Java Virtual Machine

1 Was ist die JavaVM

Die JavaVM könnte man als einen virtuellen Computer betrachten, der speziell dafür kompilierten Java-Code ausführen kann. Der Begriff "virtuell" ist daher angebracht, da die plattformabhängige VM das Laufenlassen von plattformunabhängigen Java-Code ermöglicht, also wird jeweils eine Art viruteller "Java-Computer" emuliert.

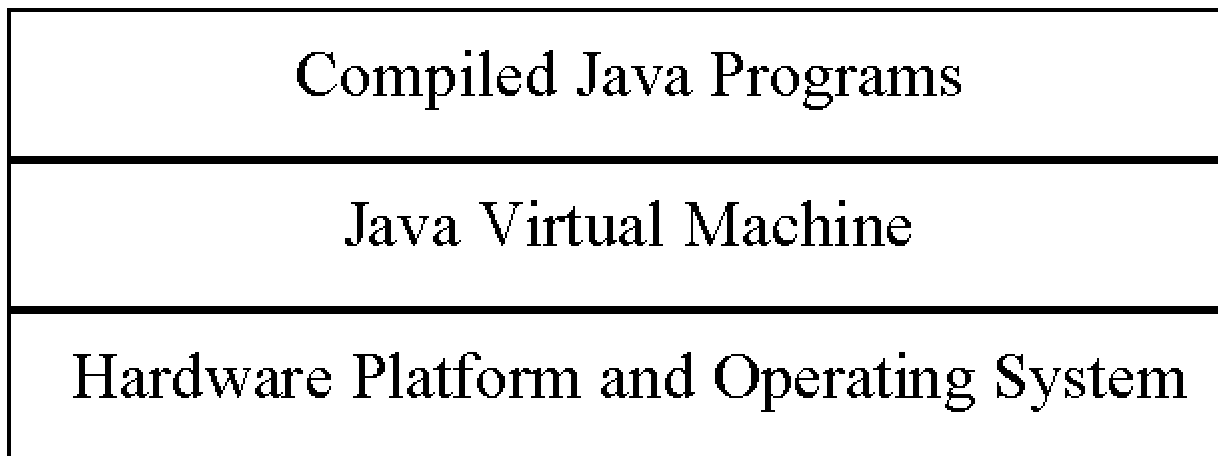

***Abbildung 1**: Einbettung der Java Virtual Machine*

Die JavaVM ist der zentrale Schlüssel zur Portierbarkeit von Java, da sie die Schnittstelle zwischen der Hardware, für die sie entwickelt wurde, und dem plattformunabhängigen Java-Code darstellt.

Es ist also nur noch nötig eine VM für jede Prozessorarchitektur zu entwickeln, während die fertigen Programme und das Java-Entwicklungssystem völlig plattformunabhängig sind.

Die fertigen Java-Programme liegen nach dem Kompilieren in einer Art Bytecode vor, den man wie eine Maschinensprache für die JavaVM betrachten kann. Der Java-Compiler, wandelt die java-Quellcodes (eine Art Textformat) in plattformunabhängige Class-Dateien in Bytecode um, welche dann von den VM's auf verschiedenen Architekuren ausgeführt werden können.

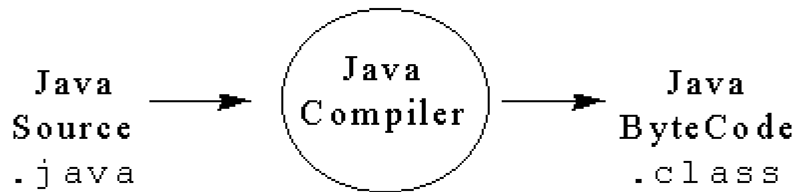

Abbildung 2*: Übersetzung von Java-Quellcode in Java-Bytecode*

2 Architektur

Die "virtuelle Hardware" der Java Virtual Machine kann man sich in vier Teile aufgeteilt vorstellen: **Register**, **Stack**, "Garbage-collected" **Heap** und **Method-Area**. Diese Teile sind genauso "virtuell" wie die Machine selbst, jedoch müssen sie in irgendeiner Form in jeder JavaVM vorhanden sein, um diese lauffähig zu halten.

Die JavaVM verarbeitet 32-bit Adressen, daher ist es auch möglich in der JavaVM bis zu 4-Gigabyte zu adressieren. Ebenso sind die Register der JavaVM 32-bit breit. Der Stack, Heap und die Method-Area befinden sich irgendwo innerhalb des 4-Gigabyte adressier-baren Speicherraums. Der exakte Ort dieser Speicheradressen bleibt jedoch den verschiedenen Implementationen der JavaVM vorbehalten.

Ein Datenwort in der JavaVM besteht immer aus 32-bits. Die JavaVM unterstützt **direkt** eine geringe Anzahl von primitiven Datentypen (siehe ***Tabelle 1***): byte (8-bits), short (16-bits), int (32-bits), long (64-bits), float (32-bits), double (64-bits) und char (16-bits). Mit Ausnahme von char, welcher ein unsigned Unicode Character ist, sind alle numerischen Typen signed. Alle Typen entsprechen denen die dem Java-Programmierer zur Verfügung stehen. Ein anderer primitiver Datentyp ist der "Object Handler", welcher aus einer 32-bit Adresse besteht, die sich auf ein Objekt auf dem Heap bezieht. Boolsche Datentypen werden auf Intergerkonstrukte abgebildet.

Datatype	Description
Byte	1-byte signed two's-complement integers
Short	2-byte signed two's-complement integers
Int	4-byte signed two's-complement integers
Long	8-byte signed two's-complement integers

Float	4-byte IEEE 754 floating-point numbers
Double	8-byte IEEE 754 floating-point numbers
Char	2-byte unsigned Unicode character
Object	4-byte referenced Java object
ReturnAddress	4-byte, used with jsr, ret, jsr_w, ret_w instructions

***Tabelle 1**: Datentypen der JavaVM*

2.1 Komponenten

Nachfolgend werden die vier virtuellen Teile der JavaVM erläutert.

2.1.1 Befehlssatz

Eine Anweisung des Java Befehlssatzes besteht aus einem Opcode, der stellvertretend für eine Operation ist, gefolgt von null oder mehreren Operanden, die Parameter oder Daten darstellen, die für die Operation benötigt werden. Die Anzahl oder Größe der Operanden wird dabei direkt durch den Opcode bestimmt. Viele Anweisungen besitzen jedoch keine Operanden und bestehen nur aus einem Opcode. Für jede Art von Operanden (Typen) gibt es einen speziellen Befehl (z.B. Addition von Integer-Zahlen: iadd), man spricht hier davon, daß der Befehlssatz typisiert ist.

2.1.1.1 Befehlsklassen

Im Folgenden wird nun der Befehlssatz der JavaVM in Klassen eingeteilt, dargestellt:

Konstanten auf den Stack pushen

bipush, sipush, ldc1, acons_null, fcons_2, ...

Lokale Variablen auf Stack laden

iload, dload, aload_0, ...

Stackwerte in lokale Variablen speichern

istore, astore, fstore_3, iinc, ...

Array-Management

newarrayy, anewarray, multinewarray, arraylength, iaload, ...

Allgemeine Stack Operationen

nop, pop, dup, dup2, dup2_x1, swap

Arithmetische Operationen und logische Operationen

iadd, lsub, fmul, ddic, irem, lneg, ishl, lshr, iand, lor, lxor, ...

Konvertierungsfunktionen

i2l, l2f, d2i, int2byte, ...

Verzweigungsfunktionen (Steuerung des Kontrollflußes)

ifeq, iflt, if_icmple, ifnull, ifnonnull, lcmp, goto, jsr, ret, ...

Methoden-Aufruf und Rückkehr

invokevirtual, invokestatic, invokeinterface, return, iretrun, ...

Manipulation von Objekten und Feldern

putfield, getfield, putstatic, getstatic, ...

Exception Handling

athrow, ...

versch. Objekt Operationen

new, newfromname, checkcast, instanceof, verifystack, ...

Monitore

monitorenter, monitorexit

Debugging

Breakpoint

opcode	*byte*	*short*	*int*	*long*	*float*	*double*	*char*	*reference*

Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl	fshl	dshl		
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		

if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lretrun*	*freturn*	*dreturn*		*areturn*

***Tabelle 2**: Typisierter Befehlsatz der JavaVM*

2.1.2 Register

In den Registern der JavaVM werden die Zustände während der Ausführung der Maschine festgehalten (siehe **Abbildung 5**). Sie sind analog zu den Register eines Prozessors.

Die Register können wie folgt eingeteilt werden:

pc -- Java Programm Counter, ein Pointer, der auf das nächste auszführende Code-Element zeigt.

optop -- ein Pointer, der auf das erste Element des Operanden-Stacks zeigt.

frame -- ein Pointer, auf die Ausführungs-Umgebung der aktuell ausgeführten Methode.

vars -- ein Pointer auf die 0. lokale Variable, der aktuell ausgeführten Methode.

2.1.3 Stack

Der JavaStack wird verwendet um Parameter für Operationen bereitzustellen (siehe **Abbildung 3**), Rückgabewerte zu empfangen, Teilergebnisse zu speichern, Parameter an Methoden zu übergeben, usw. Jeder Thread verwendet einen privaten JavaStack, welcher zur selben Zeit wie der Thread gestartet wird. Jede Methode wiederum arbeitet auf einem eigenen (lokalen) StackFrame. Jeder StackFrame besteht aus drei Komponenten: lokale Variablen, dem Execution Environment, dem Operanden Stack.

2.1.4 Heap

Der Heap (siehe **Abbildung 3**) wird von allen Threads als eine Art gemeinsamer Arbeitsspeicher verwendet. Hierbei handelt es sich um den Laufzeitdatenbereich, in dem alle Klasseninstanzen und Arrays während der Ausführung gehalten werden. Heap-Speicher für Objekte wird durch den automatischen Garbage Collector angefordert, d.h. Objekte werden nie direkt durch den Programmierer zerstört.

Wird ein Objekt nicht mehr verwendet, d.h. es gibt keinen Verweis mehr, der auf es zeigt, so wird es durch den Garbage Collector entfernt.

2.1.5 Method-Area

Die JavaVM hat eine Method-Area (siehe**Abbildung 3**), in der der kompilierte Programm-Code abgelegt wird.
Dieser Bereich wird als Teil des Heaps, beim Starten der VM angelegt. Die Method-Area beinhaltet sämtlichen Methodencode (compilierter JavaCode), sowie die Symboltabellen für die dynamischen Links.

Außerdem werden hier zusätzlich Informationen zur Fehlerbehebung und der Entwicklungsumgebung abgelegt.

2.1.6 Konstanten-Pool

Verbunden mit jeder Klasse ist ein Konstanten-Pool (siehe **Abbildung 3**).
Er enthält die Namen aller Felder, Methoden und sonstige zur Programmausführung nötige Informationen.

Java Virtual Machine

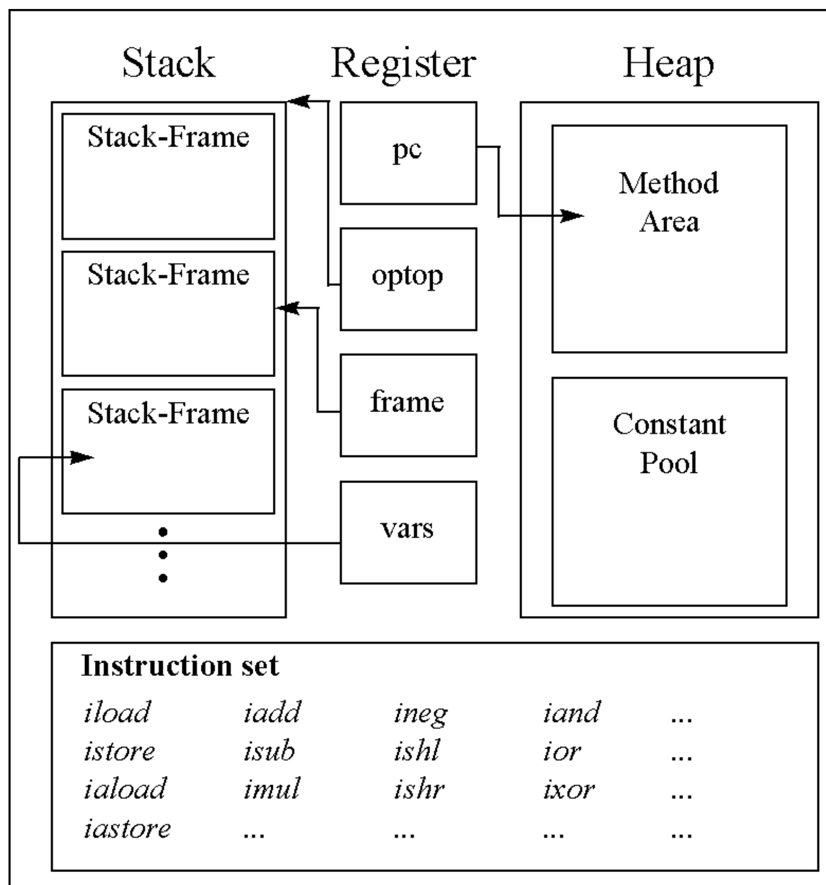

***Abbildung 3**: Aufbau der Java Virtual Machine*

3. Stackaufbau

Die JavaVM ist eine stack-basierte Maschine. Der JavaStack wird verwendet um Parameter für Operationen, Rückgabewerte, Übergabewerte an Methoden etc. zu speichern. Der JavaStack ist in StackFrames eingeteilt. Jeder StackFrame implementiert den Zustand einer einzelnen Methode. Jeder JavaStackFrame besteht aus drei Komponenten, wobei eine oder mehrere Komponenten zu einer bestimmten Zeit auch leer sein können:

lokaler Variablenbereich

Exectution Environment

OperandenStack

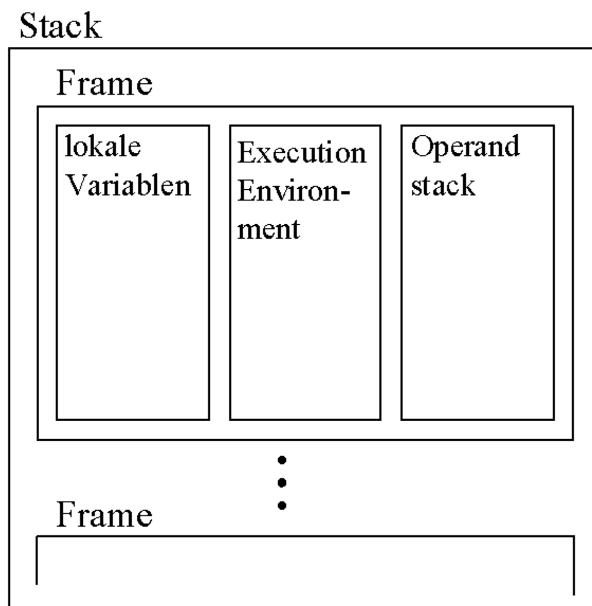

Abbildung 4: *Aufbau eines JavaStacks*

3.1 Lokale Variablen

Jeder JavaStackFrame hat eine Menge an lokalen Variablen. Sie werden durch Indizes durch das `vars`-Register adressiert und sind somit effektiv ein Array. Lokale Variablen sind ebenfalls 32-bit breit.

3.2 Execution Environment

Das Execution Environment ist die Komponente des Stack-Frames, mit der die Operationen auf dem JavaStack selbst gesteuert werden. Es enthält sowohl Zeiger auf den vorherigen Frame als auch Zeiger auf seine eigenen lokalen Variablen und Kopf und Fuß des Stacks.

3.3 Operanden Stack

Der Operandenstack ist ein 32-bit breiter FIFO Stack, der verwendet wird, um Argumente und Rückgabewerte vieler VM-Befehle zu speichern.

4. Literatur

[1] **The Lean, Mean, Virtual Machine,**

javaworld, June 1996

http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html

[2] **The Java Virtual Machine Specification**,

http://tech-www.informatik.uni-hamburg.de/
java/documentaion/vmspec/