E2 Java Virtual Machine und Bytecode

Hier sollen eher technische Informationen zusammengesfasst werden, die für die Entwicklung von Java-Programmen nicht direkt benötigt werden, jedoch zu einem *vertieften Verständnis* von Abläufen, Konfigurationen und Strukturen beitragen.

E2.1 Der Bauplan der Java Virtual Machine (JVM)

Durch die JVM wird der Java-Bytecode interpretiert und umgesetzt. Jedoch werden während der Abarbeitung des Programms durch die JVM zahlreiche weitere wichtige Leistungen automatisch erbracht, die insbesondere zur Robustheit und Sicherheit der Programmausführung beitragen. Dem Entwickler von Java-Programmen werden dadurch teilweise Aufgaben abgenommen, die in anderen Programmiersprachen explizit zu implementieren sind. Zudem übernimmt die JVM auch Überprüfungen, die Sicherheitsaspekte betreffen.

Seit dem J2SDK 1.4. wird die JVM als sogenannte Java Hot Spot VM von Sun in zwei Versionen ausgeliefert, die sich nur im JIT-Compiler unterscheiden: Es existiert ein für Server-Anwendungen und ein für Client-Anwendungen optimierter JIT-Compiler bei ansonsten identischer JVM. Die Server VM wurde optimiert zur Ausführung langlaufender Server-Anwendungen, bei denen eine hohe durchgängige Ausführungsgeschwindigkeit wichtiger ist als sparsamer Speicherverbrauch und niedrige Startzeit. Dagegen hat die Client VM Laufzeit-Vorteile bei Desktop-Anwendungen und Applets; sie ist dafür ausgelegt, die Startzeit und den Speicherverbrauch (insbesondere bei Grafik-Applikationen) zu minimieren.

E2.1.1 Komponenten der JVM

Während die JVM dem Anwender "von außen" als monolithisches Gebilde erscheint, hat sie doch eine detaillierte Struktur: Zahlreiche Einzelprogramme und –prozesse wirken bei der Abarbeitung von Java-Programmen zusammen, wie aus Abbildung E2.1 hervorgeht. Jede der einzelnen Komponenten übernimmt bestimmte Aufgaben.

- **Execution Engine:** Sie ist das "Herz" der JVM, der virtueller Software-Prozessor der virtuellen Maschine. Hier werden die im Bytecode enthaltenen Instruktionen der JVM-Spezifikation ausgeführt. Die JVM kann als Software (J2SDK) oder Hardware (Spezialchips) realisiert sein.
- **Class Loader:** Wenn im Laufe der Programmausführung weitere (z.B. importierte) Klassen benötigt werden, so besorgt der Class Loader der JVM das Lokalisieren, Laden, Überprüfen und Initialisieren der für die weitere Programmausführung benötigten Class-Files, die den Bytecode enthalten. Der Class Loader arbeitet dynamisch, d.h. erst wenn dann, wenn ein Class-File (bzw. die darin enthaltene Klasse) wirklich benötigt wird, wird sie vom Class Loader zur Laufzeit geladen. Das Java System verfügt über keinen statischen Linker – der dynamische

Class Loader wird zur Laufzeit des Programms tätig. Beim Laden sucht der Class Loader zunächst im Systempfad und in allen bei der JVM eingetragenenen Pfaden nach zu ladenden Klassen. Inerhalb des J2SE gibt es die abstrakte Klasse `java.lang.ClassLoader`, von der anwendungsspezifische Klassenlader ableitbar sind. Sollen Klassen (auch aus JAR-Files) im Netz lokalisiert und geladen werden, kann die Klasse `java.net.URLClassLoader` verwendet werden.

- **Garbage Collector:** Objekte werden von Programmen durch den Befehl "new" im Hauptspeicher auf dem Heap angelegt. Die erneute Freigabe des belegten Speichers regelt jedoch nicht das Java-Programm selbst (es gibt keine "delete"-Anweisung), sondern wird durch die periodisch laufende Speicherverwaltung der JVM geleistet (s.u.).

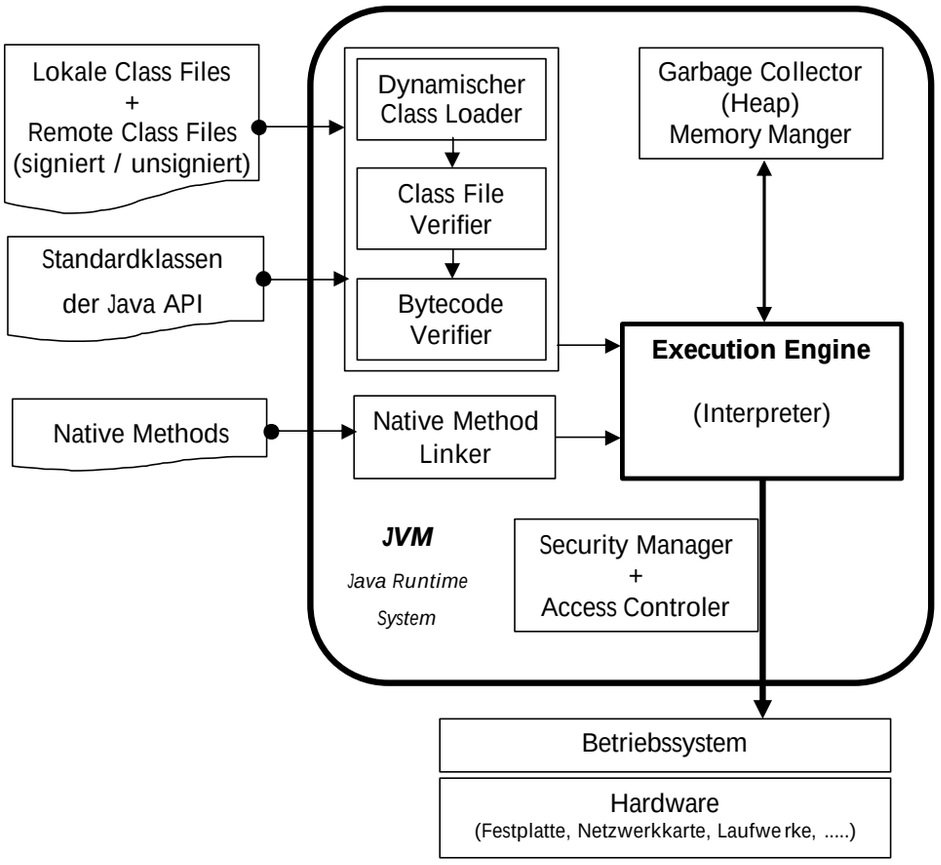

Abb. E2.1: Die Anatomie der Java Virtual Machine

- **Native Method Linker:** Er dient dem Aufruf von Routinen, die als plattformspezifischer komplilierter Maschinencode vorliegen, dh nicht als Java-Bytecode. Es handelt sich um Kompilate anderer Sprachen (z.B. mit C oder C++ programmier-

te DLLs und Libs). Das Laden des Codes passiert ebenfalls dynamisch beim Programmlauf, wenn solche Routinen aufgerufen werden.

- **Security Manager:** Für das Java-Laufzeitsystem können Sicherheitsrichtlinien (Rechte und Verbote für Java-Programme) defniert werden. Deren Einhaltung wird bei der Programmausführung vom Security Manager durchgesetzt. Dadurch unterliegen Java-Programme einer Überwachung durch die JVM und spezifische Restriktionen werden eingehalten (z.B.: Kein Zugriff auf das lokale Filesystem, kein Konnektieren an Fremdrechner via Netzwerk, kein Übertragen von Benutzerdaten, kein Abbrechen der laufenden JVM, etc.).

Anmerkung: Nachteilig ist der Class Loader Mechanismus, wenn zahlreiche Class-Files als Teil eines Applets benötigt werden. In diesem Fall müsste nach dem Laden der Webseite und des Applet-Class-Files wiederholt via Internet auf den Webserver zugegriffen werden, um zusätzliche Class-File sukzessive auf den Clientrechner herunterzuladen. Der wiederholte Netzzugriff ist zeitaufwendig und verzögerte die Ausführung des Java-Appelts. Vorteilhafter ist es, alle benötigten Class-Files in einem einzigen Java-Archiv (.jar-File) zusammenzustellen und dieses Archiv-File komplett zu laden, so dass keine nachträglichen Netzzugriffe mehr nötig sind.

Die aktuelle Java VM existiert zur Performanzoptimierung in zwei verschiedenen Auslegungen als *Java HotSpot Client VM* (Client VM) und *Java HotSpot Server VM* (Server VM). Die Server VM startet im Allgemeinen langsamer als die Client VM, läuft aber über längere Zeiten hinweg schneller als die Client VM. Auf diese Weise wird der Tatsache Rechnung getragen, dass auf einfachen Clients (lokalen Platzrechnern) meist viele kurzzeitig laufende Programme oft gestartet und beendet werden, während Serveranwendungen nach einmaligem Start lange ohne Unterbrechnung laufen. (Teil des J2SDK ist auch noch die ältere weniger performante *Classic VM*.)

Mit der J2SE 5.0 versucht *java* (Java application launcher) auf einigen Plattformen und Betriebssystemen beim Start einer Java-Anwendung festzustellen, ob die Ausführung auf einem starken *Server-Klasse*-Rechner (Definition der J2SE 5.0: mindestens zwei CPUs und 2GB RAM) stattfindet oder nicht (*server-class machine detection*). Ist dies der Fall, wird die Server VM verwendet, falls nicht die Client VM. Dieser Mechanismus greift, wenn nicht die *java*-Optionen `-client` oder `–server` spezifiziert wurden und arbeitet momentan für die Prozessorfamilien Intel-586 und Sparc-32bit auf Solaris und Linux. Dagegen ist auf Windows für Intel-586 die Client VM der Default.

Die aktuelle HotSpot-JVM verwendet adaptive *Just In Time (JIT) Kompilation* des Bytecodes: Die VM startet die Abarbeitung der Anwendung interpretativ mit einem Standard-Interpreter, analysiert aber im Hintergrund die laufenden Codestrecken auf Performanz-Engpässe; diese *hot spots* werden im Hintergrund kompiliert und danach nur noch deren Kompilat verwendet.

Performanz: In diesem Bereich versucht Microsoft Performanz-Vorteile für C# im Vergleich zu Java zu verbuchen: C#-Programme werden nie auch nur teilweise interpretiert, die CLR verfügt über keinen Interpreter; der im Falle C# produzierte Zwischencode wird bei Ausführung durch die CLR stets komplett durch einen Just in Time Compiler in Maschinesprache übersetzt. Um zusätzliche Übersetzungsvorgänge

zu vermeiden, werden einmal vom JIT-Compiler in Maschinensprache übersetzte Module im Dateisystem abgelegt; bei einer späteren erneuten Programmausführung werden sie nur noch geladen und können sofort ausgeführt werden. Durch die Fokussierung auf die Windows-Plattform erweisen sich C#-Programme auch im Bereich Grafik und Systemzugriff (Dateisystem, Schnittstellenzugriff) als schneller. Auch ist der Befehlssatz der CLR bewusst einfacher gehalten.

Garbage Collector (GC): Die Algorithmen des GC bestimmen *wesentlich* die Performaz von Java-Anwendungen, so dass diese stets weiterentwickelt werden. Der optimierte GC der HotSpot VM basiert auf Lebenszeit-orientierter Garbage Collection (*generational garbage collection*). Da in der Praxis viele Java-Objekte nur eine kurze Lebenszeit aufweisen und nur wenige Objekte sehr lange Lebenszeiten, wird der verwaltete Heap-Speicher vom GC logisch in mehrere "*Generationen*" aufgeteilt und separat bearbeitet. Die Bereiche der "jungen Generation" (kurzlebige Objekte) füllen sich schnell, jener der "alten Generation" (langlebige Objekte) langsamer. Objekte der jungen Generation, die eine bestimmte Lebensdauer überschreiten, werden bei GC-Läufen in Bereiche der älteren Generationen verlagert. Ist ein Speicherbereich aufgebraucht, wird auf ihm eine partielle Garbage Collection (minor collection) durchgeführt. Diese partielle Garbage Collection ist in der Regel viel schneller als eine komplette Garbage Collection des gesamten Heaps. Diese findet nur statt, wenn der Bereich der ältesten Objekte (tenured objects) gefüllt ist.

Auf Server-Klasse Maschinen kann auch die *Arbeitsstrategie des Garbage Collectors* (GC) durch *java*-Optionen konfiguriert (getunet) werden. So lassen sich initiale und maximale Grösse des verwalteten Heap-Speichers abweichend von Default-Einstellungen vorgeben. Ebenso kann der maximale Zeitanteil (relativ zur Gesamtzeit), der für Garbage Collection verwendet wird und der minimale Speicheranteil (relativ zur maximalen Heap-Grösse), der pro GC-Lauf freigesetzt wird bestimmen. Der VM kann mitgeteilt werden, dass GC-Unterbrechungen kürzer als eine bestimmten Zahl von Millisekunden oder ein Bruchteil der Gesamtausführungszeit erwünscht sind. Die VM wird versuchen, Heap-Grösse und GC-Parameter so anzupassen, dass dieses Ziel erreicht wird. Durch ungünstige Vorgaben wird die Zahl der GC-Läufe in der Regel zunehmen, der Systemdurchsatz durch häufige GC-Läufe reduziert werden und die Heap-Belegung zunehmen.

E2.1.2 Kontrolle der Sicherheit von Java-Programmen

Eines der Anwendungsgebiete von Java ist die Entwicklung verteilter Anwendungen und Internetapplikationen, bei denen Java-Bytecode via Intra- oder Internet an den ausführenden Rechner verschickt wird. Man spricht allgemein auch von "mobilem" Code, dh von Programmen, die in einer (möglichst gut portablen) Zwischencode-Darstellung via Netz an eine Vielzahl verschiedener, hetreogener Ziel-Plattformen gesendet und dort in gleicher Weise ausgeführt werden. Die Übersetzung in Maschinensprache geschieht dynamisch beim Client – mittels Interpretation oder möglichst effizienter Just In Time-Kompilation (JIT-Compiler).

Beim Client (Bytecode-Konsumenten) wird jedoch zuerst überprüft, ob der Bytecode den Sicherheitsanforderungen seines Formats genügt – man spricht von *Verifikation*. Erst nach erfolgreicher Verifikation darf der Bytecode auf dem Zielrechner übersetzt

und ausgeführt werden. Auf diese Weise soll der Client vor böswilligen Manipulationen des Bytecodes während der Übertragung geschützt werden, bzw vor Bytecode, der gar nicht durch den Java-Compiler sondern "von Hand" erstellt wurde – und das Sprachdesign von Java verletzt. Bei der Verwendung von mobilem Code auf ungeschützen Netzen ist die Absicherung gegen Manipulationsversuche entscheidend.

So enthält jedes in eine ("irgendwoher" aus dem Internet geladene) HTML-Seite eingebettete Applet ausführbaren Java-Bytecode. Nicht immer stehen dahinter vertrauenswürdige Autoren oder Institutionen. Somit besteht die potentielle Gefahr, dass von diesem Coding unerwünschte, gefährliche oder zerstörerische Wirkungen ausgehen, wie das Ausspähen oder Vernichten von Festplatteninhalten, das unerwünschte Übertragen von Daten an fremde Stellen oder das unkontrollierte Herstellen von Internetverbindungen.

Somit hat Java vom Standpunkt der Sicherheit neue Probleme aufgeworfen, stellt aber auch Mechanismen bereit, um die erforderliche Sicherheit zu gewährleisten. Bevor ein Java-Programm von der JVM ausgeführt werden kann, muss es deshalb mehrere Prüfungen durchlaufen, die in die JVM integriert sind. Dabei werden die J2SDK-Standard-API-Klassen (*bootstrap classes* im JAR-File `jre/lib/rt.jar`) stets als sicher (*trusted*) angesehen und keiner weiteren Prüfung unterzogen, während alle anderen ("selbstgeschriebenen") Klassen als Fremd-Klassen angesehen und überprüft werden. Dies gilt sowohl für Class-Files, die lokal auf dem jeweiligen Rechner installiert sind, als auch für Class-Files, die über das Netz (remote) auf den Rechner gelangen (zB in Applets). Beiteiligt sind (Abb E2.1):

- Der **Class File Verifier:** Er überprüft die Struktur des Class-Files hinsichtlich Konsistenz mit der offiziellen Formatspezifikation. Ferner wird geprüft auf Vorhandensein aller verwendeter Methoden und Klassen.
- Der **Byte Code Verifier:** Er prüft nach dem Laden des Bytecode-Files, ob sich der enthaltene Bytecode bei der Ausführung voraussichtlich korrekt verhalten wird, dh unter anderem ob korrekte Typisierungen und Operandenzugriffe vorliegen, keine Speicherverletzungen auftreten und Feld- und Stackgrenzen eingehalten werden (Arrayindex- und Nullreferenzüberprüfungen). Diese Sicherheitstests sind zeitintensiv und setzen die Ausführungszeit eines Programms deutlich herab, da sie eine vollständige Datenflussanalyse voraussetzen. Um zB zu überprüfen, ob jede Varaible vor der Benutzung wirklich initialisiert wurde, müssen alle möglichen Programmpfade durchlaufen und in jedem der Pfade nach Initialisierungs-Zuweisungen gesucht werden. Jedoch wird auf diese Weise garantiert, dass nur Bytecode ausgeführt wird, der die grundlegenden Regeln von Java nicht verletzt.
- Der **Security Manager:** Er kontrolliert ständig den Zugriff auf lokale Ressourcen des Betriebsystems (Netzwerkverbindungen, Dateien etc) und verbietet Aktionen, die die Sicherheit der Hardware und des Benutzers tangieren (Systemveränderung und Spionage). Bei unerlaubten ktionen werden Ausnahmen vom Typ Securityxception ausgelöst. Wie beim Class Loader gibt es auch beim Security Manger eine vorgefertigte Klasse (`java.lang.SecurityManger`), von der eigene Klassen abgeleitet werden können. Die Arbeitweise des Security Managers ist für den Rechner (ohne Programmieraufwand) konfigurierbar, es können so-

genannte "Security Policies" in Sicherheitsregel-Dateien (Policy Files) im ASCII-Format definiert werden. In diesen Dateien kann der Anwender sein eigene Sicherheitsstrategie definieren und Rechte vergeben. Eine Sicherheitsregel gewährt einer Quelle (Codesource, durch ihre URL spezifiziert) die gewünschten Rechte. Editierbar sind die Dateien von Hand oder mit Hilfe des Policy Tools des J2SDK (`policytool` im `bin`-Verzeichnis) mit grafischer Oberfläche. Beim Start liest die JVM die Sicherheitsregeln und wendet sie (in Zusammenarbeit mit dem Class Loader) auf jede Quelle individuell an. Somit können für verschiedene Quellen spezifische Schutzbereiche definiert werden, die zwischen der "strengen" Sandbox und vollem Vertrauen mit allen Zugriffsrechten angesiedelt sind.

Für die Ausführung von Applets wird dadurch das Sandbox-Prinzip durchgesetzt: Ein Appletprogramm bewegt sich bildlich wie in einem Sandkasten (abgeschottetem Bereich), dessen Grenzen es nicht verlassen kann; und in seinen Grenzen kann es nichts "Schlimmes" anrichten. Denn es verfügt nur über eng begrenzte Zugriffsmöglichkeiten auf die CPU und einen eigenen Anteil am Hauptspeicher sowie über Netzzugriff auf jenen Webserver, von dem es geladen wurde. Dagegen ist kein Zugriff auf das lokale Filesystem erlaubt und auch kein Netzzugriff auf Fremdserver. Ferner fürfen keine Betriebssystemkommandos oder externe Programme (wie der MS-DOS-Befehl format) aufgerufen werden. Nur durch ein "Zertifizieren" des Applets lassen sich diese Restriktionen lockern.

Die Sicherheitskontrollen garantieren, dass nur wirklich benötigte Class-Files in die JVM (und auf den Rechner) geladen werden und diese Class-Files ein korrektes Format und eine technisch einwandtreie Struktur haben. Nicht signierte Applet-Class-Files dürfen nicht auf kritische System-Ressourcen zuzugreifen oder gefährliche Aktionen ausführen, d.h. sind in ihrer "Sandbox" stets unter Kontrolle der JVM.

Um Applets mit erweiterten Zugriffsrechten auszustatten, müssen diese "zertifiziert" werden, wodurch Urheberschaft und integrität belegt wird. Dabei erhalten die zugehörigen Class-Files als Anhang eine digitale Signatur, aus der die Quelle des zugehörigen Codings hervorgeht. Vergeben werden die (kostenpflichtigen und jährlich zu verlängernden) Zertifikate von bekannten Institutionen und Firmen (z.B. Verisign), wobei jeder Verwender eines Zertifikats selbst registriert ist. Der Unterzeichner bürgt mit seinem Namen für die Korrektheit der Software. Einem zertifizierten Applet stehen auf dem Rechner deutlich erweiterte Zugriffsmöglichkeiten offen; auch schreibende und lesende Festplattenzugriffe sind nun erlaubt. Richtet der Appletcode dabei Schaden an, so weiß man dank Zertifikat, wer dafür haftbar zu machen ist.

E2.1.3 Die Grundstruktur des Java Bytecode

Der vom Java-Compiler aus dem Java-Programmtext erzeugte Bytecode kann als *Maschinensprache* der JVM angesehen werden. Und in der Tat verbirgt sich hinter dem Bytecode nichts anderes als eine *Menge elementarer Anweisungen*, die von der JVM verstanden und ausgeführt werden können. Diese elementaren Anweisungen (Instructions) werden als *Opcodes* (operational codes) bezeichnet. Weit mehr als hundert sind Teil der JVM-Spezifikation. Jeder Opcode hat einen leicht merkbaren, sprechenden *Namen* und eine *Nummer*. So ist zB der Opcode Nr. 96 der Befehl `iadd`

zur Addition zweier Integer-Werte. Jede Zeile Java wird in eine möglichst optimale Folge von Opcodes umgesetzt, die insgesamt den Bytecode darstellen.

Im eigentlichen Bytecode werden für die Namen der Opcodes deren Nummern eingesetzt, so dass die gesamte Anweisungsfolge als Zahlenfolge geschrieben wird. Die Zahlen werden binär dargestellt und die zugehörigen Bits (Binary Digits = Nullen und Einsen) in Bytes (Gruppen zu 8 Bits) aneinandergereiht.

Teil der J2SE ist das ***Tool*** **`javap`** (im *bin*-Verzeichnis), ein *Kommandozeilen-Disassembler* für Java-Bytecode. Dem Tool wird der Namen des `.class`-Files übergeben; ausgegeben werden Paketnamen, und enthaltene Klassen mit ihren Elementen (Attribute und Methodensignaturen). Interessant ist die *Option* `-c`: Diese veranlasst die *Ausgabe des Bytecodes* (mittels Opcode-Namen) aller enthaltenen Methoden. Die Option `-private` gibt auch alle privaten Attribute und Methodensignaturen aus. Die Option `-verbose` liefert noch weitere Informationen. Alle Aufrufoptionen gibt `javap -help` auf der Konsole aus.

Für ein File `Demo.class` lautet ein möglicher Aufruf:

```
javap -c -private Demo ↵
```

Somit lässt sich der Bytecodeaufbau betrachten und nachvollziehen. Ein auszugsweises, selbst kommentiertes Beispiel:

```
// Java-Quellcode-Anweisungen:
int x = 3;    int y = 5;     int z = x + y ;
// Zugehörige durch javap ausgegebene Opcode-Befehlsfolge:
0:   iconst_3   // Lege konstanten Integer-Wert 3 an
1:   istore_1   // Speichere diesen in Integer-Variablen Nr.1
2:   iconst_5   // Lege konstanten Integer-Wert 5 an
3:   istore_2   // Speichere diesen in Integer-Variablen Nr.2
4:   iload_1    // Lade Wert der Variable Nr.1 in den Stack
5:   iload_2    // Lade Wert der Variable Nr.1 in den Stack
6:   iadd       // Addiere Integerwerte aus Stackspeicher
7:   istore_3   // Speichere Ergebnis in Integer-Variablen Nr.3
```

Somit ist also auch der generierte Bytecode nichts Okkultes, sondern prinzipiell versteh- und entschlüsselbar. Allerdings unternimmt es natürlich kein Java-Entwickler, Bytecode zu lesen oder gar mühsam zu manipulieren. Dies ist die interne Aufgabe des Java-Compilers und der JVM.

E2.1.4 `native` – Einbinden von Nicht-Java-Methoden

Der Modifizierer `native` dient dazu, plattformspezifischen Binärcode in Java-Programmen aufzurufen. `Native`-Methoden einer Java-Klasse werden innerhalb der Klasse nur *deklariert*, während sich die eigentliche *Implementierung* in einer externen Bibliothek befindet – z.B. (unter Windows) einer in C oder C++ programmierten DLL (Dynamic Link Library) namens `TestLib.DLL`.

```
class NativeCall {
   public static void main( String[] args ){
      System.loadLibrary( "TestLib" );
      double erg = rechne( 3, 4.5 );
      IO.writeln( "Ergebnis:  " + erg );
   }
   // Deklaration der native Methoden-Schnittstelle:
   public static native double rechne( int i, double d );
}
```

Durch Aufruf der statischen Methode `System.loadLibrary()` und Übergabe des Bibliotheksnamens wird die externe Bibliothek `TestLib` geladen. In `main()` kann deren Methode `rechne()` nun aufgerufen und mit Parametern versorgt werden. Natürlich *verlieren* Java-Programme dadurch ihre *Plattformunabhängigkeit* – somit sollte man sich nur in Ausnahmefälle dieser technischen Möglichkeit bedienen.