello.java>
Generate C declarations
Loads your lib
C
JNI java_Hello_myMethod();
...
JNI java_Hello_myMethod() { … my code };
Compile
lib
libHello.so
Implement declarations

JNI Code Generation



JNI Code Generation

- Java methods with 'native' keyword call 'C' equivalent versions

```
private native void sayHello();
```

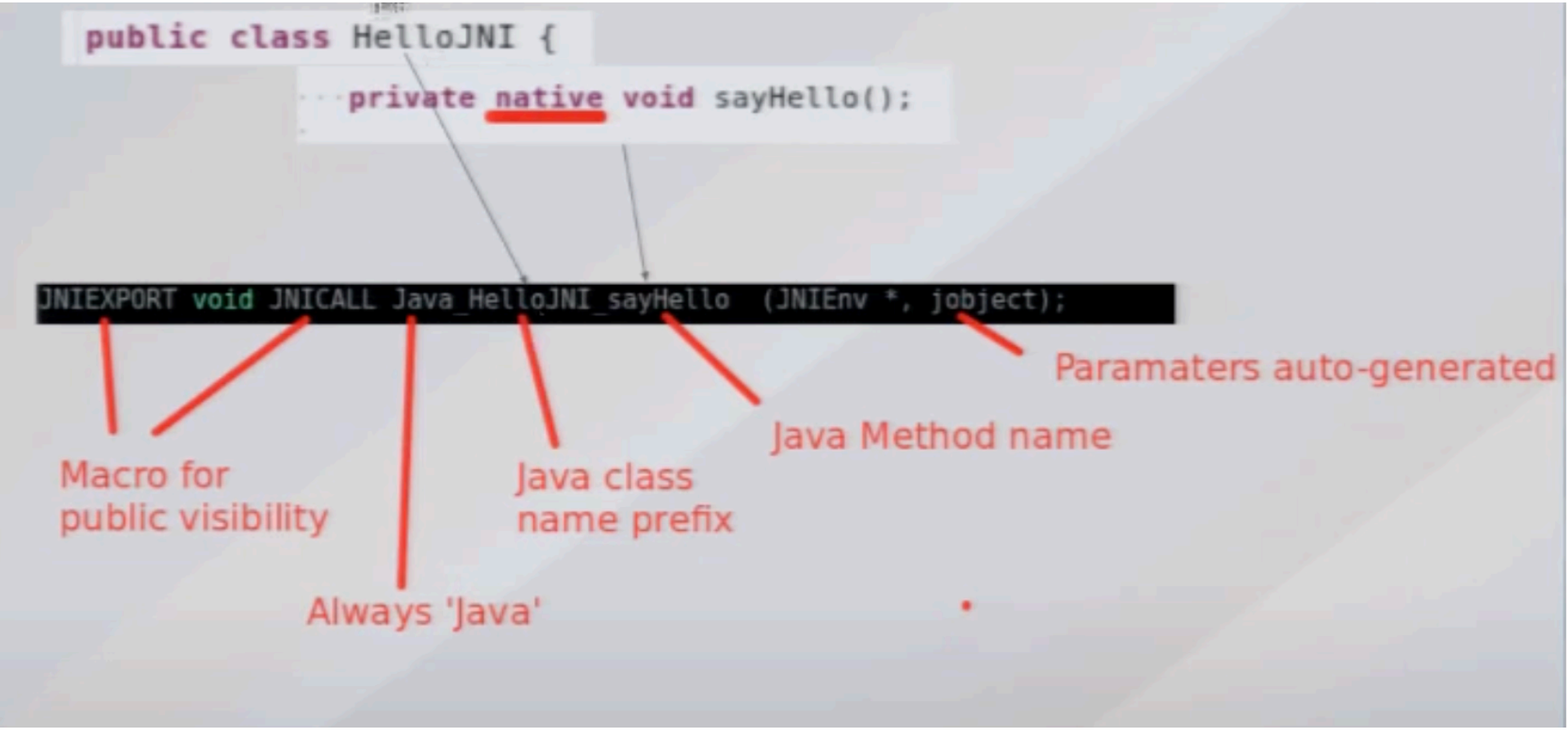
- C-Equivalent **declarations** versions are auto-generated

```
JNIEXPORT void JNICALL Java_HelloJNI_sayHello  (JNIEnv *, jobject);
```

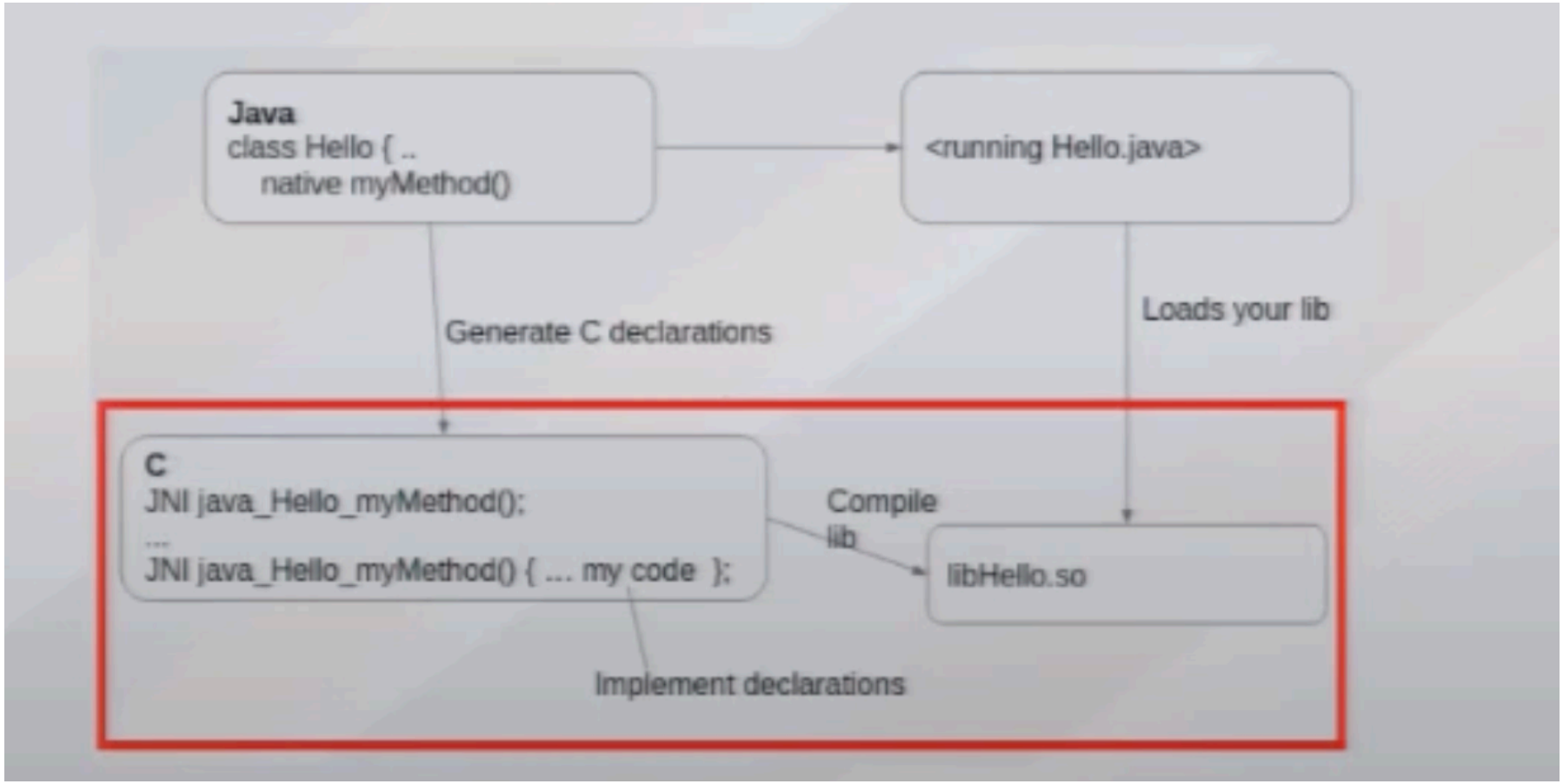
- You write the implementation of declaration:

```
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
   printf("c: Hello World!\n");
   return;
}
```

JNI Code Generation



JNI Compiling Natives



JNI Compiling Natives

Compile your JNI code as '.so' library (or .dll on windows)

- Compile your C code into a library

- gcc ... -fPIC -shared -I/… **jni.h & jni_md.h**
  - Standard 'so/dll' flags
    - Position Independent Code
    - Share with executable
  - Include 'jni.h' and 'jni_md.h' (jni & jni-linux headers)

JNI Includes

jni.h and jni_md.h

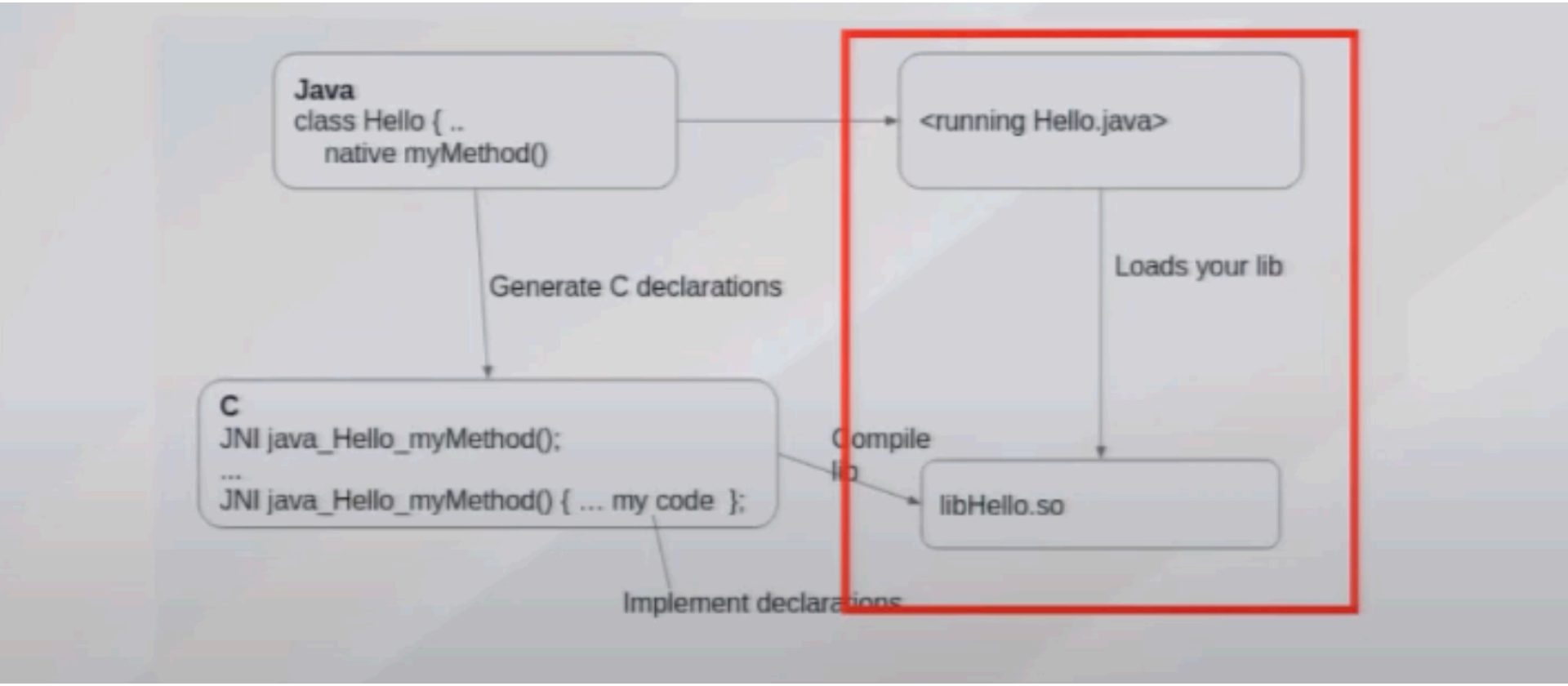
- Useful because they contain type and function definitions
  - /usr/lib/jvm/java/include/**jni.h**
  - /usr/lib/jvm/java/include/linux/**jni_md.h**
- You will be looking at them a lot
- Documentation:
  https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/functions.html

jni.h

```
54
55
56
57 typedef unsigned char   jboolean;
58 typedef unsigned short  jchar;
59 typedef short           jshort;
60 typedef float           jfloat;
61 typedef double          jdouble;
62
63 typedef jint            jsize;
64
```

```
jboolean (JNICALL *CallBooleanMethod)
  (JNIEnv *env, jobject obj, jmethodID methodID, ...);
```

JNI Load Shared Library @Runtime



JNI Load Shared Library @Runtime

Load 'lib*.so' or '*.dll" lib at runtime

- In Java, you load '.so'/'.dll' file

```
static {
    System.loadLibrary("hello"); // hello.dll (Windows) or libhello.so (Unixes
}
```

- Note,
  - On linux 'lib' is prefixed, '.so' is post-fixed automatically. So hello -> libhello.so
  - On Windows, '.dll' is post-fixed: e.g hello.dll
- Need to specify LD_LIBRARY_PATH env variable to specify where your .so is located.
  - export LD_LIBRARY_PATH=/my/bin
    java MyClass
  - Even if in the same directory

JNI Pointers

Env and jobject

```
/*
 * Example of a native call. No paramaters or return value.
 * env - is the environment that contains useful functions.
 * jobject is a pointer to the class from which we call this method.
 */
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj)
    printf("c: Hello World!\n");
    fflush(stdout);
    return;
}
```

JNIEnv JVM environment pointer

jobject is a pointer to the java object.

Ex:

(*env)->runMyMethod()

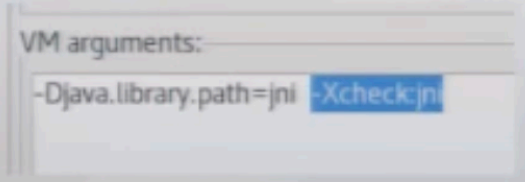
JNI Exception Handling

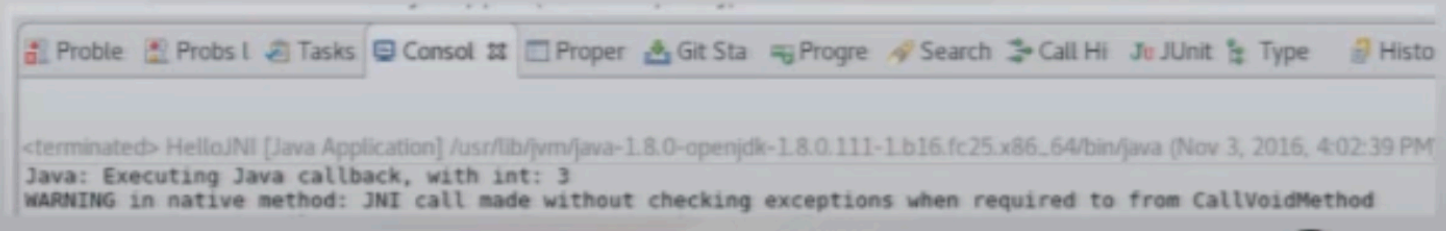
Exceptions have to be checked manually

- C doesn't have an exception checking mechanism.
- After most "*(*env)->function(..)*" calls one has to check for exceptions manually.

```
(*env)->CallVoidMethod(env,thisObj, javaMethod, 3);
if ((*env)->ExceptionCheck(env)) {
        printf("C: Exception occured when trying to call void Method :-(\n");
        return;
}
```

- Otherwise can lead to crashes due to unexpected null values or out of memory.
- VM argument to help detect errors: -Xcheck:jni
  See blog for details





JNI Exception Handling

Two JNI calls for checking for exceptions

- **Simple:**
  - **ExceptionCheck** (..)
    - Returns JNI_TRUE when exception occured. JNI_FALSE if not.
    - Useful if you intend to return the C call and let Java handle the exception.
- **Tedious:**
  - **ExceptionOccurred** (..)
    - returns pointer to exception. Null if no exceptions occurred.
    - Useful if you want to read & deal with exception itself.

JNI Calling Java from Native Code

Field and Method signatures #1

- For C to Call/Access Java :
- Need to specify method via **signature** (aka descriptor)
- J = long, B = Byte, Z=Boolean,
  See https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.3.2
- Ex method signature:
  byte myMethod(Long, Long) -> "JJ(B)"
  //note - return value 'B' is at end.
- Ex: (*env)->GetMethod..("myMethod", "JJ(J)")

JNI Calling Java from Native Code

- Signature can be found from .class files. Ex;

```
cd bin/
java -p -s HelloJNI.class
```

```
[17:30:35 bin]$ javap -p -s HelloJNI.class
Compiled from "HelloJNI.java"
public class HelloJNI {
  public HelloJNI();
    descriptor: ()V

  private native void sayHello();
    descriptor: ()V

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
```

JNI Accessing Java Methods from Native code

C can call static and instance Java methods.

Java:

```
public class HelloJNI {
  static { System.loadLibrary("hello"); // hello.dll (Windows) or libhello.so (Unixes) }
  private native void cCallingJava();
  private void callbackFromC(int cInt) {
        System.out.println("Java: Executing Java callback, with int: " + cInt);
        }
}
```

C:

```
JNIEXPORT void JNICALL Java_HelloJNI_cCallingJava (JNIEnv *env, jobject thisObj) {
  jclass javaClass = (*env)->GetObjectClass(env, thisObj);
  jmethodID javaMethod = (*env)->GetMethodID(env, javaClass, "callbackFromC","(I)V");
  if (javaMethod == NULL)     return; /* Method not found */
  /** Call back java method. For non-void, use other calls like CallIntMethod(..) etc.. */
  (*env)->CallVoidMethod(env,thisObj, javaMethod, 3);
}
```

Acquire method id

Call that method via
call that specifies return type

JNI Accessing Java Fields from Native Code

Java:

```
public class HelloJNI {
  static { System.loadLibrary("hello"); // hello.dll (Windows) or libhello.so (Unixes) }
  private String str;
  private native void accessFields(); ...
}
```

C:

```
JNIEXPORT void JNICALL Java_HelloJNI_accessFields (JNIEnv *env, jobject thisObj) {
        jclass cls = (*env)->GetObjectClass(env, thisObj); // Get ref to class
        // Find the instance field in class
        jfieldID fieldID = (*env)->GetFieldID(env, cls, "str", "Ljava/lang/String;");
        if (fieldID == NULL) return; // exception thrown
        // Can be other fields, ex: GetIntField, GetStaticObjectField  etc...
        jstring javaString = (*env)->GetObjectField(env, thisObj, fieldID);
        const char *cString = (*env)->GetStringUTFChars(env, javaString, 0);
        if (cString == NULL) return; // out of memory
        printf("C: Received string: %s\n", cString);
        fflush(stdout);
        (*env)->ReleaseStringUTFChars(env, javaString, cString);
        // Create new strinnd modify class instance variables
        jstring newJavaString = (*env)->NewStringUTF(env, "c has modified this string\n");
        (*env)->SetObjectField(env, thisObj, fieldID, newJavaString); //can be SetBooleanField(..), setStatic<TYPE>Field, etc...
}
```

Get object's class

Get "ID" of the field

Get value of field
(String being an object)

Modify field in Java

JNI Java Accessing Native Code

- Signature can be found from .class files. Ex;

cd bin/

java -p -s HelloJNI.class

```
[17:30:35 bin]$ javap -p -s HelloJNI.class
Compiled from "HelloJNI.java"
public class HelloJNI {
  public HelloJNI();
    descriptor: ()V

  private native void sayHello();
    descriptor: ()V

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
```

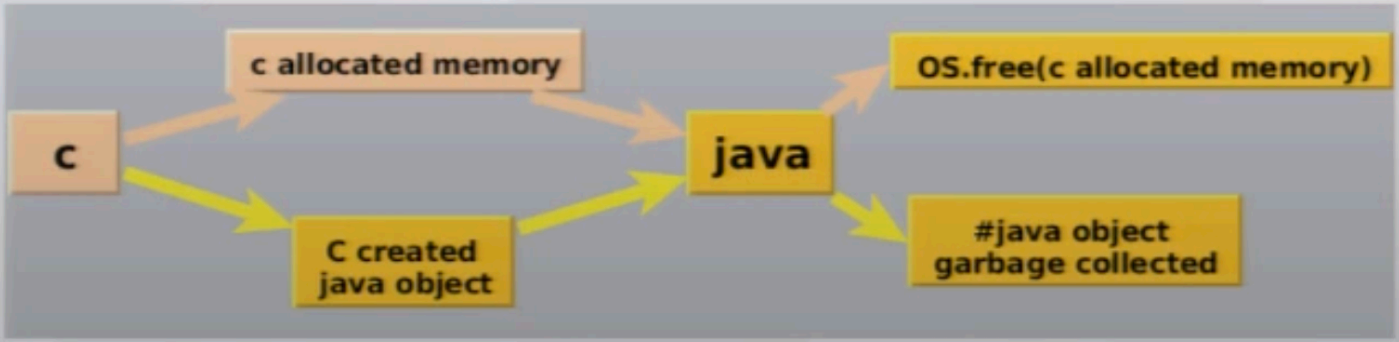
JNI Memory Management

- When you request a java object in C, you deal with one of the three references:
  - **Local references**
    - Disposed/freed after the JNI call completes
  - **Global references**
    - Never freed automatically. Can be used across JNI calls.
  - **Weak Global reference**
    - Persist between JNI calls
    - Allow Java to garbage collect something when no longer in use.
    - Example use case:
      - Cache a class that is frequently used. But a weak global reference allows Java to unload that class when it's no longer used.
    - Note, while memory used by reference is freed up, you have to delete the reference it self manually via *DeleteWeakGlobalRef()*.

JNI Memory Management

- Malloc -> free()
  (Free all the mallocs)

- New Java object -> JVM garbage collects *1
  - * With local/weak global reference.



JNI Pitfalls

- JVM loses it’s portability
- Debugging is tricky
- Data Type Mismatch
- Uses modified UTF-8 encoding. Uses only 3 bytes instead 4 bytes.
- Potential Memory Leaks / Performance Issues

Thank You!