

**Inhaltsverzeichnis**

Die virtuelle Java Maschine (JVM).....	1
1. Harte und weiche Maschinen.....	1
2. Ein Bytecode-Programm.....	2
3. Java-Assembler-Dateien.....	5
4. Die Maschinenbefehle der JVM.....	6
5. Alle Maschinenbefehle, sortiert nach Operationscode.....	8
6. Alle Maschinenbefehle, sortiert nach Assemblercode.....	11

**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 **Maschinsprache** 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-Standardklassen** (in der Version 5.0 sind das mehr als 3000 Klassen) 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 wäre.

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**:



	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
0034-0026	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 Klasse, 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 `<clinit>` einen **statischen Initialisierer**. Das **V** am Ende von Methodenbeschreibungen steht für den Rückgabotyp **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 Rei-**

### hungen 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 ('#',  $23_{16}$ ) auf den Stapel gelegt, ehe mit dem Befehl `invokestatic` (Code: B8) die Methode `druckeVerzierungszeile` 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 ; -----
```

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 Bytecode-datei **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 Bytecode-datei **Hallo12.class** und diese Bytecode-datei 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 Bytecode-datei **Hallo12.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 Bytecode-dateien (.class-Dateien) und die JVM sehr genau spezifiziert und als Standard verbreitet und durchgesetzt hat. Trotz erheblicher Anstrengungen ist es der Firma Microsoft bis heute nicht gelungen, diesen Standard zu zerstören. 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 Java-Maschine JVM:

- aload\_0** (access value load) Lädt eine Objekt-Referenz aus der nullten lokalen Variablen auf den Stapel. Dieser Befehl ist nur 1 Byte lang.
- aload 7** (access value load) Lädt eine Objekt-Referenz aus der siebten lokalen Variablen auf den Stapel. Dieser Befehl ist 2 Byte lang.
- astore\_0** (access value store) Der oberste Wert auf dem Stapel muss eine Referenz sein. Er wird vom Stapel entfernt und in die nullte lokale Variable geschrieben. Dieser Befehl ist nur 1 Byte lang.
- astore 7** (access value store) Der oberste Wert auf dem Stapel muss eine Referenz sein. Er wird vom Stapel entfernt und in die siebte lokale Variable geschrieben. Dieser Befehl ist 2 Byte lang.
- bipush 111** (byte immediate push) Der byte-Wert 111 wird zu einem int-Wert erweitert und auf den Stapel gelegt. Dieser Befehl ist nur 2 Bytes lang.
- sipush 333** (short immediate push) 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 sonstwo

- steht). Dieser Befehl ist 3 Bytes lang (von denen 2 von dem short-Wert belegt werden).
- iconst\_3** (int const 3) Der int-Wert 3 wird auf den Stapel gelegt. Dieser Befehl ist nur 1 Byte lang.
- iconst\_m1** (int const minus 1) Der int-Wert -1 wird auf den Stapel gelegt. Auch dieser Befehl ist nur 1 Byte lang.
- i2d** (int to double) Der oberste Wert auf dem Stapel muss vom Typ **int** sein. Er wird durch einen entsprechenden **double**-Wert ersetzt.
- d2i** (double to int) Der oberste Wert auf dem Stapel muss vom Typ **double** sein. Er wird durch einen entsprechenden **int**-Wert ersetzt.
- getstatic** Legt den Wert eines Klassenattributs auf den Stapel
- putstatic** Entfernt den obersten Wert vom Stapel und schreibt ihn in ein Klassattribut
- getField** Legt den Wert eines Objektattributs auf den Stapel
- putField** Entfernt den obersten Wert vom Stapel 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** grösser 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).

Nur in **Assembler-Programmen** werden **Sprungziele** durch **Label** angegeben. In **Bytecode-Dateien** 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, ca. 16,- €

Schon etwas älter, aber sehr verständlich geschrieben und eine gute Referenz.

Tim Lindholm, Frank Yellin

"The Java Virtual Machine Specification, Second Edition", Addison-Wesley 1999, ca. 56,- €

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 ein 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.

Die Liste ist **zweimal** wiedergegeben:

- Sortiert nach **OP-Codes**
- Sortiert nach **Assemblercodes**

## 5. Alle Maschinenbefehle, sortiert nach Operationscode

Assemblercode	OpCod	Kurze Beschreibung	
		D	H
NOP	= 0	0	no operation
ACONST_NULL	= 1	1	push (access value) null
ICONST_M1	= 2	2	push int value -1
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
LCONST_0	= 9	9	push long value 0
LCONST_1	= 10	A	push long value 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
DCONST_0	= 14	E	push double value 0
DCONST_1	= 15	F	push double value 1
BIPUSH	= 16	10	push byte immediate
SIPUSH	= 17	11	push short immediate
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
ILOAD	= 21	15	push int from local variable
LLOAD	= 22	16	push long from local variable
FLOAD	= 23	17	push float from local variable
DLOAD	= 24	18	push double from local variable
ALOAD	= 25	19	push access from local variable (no return addr!)
ILOAD_0	= 26	1A	push int from local variable nr 0
ILOAD_1	= 27	1B	push int from local variable nr 1
ILOAD_2	= 28	1C	push int from local variable nr 2
ILOAD_3	= 29	1D	push int from local variable nr 3
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
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
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
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!)
IALOAD	= 46	2E	push int from array
LALOAD	= 47	2F	push long from array
FALOAD	= 48	30	push float from array
DALOAD	= 49	31	push double from array
AALOAD	= 50	32	push access from array
BALOAD	= 51	33	push byte from array (or boolean)
CALOAD	= 52	34	push char from array
SALOAD	= 53	35	push short from array
ISTORE	= 54	36	pop int into local variable
LSTORE	= 55	37	pop long into local variable
FSTORE	= 56	38	pop float into local variable
DSTORE	= 57	39	pop double into local variable
ASTORE	= 58	3A	pop access into local variable (return addr ok!)
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
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
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
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
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
IASTORE	=	79	4F	pop	int	into	array			
LASTORE	=	80	50	pop	long	into	array			
FASTORE	=	81	51	pop	float	into	array			
DASTORE	=	82	52	pop	double	into	array			
AASTORE	=	83	53	pop	access	into	array			
BASTORE	=	84	54	pop	byte	into	array			
CASTORE	=	85	55	pop	char	into	array			
SASTORE	=	86	56	pop	short	into	array			
POP	=	87	57	pop	(4 Bytes)					
POP2	=	88	58	pop	(8 Bytes)					
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)	
SWAP	=	95	5F	swap	(xy -> yx)					
IADD	=	96	60	int	add	(xy -> (x+y))				
LADD	=	97	61	long	add	(xy -> (x+y))				
FADD	=	98	62	float	add	(xy -> (x+y))				
DADD	=	99	63	double	add	(xy -> (x+y))				
ISUB	=	100	64	int	subtract	(xy -> (x-y))				
LSUB	=	101	65	long	subtract	(xy -> (x-y))				
FSUB	=	102	66	float	subtract	(xy -> (x-y))				
DSUB	=	103	67	double	subtract	(xy -> (x-y))				
IMUL	=	104	68	int	multiply	(xy -> (x*y))				
LMUL	=	105	69	long	multiply	(xy -> (x*y))				
FMUL	=	106	6A	float	multiply	(xy -> (x*y))				
DMUL	=	107	6B	double	multiply	(xy -> (x*y))				
IDIV	=	108	6C	int	divide	(xy -> (x/y))				
LDIV	=	109	6D	long	divide	(xy -> (x/y))				
FDIV	=	110	6E	float	divide	(xy -> (x/y))				
DDIV	=	111	6F	double	divide	(xy -> (x/y))				
IREM	=	112	70	int	remainder	(xy -> (x%y))				
LREM	=	113	71	long	remainder	(xy -> (x%y))				
FREM	=	114	72	float	remainder	(xy -> (x%y))				
DREM	=	115	73	double	remainder	(xy -> (x%y))				
INEG	=	116	74	int	negate	(x -> (-x))				
LNEG	=	117	75	long	negate	(x -> (-x))				
FNEG	=	118	76	float	negate	(x -> (-x))				
DNEG	=	119	77	double	negate	(x -> (-x))				
ISHL	=	120	78	int	shift left	(times power of 2)				
LSHL	=	121	79	long	shift left	(times power of 2)				
ISHR	=	122	7A	int	shift right	(sign extension)				
LSHR	=	123	7B	long	shift right	(sign extension)				
IUSHR	=	124	7C	int	shift right	(zero extension)				
LUSHR	=	125	7D	long	shift right	(zero extension)				
IAND	=	126	7E	int	and					
LAND	=	127	7F	long	and					
IOR	=	128	80	int	or					
LOR	=	129	81	long	or					
IXOR	=	130	82	int	xor					
LXOR	=	131	83	long	xor					
IINC	=	132	84	increment	local variable by constant					
I2L	=	133	85	int	to long					
I2F	=	134	86	int	to float					
I2D	=	135	87	int	to double					

L2I	= 136	88	long	to	int
L2F	= 137	89	long	to	float
L2D	= 138	8A	long	to	double
F2I	= 139	8B	float	to	int
F2L	= 140	8C	float	to	long
F2D	= 141	8D	float	to	double
D2I	= 142	8E	double	to	int
D2L	= 143	8F	double	to	long
D2F	= 144	90	double	to	float
I2B	= 145	91	int	to	byte
I2C	= 146	92	int	to	char
I2S	= 147	93	int	to	short
LCMP	= 148	94	long	compare	(result 1, 0, -1)
FCMPL	= 149	95	float	compare	(result 1, 0, -1, for NaN -1)
FCMPG	= 150	96	float	compare	(result 1, 0, -1, for NaN 1)
DCMPL	= 151	97	double	compare	(result 1, 0, -1, for NaN -1)
DCMPG	= 152	98	double	compare	(result 1, 0, -1, for NaN 1)
IFEQ	= 153	99	goto if int top	is equal	0
IFNE	= 154	9A	goto if int top	is not equal	0
IFLT	= 155	9B	goto if int top	is less than	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
IF_ICMPEQ	= 159	9F	goto if int top-1	is equal	top
IF_ICMPNE	= 160	A0	goto if int top-1	is not equal	top
IF_ICMPLT	= 161	A1	goto if int top-1	is less than	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_ACMPEQ	= 165	A5	goto if access top-1	is equal	top
IF_ACMPLT	= 166	A6	goto if access top-1	is not equal	top
GOTO	= 167	A7	goto	(2 byte relative addr)	
JSR	= 168	A8	jump to subroutine	(push return addr, for finally)	
RET	= 169	A9	return from subroutine	(return addr from loc. var.!)	
TABLESWITCH	= 170	AA	goto addr in table	(with index)	
LOOKUPSWITCH	= 171	AB	goto addr in table	(with key)	
IRETURN	= 172	AC	return from function	and yield int	
LRETURN	= 173	AD	return from function	and yield long	
FRETURN	= 174	AE	return from function	and yield float	
DRETURN	= 175	AF	return from function	and yield double	
ARETURN	= 176	B0	return from function	and yield access	
RETURN	= 177	B1	return from procedure		
GETSTATIC	= 178	B2	get value of a class field		
PUTSTATIC	= 179	B3	set value of a class field		
GETFIELD	= 180	B4	get value of an object field	(instance field)	
PUTFIELD	= 181	B5	set value of an object field	(instance field)	
INVOKEVIRTUAL	= 182	B6	call object method	(instance method)	
INVOKESPECIAL	= 183	B7	call object method	(superclass, private, object initialiser)	
INVOKENONVIRTUAL	= 183	B7	old name for INVOKESPECIAL		
INVOKESTATIC	= 184	B8	call class method		
INVOKEINTERFACE	= 185	B9	call interface method		
		186	BA	unused (for historical reasons)	
NEW	= 187	BB	create an object		
NEWARRAY	= 188	BC	create an array with primitive components		
ANEWARRAY	= 189	BD	create an array with access components		
ARRAYLENGTH	= 190	BE	get the length of an array		
ATHROW	= 191	BF	throw an exception	(i.e. a throwable object)	
CHECKCAST	= 192	B0	throw an exception if an object can not be casted		
INSTANCEOF	= 193	C1	push 1 if object is instance of class, push 0 otherwise		
MONITORENTER	= 194	C2	enter a monitor		
MONITOREXIT	= 195	C3	exit a monitor		
WIDE	= 196	C4	extend loc. var. index for many instructions		
MULTIANEWARRAY	= 197	C5	create an array with arrays as components		
IFNULL	= 198	C6	goto if access value is null		
IFNONNULL	= 199	C7	goto if access value is not null		
GOTO_W	= 200	C8	goto	(4 byte relative addr)	
JSR_W	= 201	C9	jump to subroutine	(push return addr, for finally)	

## 6. 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 -> bxax)
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	
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_ACMUNE	= 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_ICMUNE	= 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 variable nr 0	
ILOAD_1	= 27 1B	push int	from local variable nr 1	
ILOAD_2	= 28 1C	push int	from local variable nr 2	
ILOAD_3	= 29 1D	push int	from local variable 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	
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)