Institut für Computertechnik

TU Wien / FAK Elektrotechnik

Projektdokumentation

EDV Orientierte Projektarbeit

LVA-Nr: 384.857

SS 2003

Abhandlung zum Java Native Interface (JNI)

Betreuer: DI Gerhard Pratl

Wien, 24. Juli 2003

Projektbearbeiter:

Matr.Nr.	Vor- und Zuname	Tel.Nr.	Email	
9425090	Klaus Salamonsberger	0699/11093099	klaus@sely.org	

Inhaltsverzeichnis

1. Einleitung

Die Abkürzung JNI steht für Java Native Interface. Dabei handelt es sich um eine Schnittstelle, die es Java ermöglicht, nicht in Java implementierte Funktionen, externe Funktionen genannt, zu verwenden. Dies ist z. B. dann notwendig, wenn Java plattformabhängige Funktionen nicht zur Verfügung stellt. Es kann aber auch dann Verwendung finden, wenn Algorithmen bereits in einer anderen Sprache implementiert wurden und somit nur noch eingebunden werden müssen. Ein weiterer Punkt ist, dass zeitkritischer Code in einer hardwarenäheren Sprache implementiert werden sollte, da er performanter ausgeführt werden kann; hardwarenähere Sprachen besitzen aufgrund des geringeren Abstraktionslevels einen geringeren Overhead und bieten somit einen Geschwindigkeitsvorteil [JNI2 pp2].

In der Version 1.0 von Java wurde nicht das JNI, sondern das NMI (Native Method Interface) verwendet. Auch von Netscape und Microsoft wurde je eine Möglichkeit zum Verwenden von Code aus anderen Programmiersprachen entwickelt; das Java Runtime Interface (JRI) von Netscape und das Raw Native Interface (RNI) und das Java/COM Interface von Microsoft. Diese wurden jedoch alle aus Kompatibilitätsproblemen bei der Einbindung in die diversen Java Virtual Machines (JVM) in den nachfolgenden Versionen von Java (beginnend bei Java 1.1) vom JNI abgelöst [TIJ1 pp945-954].

Damit in Java externe Funktionen aufgerufen werden können, müssen diese in den Klassen mittels native deklariert werden. Soll eine externe Methode in Java verwendet werden, ist jedoch darauf zu achten, dass rechtzeitig die zugehörige Bibliothek in den Speicher geladen wird. Erfolgt dies nicht, kann sie nicht geladen werden oder enthält die geladene Library die angeforderte externe Methode nicht, wird ein UnsatisfiedLinkError ausgelöst. Das Laden selbst erfolgt mittels der Methode loadLibrary aus der Klasse System. Damit die zu ladende Bibliothek gefunden werden kann, muss sie sich entweder in einem Systempfad oder im Pfad der Anwendung befinden. Es ist auch zu berücksichtigen, dass nur der Name der Bibliothek ohne deren Erweiterung (in MS-Windows „.DLL“, oder in Solaris und LINUX „.so“) und ohne deren etwaigen Präfix (in LINUX „lib“) anzugeben ist. Dies ist

deshalb notwendig, um eine Unabhängigkeit von dem verwendeten Betriebssystem zu erreichen (siehe die angegebenen Beispiele). Sollte ein Betriebssystem keine dynamischen Dateien unterstützen, müssen alle externen Methoden (`native`) mit der JVM (Java Virtual Machine) vorgelinkt werden. Damit beinhaltet bereits die JVM diese externen Methoden, und ein Laden einer zugehörigen Library wäre nicht mehr nötig. Aus Kompatibilitätsgründen sollte in Java jedoch der Befehl `loadLibrary` trotzdem implementiert werden, um eine Plattformunabhängigkeit im Java-Code gewährleisten zu können [JNI2 pp1-25].

2. Funktionalität

Das JNI ermöglicht den externen Funktionen folgende Aktivitäten:

- Erstellen, Untersuchen und Bearbeiten von Java-Objekten
- Aufrufen von Java Methoden
- Abfangen und Auslösen von Exceptions
- Laden und Anfordern von Klasseninformationen
- Untersuchen der Typen während der Laufzeit

Der Zugang zu den Objekten im Java wird dem JNI über den so genannten `JNI Interface Pointer` ermöglicht. Dabei handelt es sich um einen Zeiger auf eine Datenstruktur, welche einen Zeiger auf die möglichen JNI-Funktionen und die für den zugehörigen Thread notwendigen Daten enthält (*siehe Abbildung 2-1*). Es ist dabei unbedingt zu beachten, dass der `JNI Interface Pointer` genau einem Thread zugeordnet ist. Er darf auf keinen Fall an andere Threads übergeben werden, da die „notwendigen Daten pro Thread“ genau diesem einen Thread zugeordnet sind [JNI2 pp10].

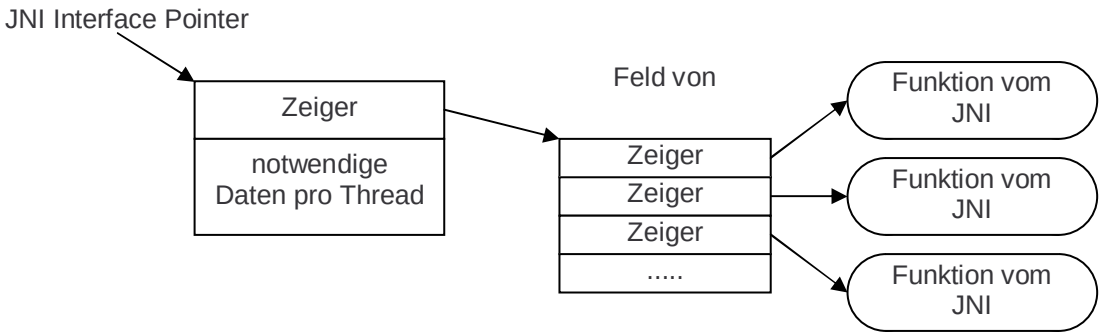

Abbildung 2-1 JNI Interface Pointer

Der `JNI Interface Pointer` wird den Funktionen in der DLL als erster Parameter übergeben. Er besitzt den Datentypen `JNIEnv` (JNI-Environment), der im JNI definiert ist und den Zugriff auf alle JNI-Funktionen ermöglicht. Weiters enthält jede externe Methode als zweiten Parameter einen Zeiger, welcher das Aufrufende Java-Objekt referenziert. Handelt es sich dabei um eine statische Methode ist es

eine Referenz der zugehörigen Klasse, andernfalls der zugehörigen Instanz. Alle weiteren Parameter sind jene, welche die in Java definierte Methode enthält; auch der Rückgabewert ist jener von der Funktion in Java.

Den im JNI verwendeten primitiven Datentypen aus Java wird das Zeichen „`j`“ zum Namen in Java vorangestellt (*siehe Tabelle 3-1*). Durch diese eigens definierten Datentypen für das JNI kann gewährleistet werden, dass die Datentypen in allen verwendeten Systemen die gleiche Größe besitzen. Wie die Erfahrung zeigt, ist dies nicht immer so, denn auf einem PC-System kann z. B. in C ein Integer 16 bit Breit oder aber auch 32 bit breit sein; dies hängt von der verwendeten Hardware ab. Um dieses Problem zu umgehen, sollten in einer Bibliothek, welche das JNI unterstützt, immer die eigens vom JNI definierten Datentypen verwendet werden. Für komplexere Datentypen wie z. B. String oder Felder gibt es ebenfalls vordefinierte Datentypen (*siehe Abbildung 2-2*).

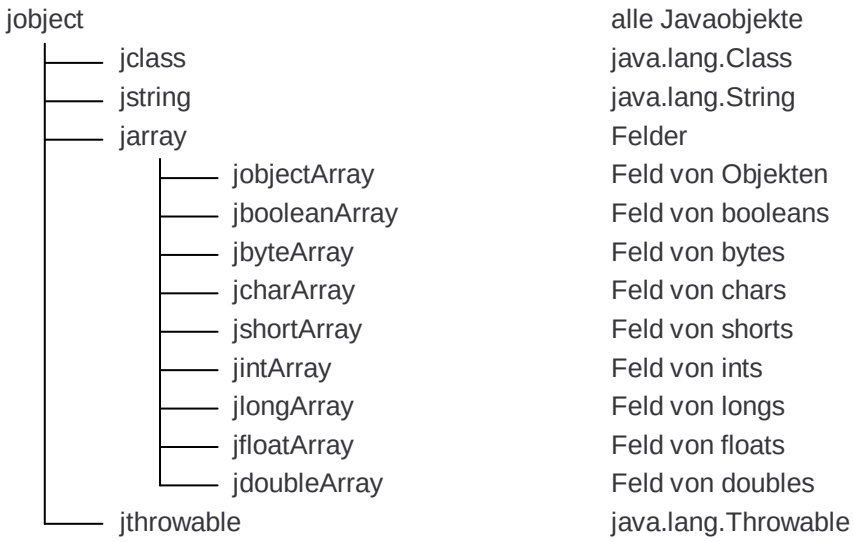

Abbildung 2-2 definierte Datentypen im JNI

Da C nicht die Möglichkeiten einer objektorientierten Sprache besitzt, ist ein `jobject` genauso wie ein `jobjectArray` definiert. Beim C++ hingegen handelt es sich um eine objektorientierte Sprache, daher kann in dieser Sprache die in *Abbildung 2-1* gezeigte Abhängigkeit tatsächlich auch aufgelöst werden.

Soll eine externe Methode in Java verwendet werden, muss als erstes die zugehörige Bibliothek geladen werden. Dies erfolgt mittels der Methode `loadLibrary` aus der Klasse `System`. Dabei ist zu beachte, dass sich die

Bibliothek entweder in einem Systempfad oder im Pfad der Anwendung befinden muss. Andernfalls kann Java die angeforderte DLL nicht finden. Es ist auch zu berücksichtigen, dass nur der Name der DLL ohne deren Erweiterung (in MS-Windows .DLL, oder in Solaris .so) anzugeben ist, da die Erweiterungen in unterschiedlichen Betriebsystemen unterschiedlich sein können und somit aus Kompatibilitätsgründen von der JVM (Java Virtual Machine) eingefügt werden. Sollte ein Betriebssystem keine dynamischen Dateien unterstützen, müssen alle Methoden, welche in Java native sind, mit der JVM vorgelinkt werden. Damit beinhaltet bereits die JVM diese externen Methoden, und ein Laden einer zugehörigen Library wäre nicht mehr nötig, wird aber im Java-Code trotzdem implementiert, jedoch mit dem Unterschied, dass in diesem Fall die Methode `loadLibrary` real nichts mehr zu tun hat [JNI2 pp1-25].

3. Das Tool JAVAH

Das Tool JAVAH ist in dem Funktionsumfang des Java Development Kits (JDK) enthalten. Es ermöglicht die Erstellung einer Headerdatei für C aus einer bestehenden Javaklasse. Daher müssen Funktionen, welche aus der zu erstellenden DLL aufgerufen werden sollen, zuerst in Java definiert werden. Eine externe Methode in Java wird als *native* deklariert und kann definitionsgemäß keinen Body besitzen, da sie nicht in Java ausprogrammiert wird.

Wird in Java eine Methode aufgerufen, welche native ist, muss zuvor dafür gesorgt werden, dass die zugehörige DLL-Datei in den Speicher geladen wurde. Ansonsten wird der Aufruf dieser Methode mit der Signalisierung eines Errors abgebrochen. Dabei handelt es sich um den `UnsatisfiedLinkError`, welcher besagt, dass die aufgerufene Methode nicht ausgeführt werden kann, da sie nicht lokalisiert werden konnte.

Das Tool JAVAH sorgt dafür, dass die externen Methoden, welche in der DLL implementiert sein sollen, korrekt definiert werden. Es erstellt automatisch die Funktionen mit deren richtigen Namen und den passenden Parametern. Weiters sorgt es dafür, dass auch Konstanten übersetzt werden.

Eine Funktionsdefinition für eine DLL im JNI hat folgende Merkmale zu erfüllen:

- Jeder Funktionsname muss mit dem Präfix „`Java_`“ beginnen.
- Daran muss der gesamten Paketname anschließen, wobei die einzelnen Teile mit einem „`_`“ zu trennen sind.
- Abgeschlossen wird der Funktionsname mit dem eigentlichen eindeutigen Namen der Java-Methode, welche mit native deklariert wurde.
- Handelt es sich um eine Methode, welche überladen wird, sind noch die beiden Trennzeichen „`__`“ und die anschließende Argumentensignatur anzufügen. Andernfalls kann auf die längere Funktionsdefinition verzichtet werden.

Mit diesen Merkmalen würde z. B. ein gültiger Funktionsname für das JNI von der folgenden Klassendefinition

```
package at.ict.net;
class IEEE1394SocketAPI {
    int native private nativeIsNodeIDValid (int i);
}
```

folgendermaßen aussehen:

```
Java_at_ict_net_IEEE1394SocketAPI_nativeIsNodeIDValid__I.
```

Im JNI kann es immer wieder vorkommen, dass eine Argumentensignatur anzugeben ist. Das erste Beispiel ist bereits bei der Funktionsdefinition vorgekommen. Sie ist aber auch dann von Relevanz, wenn von der DLL aus eine Java-Methode aufgerufen werden soll (siehe z. B. `Call<type>Method` in [JNI1 pp48]). Primitive Datentypen, wie z. B. `int` oder `long`, werden durch jeweils ein Zeichen repräsentiert, komplexere hingegen durch einen String, welcher mit einem „L“ beginnt und mit einem Strichpunkt endet. Dazwischen hat der gesamte Name der Methode, also Paketname mit Klassennamen und Methodenname, jeweils durch ein „/“ getrennt zu stehen. Für den Type String ergibt sich somit die Signatur „Ljava/lang/String;“. In der *Tabelle 3-1* sind die von Java unterstützten primitive Datentypen in Java, deren Signaturzeichen, Speicherbedarf und Namen im JNI dargestellt:

Datentyp	Signatur	Speicherbedarf	Name im JNI
void	V		void
boolean	Z	8 bit (ohne Vorzeichen)	jboolean
byte	B	8 bit (mit Vorzeichen)	jbyte
char	C	16 bit (ohne Vorzeichen)	jchar
short	S	16 bit (mit Vorzeichen)	jshort
int	I	32 bit (mit Vorzeichen)	jint
long	J	64 bit (mit Vorzeichen)	jlong
float	F	32 bit	jfloat
double	D	64 bit	jdouble

Tabelle 3-1 primitive Datentype und deren Signaturzeichen

Handelt es sich bei dem Datentypen um ein Feld, ist der zu verwendenden Struktur eine öffnende eckige Klammer („[“) voranzustellen. Weiters sind bei der Angabe von Methodennamen die Parameter zwischen zwei runden Klammern einzuklammern und der Rückgabewert ist an die schließende Klammer anzuhängen. Für das oben angegebene Beispiel würde die Methodensignatur folgendermaßen aussehen:

```
(I)I
```

Da in einem Funktionsnamen jedoch nicht alle Zeichen erlaubt sind, wie z. B. ein Strichpunkt oder ein „_“, müssen für diese Fälle spezielle Zeichenkombinationen eingeführt werden. Allgemein kann ein spezielles Unicode-Zeichen mit der Zeichensequenz _0xxxx ersetzt werden. Die vier „x“ sind dabei als vier beliebige Ziffern aufzufassen (z. B. _00065 würde das Zeichen „A“ repräsentieren). Es gibt aber auch die Möglichkeit, häufig verwendete Zeichen mit noch kürzeren Zeichenfolgen darzustellen (*siehe Tabelle 3-2*) [JNI2 pp1-25].

Zeichenfolge	repräsentierendes Zeichen
_0xxxx	Unicode des Zeichens xxxx
_1	_
_2	;
_3	[

Tabelle 3-2 repräsentative Zeichen

4. Referenzen zwischen JNI und Java

Es gibt einen bedeutenden Unterschied zwischen der Bezeichnung externe Methode und JNI-Funktion. Man spricht von einer externen Methoden immer dann, wenn es sich um den Code einer in Java als native deklarierten Methode handelt. Eine JNI-Funktion hingegen ist eine Funktion, welche das JNI dem Programmierer zur Verfügung stellt, um mit der VM zu kommunizieren.

Sollen von Java aus Teile der DLL referenziert werden, ist dies einfach mittels einer externen Methode möglich. Dieser können Parameter zur Behandlung übergeben werden, und die Ergebnisse können entweder über den Rückgabewert oder die Parameter wieder entgegengenommen werden. Ein Referenzieren in dieser Richtung wurde somit bereist über die Definition der externen Methode ermöglicht. Ein Problem könnte jedoch die Rückgabe der Parameter sein. Der einfachste Fall ist, dass man den Rückgabewert der Funktion verwendet. Da dieser jedoch nur ein primitiver Datentyp sein kann, sind die Möglichkeiten stark eingeschränkt und werden meist nicht ausreichen. Primitive Datentypen als Parameter werden immer als Übergabeparameter behandelt, was bedeutet, das diese in der externen Methode nicht verändert werden können. Objekte können jedoch verändert werden, da eine Referenz auf diese übergeben wird, und sich somit eine Veränderung auch auf die aufrufende Methode auswirkt. Um auch primitive Datentypen zurückgeben zu können, gibt es zwei Möglichkeiten: Als erstes ist es natürlich möglich, alle Parameter in einem Objekt zusammenzufassen und als ein Objekt zu übergeben. Dies würde jedoch dazu führen, dass für dieses Objekt natürlich eine eigene Klasse definiert werden müsste, was zu einem unerwünschten Aufwand führen würde. Die zweite Möglichkeit besteht darin, alle primitiven Parameter, welche Durchgangsparameter sein sollen, als Felder der Größe eins zu übergeben. Da Felder auch Objekte sind, wird an die externe Methode eine Referenz übergeben, womit die gewünschte Art der Parameterübergabe ermöglicht wird..

Es wird jedoch auch nötig sein, von der DLL aus Java-Objekte anzusprechen. Für diesen Fall liegt die Situation nicht so einfach, denn es würde wenig Sinn machen, in der DLL externe Funktionen zu definieren, welche dann in Java implementiert sind. Daher gibt es etliche JNI-Funktionen, welche es ermöglichen sollen, auf die Objekte

von Java zugreifen zu können. So ist es mit ihnen z. B. möglich neue Objekte zu erzeugen, oder auch Methoden von Klassen in Java aufzurufen.

Wie nun einzelne Objekte und Methoden von einer externen Methode heraus angesprochen werden können, wird hier nicht im Detail erläutert, da es den Rahmen dieser Arbeit sprengen würde (siehe [JNI2]), es wird nur erwähnt, dass immer sogenannte Identifier zu den gewünschten Objekten mittels speziellen JNI-Funktionen ermittelt werden müssen. Über diese Identifier ist es dann möglich, mit anderen JNI-Funktionen die gewünschten Objekte anzusprechen. Wichtig ist auch noch, dass immer unterschieden werden muss, ob es sich um eine oder keine statische Methode handelt. Denn es muss auch bei den zu verwenden JNI-Funktionen dieser Unterschied mittels unterschiedlichen Funktionsnamen berücksichtigt werden.

JNI Referenzen können in zwei Kategorien eingeteilt werden. Auf der einen Seite gibt es lokale Referenzen, welche nur solange existieren, solange die externe Methode existiert. Auf der anderen Seite gibt es globale Referenzen. Diese existieren unabhängig von der externen Methode. Objekte, welche einer externen Methode übergeben werden, und jene welche eine JNI-Funktion zurückgibt, werden immer mit lokaler Referenz übergeben. Es wird einer externen Methode jedoch ermöglicht, eine lokale Referenz in eine globale umzuwandeln. Einer JNI-Funktion können sowohl lokale wie auch globale Referenzen übergeben werden, so wie auch eine externe Methode globale und lokale Referenzen an den Aufrufer zurückgeben kann. Werden globale Referenzen verwendet, muss darauf geachtet werden, dass diese Objekte auch wieder von den externen Methoden vernichtet werden, da die *garbage collection* diese Objekte nicht mehr vernichten kann, da diese nicht wissen kann, ob dieses Objekt noch in einer externen Methode in Verwendung ist.

Wichtig ist auch jener Umstand, dass JNI-Funktionen keine Fehler des Programmierers wie z. B. ungültige Parameter oder falsch übergebene Objekte abfangen. Der Grund dafür liegt in der ansonsten sinkenden Performance, und dass dem JNI oft nicht genügend Informationen für eine derartige Überprüfung vorliegen. In einem Fehlerfall kann das JNI nicht dafür garantieren, dass es Fehlerfrei arbeitet. Im schlimmsten Fall kann auch zu einem Absturz der VM oder des Systems kommen [JNI2 pp1-25].

5. Fehlerbehandlung im JNI

Die meisten JNI-Funktionen verwenden zur Signalisierung von Fehlern Exceptions und Errors aus Java. Häufig kann auch aus dem Rückgabewert der JNI-Funktion auf einen auftretenden Fehler geschlossen werden, indem der Rückgabewert einem speziellen Wert entspricht, z. B. null. Diese Funktionen werfen zusätzlich auch noch eine Exception oder einen Error, mit welchem der Fehler genauer beschrieben wird. Das JNI stellt die Funktion

```
ExceptionOccurred ()
```

zur Verfügung um eine aufgetretene Exception abfragen zu können. Es gibt zwei Fälle, in welchen dem Programmierer dringend angeraten werden muss, diese Funktion unbedingt aufzurufen. Der erste Fall ist nach dem Aufruf einer Java-Methode aus der externen Methode heraus, da jede Methode eine Exception verursachen könnte. Der zweite Fall ist bei der Verwendung von Feldern, denn einige JNI-Funktionen zur Behandlung von Feldern geben keinen Wert an den Aufrufer zurück. Aber es könnte z. B. eine *ArrayIndexOutOfBoundsException* auftreten, welche aber wegen dem nicht vorhandenen Rückgabewert nicht erkannt werden kann. Eine aufgetretene Exception muss jedoch unbedingt behandelt werden, ansonsten wird für keine korrekte Funktion des JNI garantiert. Gibt eine JNI-Funktion einen Wert zurück, und aus welchem erkannt werden kann, dass kein Fehler aufgetreten ist, muss die Funktion zur Behandlung der Exceptions nicht aufgerufen werden.

Es kann aber bei einem multithreadfähigen System sehr sinnvoll sein, an mehreren Stellen nach möglichen aufgetretenen Fehlern nachzufragen, da z. B. eine Exception in einem Thread ausgelöst wird, welche einen anderen beeinflussen könnte.

Ist eine Exception aufgetreten kann diese in der externen Methode auf zwei Arten behandelt werden. Entweder wird ein Abbruch der externen Methode durchgeführt, was ein Weiterleiten der Exception an die aufrufende Java-Methode zur Folge hätte. Oder mit der JNI-Funktion

```
ExceptionClear()
```

kann die aufgetretene Exception abgefangen werden. In diesem Fall sollte eine Behandlung des aufgetretenen Fehler in der externen Methode erfolgen. Die Funktion ExceptionClear muss jedoch vor dem Aufruf jeder weiteren JNI-Funktion erfolgen.

Mit einer weiteren JNI-Funktion zur Fehlerbehandlung (ExceptionDescribe) kann die genaue Beschreibung des Fehlers abgerufen werden.

Tritt eine mehrfache Exception auf, können nur die eben genannten JNI-Funktionen (ExceptionOccurred, ExceptionClear und ExceptionDescribe) gefahrlos aufgerufen werden.

Eine externe Methode hat natürlich auch die Möglichkeit selbst Exceptions zu werfen. Dies wird dem Programmierer mittels den JNI-Funktionen

`Throw ()` und

`ThrowNew()`

ermöglicht. Damit ist es möglich, direkt von einer externen Methode, Exceptions zu generieren [JNI2 pp17-20] [TIJ1 pp952-953].

6. JNI und multithreading

Es muss erwähnt werden, dass Probleme, welche Multithreading mit sich bringt, wie z. B. Zugriff auf kritische Objekte, vom Programmierer zu verhindern sind. Eine einfache Variante ist, die externen Methoden als synchronized zu deklarieren. Dadurch wird automatisch das Problem auf die JVM abgeschoben, da in diesem Fall diese für eine Synchronisierung zu sorgen hat. Es besteht auch die Möglichkeit, dass in der externen Methode ein Code implementiert ist, welcher eine etwaige Synchronisation vornimmt.

In Multithreadingsystemen ist besonders darauf zu achten, dass der JNI Interface Pointer nicht an andere Threads übergeben werden darf, da der genannte Pointer auf eine Datenstruktur zeigt, welche threadabhängig ist [TIJ1 pp953] [JNI2 pp10].

Literaturverzeichnis

JAVA1 Java™ 2 Platform: Standard Edition, v1.2.2 API Specification

JAVA2 P. Heller, S. Roberts: Java 1.2, Developer's Handbook, Sybex Inc., Seite 283-356 und 411-521, 1999

JNI1 Sun: Java Native Interface - Tutorial, http://java.sun.com/docs/books/tutorial/native1.1/index.html am 6.3.2000

JNI2 JavaSoft: Java Native Interface Specification Release 1.1, http://sunsite.sut.ac.jp/java/docs/jdk1.1/jni.pdf, 6.3.2000, 1997

TIJ1 B. Eckel: Thinking in Java, 2nd edition, rev 3, http://www.codecuts.com/ads-cgi/viewer.pl/codecuts/pdfs/bruceeckel/TIJ2R3.pdf am 6.3.2000, 1999

Abbildungsverzeichnis

Tabellenverzeichnis