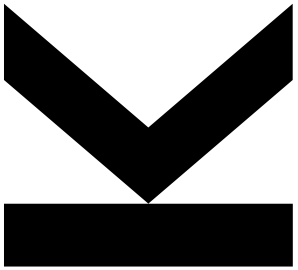
JOHANNES KEPLER
UNIVERSITY LINZ

JAVA NATIVE INTERFACE (JNI)



PR SW2 S18
Dr. Prähofer
DI Leopoldseder

WHAT IS NATIVE CODE?

- Code that is **statically compiled** to a platforms native execution format (machine code)
  - C
  - C++
  - Fortran
  - Go
  - Assembly
  - …..

VS

- Dynamic code that is typically interpreted upon invocation (and later on compiled)
  - Java
  - JavaScript
  - Python
  - R
  - Ruby
  - …

WHY NATIVE CODE?

- Java tries to solve all negative (unsafe) aspects of unsafe languages like C/C++/Fortran etc.
- Why introduce native access ?
  - JDK does not support platform-dependent features
  - Code Re-use (cannot port everything to Java)
  - Real-Time Code (Performance? , Assembly implementations)

PROBLEMS

- Java is a **managed** language
- Java **objects** are **managed** in a **garbage collected** heap
- How to **entangle** native memory with java objects?

CHALLENGES

- Method Calls
  - Java2Native
  - Native2Java
- Parameter Handling
  - Parameter Handling (in the presence of GCed objects)
- Memory Handling
  - Write & Read Java Objects
  - Object Allocation
- Garbage Collection
  - (Moving) GC
  - Roots from native

DISCLAIMER

- The slides presented here are done for Linux/Unix, however the same concepts apply for Windows / Mac, just the toolchain is different

- Linux
  - IDE
    - Eclipse
    - Intellij
    - Netbeans
  - Compiler
    - Gcc
    - LLVM (clang)
  - Library File Extension
    - .so (Shared object)
  - File Format
    - ELF

- MAC
  - IDE
    - Xcode
    - Eclipse
    - Intellij
    - Netbeans
  - Compiler
    - clang
  - Library File Extension
    - .dylib
  - File Format
    - Mach-O

- Windows
  - IDE
    - Eclipse
    - Intellij
    - Visual Studio
  - Compiler
    - Mingw
    - Cygwin
    - **Visual C++**
  - Library File Extension
    - .dll
  - File Format
    - COFF

Please refer to e.g. http://electrofriends.com/articles/jni/part-2-jni-visual-studio-setup-dll-project/ to setup Visual Studio for JNI

HELLO WORLD FROM NATIVE CODE

```java
public class HelloFromC {

    static {
        System.loadLibrary("out");
    }

    static native void sayHello();

    public static vo[illegible]main(String[] args) {
        sayHello();
    }
}
```

Load **libout.so**

Native Modifier

Java Class

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

#ifndef _Included_jni_base_HelloFromC
#define _Included_jni_base_HelloFromC
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     jni_base_HelloFromC
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL
Java_jni_base_HelloFromC_sayHello
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endi
```

Generated header file from **native** methods

```
JNIEXPORT void JNICALL Java_jni_base_HelloFromC_sayHello(JNIEnv* env,
jclass clazz){
        char* str = "Hello World.....from C";
        int ret;
        asm volatile(
            "mov %%rcx,%%rdi\n"
            "call *puts@GOTPCREL(%%rip)" :"=a"(ret) :"c"(str)
        );
        return;
}
```

Arbitrary C implementation

JNI HELLO WORLD WORKFLOW

1. Write Java code with **native** methods (**native modifier**)
2. Generate Header files for all Java classes containing native methods
3. Implement header files in C / C++
4. Compile implementations with native compiler clang/gcc to **position independent** shared library (**-fPIC** mandatory for shared libs on x86-64)
5. Load libraries at runtime (e.g. in static ctor)

WORKFLOW TOOLS

```
class Out {

    static {
        System.loadLibrary("libout");
    }

    public static native void print(String text);
}
```

Also
System.load(constantPath)

JKU

WORKFLOW TOOLS

- Java native methods (in Out.java)
  - `public static native void print(String text);`
- Compile to class file (Creates Out.class)
  - `javac Out.java`
- Generate header file (creates Out.h)
  - `javah Out`
- Implement generated header (in out.c)
  - `#include "Out.h"`
  - `JNIEXPORT void JNICALL Java_Out_print(JNIEnv* env, jclass clazz, jstring text) {...}`
- Compile native code to shared library
  - `gcc -shared -fPIC -I<path-to-jdk-install-dir>/include -o libout.so out.c`
- Set path to libraries @ JVM startup
  - `java -Djava.library.path=. Out`
- Load native library prior to first native call
  - `static { System.loadLibrary("libout"); }`

Tip: On Windows use Visual Studio !

JNI GENERATED HEADER FILES

- Virtual machine needs to be able to **bridge** a call from Java to native code
- Therefore it creates header files that follow a defined format for parameters, types, method interfaces etc

JNI GENERATED HEADER FILES

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

#ifndef _Included_Out
#define _Included_Out
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Out
 * Method:    print
 * Signature: (Ljava/lang/Stri;)V
 */
JNIEXPORT void JNICALL Java_Out_print (JNIEnv*, jclass, jstring);

#i   f __cplusplus
```

Convention: Per **native** Java method one native method Java_<className>_<MethodName>

JNIEnv: Bridge to VM

Parameters: Converted to JNI Types

Macro for Library Export

Macro for C++ Calling Convention

Class of method (non static this)

WHAT ABOUT TYPES?

- We need a defined **mapping** of Java types to native types
- **Java types are the same on all platforms**
  - Int always 2's complement signed integer with 32 bit
  - Floating points….
  - Objects…..
- Native Code: Compiler on platform decides how large types are
  - Int
  - Uint
  - Long int
  - Long long int
  - Strings ? (Cstrings, std::strings)

JNI TYPE MAPPING

Java Typ	JNI Typ	*32-bit Typ*	*64-bit Typ*	Signatur
void	void	void	void	V
byte	jbyte	signed char	signed char	B
short	jshort	short	short	S
int	jint	int	int	I
long	jlong	long long	long	J
float	jfloat	float	float	F
double	jdouble	double	double	D
boolean	jboolean	unsigned char	unsigned char	Z
char	jchar	unsigned short	unsigned short	C
java.lang.Object	jobject	*	*	Ljava/lang/Object;
java.lang.String	jstring	*	*	Ljava/lang/String;
java.lang.Class	jclass	*	*	Ljava/lang/Class;
java.lang.Throwable	jthrowable	*	*	Ljava/lang/Throwable;
java.lang.Object[]	jobjectArray	*	*	[Ljava/lang/Object;
int[]	jintArray	*	*	[I
?	jobject	*	*	L<full-class-name>;

MORE JNI TYPES

JNI Type	Usage
`Jsize`	Array Lengths
`Jweak`	Weak References
`Jvalue`	Base for all primitive types
`jfieldID`	ID for fields
`jmethodID`	ID for methods
`JNIEnv`	Interface to the JVM

- Types defined in **jni.h**
- Actual types in native code are platform dependent and architecture dependent
- Inheritance tree the same in native code

JAVA REFERENCES

- What is an object in the Java world
  - A pointer to a memory location that contains the object
    - VM defines memory layout (architecture and platform dependent)
    - VM utilizes pointer address for GC, etc.
  - Pointer is understood from the VM
    - GC understands pointers
  - Java has **security guarantees**, cannot give java memory to untrusted and unknown native code
- What should be the semantic of a pointer to a Java object from native code?
  - GC cannot visit native code
  - VM cannot stop native code (safepoint concept)

OBJECT HANDLES

- Every reference from native code to a Java object is just a **handle**
  - Native code never has actual pointer to Java object but a handle to the object (access monitored by VM)
- 3 different types of handles (based on their lifetime)
  - Local
    - Dies when native frame goes out of scope
    - Parameters of native methods always local handle
    - Can be deleted with `(*env)->DeleteLocalRef(env, obj)`
  - Global
    - Survives native frame boundaries
    - Needs to be handled manually: `NewGlobalRef`, `DeleteGlobalRef`
  - Weak
    - Same as global except when Java **Garbage Collector** wants to collect the original Object the handle becomes null

OBJECT HANDLE EXAMPLE

```
void Java_Foo_bar(JNIEnv* env, jclass clazz, jobject obj) {
static jobject last_obj = NULL;

if(last_obj != NULL) {
     (*env)->DeleteGlobalRef(env, last_obj);
}
last_obj = (*env)->NewGlobalRef(env, obj);

printf("obj %s last_obj\n", obj == last_obj ? "==" : "!=");

printf("obj is %s the same as last_obj\n",
     (*env)->IsSameObject(obj, last_obj) ? "indeed" : "not");
}
```

```
Output:
obj != last_obj
obj is indeed the same as last_obj
```

JKU

JAVA MEMORY ACCESS: FIELDS

- Search for class

```
jclass GetObjectClass(JNIEnv*, jobject obj)
jclass FindClass(JNIEnv*, char* name)
```

- Search for field (with class and name)

```
jfieldID GetFieldID(JNIEnv*, jclass clazz, char* name, char* sig)
```

- Read/Write Field

```
<T> Get<T>Field(JNIEnv*, jobject obj, jfieldID field);
void Set<T>Field(JNIEnv*, jobject obj, jfieldID field, <T> value);
```

Different types

```
JNIEnv* env = ...
jobject person = ...
jclass clazz = (*env)->GetObjectClass(env, person);
jfieldID field = (*env)->GetFieldID(env, clazz, "age", "I");
jint age = (*env)->GetIntField(env, person, field);
(*env)->SetIntField(env, person, field, age + 1);
(*env)->DeleteLocalRef(clazz);
```

JKU

MEMORY ACCESS: ARRAYS

- 3 Access Kinds
  - Element wise

```
<T> Get<T>ArrayElement(JNIEnv*, j<T>Array array, jsize index)
void set<T>ArrayElement(JNIEnv*, j<T>Array array, jsize index, <T> value)
```

  - Region wise

```
Get<T>ArrayRegion(JNIEnv*, j<T>Array array, jsize start, jsize length, <T>* buffer)
Set<T>ArrayRegion(JNIEnv*, j<T>Array array, jsize start, jsize length, <T>* buffer)
```

  - Array wise

```
<T>* get<T>ArrayElements(JNIEnv*, j<T>Array array)
void Release<T>ArrayElements(JNIEnv*, j<T>Array array, <T>* elems, jint mode)
```

0……..copy element in array and release buffer
JNI_COMMIT…copy element in array and do not release buffer
JNI_ABORT…..copy element not back and release buffer

JKU

ARRAY ACCESS EXAMPLE

```
JNIEnv* env = ...; jintArray array = …;

//Access each element individually
for(jsize i = 0; i < (*env)->GetArrayLength(env, array); i++) {
jint value = (*env)->GetIntArrayElement(env, array, i);
(*env)->SetIntArrayElement(env, array, i, value + 1);
}

//Access the entire array at once
jint* native_array = (*env)->GetIntArrayElements(env, array);
for(jsize i = 0; i < (*env)->GetArrayLength(env, array); i++) {
    native_array[i]++;
}
(*env)->ReleaseIntArrayElements(env, array, native_array, 0);

//Access chunks of the array
jint* chunk = (jint*) calloc(16, sizeof(jint);
for(jsize i = 0; i < (*env)->GetArrayLength(env, array) / 16; i++) {
(*env)->GetIntArrayRegion(env, array, i*16, 16, chunk);
for(int i = 0; i < 16; i++) {
    chunk[i]++;
}
(*env)->SetIntArrayRegion(env, array, i*16, 16, chunk);
}
free(chunk);
```

STRINGS

- Read access like arrays

```
jsize GetStringLength(JNIEnv*, jstring string)
jchar* GetStringChars(JNIEnv*, jstring string)
void ReleaseStringChars(JNIEnv*, jstring string, jchar* chars)
```

Content **not** written back

```
JNIEXPORT void JNICALL Java_Out_print(JNIEnv* env, jclass clazz, jstring text) {
if(text != NULL) {
    jsize length = (*env)->GetStringLength(env, text);
    jchar* characters = (*env)->GetStringChars(env, text);
    char* native_characters = calloc(length + 1, sizeof(char));
    for(jsize i = 0; i < length; i++) {
        native_characters[i] = (char) characters[i]; //assume ASCII only
    }
    native_characters[length] = '\0';
    (*env)->ReleaseStringChars(env, text, characters);
    printf("%s", native_characters);
    free(native_characters);
}
}
```

Clazz and **text** are local handles and therefore to not require explicit deletion

JKU

NATIVE 2 JAVA CALLS

- Find class
- Find Method

"(" Param0Sig Param1Sig…) ReturnSig

```
jmethodID GetMethodID(JNIEnv*, jclass clazz, char* name, char* sig)
```

- Call

```
<T> Call<T>Method(JNIEnv*, jobject thiz, jmethodID method, ...)
<T> CallNonvirtual<T>Method(JNIEnv*, jobject thiz, jclass clazz,
      jmethodID method, ...)
<T> CallStatic<T>Method(JNIEnv*, jclass clazz, jmethodID method, ...)
```

Parameters

- Object allocation

```
jobject NewObject(JNIEnv*, jclass clazz, jmethodID constructor, ...)
jstring NewString(JNIEnv*, jchar* chars, jsize length)
j<T>Array New<Type>Array(JNIEnv*, jsize length)
jobject AllocObject(JNIEnv*, jclass clazz)
```

Create object, no constructor is called

JKU

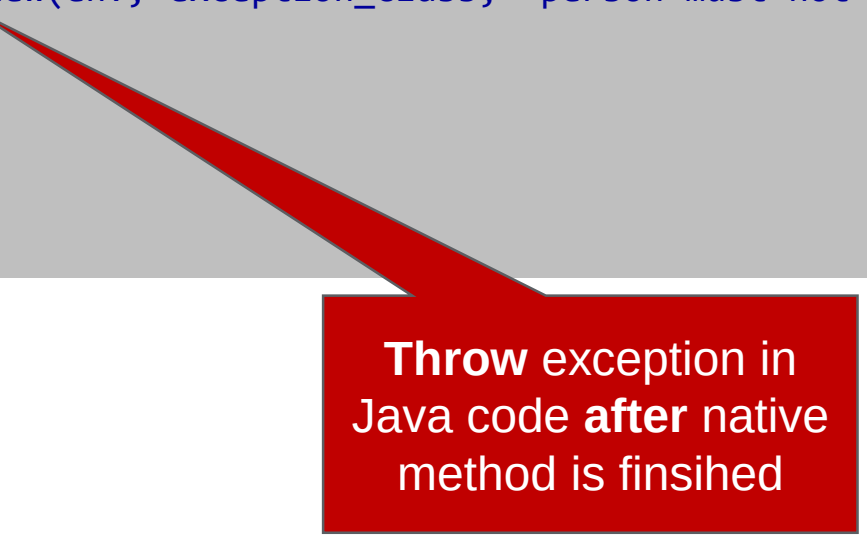
NATIVE 2 JAVA CALLS EXAMPLE

```
JNIEnv* env = ...;
jobject person1 = ...; jobject person2 = ...;

jclass object_class = (*env)->FindClass(env, " Ljava/lang/Object");

const char* sig = "(Ljava/lang/Object;)Z";
jmethodID equals_method = (*env)->GetMethodID(env, object_class, "equals", sig);

jboolean equals = (*env)->CallBooleanMethod(env, person1, equals_method, person2);

(*env)->DeleteLocalRef(env, object_class);
```

(Object): boolean

EXCEPTION HANDLING

```
JNIEXPORT void JNICALL Java_Person_raiseSalary
                  (JNIEnv* env, jclass clazz, jobject person, jdouble factor) {
if(person == NULL) {
     char* exception_name = "java/lang/NullPointerException";
     jclass exception_class = (*env)->FindClass(env, exception_name);
     (*env)->ThrowNew(env, exception_class, “person must not be null”);
     return;
}
...
}
```



JKU

MORE JNI METHODS

Method	Description
`ExceptionOccurred`	Check if the last invocation raised an exception
`ExceptionDescribe`	Produce a descriptive string for the exception
`IsSameObject`	Check for reference equality
`IsInstanceOf`	Check if object is instance of type

https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

EXCURSE SAFEPOINTS

- When does a Garbage Collection happen in Java?
  - Depends on the GC
    - Serial
    - Parallel CMS
    - G1
    - Shenendoa
    - Z GC
    - ….
  - GC's mostly have **stop-the-world** sections where the GC **stops** all application **threads** and pauses them to move objects around in the heap
  - What if JNI code is executed?
    - No problem, JNI is considered a stop the world as all objects are referenced via handles **except JNI critical**

CRITICAL JNI CODE

- JNI API to for accessing Java on-heap memory of primitive Arrays like Get<T>ArrayElements except that primitive array is returned

```
void* GetPrimitiveArrayCritical(JNIEnv*, jarray array, jboolean *isCopy)
void ReleasePrimitiveArrayCritical(JNIEnv*, jarray array, void *carray, jint mode)
```

```
jint len = (*env)->GetArrayLength(env, arr1);
jbyte *a1 = (*env)->GetPrimitiveArrayCritical(env, arr1, 0);
jbyte *a2 = (*env)->GetPrimitiveArrayCritical(env, arr2, 0);
/* We need to check in case the VM tried to make a copy. */
if (a1 == NULL || a2 == NULL) {
  ... /* out of memory exception thrown */
}
memcpy(a1, a2, len);
(*env)->ReleasePrimitiveArrayCritical(env, arr2, a2, 0);
(*env)->ReleasePrimitiveArrayCritical(env, arr1, a1, 0);
```

WHAT IS ALLOWED INSIDE CRITICAL REGIONS?

- **NOTHING**
  - Code is considered a critical region
  - “Do not run for an extended period of time” … (O(?))
  - Must not call other JNI functions
  - Must not call JNI System
  - Avoid calling into Java
  - Avoid calling into blocking Code

- Rule of thumb: Only use critical if you know what you are doing, in general only very advanced engineers should use this API

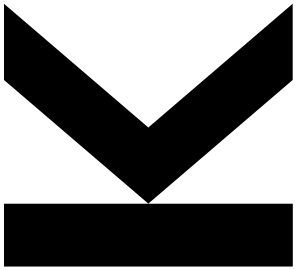
Further readings: https://shipilev.net/jvm-anatomy-park/9-jni-critical-gclocker/

SUMMARY

- JNI allows you to call native code from Java
- For every Java type there is a JNI type
- Native code can use Java objects and call java methods
  - Object handles only (except critical)
  - Manual memory management with java objects

JOHANNES KEPLER
UNIVERSITY LINZ

THANK YOU



JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Strasse 69
4040 Linz, Austria
www.jku.at

JOHANNES KEPLER
UNIVERSITY LINZ