Inhaltsverzeichnis

Die virtuelle Java Maschine (JVM)

1. Harte und weiche Maschinen

Stellen Sie sich einen Computer **ohne ein Betriebssystem** vor, z.B. einen mit einem Pentium-Prozessor. Theoretisch ist es möglich, Programme zu schreiben, die von einer solchen "nackten Hardware" ausgeführt werden können. Solche Programme dürften nur Maschinenbefehle enthalten, die ein Pentium-Prozessor ausführen kann.

Ein solches Programm zu schreiben wäre **sehr aufwendig**. Um nur ein paar Zeichen auszugeben oder einzulesen (zum Bildschirm, von der Tastatur oder zu/von einer Datei auf der Festplatte) müsste man **sehr viele** Pentium-Maschinenbefehle in das Programm einbauen. Die Entwicklung eines leistungsfähigen Programms würde vermutlich viele Jahre dauern, deshalb kommen solche Programme in der Praxis nur selten vor.

Jetzt stellen Sie sich einen Computer **mit einem Betriebssystem** vor, z.B. einen Computer mit einem Pentium-Prozessor und einem Linux-Betriebssystem. Für eine solche Maschine ein Programm zu erstellen ist schon wesentlich **einfacher**. Denn ein solches Programm kann zwei Arten von Befehlen enthalten: **Maschinenbefehle**, die der Prozessor direkt ausführen kann und Aufrufe von Unterprogrammen (oder ähnlichen Programmteilen), die zum **Betriebssystem** gehören. Um z.B. Daten **einzulesen** oder **auszugeben** würde man entsprechende Teile des Betriebssystems aufrufen, statt die zahlreichen und komplizierten Maschinenbefehle, die dazu nötig sind, in das Programm einzubauen.

Ein Betriebssystem enthält viele **komplizierte**, **nützliche** und **leistungsfähige** Befehlsfolgen (und manchmal auch ein paar **unnütze**, **fehlerhafte** und **schädliche** Befehlsfolgen) und erspart es einem, diese Befehlsfolgen in jedes einzelne Programm einzubauen, in dem man sie benötigt. Ein Betriebssystem hat aber auch einen **Nachteil**: Ein Programm, welches auf einem Pentium-Rechner unter **Linux** ausführbar ist, kann nicht unter einem **Windows**-Betriebssystem ausgeführt werden, selbst wenn man den gleichen Pentium-Rechner verwendet. Denn das Programm ruft Unterprogramme (oder ähnliche Programmteile) auf, die es nur unter **Linux** aber nicht unter **Windows** gibt. Und umgekehrt kann ein **Windows**-Programm nicht unter **Linux** ausgeführt werden (zumindest nicht ohne spezielle Software wie z.B. VMware). Ein Programm von einem Betriebssystem auf ein anderes zu **portieren** ist in aller Regel sehr schwierig und aufwendig und in einigen Fällen sogar praktisch kaum möglich.

Unter einer (klassischen) **Plattform** versteht man häufig eine Kombination aus einem **Prozessor** und einem **Betriebssystem**, z.B. einen i386-Prozessor unter Linux oder einen i386-Prozessor unter Windows2000 oder einen M68-Prozessor unter Linux etc.

Um Programme **leicht und selbstverständlich** portierbar zu machen, hat die Firma Sun eine so genannte **virtuelle Java Maschine** (JVM, Java virtual machine) definiert. Das ist eine Maschine, die eine gewisse Ähnlichkeit mit einem **Prozessor** (z.B. einem Pentium-Prozessor oder einem Motorola M68-Prozessor oder einem Sparc-Prozessor etc.) hat. Sie "kennt" ca. 200 verschiedene Maschinenbefehle, mit denen man z.B. zwei **int**-Werte addieren, zwei **double**-Werte multiplizieren, die Referenz eines String-Objekts auf den Stapel laden oder den obersten Wert vom Stapel entfernen und in einer Variablen speichern

kann (ganz ähnlich wie bei anderen Prozessoren auch). Die **Maschinensprache** der **JVM** heißt **Bytecode** (weil viele Maschinenbefehle nur 1 oder 2 Byte lang sind) und die **Maschinenprogramme** der JVM (die den **.exe**-Dateien unter **Windows** entsprechen) bezeichnet man als **Bytecode-Programme**.

Zwischen klassischen **Prozessoren** und der **JVM** gibt es aber nicht nur viele **Ähnlichkeiten**, sondern auch ein paar sehr wichtige **Unterschiede**:

1. Ein **Bytecode-Programm** darf nur **Maschinenbefehle der JVM** und keine Aufrufe irgendwelcher Linux- oder Windows-Befehle enthalten. Allerdings gehören alle **Java-Standard-Klassen** und **-Schnittellen** (in der Version 7.0 sind das mehr als 4000 Klassen und Schnittstellen) zur **JVM** und können von jedem Bytecode-Programm benutzt werden. Diese Standardklassen leisten vieles von dem, was sonst ein Betriebssystem erledigt (z.B. das **Ein**- und **Ausgeben** von Daten, insbesondere über das **Internet**).

2. Die JVM ist **stark typisiert**. Das bedeutet z.B. dass man (weder aus Versehen noch absichtlich) einen **int**-Wert zur Referenz eines **String**-Objekts addieren oder einen **long**-Wert mit einem **float**-Wert vergleichen kann. Es ist (aus gutem Grund) noch nicht einmal möglich, die **Referenz** eines Objekts zum Bildschirm oder in eine Datei **auszugeben** ("Referenzen sind **Privatsache** des **Ausführers** und gehen weder den **Programmierer** noch den **Benutzer** etwas an").

3. Die JVM ist **objektorientiert**. Klassen werden von der JVM **automatisch geladen** (wenn sie gebraucht werden) und ein Objekt einer beliebigen Klasse kann man im wesentlichen mit **einem** einzigen Maschinenbefehl (**new**) erzeugen. Klassische Prozessoren sind nicht objektorientiert, das Laden einer Klasse muss ausdrücklich befohlen werden und zum Erzeugen eines Objekts sind viele Maschinenbefehle nötig.

4. Wenn die **JVM** eine Klasse lädt, führt sie zahlreiche **Sicherheits**- und **Plausibilitätsprüfungen** durch und wirft eine Ausnahme (z.B. eine **SecurityException**), wenn mit der Klasse "irgend etwas nicht stimmt". Damit ist es sehr viel schwieriger, einen **Virus** für die **JVM** zu programmieren als für klassische Plattformen. Und falls es einem Programmierer doch einmal gelingen sollte, einen Virus für die JVM zu konstruieren, ist es wahrscheinlich leichter, die JVM durch den Einbau zusätzlicher Prüfungen zu "immunisieren" als das bei einer klassischen Plattform der Fall ist.

Das Wort **virtuell** im Namen der **JVM** deutet darauf hin, dass die **JVM** heute meistens nicht als **Chip** realisiert wird (wie das bei klassischen Prozessoren üblich ist), sondern durch **Programme**, die auf klassischen Plattformen laufen Für jede verbreitete Plattform gibt es mindestens ein Programm, welches auf dieser Plattform eine **JVM** (oder mehrere nebenläufige **JVMs**) realisiert.

Es gibt aber schon heute auch Chips, die die JVM "in Silikon implementieren" (im SWE-Labor kann man sich eine solche "harte" JVM ansehen). Ob Java-Programme in Zukunft verstärkt von solchen speziellen Chips oder weiterhin von Programmen auf klassischen Prozessoren ausgeführt werden ist schwer vorherzusehen. Vor allem ökonomische Gründe werden entscheiden, welche Ausführungsweise vorherrschen wird.

2. Ein Bytecode-Programm

Ein **Java-Compiler** übersetzt **Java-Quelldateien** (.java-Dateien) in **Bytecode-Dateien** (.class-Dateien). Hier ein Beispiel für eine **Java-Quelldatei**:

```
 1 // Datei Hallo11.java
 2 /* ------------------------------------------------------------------------
 3 Ein Hallo-Programm, welches aus 5 Klassen besteht (aus der Hauptklasse
 4 Hallo11 und den Klassen Hallo02, Hallo03, AM01 und String).
 5 ------------------------------------------------------------------------ */
 6 import de.tfh_berlin.grude.einaus.AM01;
 7
 8 class Hallo11 {
 9    static {AM01.pln("Statischer Initialisierer wurde ausgefuehrt!");}
10
11    static public void main(String[] susi) {
12       final int  wieViele = 12;
13       final char zeichen  = '#';
14       Hallo03.druckeVerzierungszeile(wieViele, zeichen);
15       Hallo02.druckeTextzeile();
16       Hallo03.druckeVerzierungszeile(wieViele, zeichen);
17    } // main
18 } // class Hallo11
```

Hier die Klassendatei **Hallo11.class**, die der Java-Compiler **javac** (von Sun) aus der Quelldatei **Hallo11.java** erzeugt hat:

```
19 000000  CA FE BA BE  00 00 00 2E  00 23 0A 00  07 00 11 0A  Êþº¾.....#......
20 000010  00 12 00 13  0A 00 14 00  15 08 00 16  0A 00 17 00  ................
21 000020  18 07 00 19  07 00 1A 01  00 06 3C 69  6E 69 74 3E  ..........<init>
22 000030  01 00 03 28  29 56 01 00  04 43 6F 64  65 01 00 0F  ...()V...Code...
23 000040  4C 69 6E 65  4E 75 6D 62  65 72 54 61  62 6C 65 01  LineNumberTable.
24 000050  00 04 6D 61  69 6E 01 00  16 28 5B 4C  6A 61 76 61  ..main...([Ljava
25 000060  2F 6C 61 6E  67 2F 53 74  72 69 6E 67  3B 29 56 01  /lang/String;)V.
26 000070  00 08 3C 63  6C 69 6E 69  74 3E 01 00  0A 53 6F 75  ..<clinit>...Sou
27 000080  72 63 65 46  69 6C 65 01  00 0C 48 61  6C 6C 6F 31  rceFile...Hallo1
28 000090  31 2E 6A 61  76 61 0C 00  08 00 09 07  00 1B 0C 00  1.java..........
29 0000A0  1C 00 1D 07  00 1E 0C 00  1F 00 09 01  00 2C 53 74  .............,St
30 0000B0  61 74 69 73  63 68 65 72  20 49 6E 69  74 69 61 6C  atischer Initial
31 0000C0  69 73 69 65  72 65 72 20  77 75 72 64  65 20 61 75  isierer wurde au
32 0000D0  73 67 65 66  75 65 68 72  74 21 07 00  20 0C 00 21  sgefuehrt!.. ..!
33 0000E0  00 22 01 00  07 48 61 6C  6C 6F 31 31  01 00 10 6A  ."...Hallo11...j
34 0000F0  61 76 61 2F  6C 61 6E 67  2F 4F 62 6A  65 63 74 01  ava/lang/Object.
35 000100  00 07 48 61  6C 6C 6F 30  33 01 00 16  64 72 75 63  ..Hallo03...druc
36 000110  6B 65 56 65  72 7A 69 65  72 75 6E 67  73 7A 65 69  keVerzierungszei
37 000120  6C 65 01 00  05 28 49 43  29 56 01 00  07 48 61 6C  le...(IC)V...Hal
38 000130  6C 6F 30 32  01 00 0F 64  72 75 63 6B  65 54 65 78  lo02...druckeTex
39 000140  74 7A 65 69  6C 65 01 00  1F 64 65 2F  74 66 68 5F  tzeile...de/tfh_
40 000150  62 65 72 6C  69 6E 2F 67  72 75 64 65  2F 65 69 6E  berlin/grude/ein
41 000160  61 75 73 2F  41 4D 30 31  01 00 03 70  6C 6E 01 00  aus/AM01...pln..
42 000170  15 28 4C 6A  61 76 61 2F  6C 61 6E 67  2F 4F 62 6A  .(Ljava/lang/Obj
43 000180  65 63 74 3B  29 56 00 20  00 06 00 07  00 00 00 00  ect;)V. ........
44 000190  00 03 00 00  00 08 00 09  00 01 00 0A  00 00 00 1D  ................
45 0001A0  00 01 00 01  00 00 00 05  2A B7 00 01  B1 00 00 00  ........*·..±...
46 0001B0  01 00 0B 00  00 00 06 00  01 00 00 00  08 00 09 00  ................
47 0001C0  0C 00 0D 00  01 00 0A 00  00 00 36 00  02 00 03 00  ..........6.....
48 0001D0  00 00 12 10  0C 10 23 B8  00 02 B8 00  03 10 0C 10  ......#¸..¸.....
49 0001E0  23 B8 00 02  B1 00 00 00  01 00 0B 00  00 00 12 00  #¸..±...........
50 0001F0  04 00 00 00  0E 00 07 00  0F 00 0A 00  10 00 11 00  ................
51 000200  11 00 08 00  0E 00 09 00  01 00 0A 00  00 00 1E 00  ................
52 000210  01 00 00 00  00 00 06 12  04 B8 00 05  B1 00 00 00  .........¸..±...
53 000220  01 00 0B 00  00 00 06 00  01 00 00 00  09 00 01 00  ................
54 000230  0F 00 00 00  02 00 10                              .......
```

Die folgenden Hinweise sollen die **Struktur einer Klassendatei** ("was steht wo?") anhand der obigen Beispieldatei (Hallo11.class) deutlich machen:

```
                                                                Länge (in Byte)
0000-0003 Magische Zahl,  Wert 0xCAFEBABE                          4
0004-0005 Kleine Version, Wert  0                                  2
0006-0007 Grosse Version, Wert 46                                  2
0008-0009 Anzahl Einträge in der Konstanten-Tabelle + 1 (hier: 35)  2
000A-0185 Konstanten-Tabelle (34 Einträge unterschiedlicher Länge)
000A-000E KT01: Infos zu einer Methode, Indizes  7, 17             5
000F-0013 KT02: Infos zu einer Methode, Indizes 18, 19             5
0014-0018 KT03: Infos zu einer Methode, Indizes 20, 21             5
0019-001B KT04: Index eines Strings, Wert 22                       3
001C-0020 KT05: Infos zu einer Methode, Indizes 23, 24             5
0021-0023 KT06: Index einer Klasse, Wert 25                        3
0024-0026 KT07: Index einer Klasse, Wert 26                        3
0027-002F KT08: Ein String, Länge  6, "<init>"                     3 + 6
0030-0035 KT09: Ein String, Länge  3, "()V"                        3 + 3
0036-000C KT10: Ein String, Länge  4, "Code"                       3 + 4
003D-004E KT11: Ein String, Länge 15, "LineNumberTable"            3 + 15
004F-0056 KT12: Ein String, Länge  4, "main"                       3 + 4
0057-006E KT13: Ein String, Länge 22, "([Ljava/lang/String;)V"     3 + 22
006F-0079 KT14: Ein String, Länge  8, "<clinit>"                   3 + 8
007A-0086 KT15: Ein String, Länge 10, "SourceFile"                 3 + 10
0087-0095 KT16: Ein String, Länge 12, "Hallo11.java"               3 + 12
0096-009A KT17: Name und Typ, Indizes 8,   9                       5
009B-009D KT18: Index einer Klasse, Wert  27                       3
009E-00A2 KT19: Name und Typ, Indizes 28, 29                       5
00A3-00A5 KT20: Index einer Klasse, Wert  30                       3
00A6-00AA KT21: Name und Typ, Indizes 31,  9                       5
00AB-00D9 KT22: Ein String, Länge 44, "Statischer Initialisi ..."  3 + 44
00DA-00DC KT23: Index einer Klasse, Wert  32                       3
00DD-00E1 KT24: Name und Typ, Indizes 33, 34                       5
00E2-00EB KT25: Ein String, Länge  7, "Hallo11"                    3 + 7
00EC-00FE KT26: Ein String, Länge 16, "java/lang/Object"           3 + 16
00FF-0108 KT27: Ein String, Länge  7, "Hallo03"                    3 + 7
0109-0121 KT28: Ein String, Länge 22, "druckeVerzierungszeile"     3 + 22
0122-0129 KT29: Ein String, Länge  5, "(IC)V"                      3 + 5
012A-0133 KT30: Ein String, Länge  7, "Hallo02"                    3 + 7
0134-0145 KT31: Ein String, Länge 15, "druckeTextzeile"            3 + 15
0146-0167 KT32: Ein String, Länge 31, "de/tfh_berlin/grude ..."    3 + 31
0168-016D KT33: Ein String, Länge  3, "pln"                        3 + 3
016E-0185 KT34: Ein String, Länge 21, "(Ljava/lang/Object;)V"      3 + 21
0186-0187 Erreichbarkeits-Bits, Wert 0x0020                        2
0188-0189 Diese Klasse,    Index 6                                 2
018A-018B Die Superklasse, Index 7                                 2
018C-018D Anzahl der Schnittstellen, Wert 0                        2
          Schnittstellen-Tabelle     (mit 0 Einträgen)             0
018E-018F Anzahl der Attribute,      Wert 0                        2
          Attribut-Tabelle           (mit 0 Einträgen)             0
0190-0191 Anzahl der Methoden, Wert 3                              2
0192-022C Methoden-Tabelle (3 Einträge unterschiedlicher Länge)
0192-01BC MT01: Name 8 ("<init>"), Beschreibung 9 ("()V"),
                1 Attribut, Name 10 ("Code"), Länge 29             6 + 29
01BD-0201 MT02: Name 12 ("main"), Beschr. 13 ("([Ljava/lang..."),
                1 Attribut, Name 10 ("Code"), Länge 54             6 + 54
0202-022C MT03: Name 14 ("<clinit>"), Beschr. 9 ("()V"),
                1 Attribut, Name 10 ("Code"), Länge 30             6 + 30
022D-022E Anzahl Attribute dieser Klase, Wert 1                    2
022F-0236 Attribut-Tabelle (mit 1 Eintrag)
022F-0236 AT01: Name 15 ("SourceFile"), Attribut-Länge, Wert 2,
                Attribut-Wert ist der Index 16 ("Hallo11.java")    6 + 2
```

Der Name **<init>** bezeichnet einen **Konstruktor** und **<cinit>** einen **statischen Initialisierer**. Das **V** am Ende von Methodenbeschreibungen steht für den Rückgabetyp **void**. Primitive Typen wie **int**, **char** etc. haben einzelne große Buchstaben wie **I**, **C** etc. als Namen. Klassentypen werden immer mit ihrem **vollen Namen** bezeichnet (zwischen **L** und einem **Semikolon** eingeschlossen, die einzelnen Teile durch **Schrägstriche** statt durch Punkte getrennt), z.B. **Ljava/lang/String;** . Der Name **[Ljava/lang/String;** (ohne eine schließende eckige Klammer) bezeichnet den Typ **Reihung von String-Objekten**. Entsprechend steht **[I** für **Reihung von int-Werten** und **[[C** für **Reihung von Reihungen von char-Werten**.

Um das Beispiel vollständig entschlüsseln zu können, muss man sich auch mit den Maschinenbefehlen vertraut machen, aus denen die Code-Attribute der Methoden bestehen. Im Beispiel belegt der Code der **main**-Methode **54** Bytes. Im Byte Nr. **01D3** steht der Befehl **10** (**bipush**, byte immediate push), mit dem die Zahl **0C** (12_{10}) auf den Stapel gelegt wird. Danach wird mit dem gleichen Befehl ein Nummernzeichen ('#', $\mathbf{23}_{16}$) auf den Stapel gelegt, ehe mit dem Befehl **invokestatic** (Code: **B8**) die Methode **drucke-Verzierungszeile** aufgerufen wird.

3. Java-Assembler-Dateien

Nicht alle Menschen lesen gerne **hexadezimale Darstellungen** und einige haben Schwierigkeiten, **Maschinenprogrammen** direkt von Hand zu schreiben Deshalb hat man **Assembler**-Sprachen erfunden, mit denen man Maschinenprogramme zumindest ein bisschen **lesbarer** darstellen kann.

Mit einem **Backassembler** kann man (schwer lesbare) **Maschinenprogramme** in (etwas leichter lesbare) **Assembler-Programme** "zurückübersetzen". Das folgende Assembler-Programm (Datei **Hallo11.j**) wurde mit dem Backassembler **JasminVisitor** aus der oben wiedergegebenen Bytecodedatei **Hallo11.class** erzeugt. Die fetten Hervorhebungen wurden nachträglich von Hand hinzugefügt:

```
 1 ;; Produced by JasminVisitor (BCEL)
 2 ;; http://bcel.sourceforge.net/
 3 ;; Thu Sep 19 21:25:21 CEST 2002
 4
 5 .source Hallo11.java
 6 .class  Hallo11
 7 .super java/lang/Object
 8
 9
10 ; ---------------------------------------------------------------------------
11 .method  <init>()V
12 .limit stack  1
13 .limit locals 1
14 .var 0 is this LHallo11; from Label0 to Label1
15
16 Label0:
17 .line 8
18   aload_0
19   invokespecial java/lang/Object/<init>()V
20 Label1:
21   return
22
23 .end method ; <init>
24 ; ---------------------------------------------------------------------------
25 .method public static main([Ljava/lang/String;)V
26 .limit stack  2
27 .limit locals 3
28 .var 0 is arg0 [Ljava/lang/String; from Label0 to Label1
29
30 Label0:
31 .line 14
32   bipush 12
33   bipush 35
34   invokestatic Hallo03/druckeVerzierungszeile(IC)V
35 .line 15
36   invokestatic Hallo02/druckeTextzeile()V
37 .line 16
38   bipush 12
39   bipush 35
40   invokestatic Hallo03/druckeVerzierungszeile(IC)V
41 Label1:
42 .line 17
43   return
44
45 .end method ; main
46 ; ---------------------------------------------------------------------------
47 .method static <clinit>()V
48 .limit stack  1
```

```
49 .limit locals 0
50
51 .line 9
52   ldc "Statischer Initialisierer wurde ausgefuehrt!"
53   invokestatic de/tfh_berlin/grude/einaus/AM01/pln(Ljava/lang/Object;)V
54   return
55
56 .end method ; <clinit>
57 ; ---------------------------------------------------------------------
```

Erläuterungen zur Notation in Jasmin-Programmen:

Jasmin-Notation	**Bedeutung**
`.method <init>()V`	Konstruktor mit 0 Parametern
`.method static <clinit>()V`	Statischer Initialisierer (in Java: `static { ... }`)
`java/lang/Object`	Voller Name der Klasse `Object`
`Ljava/lang/String;`	Der Typ `String`
`B, C, D,` `F, I, J, S, Z`	Namen der primitiven Typen `byte`, `char`, `double`, `float`, `int`, `long` (J), `short`, `bool` (Z)
`[B`	Der Typ Reihung von `byte`
`[[[B`	Der Typ Reihung von Reihung von Reihung von `byte`
`[Ljava/lang/String;`	Der Typ Reihung von `String`
`.method public add(II)I`	Eine Objektmethode namens `add` mit 2 `int`-Parametern und dem Ergebnistyp `int`.
`.method public static main ([Ljava/lang/String;)V`	Eine Klassenmethode namens `main` mit 1 Parameter vom Typ `String[]` und Ergebnistyp `void` (V)
`.limit stack 5`	Diese Methode braucht maximal 5 Worte auf dem Stapel
`.limit locals 3`	Diese Methode erzeugt maximal 3 lokale Variablen (auch Parameter zählen als lokale Variablen)
`.line 15`	Entsprechende Zeile im Java-Quellprogramm (falls vorhanden)
`Label10:`	Mögliches Ziel eines Sprungbefehls (z.B. `goto Label10`)

Mit einem **Assembler** kann man **Assembler-Programme** in **Maschinenprogramme** umwandeln. Mit dem Assembler **Jasmin** kann man das obige Assembler-Programm **Hallo11.j** wieder in eine Bytecodedatei **Hallo11.class** übersetzen.

Aufgabe: Schreiben Sie ein **Java-Quellprogramm** namens **Hallo12** welches "Hallo" zur Standardausgabe ausgibt (oder etwas ähnliches leistet). Übersetzen Sie die Datei **Hallo12.java** mit dem Compiler **javac** in eine Bytecodedatei **Hallo12.class** und diese Bytecodedatei mit dem **Backassembler JasminVisitor** in ein Assembler-Datei **Hallo12.j**. Verändern Sie die Assembler-Datei mit einem Editor ein bisschen (indem Sie darin z.B. den String "Hallo" durch "Hallo Sonja!" ersetzen). Übersetzen Sie die veränderte Assembler-Datei **Hallo12.j** mit dem **Assembler Jasmin** in eine Bytecodedatei **Hall012.class** und lassen Sie die vom Java-Interpreter **java** ausführen.

4. Die Maschinenbefehle der JVM

Der Firma Sun gebührt großes Lob unter anderem dafür, dass sie das **Format von Bytecodedateien** (.class-Dateien) und die **JVM** sehr genau **spezifiziert** und als **Standard** verbreitet und durchgesetzt hat. Trotz erheblicher Anstrengungen ist es der Firma Microsoft nicht gelungen, diesen Standard zu zerstören (und sie hat aufgehört, es zu versuchen). Leider hat die Firma Sun es aber unterlassen, für die JVM außer der (schwer lesbaren) **Maschinensprache** auch eine (etwas leichter lesbare) **Assemblersprache** zu definieren. Die Assemblersprache in der obigen Datei **Hallo11.j** (erzeugt vom Backassembler **JasminVisitor**) ist deshalb leider nicht "offiziell" oder standardisiert und es gibt einige Varianten dieser Sprache (das betrifft aber nur die **Assemblersprache**, nicht die **Maschinensprache**).

Hier ein paar kurze Erläuterungen zu einigen Befehlen der JVM:

aload_0 (access value load, 1 Byte) Lädt eine Objekt-Referenz aus der lokalen Variablen 0 auf den Stapel.

aload 7 (access value load, 2 Byte) Lädt eine Objekt-Referenz aus der lokalen Variablen 7 auf den Stapel.

astore_0 (access value store, 1 Byte) Der oberste Wert auf dem Stapel muss eine Referenz sein. Er wird vom Stapel entfernt und in die nullte lokale Variable geschrieben.

astore 7 (access value store, 2 Byte) Der oberste Wert auf dem Stapel muss eine Referenz sein. Er wird vom Stapel entfernt und in die lokale Variable 7 geschrieben.

bipush 111 (byte immediate push, 2 Byte) Der byte-Wert 111 wird zu einem int-Wert erweitert und auf den Stapel gelegt.

sipush 333 (short immediate push, 3 Byte) Der short-Wert 333 wird zu einem int-Wert erweitert und auf den Stapel gelegt. "immediate" (direkt) bedeutet, dass der Operand des Befehls direkt in dem Befehl selbst enthalten ist (und nicht in einer Variablen, auf dem Stapel oder sonst wo steht).

iconst_3 (int const 3, 1 Byte) Der int-Wert **3** wird auf den Stapel gelegt.

iconst_m1 (int const minus 1, 1 Byte) Der int-Wert -1 wird auf den Stapel gelegt.

i2d (int to double, 1 Byte) Der oberste Wert auf dem Stapel muss vom Typ **int** sein. Er wird durch einen entsprechenden **double**-Wert ersetzt.

d2i (double to int, 1Byte) Der oberste Wert auf dem Stapel muss vom Typ **double** sein. Er wird durch einen entsprechenden **int**-Wert ersetzt.

getstatic (3 Byte) Legt den Wert eines Klassenattributs auf den Stapel

putstatic (3 Byte) Entfernt den obersten Stapelwert und schreibt ihn in ein Klassattribut

getField (3 Byte) Legt den Wert eines Objektattributs auf den Stapel

putField (3 Byte) Entfernt den obersten Stapelwert und schreibt ihn in ein Objektattribut.

if_icmpge label (if int compare greater or equal) Der oberste Wert **n2** und der zweitoberste Wert **n1** auf dem Stapel müssen **int**-Werte sein. Sie werden vom Stapel entfernt und verglichen. Wenn **n1** groesser oder gleich **n2** ist, wird zum angegebenen **label** gesprungen (sonst passiert nichts weiter). Statt mit **ge** (greater or equal) gibt es diesen Befehl auch mit **gt** (greater than), **le** (less or equal), **lt** (less than), **eq** (equal) und **ne** (not equal).

invokestatic (invoke a static method, 3 Byte) Ruft eine Klassenmethode auf.

invokevirtual (invoke a virtual method, 3 Byte) Ruft eine Objektmethode auf.

Nur in **Assembler-Programmen** werden **Sprungziele** durch **Label** angegeben. In **Bytecodedateien** ("in Wirklichkeit") werden **Sprungziele** durch **Byte-Nummern** (relativ zum Anfang der betreffenden Sprunganweisung) angegeben. Eine wichtige Aufgabe eines Assemblers wie **Jasmin** ist es, **Label**-Sprungziele in **Byte-Nummern** umzurechnen (und dem Assembler-Programmierer diese fummelige und fehleranfällige Arbeit abzunehmen).

Weitere Informationen zur JVM und ihren Maschinenbefehlen findet man in den folgenden beiden Büchern:

Jon Meyer, Troy Downing
"Java Virtual Machine", O'Reilly 1997, neu ca. 112,- €, gebraucht ab ca. 3,- €
Schon etwas älter, aber sehr verständlich geschrieben und eine gute Referenz.

Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley
"The Java Virtual Machine Specification, Java SE 7 Edition", Addison-Wesley 2013, ca. 68,- €
Die verbindliche Spezifikation mit den letzten Klarstellungen der Firma Sun.

Von beiden Büchern gibt es auf der Homepage von Sun kostenlose HTML-Versionen zum runterladen.

Hier eine Liste mit folgenden Angaben **zu jedem der 200 Maschinenbefehl** der JVM:

- Ein (inoffizieller) **Assemblercode** für den Befehl
- Der (offizielle) **OP-Code** des Befehls (dezimal und hexadezimal)
- Eine (sehr) kurze **Beschreibung**, was der Befehl bewirkt.

5. Alle Maschinenbefehle, sortiert nach Assemblercode

```
Assemblercode       OpCod Kurze Beschreibung
                     D  H
aaload           =  50 32 push access from array
aastore          =  83 53 pop  access into array
aconst_null      =   1  1 push (access value) null
aload            =  25 19 push access from local variable      (no return addr!)
aload_0          =  42 2a push access from local variable nr 0 (no return addr!)
aload_1          =  43 2b push access from local variable nr 1 (no return addr!)
aload_2          =  44 2c push access from local variable nr 2 (no return addr!)
aload_3          =  45 2d push access from local variable nr 3 (no return addr!)
anewarray        = 189 bd create an array with access    components
areturn          = 176 b0 return from function and yield access
arraylength      = 190 be get the length of an array
astore           =  58 3a pop  access into local variable      (return addr ok!)
astore_0         =  75 4b pop  access into local variable nr 0
astore_1         =  76 4c pop  access into local variable nr 1
astore_2         =  77 4d pop  access into local variable nr 2
astore_3         =  78 4e pop  access into local variable nr 3
athrow           = 191 bf throw an exception (i.e. a throwable object)
baload           =  51 33 push byte   from array (or boolean)
bastore          =  84 54 pop  byte   into array
bipush           =  16 10 push byte  immediate
caload           =  52 34 push char   from array
castore          =  85 55 pop  char   into array
checkcast        = 192 b0 throw an exception if an object can not be casted
d2f              = 144 90 double to float
d2i              = 142 8e double to int
d2l              = 143 8f double to long
dadd             =  99 63 double add       (xy -> (x+y))
daload           =  49 31 push double from array
dastore          =  82 52 pop  double into array
dcmpg            = 152 98 double compare (result 1, 0, -1, for nan  1)
dcmpl            = 151 97 double compare (result 1, 0, -1, for nan -1)
dconst_0         =  14  e push double value  0
dconst_1         =  15  f push double value  1
ddiv             = 111 6f double divide    (xy -> (x/y))
dload            =  24 18 push double from local variable
dload_0          =  38 26 push double from local variable nr 0
dload_1          =  39 27 push double from local variable nr 1
dload_2          =  40 28 push double from local variable nr 2
dload_3          =  41 29 push double from local variable nr 3
dmul             = 107 6b double multiply  (xy -> (x*y))
dneg             = 119 77 double negate    ( x -> ( -x))
drem             = 115 73 double remainder (xy -> (x%y))
dreturn          = 175 af return from function and yield double
dstore           =  57 39 pop  double into local variable
dstore_0         =  71 47 pop  double into local variable nr 0
dstore_1         =  72 48 pop  double into local variable nr 1
dstore_2         =  73 49 pop  double into local variable nr 2
dstore_3         =  74 4a pop  double into local variable nr 3
dsub             = 103 67 double subtract  (xy -> (x-y))
dup              =  89 59 duplicate   top         (   x -> xx)
dup_x1           =  90 5a duplicate   top 2 below (  ax -> xax)
dup_x2           =  91 5b duplicate   top 3 below ( bax -> xbax)
dup2             =  92 5c duplicate 2 top         (  xy -> xyxy)
dup2_x1          =  93 5d duplicate 2 top 3 below ( axy -> xyaxy)
dup2_x2          =  94 5e duplicate 2 top 4 below (baxy -> xybaxy)
f2d              = 141 8d float  to double
f2i              = 139 8b float  to int
f2l              = 140 8c float  to long
fadd             =  98 62 float  add       (xy -> (x+y))
faload           =  48 30 push float  from array
fastore          =  81 51 pop  float  into array
fcmpg            = 150 96 float  compare (result 1, 0, -1, for nan  1)
fcmpl            = 149 95 float  compare (result 1, 0, -1, for nan -1)
fconst_0         =  11  b push float  value  0
fconst_1         =  12  c push float  value  1
fconst_2         =  13  d push float  value  2
fdiv             = 110 6e float  divide    (xy -> (x/y))
```

```
fload            =  23 17 push float  from local variable
fload_0          =  34 22 push float  from local variable nr 0
fload_1          =  35 23 push float  from local variable nr 1
fload_2          =  36 24 push float  from local variable nr 2
fload_3          =  37 25 push float  from local variable nr 3
fmul             = 106 6a float  multiply  (xy -> (x*y))
fneg             = 118 76 float  negate    ( x -> ( -x))
frem             = 114 72 float  remainder (xy -> (x%y))
freturn          = 174 ae return from function and yield float
fstore           =  56 38 pop  float  into local variable
fstore_0         =  67 43 pop  float  into local variable nr 0
fstore_1         =  68 44 pop  float  into local variable nr 1
fstore_2         =  69 45 pop  float  into local variable nr 2
fstore_3         =  70 46 pop  float  into local variable nr 3
fsub             = 102 66 float  subtract  (xy -> (x-y))
getfield         = 180 b4 get value of an object field (instance field)
getstatic        = 178 b2 get value of a  class  field
goto             = 167 a7 goto (2 byte relative addr)
goto_w           = 200 c8 goto (4 byte relative addr)
i2b              = 145 91 int    to byte
i2c              = 146 92 int    to char
i2d              = 135 87 int    to double
i2f              = 134 86 int    to float
i2l              = 133 85 int    to long
i2s              = 147 93 int    to short
iadd             =  96 60 int    add       (xy -> (x+y))
iaload           =  46 2e push int    from array
iand             = 126 7e int    and       (xy -> (x&y))
iastore          =  79 4f pop  int    into array
iconst_0         =   3  3 push int    value  0
iconst_1         =   4  4 push int    value  1
iconst_2         =   5  5 push int    value  2
iconst_3         =   6  6 push int    value  3
iconst_4         =   7  7 push int    value  4
iconst_5         =   8  8 push int    value  5
iconst_m1        =   2  2 push int    value -1
idiv             = 108 6c int    divide    (xy -> (x/y))
if_acmpeq        = 165 a5 goto if access top-1 is equal            top
if_acmpne        = 166 a6 goto if access top-1 is not equal        top
if_icmpeq        = 159 9f goto if int top-1    is equal            top
if_icmpge        = 162 a2 goto if int top-1    is greater or equal top
if_icmpgt        = 163 a3 goto if int top-1    is greater than     top
if_icmple        = 164 a4 goto if int top-1    is less    or equal top
if_icmplt        = 161 a1 goto if int top-1    is less    than     top
if_icmpne        = 160 a0 goto if int top-1    is not equal        top
ifeq             = 153 99 goto if int top      is equal            0
ifge             = 156 9c goto if int top      is greater or equal 0
ifgt             = 157 9d goto if int top      is greater than     0
ifle             = 158 9e goto if int top      is less    or equal 0
iflt             = 155 9b goto if int top      is less    than     0
ifne             = 154 9a goto if int top      is not equal        0
ifnonnull        = 199 c7 goto if access value is not null
ifnull           = 198 c6 goto if access value is     null
iinc             = 132 84 increment local variable by constant
iload            =  21 15 push int    from local variable
iload_0          =  26 1a push int    from local vairable nr 0
iload_1          =  27 1b push int    from local vairable nr 1
iload_2          =  28 1c push int    from local vairable nr 2
iload_3          =  29 1d push int    from local vairable nr 3
imul             = 104 68 int    multiply  (xy -> (x*y))
ineg             = 116 74 int    negate    ( x -> ( -x))
instanceof       = 193 c1 push 1 if object is instance of class, push 0 otherwise
invokeinterface  = 185 b9 call interface method
invokenonvirtual = 183 b7 old name for invokespecial
invokespecial    = 183 b7 call object method (superclass, private, object initialiser)
invokestatic     = 184 b8 call class     method
invokevirtual    = 182 b6 call object method (instance method)
ior              = 128 80 int    or        (xy -> (x|y))
irem             = 112 70 int    remainder (xy -> (x%y))
ireturn          = 172 ac return from function and yield int
ishl             = 120 78 int    shift left  (times power of 2)
ishr             = 122 7a int    shift right (sign extension)
```

```
istore           =  54 36 pop  int    into local variable
istore_0         =  59 3b pop  int    into local variable nr 0
istore_1         =  60 3c pop  int    into local variable nr 1
istore_2         =  61 3d pop  int    into local variable nr 2
istore_3         =  62 3e pop  int    into local variable nr 3
isub             = 100 64 int    subtract  (xy -> (x-y))
iushr            = 124 7c int    shift right (zero extension)
ixor             = 130 82 int    xor
jsr              = 168 a8 jump to subroutine (push return addr, for finally)
jsr_w            = 201 c9 jump to subroutine (push return addr, for finally)
l2d              = 138 8a long   to double
l2f              = 137 89 long   to float
l2i              = 136 88 long   to int
ladd             =  97 61 long   add       (xy -> (x+y))
laload           =  47 2f push long   from array
land             = 127 7f long   and
lastore          =  80 50 pop  long   into array
lcmp             = 148 94 long   compare (result 1, 0, -1)
lconst_0         =   9  9 push long   value  0
lconst_1         =  10  a push long   value  1
ldc              =  18 12 push constant (int, float, string)
ldc_w            =  19 13 push constant (int, float, string), wide index
ldc2_w           =  20 14 push constant (long, double),       wide index
ldiv             = 109 6d long   divide    (xy -> (x/y))
lload            =  22 16 push long   from local variable
lload_0          =  30 1e push long   from local variable nr 0
lload_1          =  31 1f push long   from local variable nr 1
lload_2          =  32 20 push long   from local variable nr 2
lload_3          =  33 21 push long   from local variable nr 3
lmul             = 105 69 long   multiply  (xy -> (x*y))
lneg             = 117 75 long   negate    ( x -> ( -x))
lookupswitch     = 171 ab goto addr in table (with key)
lor              = 129 81 long   or
lrem             = 113 71 long   remainder (xy -> (x%y))
lreturn          = 173 ad return from function and yield long
lshl             = 121 79 long   shift left  (times power of 2)
lshr             = 123 7b long   shift right (sign extension)
lstore           =  55 37 pop  long   into local variable
lstore_0         =  63 3f pop  long   into local variable nr 0
lstore_1         =  64 40 pop  long   into local variable nr 1
lstore_2         =  65 41 pop  long   into local variable nr 2
lstore_3         =  66 42 pop  long   into local variable nr 3
lsub             = 101 65 long   subtract  (xy -> (x-y))
lushr            = 125 7d long   shift right (zero extension)
lxor             = 131 83 long   xor
monitorenter     = 194 c2 enter a monitor
monitorexit      = 195 c3 exit  a monitor
multianewarray   = 197 c5 create an array with arrays as components
new              = 187 bb create an object
newarray         = 188 bc create an array with primitive components
nop              =   0  0 no operation
pop              =  87 57 pop  (4 bytes)
pop2             =  88 58 pop  (8 bytes)
putfield         = 181 b5 set value of an object field (instance field)
putstatic        = 179 b3 set value of a  class  field
ret              = 169 a9 return from subroutine (return addr from loc. var.!)
return           = 177 b1 return from procedure
saload           =  53 35 push short  from array
sastore          =  86 56 pop  short  into array
sipush           =  17 11 push short immediate
swap             =  95 5f swap (xy -> yx)
tableswitch      = 170 aa goto addr in table (with index)
wide             = 196 c4 extend loc. var. index for many instructions
                   186 ba Unbenutzt (aus historischen Gründen)
```