The Java Series

The JAVA Native Interface

What is JNI?

The **JAVA Native Interface** allows Java code to operate with applications written in other languages.

- You can call functions and methods implemented in other languages.
- Your native code can manipulate Java objects (invoke methods, create objects, etc.)
- Throw and catch exceptions between native and Java code.
- Manipulate threads, etc.

The **invocation API** allows you to embed the Java Virtual Machine in your native applications.

What is JNI for?

- You may need to use native methods to:
  - Use platform dependant features from your Java programs: access specific hardware, features not supplied in the standard Java classes, etc.
  - Integrate an already existing (legacy) application into a Java application.
  - Implement time-critical portions of code in lower-level programming languages and access them from Java.
- You may need to use the invocation API:
  - To include a JVM within an application, so that it can execute Java code.
- Of course, native code is not portable!!!

What we will see

We are going to:

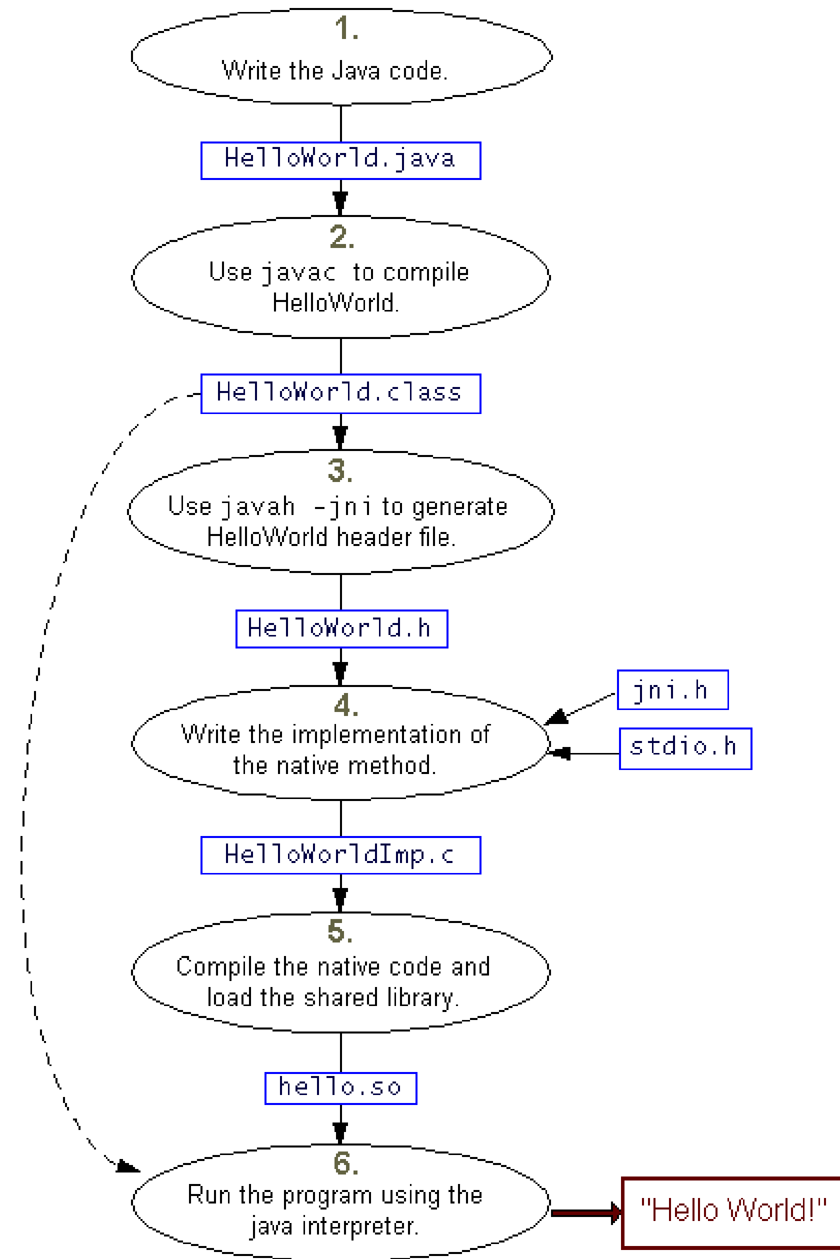
- Provide native implementations for methods in our Java classes.
- Manipulate Java objects from the native implementations: interact with the JVM
- Interchange exceptions between Java and native code
- Include a JVM in a native program and have it executing arbitrary Java code.

The MAIN idea

To provide native implementations to Java methods:

- In your Java class declare your method as **`native`**.
- Compile your Java class.
- Generate C/C++ headers using the **`javah`** utility.
- Implement (in C or C++) the functions declared in the generated headers.
- In the implementation use the variables provided in the functions. They refer to the JVM making the invocation.
- Compile the native code as a shared library.
- Load the library from your Java application.
- Use your Java class as any other class.

The MAIN idea



Scenario 1: Java Part

This method will have a native implementation.
No implementation is provided in Java

```
public class Test {
        public native void saySomeThing();
        static {
                System.loadLibrary("test");
        }
}
```

Whenever this class is loaded we load
the library containing the implementation.
Depending on the system this looks for **libtest.so**

```
public class App {
        public static void main (String[] args) {
                Test t = new Test();
                t.saySomeThing();
        }
}
```

When invoking the method we don t really
care how it is implemented. This is OO!!

Scenario 1: Native Impl

Compile and generate header files

```
[]> javac *.java
[]> javah -jni Test
```

Include jni header, and generated header

```
-- File Test.c --
#include <jni.h>
#include "Test.h"
#include <stdio.h>
JNIEXPORT void JNICALL
  Java_Test_saySomeThing(JNIEnv *env, jobject obj) {
    printf ("Just saying something\n!");
    return;
}
```

Create function as generated by `javah`
Convention: `Java_`*`ClassName`*`_`*`MethodName`*

First arg: pointer to the caller JVM
Second arg: reference to the caller object instance.
Other args: method arguments (none here)

Scenario 1: Compiling and Running

```
[]> ls -l
total 12
-rw-r--r--   1 rramos   cp              357 Apr 14 14:54 App.class
-rw-r--r--   1 rramos   cp              113 Apr 14 14:54 App.java
-rw-r--r--   1 rramos   cp              183 Apr 12 10:30 Test.c
-rw-r--r--   1 rramos   cp              390 Apr 14 14:54 Test.class
-rw-r--r--   1 rramos   cp              361 Apr 12 10:21 Test.h
-rw-r--r--   1 rramos   cp              107 Apr 14 14:54 Test.java
lrwxr-xr-x   1 rramos   cp               87 Apr 12 10:24 include ->
/afs/cern.ch/asis/packages/JAVA/jdk-1.2/i386_redhat51/usr.local/libexec/jdk/1.2/include
-rwxr-xr-x   1 rramos   cp             5061 Apr 12 10:30 libtest.so

[]> gcc -shared -I./include -I./include/linux Test.c -o libtest.so
[]> setenv LD_LIBRARY_PATH .
[]> java App
```

This lib will contain our implementation

Set up logistics for compiler to find include files and for the system to find the native libraries.

The Mechanism

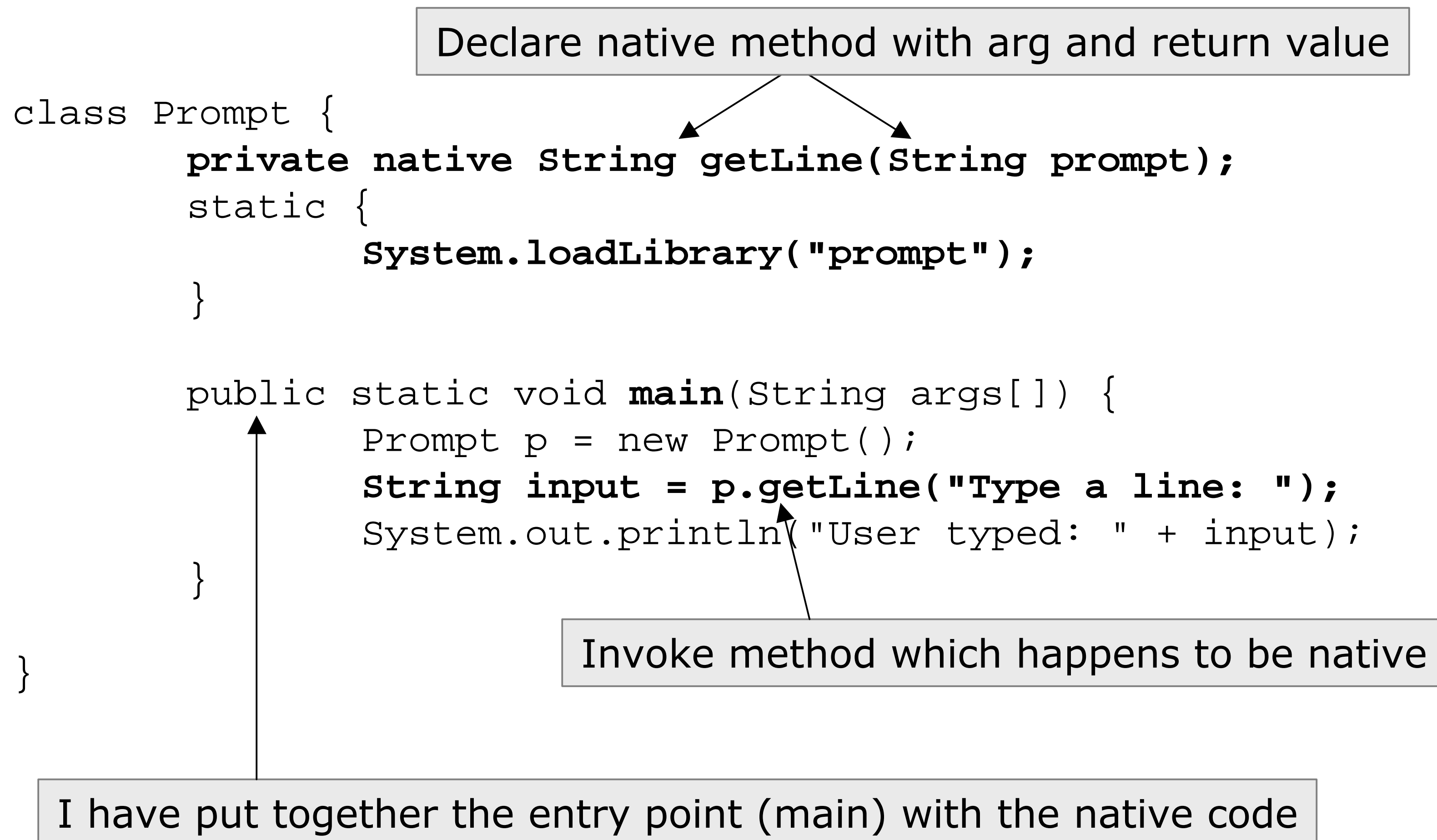
The JNI mechanism looks automatically for the functions as declared in the generated header file, following the convention.

In the implementation you have available:

- The invoker JVM. Use it to interact with the JVM: create new objects, access methods, etc.
- A reference to the instance containing the invoked native method: use it to invoke methods on that instance, etc.

Be aware of the logistics to make it happen (paths to include and lib directories, etc.)

Sce 2: Interchanging info



Sce 2: Native code in C

```
#include <stdio.h>
#include <jni.h>
#include "Prompt.h”

JNIEXPORT jstring JNICALL
  Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt){

       char buf[128];
       const char *str = (*env)->GetStringUTFChars(env, prompt, 0);
       printf("%s", str);
       (*env)->ReleaseStringUTFChars(env, prompt, str);
       scanf("%s", buf);
       return (*env)->NewStringUTF(env, buf);
}
```

Follow convention, add argument to receive string as indicated in Prompt.h

Since Java strings are different from C strings, JEnv provides us with methods to handle them

Why do we have to release it? What is the scope of the garbage collector?

JNIEnv

With JNIEnv we have access to the invoker s JVM.

We also have other functions to interact accordingly with the JVM:

- Convert arrays and strings (
- Invoke instance and class methods.
- Access instance and class variables.
- Create new instances.
- Interact with threads.
- etc

see:

http://java.sun.com/docs/books/tutorial/native1.1/summary/index.html

Scenario 3: In C++

```
#include <iostream.h>
#include <jni.h>
#include "Prompt.h”

JNIEXPORT jstring JNICALL
  Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt){
        char buf[128];
        const char *str = env->GetStringUTFChars(prompt, 0);
        printf("%s", str);
        env->ReleaseStringUTFChars(prompt, str);
        scanf("%s", buf);
        return env->NewStringUTF(buf);
}
```

The function to implement is still the same

However, we access JNIEnv in an OO way:
-> Operator
no *env argument.

Scenario 4: Interacting with JVM

Just a regular class

```
class Thing {
        private int value = 0;
        public Thing (int i) { value = i; }
        public int add (int i) { value = value + i; return value; }
}

public class Test {
        public native int calculation (Thing t);
        public static void main (String[] args) {
                Thing t = new Thing(32);
                Test  test = new Test();
                int i = test.calculation (t);
                System.out.println("In Java: Calculated result "+i);
        }
        static { System.loadLibrary("test"); }
}
```

A native method with an object argument

The instance t will be available in the native method implementation

Manipulating objects from native code

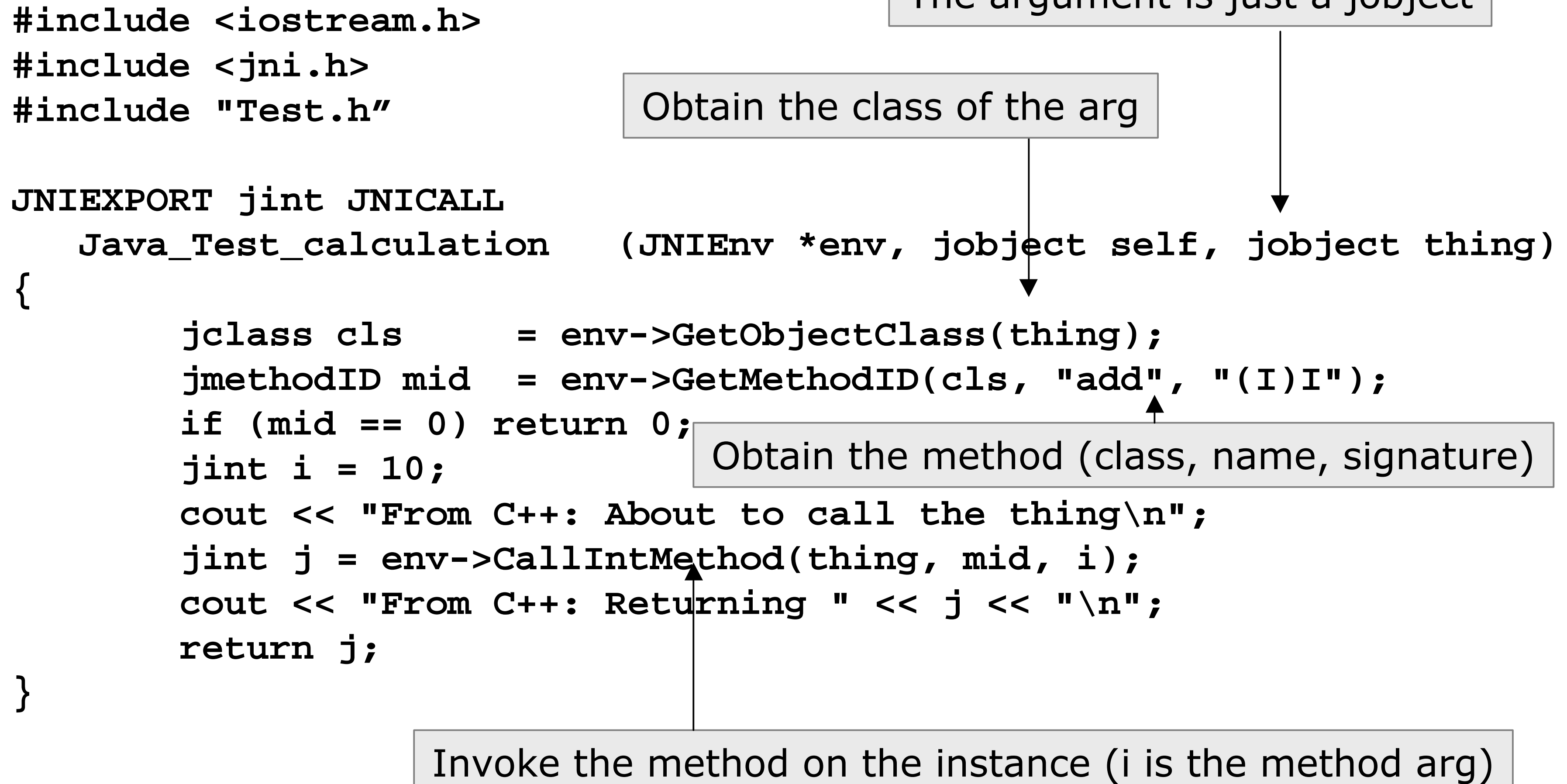
Objects in the native code are accessed through the JNIEnv reference.

In C (or C++) any object is accessed through the generic `jobject` type.

We must build dynamically the invocation to the object:

1. Ask JNIEnv to find the class.
2. Ask JNIEnv to find the method.
3. Ask JNIEnv to invoke the method.

Sce 4: The Native Code



Note that:

```
# From a class we obtain a method
jmethodID mid  = env->GetMethodID(cls, "add", "(I)I");
```

`cls` is the class obtained with env->getObjectClass(thing)

`add` is the name of the method we want to invoke

`(I)I` is the signature of the method (to distinguish between overloaded methods). Requires an int as argument and returns an int as a result.

Use javap to obtain signatures:

```
[]> javap -s -p Test
Compiled from Test.java
class Thing extends java.lang.Object {
    private int value;         /*   I   */
    public Thing(int);         /*   (I)V   */
    public int add(int);       /*   (I)I   */
}
```

javap example

```
[]> javap -s -p Test

Compiled from Test.java
public class Test extends java.lang.Object {
    static {};
        /*   ()V   */
    public Test();
        /*   ()V   */
    public native int calculation(Thing);
        /*   (LThing;)I   */
    public static void main(java.lang.String[]);
        /*   ([Ljava/lang/String;)V   */
}
```

References and Garbage Colector

You are responsible for the memory management of your native code. The rules are:

- Whatever is given to you (method arguments) is handled by the JVM s garbage collector.
- Whatever you create (C vars, Java objects through Jenv, etc.) is your responsibility: you must explicitly remove it from memory (string example in step 2)

Also, to access references to arguments (jobject, etc.) JNIEnv creates a local copy which is removed by the gc whenever the function exits => It will not be the same across function calls

To have it visible you have explicitly make it Global

Local & Global references

```
/* This code is illegal */
static jclass cls = 0;
static jfieldID fld;

JNIEXPORT void JNICALL
Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)
{
  ...
  if (cls == 0) {
    cls = (*env)->GetObjectClass(env, obj);
    if (cls == 0) {
          ... /* error */
    }
    fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
  }
  /* access the member variable using cls and fid */
  ...
}
```

Trying to cache the class and field id for future calls

But obj will be different at the next invocation

This leads to wrong results or a JVM crash

Local & Global References

```
/* This code is correct. */
static jclass cls = 0;
static jfieldID fld;
JNIEXPORT void JNICALL
Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)
{
   ...
   if (cls == 0) {
      jclass cls1 = (*env)->GetObjectClass(env, obj);
      if (cls1 == 0) {
          ... /* error */
      }
      cls = (*env)->NewGlobalRef(env, cls1);
      if (cls == 0) { ... /* error */    }
      fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
      /* access the member variable using cls and fid */
      ...
}
```

The reference now will exist until I explicitly DeleteGlobalRef somewhere else.

Sce 5: Throwing exceptions

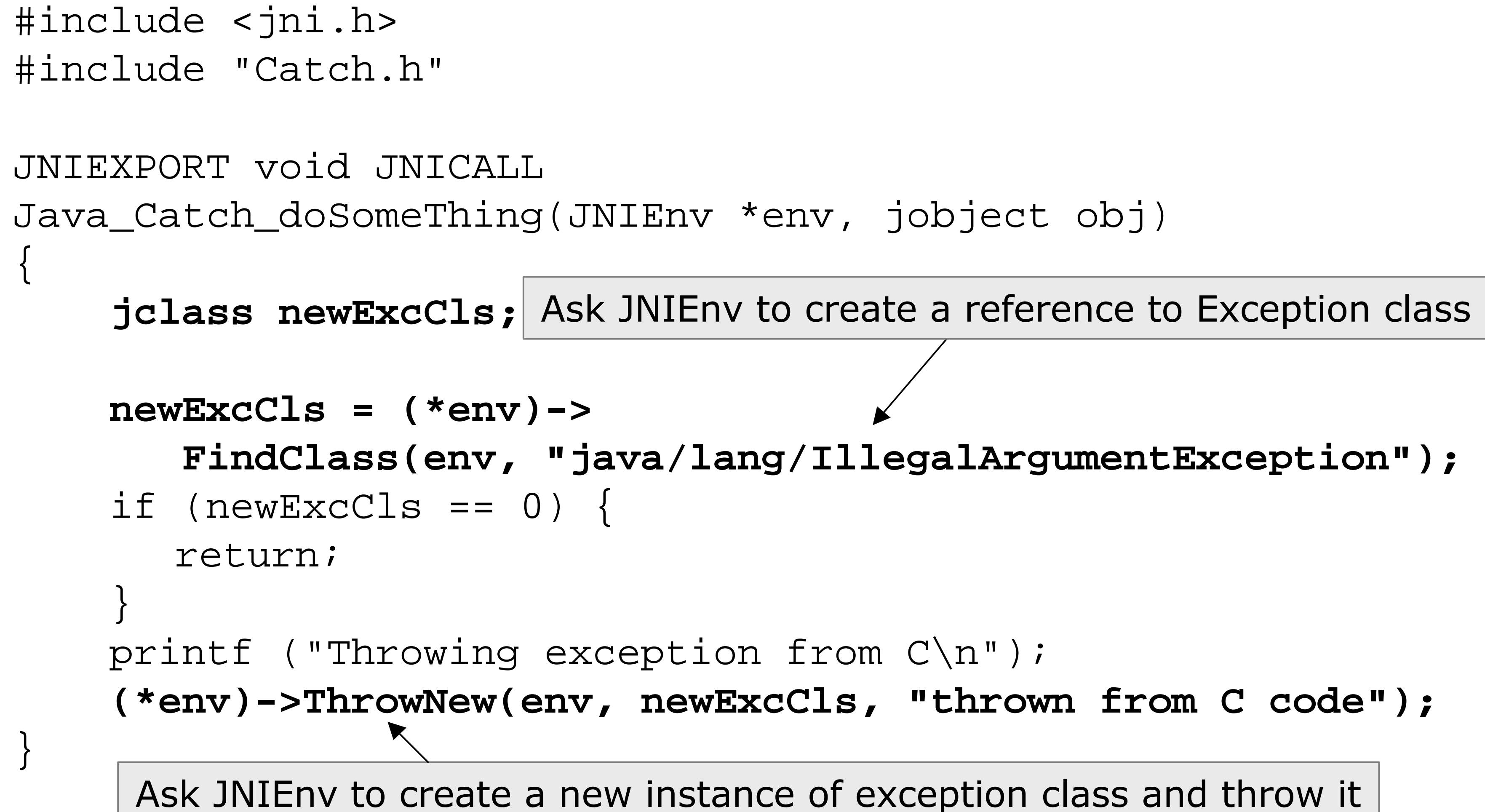
Declare a native method throwing an exception

```
public class Catch {
  private native void doSomeThing()
          throws IllegalArgumentException;

  public static void main(String args[]) {
    Catch c = new Catch();
    try {
      c.doSomeThing();
    } catch (Exception e) {
      System.out.println("Exception caught in Java:\n  " + e);
    }
  }
  static {
    System.loadLibrary("mystuff");
  }
}
```

Catch the exception whenever invoking the native method

Sce 5: Throwing exceptions



Sce 6: Catching Exceptions

A regular Java method throwing an exception

```
class Throw {
  private void callback() throws NullPointerException {
    throw new NullPointerException("thrown in Throw.callback");
  }
  private native void doSomeThing();

  public static void main(String args[]) {
    Throw c = new Throw();
    c.doSomeThing();
  }
  static {
    System.loadLibrary("mystuff");
  }
}
```

A regular native method

In the implementation of the native method we will invoke the callback and catch the Exception it generates

Sce 6: Catching Exceptions

```
#include <jni.h>
#include "Throw.h"

JNIEXPORT void JNICALL Java_Throw_doSomeThing(JNIEnv *env, jobject obj)
{
  jclass cls = (*env)->GetObjectClass(env, obj);
  jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "()V");
  jthrowable exc;
  if (mid == 0) {
    return;
  }
  (*env)->CallVoidMethod(env, obj, mid);
  exc = (*env)->ExceptionOccurred(env);
  if (exc) {
    printf ("Exception caught in C\n");
    (*env)->ExceptionDescribe(env);
    (*env)->ExceptionClear(env);
  }
}
```

Invoke a method on invoker object

Catch and deal with the exception. Note that since we are outside Java we cannot be forced to deal with the exception

Some Generalities

- There is a lot of manipulation to do from native code every time we want to interact with the invoking JVM.
- Due to the nature of native code there are some Java features which we don t have:
  - There is no type checking at compilation time when we invoke Java methods from native code.
  - Cannot force native code to catch exceptions thrown by Java methods invoked from native code.
  - Automatic memory management is limited (only through the local references mechanism)
  - Memory allocated outside the JVM is out of the scope of the garbage collector. We have to take care of it.

The invocation API

- It s an API through which we can include a JVM in our applications.
- This enables our application to run Java code
- For instance, if our application is a web browser:
  - Use the invocation API in the www browser source code
  - We include a mechanism to find classes across the network and not only in the local path.
  - Whenever we obtain an applet class from the network we create an instance and look for the init method.
- If our application is a www server:
  - Use the invocation API to execute Java code upon a CGI request -> Servlets
- ETC, ..

Sce 7: The invocation API (1)

```
#include <jni.h>
#define PATH_SEPARATOR ':'
#define USER_CLASSPATH "." /* where Prog.class is */
main() {
    JNIEnv *env;    JavaVM *jvm; JDK1_1InitArgs vm_args; jint res;
    jclass cls;    jmethodID mid; jstring jstr;  jobjectArray args;
    char classpath[1024];

    /* IMPORTANT: specify vm_args version # if JDK1.1.2 and beyond */
    vm_args.version = 0x00010001;
    JNI_GetDefaultJavaVMInitArgs(&vm_args);
    /* Append USER_CLASSPATH to the end of default system class path */
    sprintf(classpath, "%s%c%s%c%s",
            vm_args.classpath, PATH_SEPARATOR, USER_CLASSPATH,
                               PATH_SEPARATOR, "./lib/classes.zip");
    vm_args.classpath = classpath;

    res = JNI_CreateJavaVM(&jvm,&env,&vm_args);
    if (res < 0) {
        fprintf(stderr, "Can't create Java VM\n");
        exit(1);
    }
```

Initialize args for JVM

Initialize CLASSPATH (in vm_args)

Create a JAVA VIRTUAL MACHINE

Sce 7: The invocation API (2)

```
cls = (*env)->FindClass(env, "Prog");
if (cls == 0) { fprintf(stderr, "Can't find Prog class\n"); exit(1);}

mid = (*env)->GetStaticMethodID(env, cls, "main",
   "([Ljava/lang/String;)V");
if (mid == 0) {  fprintf(stderr, "Can't find Prog.main\n"); exit(1);}

jstr = (*env)->NewStringUTF(env, " from C!");
if (jstr == 0) { fprintf(stderr, "Out of memory\n"); exit(1);  }

args = (*env)->NewObjectArray(env, 1,
                     (*env)->FindClass(env, "java/lang/String"), jstr);
if (args == 0) {  fprintf(stderr, "Out of memory\n");  exit(1);  }
(*env)->CallStaticVoidMethod(env, cls, mid, args);

(*jvm)->DestroyJavaVM(jvm);
```

1. Find **Prog** class

2. Find **main** method

3. Create args for **main** method

4. Invoke main method

5. Destroy JVM

Sce 7: The invocation API

```
public class Prog {

   public static void main (String[] args) {
      System.out.println ("This is the main method");
      System.out.println ("First argument is: "+args[0]);
   }
}
```

This is just a regular Java class

The invocation API

Note that:

- **We set up** initialization args for the JVM
- When creating the JVM **we obtain** also a JNIEnv reference.
- Once we have the JNIEnv reference **we manipulate** it as in the other examples.
- But NOW our program CONTROLS the JVM.
- Before our functions where waiting to be invoked from an already existing JVM.
- **We decide** what class(es) to load. In the example the Prog class can be any regular Java class!!
- **We decide** what method(s) to invoke.

Conclusion

- JNI enables our Java programs to invoke functions in ANY native library.
- If you have an already existing native library, just make a C or C++ JNI wrapping.
- C or C++ programs can
  - Manipulate objects of a foreign JVM.
  - Interchange information with a foreign JVM.
  - Catch/Throw exceptions from/to a foreign JVM.
- We can include a JVM in out C or C++ programs.
- Have to be careful with a few aspect (type checking, exception handling, memory management, etc.) -> LEAVE MEMORY MANAGENT to the JVM as much as possible.
- It s a little bit cumbersome (finding classes, methods, building invocations, etc.) -> MAKE YOUR JNI PROGRAMS SUCCINT.

Summary

We have seen how to:

- Invoke native methods from Java.
- Manipulate Java objects and exceptions from native code
- Include a JVM in an application.

Also, with JNI we can:

- Interact with Java threads from within native code.