Fachhochschule
Münster University of
Applied Sciences

Java Native Interface

Einbinden von *nativen C/C++* Bibliotheken via JNI in Java Klassen.

Prof. Dr. Nikolaus Wulff

Java Erweiterungen

Java lässt sich erweitern durch:

- Integration nativen C Codes – und erlaubt somit den Zugriff auf andere Hochsprachen und deren Bibliotheken wie z. B. *C/C++*, *Fortran*, *Assembler* etc.
- Ausführen anderer Sprachen innerhalb der JVM, wie z.B. *Scala*, *Lisp* oder *Groovy*.
- Erweiterungen durch *Domain Specific Languages* (DSL)
  - z.B. mit der Java Scripting Engine
  - Eigene Java Quellcode-Generierung z.B. mit AntLR oder JavaCC und EBNF Grammatiken.
- Byte-Code Generierung z.B. mit ASM oder BCEL

Nummerik Bibliothek

- Als Beispiel sollen die Klasse *Differentiator* beliebige Java Funktionen per JNI differenzieren.
- Die eigentliche Implementierung erfolgt in Form einer zu entwickelnden C Bibliothek.
- Die Java Virtuelle Maschine (JVM) lädt auf Anforderung C Bibliotheken und bindet die enthaltenen Funktionen in die Java Anwendung ein.
- Bereits fertig entwickelte Altanwendungen lassen sich so in Java integrieren und wiederverwenden.
  - Native Datenbanktreiber können z.B. als C Routinen eingebunden werden, oder aber auch graphische Fenster-systeme des OS können direkt verwendet werden...

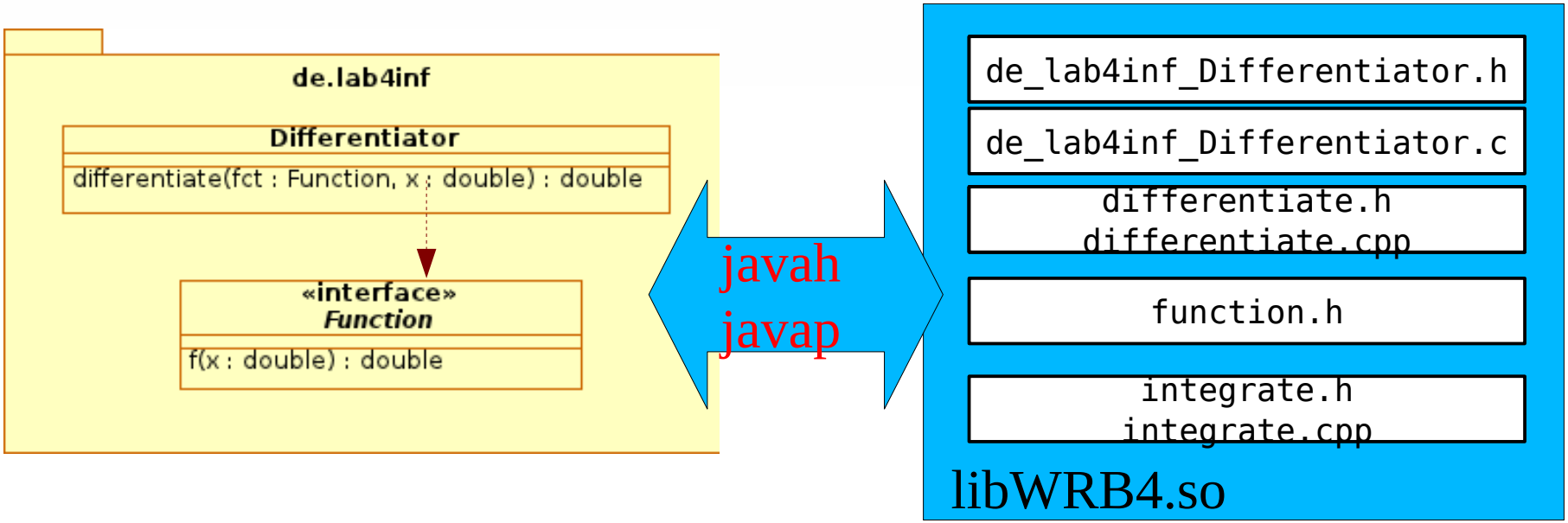
Native Code

- Java bietet die Möglichkeit Klassen mit Methoden zu schreiben, die in C realisiert werden.
- Hierzu werden die Methoden mit dem Schlüsselwort **`native`** markiert, ohne das eine Implementierung in Java erfolgt.

```
public class Differentiator {
    /**
     * Differentiate the given function.
     * @param fct to differentiate
     * @param x the function argument
     * @return f'(x)
     */
    public native double differentiate(
        final Function fct, final double x);
}
```

C Header-Dateien generieren

- Mit dem Tool **javah** wird zu Differentiator.java ein C Header *PackageName*_Differentiator.h generiert.
- Die Datei *PackageName*_Differentiator.c dient als Adapter zu den C/C++ Funktionen.



de_lab4inf_Differentiator.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class de_lab4inf_jni_Differentiator */

#ifndef _Included_de_lab4inf_Differentiator
#define _Included_de_lab4inf_Differentiator
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     de_lab4inf_Differentiator
 * Method:    differentiate
 * Signature: (Lde/lab4inf/Function;D)D
 */
JNIEXPORT jdouble JNICALL
Java_de_lab4inf_Differentiator_differentiate
  (JNIEnv *, jobject, jobject, jdouble);

#ifdef __cplusplus
}
#endif
#endif
```

```
double differentiate(Function fct, double x)
```

JVM

this Zeiger

Funktionsargumente

Differentiator.c howto?

- Wie kann jetzt die C Methode implementiert werden, so dass auch Java Funktionen differenziert werden?
- Hierzu dienen die C++ Klassen ***Function*** (für “normale C Funktionen”) und ***JavaFunction***.
- Es wird nicht nur der Wert *x*, sondern auch eine Möglichkeit benötigt auf die unbekannte Java Funktion – eine Java Schnittstelle – zuzugreifen.
- In der C++ Klasse ***JavaFunction***, wird intern die Java Reflection API aufgerufen, um einen Funktionszeiger auf die eigentliche Java Implementierung zu erhalten.
- ***JavaFunction*** ist lediglich ein Adapter zwischen der C/C++ und der Java Welt.

Java Reflection

- Jedem Java Objekt ist eindeutig ein Class Objekt zugeordnet.
  - Class<?> c = obj.getClass()
- Dieses Class Objekt enthält Metainformationen und bietet „Java FunctionPointer“ (JFP):
  - Method m = c.getDeclaredMethod(name, ...)
  - Field f = c.getDeclaredField(name)
- Dieser JFP erlaubt es ein Objekt zu manipulieren:
  - double x = f.getDouble(obj)
  - m.invoke(obj, ...)

Native Code API

- Alle C Methoden erhalten eine Referenz auf die JVM Umgebung und das entsprechende Java Objekt, dessen native Methode entwickelt wird.
- Dieses „Umgebungsobjekt“ ist eine C Struktur, die verschieden FunctionPointer anbietet, um Methoden und Felder beliebiger Objekte zu addressieren.
- Ermitteln des Class Objekts von obj

```
jclass clazz = (*env)->GetObjectClass(env, obj);
```

- Ermitteln der Methode „double getValue()“ :

```
jmethodID method =
    (*env)->GetMethodID(env, clazz, "getValue", "()D");
```

Signatur

Unterschied C und C++

- Achtung: je nach dem ob der C oder C++ Compiler eingebunden wird, ist die Deklaration in <jni.h> auf Grund der bedingten Compilierung eine andere!
- Werden C Struckturen verwendet gilt:

```
jmethodID method;
jclass clazz = (*env)->GetObjectClass(env, obj);
method = (*env)->GetMethodID(env, clazz, "getValue", "()D");
```

- Während bei C++ mit Klassen gearbeitet wird:

```
jmethodID method;
jclass clazz = env->GetObjectClass(obj);
method = env->GetMethodID(clazz, "getValue", "()D");
```

- Die Endung *.c oder *.cpp entscheidet zwischen einer C oder C++ Übersetzung der Sourcen!

Zugriff auf Felder

- Felder können direkt aus C addressiert werden.
- Es wird keine Rücksicht auf private genommen!

```
jfieldID field;

field = (*env)->GetFieldID(env, clazz, "x", "D");

(*env)->SetDoubleField(env, obj, field, 1.0);
```

- Die zu verwendenen C/C++ Strukturen, Klassen und Funktionen sind in der Datei **<jni.h>** deklariert.

Aufruf von Funktionen

- Aufruf der Methode „getValue“:

```
double x;

x =  (*env)->CallDoubleMethod(env, obj, method);
```

- Gegenstück „setValue“:

```
method = (*env)->GetMethodID(env, clazz, "setValue", "(D)V");

(*env)->CallVoidMethod(env, obj, method, x);
```

Signatur

- Die Signatur dient dazu überladene Methoden eindeutig zu identifizieren, und muss genau angegeben und vorher ermittelt werden (mit ***javap***).

Java Disassembler

- Der Java Disassembler ***javap*** gestattet es in das Innere von Java Klassen zu schauen.
- Mit den Parametern -p -s liefert javap die Signatur Informationen:

```
~/workspace/WRB$ javap -s -p de.lab4inf.Differentiator
Compiled from "Differentiator.java"

public class de.lab4inf.Differentiator extends java.lang.Object
public de.lab4inf.Differentiator();
  Signature: ()V
public native double differentiate(de.lab4inf.Function, double);
  Signature: (Lde/lab4inf/Function;D)D
static {};
  Signature: ()V
}
```

Objekterzeugung

- Primitive Datentypen können dirket zurückgegeben werden. Objekte müssen erst erzeugt werden:

```
jclass   clazz;
jmethodID ctor;
jobject complex;

clazz = (*env)->FindClass(env, "math/Complex");
assert(clazz != NULL);


ctor  = (*env)->GetMethodID(env, clazz, "<init>", "()V");
assert(ctor != NULL);

complex =  (*env)->NewObject(env, clazz, ctor);
assert(complex != NULL);

return complex;
```

Konstruktor Signatur

Einbinden der C Bibliothek

- Vor dem Verwenden von nativen Klassen muß die zugehörige C Bibliothek geladen werden.
- Diese hat immer den generischen Namen
  - libXXX.so für Unixe oder libXXX.dll für Windows
- Geladen wird sie am Besten beim Start der Anwendung mit dem Aufruf:
  - System.loadLibrary("XXX");
  - gesucht wird im **java.library.path**.
  - java -Djava.library.path=.... setzt den Pfad.
- Oder nicht portabel mit expliziter Pfadangabe:
  - System.load("/pfad/libXXX.so");

Warnungen

- Mit dem C Code ist es möglich auf jedes Feld und jede Methode der JVM zuzugreifen.
- Sogar eigene Semaphore, Mutexe und critical Sections können geschrieben werden.Es ist möglich die gesamte JVM lahmzulegen. Es ist daher Vorsicht angebracht. Besser keine eigenen Synchronisations-mechanismen in C einbinden!
- Kein malloc und free etc.
- Keine globalen Referenzen auf Objekte!
- Objekte mit Java Mitteln erzeugen, damit der Garbage Collector den Speicher verwaltet.

Java und C ein gutes Gespann

- Mit JNI bleiben keine Wünsche mehr offen. Mit C ist die gesamte Java Reflection API aufrufbar.
- Ein guter Programmierer kann bis auf Betriebs-systemebene gehen, sogar Assembler ist möglich.
- Die Eclipse ist ein gutes Beispiel für Java und C.
- Eclipse verwendet weder AWT noch Swing für die GUI sondern das selbstentwickelte SWT.
- SWT ist eine Bibliothek, die aus graphischen Klassen besteht, die vollkommen in C implementiert sind und direkt delegieren an die Widgets des jeweiligen Betriebsystems.