SEMINARARBEIT

Interkompatibilität von Programmiersprachen: Java Native Interface

durchgeführt am Masterstudiengang
Angewandte Informatik
Universität Salzburg

vorgelegt von
Christian Sesser

Studiengangsleiter: O.Univ.-Prof. Dr. Peter Zinterhof

Betreuer: Univ.-Prof. Dr. Wolfgang Pree

Salzburg, Juli 2012

Inhaltsverzeichnis

1 Einleitung 4
1.1 Begriffserklärung 4
1.1.1 Die Java-Plattform 4
1.1.2 Host-Umgebung 4
1.2 Die Bedeutung von JNI 4
2 Vorgehensweise mit JNI 5
2.1 Deklaration der nativen Methode 6
2.2 Kompilierung der Java Klassen 6
2.3 Erstellung der Header-Datei für die nativen Methoden 7
2.4 Implementierung der nativen Methoden 7
3 JNI – ein Überblick 7
3.1 JNIEnv, jobject – Die zwei Standard Parameter 8
3.2 Datenaustausch zwischen Java und Native 8
3.3 Garbage Collector 9
3.4 Zugriff auf Objekte 10
4 Häufige Fehler und Tücken mit JNI 11
4.1 Überprüfung auf Fehlerbedingungen 11
4.2 Übergabe von ungültigen Argumenten an JNI-Funktionen 11
4.3 Die Abgrenzung von Java Applikation und nativem Code 11
4.4 Verletzung von Zugriffsregeln (Umgehung der Datenkapselung) 12
4.5 Beibehalten von Ressourcen der virtuellen Maschinen 12
5 Exceptions 12
5.1.1 Checking for Exceptions 12
5.1.2 Exceptionhandling 14
6 Einbindung bestehender Bibliotheken 15
6.1 One-to-One Mapping 15
6.2 Shared Stubs 15
7 JNI und Threads 16
7.1 Monitor Entry und Exit 16
7.2 Monitor Wait und Notify 17
8 Von nativen auf Java Code zugreifen 18

I. Abbildungsverzeichnis 19
II. Tabellenverzeichnis 19
III. Literaturverzeichnis 19

1 Einleitung

1.1 Begriffserklärung

1.1.1 Die Java-Plattform

Die Java-Plattform ist eine Programmierumgebung, die aus der Java Virtual Machine (JVM) und dem Java Application Programming Interface (API) besteht.

Java-Anwendungen werden in der Programmiersprache Java geschrieben und in eine Hardware-unabhängige binäre Class-Datei (*.class) kompiliert. Diese Class-Datei kann auf jeder JVM Implementierung ausgeführt werden. Die Java API besteht aus einer Reihe von vordefinierten Class-Dateien. Jede Implementierung der Java-Plattform garantiert die Programmiersprache Java, die virtuelle Maschine und API zu unterstützen.

1.1.2 Host-Umgebung

Der Begriff Host-Umgebung schließt das Host-Betriebssystem, eine Reihe von nativen Bibliotheken, und die CPU-Befehlssatz ein. Native Anwendungen sind in nativen Programmiersprachen wie C und C++ geschrieben, werden in Host-spezifischen binären Code kompiliert, und mit nativen Bibliotheken verlinkt. Native Anwendungen und nativen Bibliotheken sind in der Regel abhängig von einer bestimmten Host-Umgebung.

Beispielsweise wird eine C-Anwendung, welche für ein Betriebssystem entwickelt wurde, in der Regel nicht auf anderen Betriebssystemen laufen.

Java-Plattformen setzen üblicherweise auf eine Host-Umgebung auf. Zum Beispiel ist die Java Runtime Environment (JRE) ein Sun Produkt, das die Java-Plattform auf Betriebssystemen wie Windows und Solaris unterstützt. Die Java-Plattform bietet eine Reihe von Funktionalitäten, auf die sich Anwendungen, unabhängig von der zugrunde liegenden Host-Umgebung, verlassen können. [1]

1.2 Die Bedeutung von JNI

Die Maschinenunabhängigkeit ist ein großes Plus von Java, doch wird durch den Zwischenschritt über die JVM zwangsläufig Geschwindigkeit eingebüßt. Für viele Applikation spielt dies keine schwerwiegende Rolle, doch gibt es Fälle, in denen ein Geschwindigkeitsverlust nicht akzeptierbar ist. In solchen Fällen könnte der Wunsch aufkommen, gewisse Module native zu programmieren.

Geschwindigkeit ist aber nicht der einzige Grund, bei dem das Verlangen groß sein könnte, nativen Code in Java zu integrieren. Es gibt Fälle, in denen mit Java am Anschlag gelaufen wird, da das Java API keine Host abhängige Funktionen zur Verfügung stellt.

Eine andere Situation, in der JNI zum Einsatz kommen könnte, ist jene, in der eine bestehende native Library nicht ein zweites Mal geschrieben werden möchte. Wenn gewisse Funktionalitäten in nativem Code bereits existieren, dann kann es ein großer Mehraufwand bedeuten, diese in die Java Sprache umzuschreiben. Es wäre doch bequem, diese gleich aus der Java Applikation anzusprechen. JNI macht dies möglich.

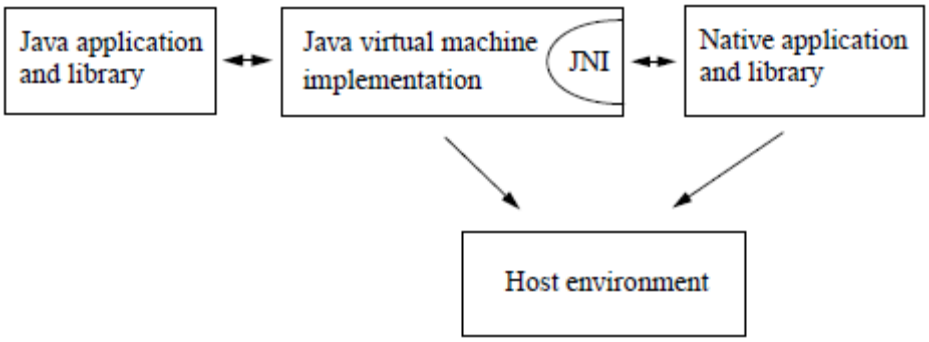

Abbildung 1. JNI – Überblick [1]

Abbildung 1 zeigt die Zusammenhänge von JNI. Wie ersichtlich wird, ist mittels JNI eine Kommunikation in beide Richtungen möglich. Einerseits können aus Java-Applikationen nativer Code aufgerufen werden, andererseits kann die native Seite auf Java-Funktionalitäten zurückgreifen.

Durch den Einsatz von JNI gehen zwei Vorteile von Java verloren:

- Aufgrund des nativen Teiles verliert die Applikation ihre Maschinenunabhängigkeit. Der Java-Teil lässt sich zwar weiterhin auf verschiedene Maschinen übertragen, der native Anteil muss jedoch auf Betriebssysteme und Befehlssätze neu angepasst werden.
- Die Typen-Sicherheit und -Prüfung, welche von Java großzügig unterstützt wird, hinterlässt bei nativen Sprachen wie C und C++ die Nachsicht. Auf diesen Punkt wird in einem späteren Kapitel näher eingegangen.

2 Vorgehensweise mit JNI

Dieses Kapitel beschreibt die Vorgehensweise zur Erstellung einer Java Anwendung, die mittels JNI eine C-Funktion zur Konsolenausgabe von „Hello World!“ aufruft. Abbildung 2 stellt den Entwicklungsablauf da, welcher aus folgenden Punkten besteht:

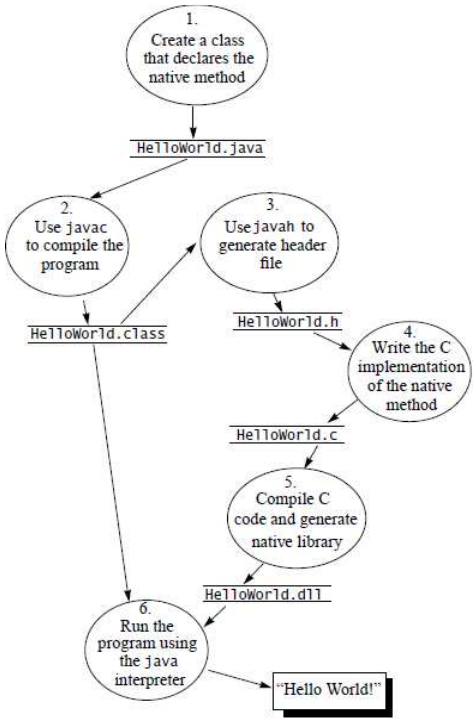

Abbildung 2. Schritte zur Entwicklung einer JNI Anwendung [1]

2.1 Deklaration der nativen Methode

Erstellen einer Klasse (HelloWorld.java), welche die native Methode „print()“deklariert:

```
class HelloWorld {
      private native void print();
      public static void main(String[] args) {
            new HelloWorld().print();
      }
      static {
            System.loadLibrary("HelloWorld");
      }
}
```

Das Attribut native weist darauf hin, dass diese Methode in einer anderen Sprache implementiert wird. In diesem Fall in einer eigenen C-Datei. Aus diesem Grund wird die Methodendeklarierung auch mit einem Strichpunkt beendet. Die statische Anweisung „`System.loadLibrary("HelloWorld");`“ veranlasst das Laden der nativen Bibliothek, bevor die Main-Methode gestartet wird.

2.2 Kompilierung der Java Klassen

Nach der Erstellung der Klassen, können die Java Quelldateien mit Hilfe des „javac“ Compilers in Java Class-Dateien übersetzt werden. Damit ist der Teil auf der Java Seite bereits abgeschlossen.

2.3 Erstellung der Header-Datei für die nativen Methoden

Als nächstes wird mit dem Tool „javah“ eine C Header-Datei generiert. Diese beinhaltet die Funktionsprototypen der nativen Methodenimplementierung im JNI-Format. Der Befehl lautet „`javah -jni HelloWorld`“ und erstellt aus „`HelloWorld.class`“ „`HelloWorld.h`“. Der wichtige Teil der Header-Datei lautet:

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

Er beschreibt genau die Methode, wie sie in C/C++ implementiert werden muss.

2.4 Implementierung der nativen Methoden

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"
JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, jobject obj)
{
	printf("Hello World!\n");
	return;
}
```

Dieses C-Programm bindet drei Header-Dateien ein:

- *jni.h* – Diese Header-Datei beinhaltet Informationen für den Aufruf von JNI Funktionen. Sie muss immer inkludiert werden.
- *stdio.h* – Dies Standard Ein- Ausgabe Header-Datei wird für die Funktion *printf* benötigt.
- *HelloWorld.h* – Die erzeugte Header-Datei mit den Methodenprototypen.

Die beiden Funktionsparameter *JNIEnv* und *jobject* werden in diesem Beispiel nicht benötigt. Auf sie wird in einem späteren Kapitel eingegangen.

Anschließend muss der C Quellcode kompiliert und die native Library gebildet werden. [1]

3 JNI – ein Überblick

In diesem Kapitel werden die Funktionsweise und Abläufe von JNI erläutert. Hierzu wird noch einmal der C Prototyp einer nativen Methode herangezogen:

```
JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, jobject obj)
```

JNIEXPORT und JNICALL sind zwei Makros, denen hier keine große Aufmerksam zukommen soll. Sie stellen sicher, dass die Funktion von der nativen Library exportiert wird und der Compiler die Aufrufkonvention einhält.

3.1 JNIEnv, jobject – Die zwei Standard Parameter

Die zwei Standardparameter `(JNIEnv *env, jobject obj)` kommen in jeder Methode vor. Der erste, der so genannte JNIEnv Interface Pointer, ist ein Pointer, der an die Stelle zeigt, wo ein Pointer abgelegt ist, welcher auf die JNI Funktions-Tabelle verweist. In dieser Tabelle befinden sich Pointer, welche zu den eigentlichen JNI Funktionen führen. Diese JNI Funktionen werden gebraucht, um von nativen Methoden aus auf die Datenstruktur der JVM zuzugreifen. Diese Funktionen stellen also die eigentliche Schnittstelle zwischen nativen und Java-Code dar. Als Beispiel können die Funktionen GetStringUTFChars, GetObjectClass oder auch GetMethodID angefügt werden. Abbildung 3 zeigt die Zusammenhänge des JNIEnv Interface Pointers.

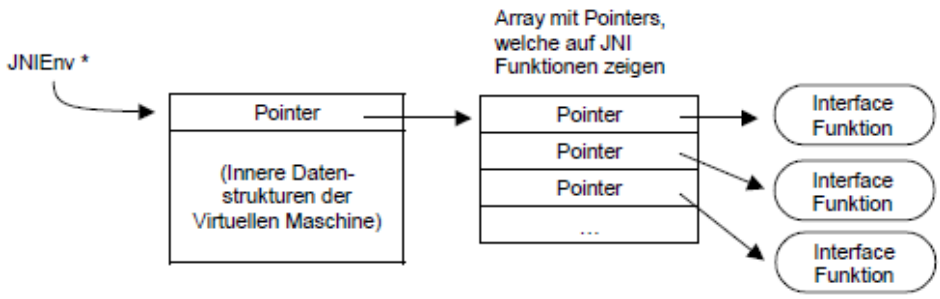

Abbildung 3. Übersicht JNIEnv Pointers

Der zweite Standardparameter hängt davon ab, ob es sich um eine statische oder um eine Instanzmethode handelt.

Bei Instanzmethoden, handelt es sich beim Parameter um eine Referenz auf das Objekt, auf dem die Methode aufgerufen worden ist. Hingegen bei einer statischen Methode, fungiert er als Referenz auf die Klasse, in der sie definiert ist.

3.2 Datenaustausch zwischen Java und Native

Für den Austausch primitiver Datentypen zwischen Java und nativem Code wurden in JNI spezielle native Datentypen spezifiziert, welche mit den Typen aus Java korrespondieren. Bei den primitiven Typen geht ein int in einen jint über oder einen char in einen jchar. Tabelle 1, die zusätzlich auch noch Aufschluss über die Größe gibt, vervollständigt das Set der primitiven Typen:

Java Language Type	Nativer Type	Beschreibung
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

Tabelle 1. Mapping von primitiven Datentypen [1]

Bei den Referenzen wurden die Referenztypen auf eine kleine Anzahl beschränkt. Die wichtigsten sind jclass und jstring, welche in der Hierarchie unter jobject liegen. Der Vollständigkeit wegen seien noch jarray und jthrowable erwähnt, doch auf diese soll hier nicht eingegangen werden. Grundsätzlich werden alle Java-Objekte auf der nativen Seite als jobject gehandhabt. Der Zugriff auf Methoden und Felder eines Objektes wird in einem späteren Kapitel erklärt. All diese nativen Datentypen werden mit der bereits in Kapitel 2.4 erwähnten *jni.h* Header-Datei importiert.

Wie bereits aus Java bekannt, werden primitive Datentypen als Kopie und Objekte als Referenz Methoden übergeben. Primitive Typen werden also zwischen JVM und dem nativen Code kopiert.

Referenzen, welche dem nativen Code übergeben werden, zeigen nicht direkt auf Objekte, sondern nur auf einen Pointer, welcher dann auf das eigentliche Objekt verweist. Mehr Details dazu im nachfolgenden Kapitel.

3.3 Garbage Collector

Der Garbage Collector (GC) bereinigt den Speicher von Objekten, auf denen keine Referenz mehr besteht. Innerhalb von Java stellt dies kein Problem dar. Wie der GC erfährt, ob Referenzen, die nativem Code übergeben wurden, noch verwendet werden, wird in diesem Kapitel geklärt.

Sobald eine native implementierte Methode aufgerufen wird, wird eine Tabelle (Registry) erstellt, in der alle Referenzen eingetragen werden. Objekte, welche durch die Tabelleneinträge referenziert werden, können vom GC nicht weggeräumt werden. Neue instanziierte Objekte werden ebenfalls in diese Registry eingetragen. Sobald die native Methode beendet ist, wird die Tabelle gelöscht und somit verschwinden auch die Referenzen auf die Objekte.

Der GC löscht aber nicht nur Objekte aus dem Speicher, sonder räumt diesen nach dem Löschen auch auf. Zum Aufräumen müssen zwangsläufig Objekte verschoben werden, doch was passiert nun, wenn im nativen Code noch eine Referenz gehalten wird? Diese Referenz wäre nach dem Verschieben nicht mehr gültig.

Damit dies nicht der Fall ist, der GC aber dennoch seine Arbeit machen kann, werden nicht direkt Pointer auf Objekte übergeben, sondern nur Referenzen auf Pointer, welche auf die Objekte zeigen. So können die Objekte beliebig verschoben werden und die Virtuelle Maschine muss nur die Pointer anpassen (siehe Abbildung 4)

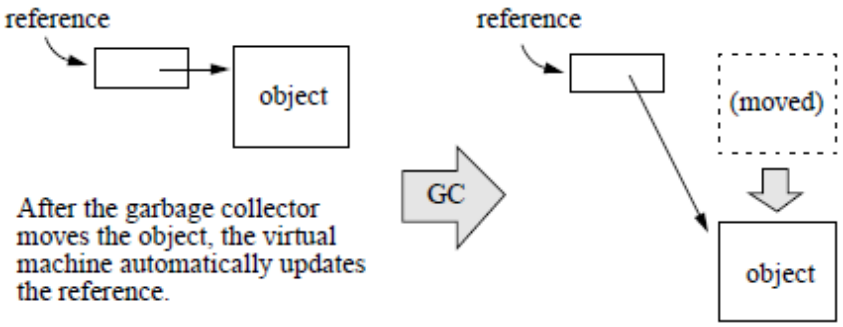

Abbildung 4. Verschiebung von Objekten, während nativer Code eine Referenz hält

3.4 Zugriff auf Objekte

Innerhalb der Implementierung einer nativen Methode, ist die Referenz auf das Objekt, das einzige was zur Verfügung steht. Es muss also zuerst eine Referenz auf die Klasse des Objekts geholt werden. Mit Hilfe dieser und dem Feld- oder Methodennamen kann anschließend die sogenannte Feld ID (jfieldID) oder Methoden ID (jmethodID) geholt werden. Zusammen mit einer solchen ID und der Referenz auf das Objekt kann dann auf die Objektdaten und -Methoden zugegriffen werden. Anhand der Beispiele soll dies verdeutlicht werden:

- Holen der Referenz auf die Objektklasse:
  ```
  jclass cls = (*env)->GetObjectClass(env, mObj);
  ```
- Holen der Feld- und Methoden ID (int x und void output(int z):
  ```
  jfieldID fid = (*env)->GetFieldID(env, cls, "x", "I");
  jmethodID mid = (*env)->GetMethodID(env, cls, "output", "(I)V");
  ```
- Lesen des int-Feldes x:
  ```
  x = (*env)->GetIntField(env, mObj, fid);
  ```
- Aufruf der void-Funktion output:
  ```
  (*env)->CallVoidMethod(env, mObj, mid, x);
  ```

4 Häufige Fehler und Tücken mit JNI

4.1 Überprüfung auf Fehlerbedingungen

Häufig treten Probleme bei der Ausführung eines Programmes auf, wenn nicht nach Fehlerbedingungen geprüft wird. Im Gegensatz zu Java, verfügen native Programmiersprachen über keine Standard-Exception-Mechanismen. Weiters stützt sich JNI nicht auf native Exception-Mechanismen (wie zB C++-Exceptions). Um eine robuste Software zu erhalten, müssen als Folge nach jedem Aufruf von JNI-Funktionen, die möglicherweise eine Exception auslösen, Kontrollabfragen durchgeführt werden.

4.2 Übergabe von ungültigen Argumenten an JNI-Funktionen

JNI-Funktionen erkennen oder beheben keine ungültigen Argumente. Erwartet eine JNI-Funktion eine Referenz, erhält jedoch NULL oder (jobject) 0xFFFFFFFF, so ist das resultierende Verhalten nicht definiert. Konkret könnte dies entweder zu falschen Ergebnissen oder zum Absturz der virtuellen Maschine führen. Ab Java 2 SDK Version 1.2 existiert die Kommandozeilen-Option „Xcheck: jni“. Diese Option weist die virtuelle Maschine an, Situationen aufzuspüren, in denen nativer Code JNI-Funktionen mit falschen Argumenten aufruft. Diese Überprüfung der Gültigkeit von Argumenten verursacht eine erhebliche Menge an Overhead und ist deswegen standardmäßig nicht aktiviert.

4.3 Die Abgrenzung von Java Applikation und nativem Code

Beim Designen einer Java-Anwendung, welche nativen Code mit einbezieht, stellt sich die Frage, wie viel und was in den nativen Code ausgelagert werden soll. Die Grenzen zwischen dem nativen Code und dem Rest der Anwendung in der Programmiersprache Java sind anwendungsspezifisch Es gibt jedoch einige allgemein gültige Grundsätze:

- Die Grenzen sollten einfach gehalten werden. Komplexe Ablaufsteuerungen, die zwischen der JVM und dem nativem Code hin und her gehen, sind schwer zu debuggen und zu verwalten. Außerdem können diese leistungsoptimierte JVM-Implementierungen beeinträchtigen.

- Der Code auf der nativen Seite sollte minimal gehalten werden. Es gibt zwingende Gründe, dies zu tun. Nativer Code ist weder portierbar, noch typsicher. JNI ermöglicht den Zugriff auf Funktionen der virtuellen Maschine, wie das Laden von Klassen-, Objekt-Erstellung, Methodenaufrufe, Thread-Synchronisation, etc. Manchmal ist es verlockend, komplexe Interaktionen mit Java-VM-Funktionalitäten in nativem Code zu realisieren, obwohl diese in der Tat einfacher in Java zu erreichen sind. In gutem Software-Engineering werden solche Teile auf ein Minimum gehalten.

- Des Weiteren sollte der native Code isoliert gehalten werden. Native Methoden können im gleichen Package oder in einer eigenen Klasse zum Rest der Anwendung

getrennt werden. Dieses Package oder diese Klasse bildet damit sozusagen die "Portierung Schicht" für die Anwendung.

4.4 Verletzung von Zugriffsregeln (Umgehung der Datenkapselung)

JNI setzt keine Zugriffsregeln auf gekapselte Klassen, Felder oder Methoden durch. Somit können damit die auf Java Ebene eingesetzten Modifiers private, protected, final, usw. umgangen werden, auch wenn diese zu einer IllegalAccessException führen würden. Diese Freizügigkeit war eine bewusste Design-Entscheidung, da mit nativem Code ohnehin jede Speicherstelle im Heap manipuliert werden kann. Es ist zu berücksichtigen, dass das Umgehen dieser Zugriffsüberprüfung unerwünschte Auswirkungen auf die Programmausführung haben kann.

4.5 Beibehalten von Ressourcen der virtuellen Maschinen

Eine häufige Fehlerquelle in nativen Methoden ist, Ressourcen der virtuellen Maschinen nicht wieder freizugeben. [1]

5 Exceptions

Im vorherigen Kapitel wurde darauf hingewiesen, dass nativer Code nach dem Aufruf von JNI-Funktionen auf mögliche Fehler prüfen soll. In diesem Kapitel wird darauf eingegangen, wie diese Fehler aufgespürt werden können und darauf reagiert werden soll. Auf willkürliche Fehler, die in der nativen Geschäftslogik passieren, wird nicht eingegangen (zB Fehler nach Systemaufrufen).

5.1.1 Checking for Exceptions

Es existieren zwei Möglichkeiten zu prüfen, ob ein Fehler aufgetreten ist:

1. Die meisten JNI-Funktionen verwenden einen eindeutigen Return-Wert (zB NULL), um hinzuweisen, dass ein Fehler aufgetreten ist. Der Fehlercode gibt auch Aufschluss darüber, dass sich eine nicht abgearbeitete Exception im aktuellen Thread befindet (Fehlercodes im Rückgabewert ist gängige Praxis in C).
   Das folgende Beispiel zeigt, wie der Rückgabewert der Methode Get-FieldID bei NULL auf Fehler überprüft wird. Das Beispiel besteht aus zwei Teilen: einer Klasse Window mit einigen Beispielfeldern (handle, length, width) und einer nativen Methode, welche die Field-IDs dieser Felder zwischenspeichert. Obwohl diese Felder in der Window-Klasse existieren, müssen für mögliche Fehler bei GetFieldID vorgesorgt werden. Beispielsweise könnte die virtuelle Maschine nicht ausreichend Speicher allokieren, um eine Field-ID zu repräsentieren. [1]

```
/* a class in the Java programming language */
public class Window {
        long handle;
        int length;
        int width;
        static native void initIDs();
        static {
                initIDs();
        }
}
```

```
/* C code that implements Window.initIDs */
jfieldID FID_Window_handle;
jfieldID FID_Window_length;
jfieldID FID_Window_width;
JNIEXPORT void JNICALL
Java_Window_initIDs(JNIEnv *env, jclass classWindow)
{
        FID_Window_handle =
                (*env)->GetFieldID(env, classWindow, "handle", "J");
        if (FID_Window_handle == NULL) { /* important check. */
                return; /* error occurred. */
        }
        FID_Window_length =
                (*env)->GetFieldID(env, classWindow, "length", "I");
        if (FID_Window_length == NULL) { /* important check. */
                return; /* error occurred. */
        }
        FID_Window_width =
                (*env)->GetFieldID(env, classWindow, "width", "I");
/* no checks necessary; we are about to return anyway */
}
```

2. Bei Verwendung von JNI-Funktionen, deren Rückgabewert keine Fehler denunzieren können, muss sich der native Code auf ausgelöste Exceptions zur Fehlerüberprüfung verlassen. Die JNI-Funktion ExceptionOccurred überprüft, ob der aktuelle Thread eine Exception ausgelöst hat. Beispielsweise kann die JNI-Funktion CallIntMethod im Rückgabewert keine Fehler codieren. Typische Fehlercodes, wie NULL oder -1, funktionieren nicht, weil sie gültige Rückgabewerte einer Integer Methode sein könnten. Im anschließenden Beispiel betrachten wird die Java-Methode floor(), und einen nativen Code, der diese aufruft. Der native Code überprüft, ob eine Exception im aktuellen Kontext aufgetreten ist, bevor das Ergebnis weiterverwendet wird. [1]

```
public class Fraction {
        // details such as constructors omitted
        int over, under;
        public int floor() {
                return Math.floor((double)over/under);
        }
}
```

```
/* Native code that calls Fraction.floor. Assume method ID
MID_Fraction_floor has been initialized elsewhere. */

void f(JNIEnv *env, jobject fraction)
{
      jint floor = (*env)->CallIntMethod(env, fraction,
                                         MID_Fraction_floor);
      /* important: check if an exception was raised */
      if ((*env)->ExceptionCheck(env)) {
            return;
      }
      ... /* use floor */
}
```

5.1.2 Exceptionhandling

Nativer Code kann anstehende Exception auf zwei Arten verarbeiten:

- Die native Methode kann sich beenden und die Exception beim Aufrufenden behandeln lassen.
- Der native Code kann die Exception durch den Aufruf ExceptionClear löschen und anschließend seinen eigene Exception-Handling-Code ausführen.

Es ist äußerst wichtig, auf ausgelöste Exceptions zu überprüfen, diese zu bearbeiten und zu löschen, bevor nachfolgenden JNI-Funktionen ausgeführt werden. Bleiben ausgelöste Exceptions unbehandelt, so kann dies in späteren JNI-Funktionen zu unerwarteten Auswirkungen führen.

Oft ist es auch notwendig, nach ausgelösten Exceptions mittels speziellen JNI-Funktionen Ressourcen der JVM wieder freizugeben. Im folgenden Beispiel wird der Inhalt eines Strings der Methode GetStringChars übernommen. Wenn eine Exception ausgelöst wird, wird ReleaseStringChars aufgerufen und die Funktion mittels return sofort verlassen, ohne die Exception zu löschen. [1]

```
JNIEXPORT void JNICALL
Java_pkg_Cls_f(JNIEnv *env, jclass cls, jstring jstr)
{
      const jchar *cstr = (*env)->GetStringChars(env, jstr);
      if (c_str == NULL) {
            return;
      }
      ...
      if (...) { /* exception occurred */
            (*env)->ReleaseStringChars(env, jstr, cstr);
            return;
      }
      ...
      /* normal return */
      (*env)->ReleaseStringChars(env, jstr, cstr);
}
```

6 Einbindung bestehender Bibliotheken

6.1 One-to-One Mapping

Beim One-to-one Mapping wird eine Wrapper-Klasse erstellt. Zuerst wird eine native Methode in Java definiert, welche mehr oder weniger gleiches Erscheinen hat, wie die zu wrappende Funktion. Anschließend wird die Java native Methode in C/C++ implementiert. Bei der Implementierung läuft es darauf hinaus, dass die zu wrappende Funktion mit den übergebenen Argumenten aufgerufen wird. Dabei kann es vorkommen, dass gewisse Argumente zuerst noch bearbeitet werden müssen – man denke zum Beispiel an einen String, der zuerst noch konvertiert werden muss – bevor sie der Funktion weiter gegeben werden können. Dieses Verfahren mag gut sein, so lange es sich nur um einzelne Funktionen mit einfachen Parametern und Rückgabewerten handelt. Es wird sehr schnell aufwändig und kompliziert, wenn es sich bei den Parametern nicht nur um primitive Typen handelt, sondern um Pointers auf Strukturen.

Ein Beispiel für One-to-one Mapping soll an der Funktion atol gezeigt werden, welche einen String in einen Dezimalwert umwandelt. Die Wrapper-Klasse sieht dann folgendermaßen aus:

```
public class C {
        public static native int atol(String str);
        ...
}
```

Und die dazugehörige native Implementierung:

```
JNIEXPORT jint JNICALL
Java_C_atol(JNIEnv *env, jclass cls, jstring str)
{
        const char *cstr = env->GetStringUTFChars(str, 0);
        if (cstr == NULL) {
                return 0; /* out of memory */
        }
        int result = atol(cstr);
        env->ReleaseStringUTFChars(str, cstr);
        return result;
}
```

6.2 Shared Stubs

Der Ansatz mit Shared Stubs ist einiges bequemer, dafür nicht mehr ganz so schnell wie One-to-one Mapping. Es ist aufwändiger Shared Stubs zu programmieren, da über solche gleich mehrere native Funktionen aufgerufen werden können, doch anschließend ist ihr Einsatz sehr praktisch. Sie übernehmen ganz allgemein die Umwandlung der Parameter und der richtige Aufruf der Funktionen. Von Java wird bereits eine solche Shard Stub Klasse zur Verfügung gestellt und zwar ist dies die Klasse java.lang.CFunction. Mit ihrer Hilfe ist es möglich,

Funktionen aus bestehenden nativen Libraries aufzurufen, ohne eine einzige Zeile native Code schreiben zu müssen.

Als Beispiel für den Gebrauch von Shared Stubs soll wieder die Funktion atol seine Dienste leisten. Einfachheitshalber wird die bereits implementierte Klasse CFunction herbeigezogen:

```
public calss C {
      private static CFunction c_atol =
            new CFunction(  "msvcrt.dll",// native library name
                            "atol", // C function name
                            "C"); // calling convention
      public static int atol(String str){
            return c_atoll.callInt(new Object() {str});
      }
      ...
}
```

Für welche Variante entschieden wird, ist Situation bedingt. Steht die Geschwindigkeit wirklich an vorderster Stelle, empfiehlt sich One-to-one Mapping, da bei dieser Variante kein Overhead bei der Argumentumwandlung und beim Aufruf der Funktion entsteht. Müssen aber gleich ganze Bibliotheken eingebunden werden, könnte die Entscheidung positiv für Shared Stubs ausfallen.

7 JNI und Threads

Beim Schreiben von nativen Methoden, die in einer Multithread-Umgebung ausgeführt werden sollen, gibt es gewisse Einschränkungen, die beachtet werden müssen:

- Ein JNIEnv Pointer ist nur in dem Thread gültig, dem er zugeordnet ist. Er darf weder anderen Threads übergeben werden, noch zwischengespeichert werden, um ihn in mehreren Threads zu verwenden. Die JVM übergibt beim Aufruf nativer Methoden denselben JNIEnv Pointer des jeweiligen Threads.
- Lokale Referenzen sind nur in dem Thread gültig, der sie geschaffen haben. Sie dürfen nicht von einem Thread zu einem anderen übergeben werden. Lokale Referenzen sollten zu globale Referenzen umgewandelt werden, wenn die Möglichkeit besteht, dass mehrere Threads die gleichen Referenzen verwenden. [1]

7.1 Monitor Entry und Exit

Monitore (Lock, Mutex, Semaphore) sind die primitiven Synchronisations-Mechanismen auf der Java-Plattform. Jedes Objekt kann dynamisch mit einem Monitor verbunden sein. JNI ermöglicht die Synchronisation mit diesen Monitoren, damit die Umsetzung der Funktionalität einem synchronisierten Block in der Programmiersprache Java entspricht: [1]

```
synchronized (obj) {
...                 // synchronized block
}
```

Die JVM garantiert, dass ein Thread den Monitor mit dem Objekt obj verbunden erwirbt, bevor es Anweisungen aus dem Block ausführt. Somit existiert höchstens ein Thread, der den Monitor hält und innerhalb dem synchronisierten Block Code ausführt.

Nativer Code kann JNI-Funktionen verwenden, um gleichwertige Synchronisation auf JNI Referenzen durchzuführen. Mit der MonitorEnter Funktion wird der Monitor gestartet und mit der MonitorExit Funktion wird der Monitor wieder verlassen: [1]

```
if ((*env)->MonitorEnter(env, obj) != JNI_OK) {
        ... /* error handling */
}
...         /* synchronized block */
if ((*env)->MonitorExit(env, obj) != JNI_OK) {
        ... /* error handling */
};
```

7.2 Monitor Wait und Notify

Die Java-API enthält eine Reihe weiterer Methoden, die nützlich für die Thread-Synchronisation sind, nämlich Object.wait, Object.notify, und Object.notifyAll. Es werden keine JNI-Funktionen zu Verfügung gestellt, die direkt diesen Methoden entsprechen, weil Monitor wait- und notify-Operationen nicht so Performancekritisch wie Monitor-Entry und Monitor-Exit Operationen sind. Nativer Code kann stattdessen den JNI Methodenaufrufmechanismus verwenden, um die entsprechenden Methoden in der Java-API aufrufen: [1]

```
/* precomputed method IDs */
static jmethodID MID_Object_wait;
static jmethodID MID_Object_notify;
static jmethodID MID_Object_notifyAll;
void
JNU_MonitorWait(JNIEnv *env, jobject object, jlong timeout)
{
        (*env)->CallVoidMethod(env, object, MID_Object_wait,timeout);
}
void
JNU_MonitorNotify(JNIEnv *env, jobject object)
{
        (*env)->CallVoidMethod(env, object, MID_Object_notify);
}
void
JNU_MonitorNotifyAll(JNIEnv *env, jobject object)
{
        (*env)->CallVoidMethod(env, object, MID_Object_notifyAll);
}
```

Wir gehen davon aus, dass die Methode-IDs für Object.wait, Object.notify und Object.notifyAll bereits anderweitig ermittelt und in globalen Variablen zwischengespeichert wurden. Wie in der Programmiersprache Java, können die oben genannten Monitor-Funktionen nur aufgerufen werden, wenn man den Monitor hält, der mit dem jobject Argument assoziiert ist. [1]

8 Von nativen auf Java Code zugreifen

Es wurde bereits gezeigt, wie nativer Code, welcher aus Java gestartet wurde, wiederum auf Java-Funktionen zugreift.

Dies ist auch aus C/C++ Applikationen aus möglich. Jedoch müssen hier einige Punkte berücksichtigt werden:

Java-Bytecode kann bekannter Weise nur in einer JVM ausgeführt werden. Da eine VM typischerweise native implementiert ist, ist es problemlos möglich, aus einer beliebigen C Applikation auf dessen Library zuzugreifen. Man führe sich nur kurz das *java* Kommando vor Augen. Ein einfaches C Programm, das mit der JVM gelinkt ist. Es startete eine JVM und ruft anschließend die main-Methode der über die Kommandozeile übergebenen Klasse auf. Es ist also möglich, aus einer C/C++ Applikation eine JVM zu starten. Die Methoden, welche anschließend aufgerufen werden, können frei gewählt werden; es gibt keinen Grund, warum es die main-Methode sein muss. Dabei ist wieder zu beachten, dass auch hier die Datenkapselung von Java umgangen wird. Auf diese Problematik wurde bereits bei 4.4 eingegangen. [1]

I. Abbildungsverzeichnis

Abbildung 1. JNI – Überblick [1] 5
Abbildung 3. Schritte zur Entwicklung einer JNI Anwendung [1] 6
Abbildung 4. Übersicht JNIEnv Pointers 8
Abbildung 5. Verschiebung von Objekten, während nativer Code eine Referenz hält 10

II. Tabellenverzeichnis

Tabelle 1. Mapping von primitiven Datentypen [1] 9

III. Literaturverzeichnis

Liang, S. (1999). *The Java Native Interface*. Reading, Massachusetts: Addison Wesley Longman, Inc.[1]