Software-Engineering Seminar

Berufsakademie Stuttgart – Außenstelle Horb
Studiengang Informationstechnik

Java Virtual Machine (JVM) – Architektur

Autoren:
Ing.-Assistent (BA) Olga Illenseer
Ing.-Assistent (BA) Timo Siefert

Betreuender Dozent:
Prof. Dr.-Ing. Olaf Herden

Projektbearbeitungszeit:
Oktober 2005

Abstract

Im Rahmen der Software-Engineering II Vorlesung im 5. Semester wurde diese Ausarbeitung über die Architektur der Java Virtual Machine (JVM) erstellt. Hauptsächlich wird in diesem Dokument der Grundaufbau, sprich die wichtigsten Komponenten einer JVM, und ihre Eigenschaften betrachtet.

Folgende Komponenten werden in ihrem schematischen Grundaufbau erläutert.

- Class-Loader
- Execution-Engine
- Speicherverhalten der JVM
- Speicherbereiche der JVM
- Class-Datei
- Datentypen der JVM

Diese Ausarbeitung ist keine detaillierte Beschreibung einer JVM, sondern stellt einen groben Überblick über die einzelnen Bestandteile dar.

Abkürzungsverzeichnis

CL	Class-Loader
JVM	Java Virtual Machine
PC	Programm Counter
EE	Execution-Engine
JRE	Java Runtime Enviroment
NMI	Native Method Interface
JIT	Just-In-Time

Inhaltsverzeichnis

Abstract **2**
Abkürzungsverzeichnis **3**
Inhaltsverzeichnis **4**
Abbildungsverzeichnis 5
Tabellenverzeichnis 5
1 Einleitung **6**
2 Aufbau der JVM **7**
2.1 Laufzeitumgebung 7
2.2 Class-Loader 8
2.2.1 Laden 8
2.2.2 Linken 8
2.2.3 Initialisieren 9
2.3 Register 10
2.4 Execution-Engine 11
2.4.1 Interpretation 11
2.4.2 Just-In-Time (JIT) Kompilation 11
2.4.3 Adaptive Optimierung 12
2.5 Native Method Interface 12
3 Speicher **13**
3.1 Java Stack 13
3.1.1 Lokale Variablen 14
3.1.2 Operanden Stack 14
3.1.3 Frame Daten 14
3.2 Native Method Stack 14
3.3 Heap 15
3.4 Programm Counter 15
3.5 Method Area 15
3.5.1 Constant Pool 16
3.5.2 Felderinformationen 16
3.5.3 Methoden Informationen 16
3.5.4 Referenz auf Instanz von Class Loader 16
3.5.5 Referenz auf Instanz von Class 16
3.5.6 Beispiel 16
4 Spezifikationen **18**
4.1 CLASS-Datei 18
4.1.1 Konstantenpool 19
4.1.2 Felder 20
4.1.3 Methoden 21
4.1.4 Attribute 21
4.2 Datentypen 21
4.2.1 Ordinale Typen 23
4.2.2 Gleitkomma Typen 23
4.2.3 Referenz Typen 23
4.2.4 Sonstige Typen 23
4.3 Lebenszyklus von Objekten und Klassen 23
5 Zusammenfassung **24**
6 Quellenangaben **25**
6.1 Literaturverzeichnis 25
6.2 Internetlinks 25

Abbildungsverzeichnis

Abbildung 1 - Aufbau der JVM 7
Abbildung 2 - Speicherübersicht 13
Abbildung 3 - Methodenaufrufe 14
Abbildung 4 - Datentypen in Java 22

Tabellenverzeichnis

Tabelle 1 - Tag Eintrag im Konstantenpool 20
Tabelle 2 - Access-Flags 20
Tabelle 3 - Gesamtübersicht Datentypen 22

1 Einleitung

Jede Software benötigt Hardware, auf welcher diese ausgeführt werden kann. In der Regel wird der Quellcode des Programmierers durch den Compiler in Maschinencode umgewandelt. Dieser Maschinencode ist allerdings plattformabhängig, sprich er kann nur auf einer bestimmten Hardware ausgeführt werden. Java unterscheidet sich in der Hinsicht von anderen Programmiersprachen, da durch den Java-Compiler nicht Maschinencode, sondern der Java-Bytecode für eine abstrakte Maschine erzeugt wird. Dieser Bytecode kann von der Java Virtual Maschine (JVM), einer Laufzeitumgebung für unterschiedliche Hardware, ausgeführt werden. Somit stellt die JVM die Schnittstelle zwischen dem Bytecode und dem Zielsystem her – die Eigenschaft macht Java zu einer plattformunabhängigen Programmiersprache – da anhand der Spezifikationen für die JVM für diverse Zielsysteme eine JVM erstellt werden kann.

Für den Begriff der JVM gibt es drei unterschiedliche Definitionen:

- abstrakte Computer Spezifikation der JVM
- konkrete Implementierung der JVM
- Laufzeitinstanz der JVM

Die **abstrakte Computer Spezifikation** der JVM ist das Konzept der JVM, die Bestandteile dieser und der Aufbau von class-Dateien (Java Bytecode). Die **konkrete Implementierung** der JVM hat die Aufgabe, die class-Dateien einzulesen und das Programm zum Ausführen zu bringen. Für jedes Java Programm, welches auf dem Zielsystem ausgeführt werden soll, ist eine eigene **Laufzeitinstanz** nötig.

Diese Ausarbeitung beschäftigt sich hauptsächlich mit der **abstrakten Computer Spezifikation** der JVM.

Die JVM trennt die von ihr erzeugten Prozesse (Threads) vom Betriebsystem ab, wodurch es nicht möglicht ist, mit Mitteln des Betriebssystem die Prozesse zu kontrollieren. Zerstört ein Prozess der JVM das Betriebssystem, so muss die gesamte JVM beendet werden.
Der Vorteil der JVM ist außer der Plattformunabhängigkeit der Sicherheitsvorteil. Die JVM überwacht zur Laufzeit die Programmausführung und verhindert zum Beispiel dadurch, dass ein anderes Programm Daten in fremde Speicherbereiche schreibt oder über Arraygrenzen hinaus Informationen liest.

2 Aufbau der JVM

Die Spezifikation der JVM beschreibt das Verhalten der JVM im Bezug auf Speicherbereich, Teilsysteme, Datentypen und dem Befehlssatz. Diese Beschreibung gibt aber nur das Verhalten der JVM nach außen hin an, nicht wie diese konkret implementiert werden soll – so gesehen kann die JVM als Blackbox betrachtet werden, bei der für eine Eingabe eine definierte Ausgabe durchgeführt werden muss. So gesehen muss sich ein Programmierer nicht um die konkrete Implementierung der JVM Gedanken machen, sondern kann von einem gleichen Verhalten sämtlicher JVMs für unterschiedliche Zielsysteme ausgehen.

Die JVM besteht aus mehreren Bestandteilen, welche in Abbildung 1 skizziert und im folgenden näher beschrieben werden.

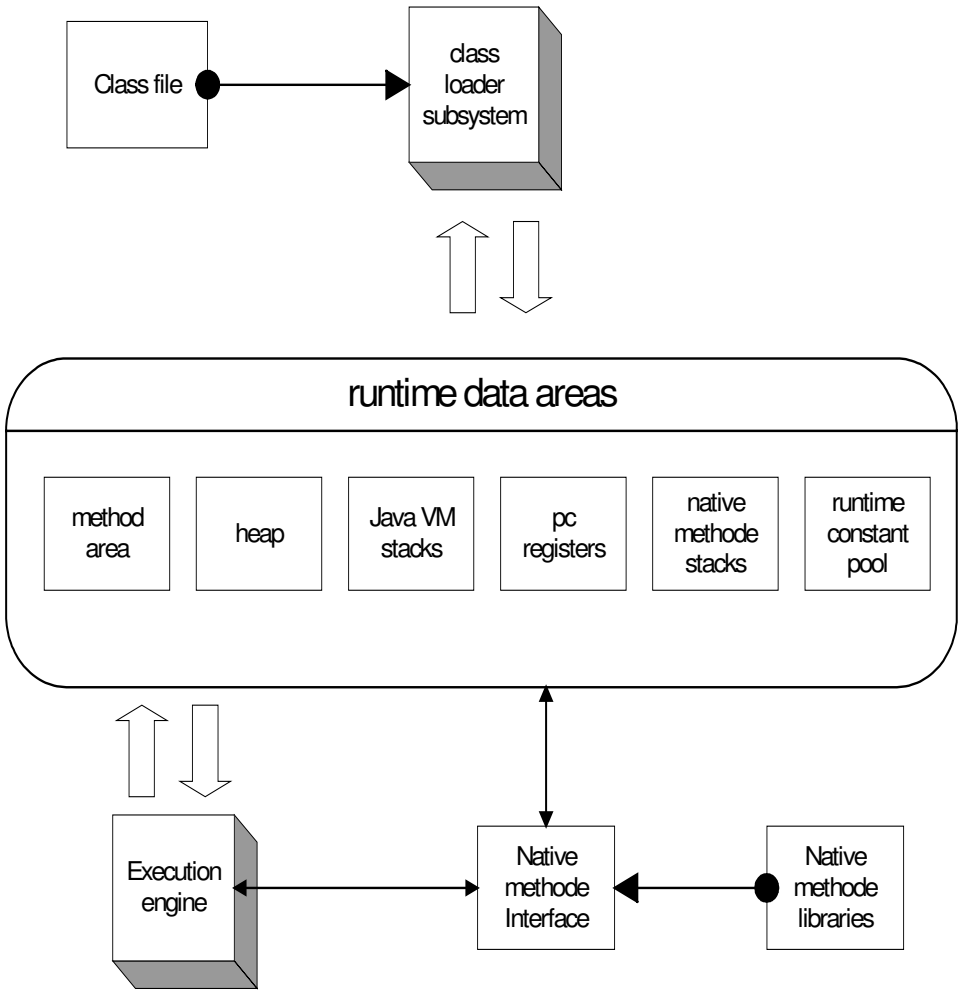

Abbildung 1 - Aufbau der JVM

2.1 Laufzeitumgebung

Unter Laufzeitumgebung (JRE – Java Runtime enviroment) versteht man ein Softwarepaket, welches zur Ausführung von Java-Programmen benötigt wird. In diesem Paket befindet sich die JVM für eine bestimmte Maschine und die wichtigsten Bibliotheken. Sinnvollerweise ist die JVM in mehrere Module aufgeteilt – welche je nach Bedarf ausgeführt werden.

2.2 Class-Loader

Der Class-Loader ist ein wichtiger Bestandteil der JVM – er übernimmt die Aufgabe, Klassen, welche für die Programmausführung benötigt werden, zu finden und zu laden. Weiter hat der Class-Loader die Aufgabe, sich um die Korrektheit der geladenen Klasse zu kümmern, auch muss er den benötigten Speicher reservieren und initialisieren und sich um die Auflösung (Resolution) von symbolischen Referenzen kümmern.

Die JVM kennt zwei unterschiedliche Arten von Class-Loadern, den Bootstrap Class-Loader und den User-Defined-Class-Loader, wobei der Boostrap Class-Loader der Standard-Class-Loader der JVM ist.
Jede JVM muss in der Lage sein, Klasseninformationen aus den class-Dateien, welche laut Java-Spezifikation aufgebaut sein müssen, auslesen zu können. Für das Auslesen der Informationen muss jede JVM einen Bootstrap Class-Loader besitzen.
Der User-Defined Class-Loader ist ein Class-Loader, welcher vom Programmierer entworfen wird, und ist somit nicht Bestandteil der JVM, sondern der Applikation. Der benutzerdefinierte Class-Loader wird von der Klasse java.lang.Class-Loader abgeleitet. Mit den zur Verfügung stehenden Methoden kann ein Programmierer eine neue Klasse erzeugen, welche die übergebenen binären Daten enthalten soll (defineClass).
„Mit einem eigenen ClassLoader kann man erreichen, dass Klassen neu geladen werden. Da Klassen in Java nicht nur über ihr Paket und ihren Namen, sondern auch über ihren ClassLoader identifiziert werden, lassen sich mehrere Versionen einer Klasse laden ohne dass Konflikte auftreten. Man muss lediglich immer, wenn man eine Klasse neu laden will, eine neue Instanz eines ClassLoaders verwenden.“ [Self05]

Von geladenen Klassen ist es möglich, eine Instanz (Instanziierung) zu erzeugen. Im gleichen Zug können auch Klassen wieder freigegeben werden (Finalisierung).

2.2.1 Laden

Im Schritt des Ladens einer Klasse hat der Class-Loader die Aufgabe, die Klasseninformationen zu finden und zu importieren. Hierzu muss der entsprechende Class-Loader (Bootstrap oder User-Definded Loader) zu einem Klassennamen die entsprechende Binärdatei finden und diese dann in die Method Area importieren.
Im Schritt des Ladens wird eine Instanz von java.lang.Class erstellt, welche die geladene Klasse vertritt.
Für den Class-Loader ist es auch möglich, andere Formate als Java class-Dateien zu erkennen und zu laden.

2.2.2 Linken

Der Schritt des Linkens unterteilt sich in drei Unterschritte.

2.2.2.1 Verifizierung

Im Schritt der Verifizierung wird die zu ladende Klasse auf ihre Korrektheit hin überprüft. Dieser Schritt kann je nach Architektur schon während des Ladens oder erst bei der Initialisierung durchgeführt werden.

Beim Prüfen wird z.B. beachtet, dass final-Klassen keine Nachkommen besitzen, oder dass final-Methoden nicht überschrieben werden. Ein wesentlicher Schritt ist die Überprüfung des Bytecodes auf Korrekheit, z.B. das kein `int` einem `string` zugewiesen wird. Im Fehlerfall wird von der JVM eine definierte Exception geworfen.

2.2.2.2 Vorbereitung

Im Vorgang der Vorbereitung wird der benötigte Speicher für die Klassenvariablen reserviert und initialisiert, d.h. je nach Datentyp mit dem Wert false, null, 0 oder 0.0 belegt.

2.2.2.3 Auflösen (Resolution)

Die beim Kompilieren des Java-Codes erzeugten class-Dateien, welche mittels der symbolischen Referenz auf Klassen, Methoden und Felder im Konstantenpool miteinander verbunden sind, werden beim Auflösen durch direkte Referenzen transformiert. Das Umwandeln der symbolischen Referenzen in direkte Referenzen ist wichtig, damit das Programm diese symbolischen Referenzen verwenden kann. Beim Laden wird vom CL für jede Referenz auf fremde Klassen der voll qualifizierte Klasen-, Methoden- oder Feldname in den Konstantenpool geschrieben. In der Auflösungsphase werden diese Einträge mittels eines „Zeigers“ auf den Eintrag in der Method Area ersetzt. Eine Überprüfung der Existenz des referenzierten Eintrags muss in diesem Schritt durchgeführt werden.

2.2.3 Initialisieren

Im letzten Schritt, dem Initialisieren, wird die Klasse oder das Interface mit Werten vorbelegt, welche vom Programmierer als Startwerte vorgegeben wurden.
Das Initialisieren muss genau dann ausgeführt werden, wenn ein Typ das erste Mal verwendet wird, z.B.:

- Erzeugung eines Objektes mit new
- Verwendung / Wertzuweisung einer statischen Methode
- Initialisierung einer Unterklasse
- Aufruf der main()- Methode
- Verwendung von Methoden aus der Reflection-API

Weiter werden beim Initialisieren statische Variablen mit dem Startwert vorinitialisiert, wenn dieser ungleich `null` ist. Der Wert `null` wird automatisch allen Variablen in der Vorbereitungsphase, wenn der Speicher reserviert wird, vergeben.

Beim Initialisieren von statischen Variablen wird zwischen zwei Arten, dem Class Variable Initializer und dem Static Initializer unterschieden, welche in folgenden Beispielen dargestellt werden.
Class Variable Initializer:

```
class Beispiel1{
      static int i = 3*6*Math.random();
}
```

Static Initializer:

```
class Beispiel2{
      static int i;
      static{
            i = 3*6*Math.random();
```

```
	}
}
```

Der Java-Compiler sammelt die Informationen der Initialisierung (von beiden Initializer) in der Methode <clinit> (Methodenname inklusiv der eckigen Klammern), welche nur von der JVM aufgerufen werden kann.

Beispiel für `<clinit>`-Methode:

```
class Beispiel3{
	static int a;
	static int b;
	static int c = 3;
	final static int d = 4;

	// Statischer Initializer
	static{
		b = 2;
	}
}
```

Für die Beispielklasse `Beispiel3` würde der Compilier folgende `<clinit>`-Methode erzeugen:

```
static void <clinit>{
	b = 2;
	c = 3;
}
```

In der erzeugten Methode taucht die Variable `a` nicht auf, da diese nicht mit einem Wert ungleich null initialisiert wird. Die Variable `d` wird ebenfalls nicht aufgeführt, da sich diese durch `final static` als Konstante und nicht als statische Variable ausgibt.

Jeder Class-Loader verwaltet seine eigenen Namespaces, in dem alle Klassennamen liegen, welcher der Class-Loader geladen hat. Beim Auflösen von Referenzen zwischen zwei Klassen müssen beide Klassen vom gleichen Class-Loader geladen sein. Dadurch ist es möglich, dass mehrere Class-Loader die gleiche Klasse laden.

2.3 Register

Die JVM besitzt keine Register, in welche die Daten zur Ausführung von Operationen geladen werden können. Aus diesem Grund werden die Operationen auf dem Operandenstapel ausgeführt. Allerdings besitzt die JVM zur internen Verwaltung vier Register, den Programm Counter (PC), das VARS-Register, das OPTOP- und das FRAME-Register.

Mittels der Information des PC, welches die Adresse des nächsten auszuführenden Opcodes beinhaltet, wird dieser Code eingelesen (siehe *„Programm Counter“*). Das OPTOP-Register zeigt auf den ersten Wert im Operanden-Stack (siehe „Stack“), welcher zur Auswertung von arithmetischen Ausdrücken dient. Das FRAME-Register verweist auf den aktuellen Methodenrahmen. Das VARS-Register hat die Aufgabe, auf die lokalen Variablen der aktuellen Methode zu zeigen.

Dadurch, dass laut Spezifikation der JVM keine Register benötigt werden, kann auch eine JVM für Prozessoren mit wenigen Registern realisiert werden, welche effizient Programme ablaufen lassen kann.

2.4 Execution-Engine

Die Execution-Engine (EE) ist die wichtigste Komponenten der JVM, da diese für das Ausführen des Java Bytecodes verantwortlich ist. In der JVM-Spezifikation ist das Verhalten der Execution-Engine exakt beschrieben, wie sich die Execution-Engine beim Ausführen eines Befehls verhalten soll, allerdings bleibt offen, wie dies zu realisieren ist.
Die EE ist in der Lage, Bytecode zu interpretieren, nativen Code und Quellcode zu kompilieren.

Für den Begriff der Exection-Engine gibt es drei unterschiedliche Bedeutungen:

- abstrakte Computer Spezifikation der Execution-Engine
- konkrete Implementierung der Execution-Engine
- Laufzeitinstanz der Execution-Engine

Mit der **abstrakten Spezifikation** wird wieder das Verhalten der Execution-Engine für einzelne Instruktionen der JVM bestimmt. Die **konkrete Implementierung** der EE kann als reine Softwareapplikation oder als Hardwareversion realisiert werden – natürlich ist auch eine Mischform aus beidem möglich. In der konkreten Implementation kann Code interpretiert oder „Just-in-time“ compiliert werden (Ausführen von nativem Code). Jeder Thread, welcher durch eine Applikation gestartet wird, stellt eine **Laufzeitinstanz** der EE dar.

Der Bytecode für die JVM besteht aus einer Sequenz von Instruktionen, wobei jede dieser Instruktionen aus dem Opcode und folgenden Operanden besteht. Der Opcode gibt der JVM an, welche Aktion von ihrer Seite ausgeführt werden soll und der Operand stellt für die Aktionen nötigen Informationen zur Verfügung. Die Anzahl und der Typ der Operanden einer Instruktion wird durch den Opcode bestimmt. Die möglichen Befehle des Opcodes werden Befehlssatz der JVM oder Instruction Set genannt.

Für das Ausführen von Bytecode durch die EE haben sich verschiedene Techniken entwickelt. Da in der Spezifikation keine Aussage über die Ausführung von Bytecode getroffen wird, stehen verschiedene Möglichkeiten zur Verfügung. Die folgenden drei Ansätze sind häufig verwendete Methoden, welche nicht direkt mit der Architektur der JVM zu tun haben, werden aber aus Gründen der Vollständigkeit hier erwähnt.

2.4.1 Interpretation

Die einfachste Technik zur Ausführung von Bytecode ist das Interpretationsverfahren, bei dem die Instruktion aus dem Bytecode gelesen und ausgeführt wird. Der Code wird nicht in nativen Code kompiliert, wobei auch keine Codeoptimierung durchgeführt werden kann. Dieses Verfahren wird in heutigen JVM’s nicht mehr eingesetzt, da das Interpretationsverfahren für heutige Anwendungen zu langsam ist.

2.4.2 Just-In-Time (JIT) Kompilation

Ein schnellerer und moderner Ansatz der Ausführung von Bytecode ist die Just-In-Time Kompilation, bei der Methoden bei ihrem Aufruf in nativen Code übersetzt werden. Bei

diesem Verfahren besteht die Möglichkeit der Codeoptimierung, da dem Compiler zur Laufzeit bereits Informationen über das Verhalten des Codes vorliegt.
Diese Technik wird in heutigen JVM's häufig eingesetzt, da sie relativ leicht zu implementieren ist.

2.4.3 Adaptive Optimierung

Der modernste Ansatz der Codeausführung ist das Verfahren der adaptiven Optimierung, bei dem das Interpretations- und JIT-Verfahren miteinander vereint werden. Durch dieses Verfahren ist ein maximaler Performancegewinn möglich.
Bei der adaptiven Optimierung werden die Instruktionen durch den Interpreter ausgeführt, wobei das Verhalten des Codes von der JVM beobachtet wird. Durch das beobachten ist ein Auffinden von „Hot Spots" möglich. Hot Spots sind Codezeilen die ca. 80 – 90 % der Ausführungszeit benötigten, aber insgesamt nur 10 – 20 % vom Quellcode ausmachen. Diese „Hot Spots" werden mit starker Optimierung in nativen Code kompiliert.

2.5 Native Method Interface

Das Native Method Interface (NMI) ist eine Schnittstelle zwischen Java und dem Betriebssystem, über welche betriebssystemspezifische Methoden aufgerufen werden können (z.B. Windows DLL).
Eine Spezifikation über diese Schnittstelle existiert nicht und hängt von den jeweils verwendeten Bibliotheken ab. Sun hat eine eigene Spezifikation für diese Schnittstelle entwickelt, das Java Native Interface (JNI), welche ein Vorschlag ist, mit dem Ziel als Standard definiert zu werden.
Der Programmierer einer Virtuellen Maschine kann eine eigene Schnittstelle zum Betriebssystem spezifizieren.

Der große Nachteil bei der Verwendung von nativen Methoden ist der Verlust der Plattformunabhängigkeit des Java-Codes, da sich diese Methoden auf eine bestimmte Plattform / Betriebssystem beziehen.

Bei der Implementierung eines NMI muss darauf geachtet werden, dass zum Beispiel der Garbage Collector keine Objekte löscht, welche von nativen Methoden benötigt werden.

3 Speicher

Im folgenden Kapitel werden die einzelnen in Abbildung 2 skizzierten Speicherbereiche der JVM genauer beschrieben.

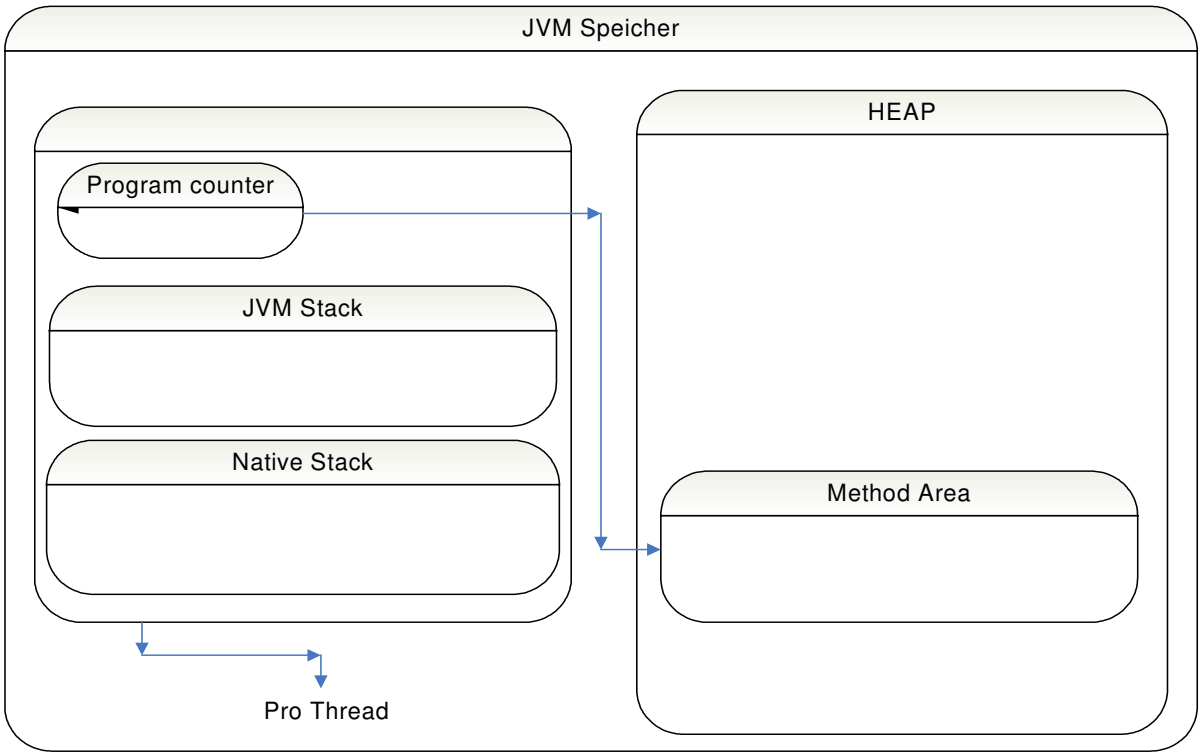

Abbildung 2 - Speicherübersicht

3.1 Java Stack

Jeder Thread, der erzeugt wird, hat einen eigenen Java Stack. Dieser Java Stack wird bei der Erzeugung des Threads erstellt. Jeder Thread kann nur auf seinen eigenen Java Stack zugreifen. Auf dem Java Stack werden zu den einzelnen Methoden Stack Frames angelegt. Die Stack Frames legt und entfernt die JVM durch die Befehle push und pop. Mit push wird ein Stack Frame auf der Spitze des Java Stacks angelegt. Mit pop wird das oberste Stack Frame vom Java Stack entfernt. Der Stack Frame der sich ganz oben auf dem Java Stack befindet, ist der aktuelle Stack Frame und gehört zu der aktuellen Methode. Wird eine Methode durch normales zurückkehren oder das Werfen einer Exception beendet, so wird der aktuelle Stack Frame verworfen.

Der Stack Frame enthält den Zustand von nicht nativen Methodenaufrufen, deren lokal Variablen, Übergabeparameter, den möglichen Rückgabewert sowie Methoden interner Berechnungsergebnisse. Die Einträge sind in 32-Bit Worten organisiert. Jeder Frame enthält alle Eingangs- und Ausgangsdaten der verschiedenen Berechnungen, Referenz auf runtime constant pool, implementierungsspezifische Informationen und Debugging Informationen. Der Stack Frame besteht aus drei Teilen:

- Lokale Variablen
- Operanden Stack
- Frame Daten

Die Größen dieser Teile sind außer bei den Frame Daten jeweils von der jeweiligen Methode abhängig. Die Frame Daten selber sind von der Implementierung abhängig.

3.1.1 Lokale Variablen

Im Stack Frame werden die lokalen Variablen der jeweiligen Methode gespeichert.
Die lokalen Variablen sind als 0-basiertes Array aus Wörtern organisiert und enthalten Parameter und lokale Variablen einer Methode.

3.1.2 Operanden Stack

Der Operanden Stack ist ebenfalls als Array aus Wörtern organisiert. Er besteht aus denselben Datentypen wie die lokalen Variablen.

3.1.3 Frame Daten

Die Frame Daten bestehen aus:

- einem „Zeiger“ auf den Constant Pool
- Informationen über den vorherigen Stack Frame
- Referenzen auf die Exception Tabelle der Methode

3.2 Native Method Stack

Der Native Method Stack ist ein implementierungsspezifischer Speicherbereich zur Behandlung von nativen Methodenaufrufen. Dieser hat dabei die gleiche Aufgabe wie der Stack, jedoch für die nativen Methoden. Dieser Stack ist nur einmal je JVM vorhanden und wird von allen Threads gemeinsam genutzt. Wird eine native Methode aufgerufen, so verlässt der Thread den Java Stack und begibt sich in den Native Method Stack.

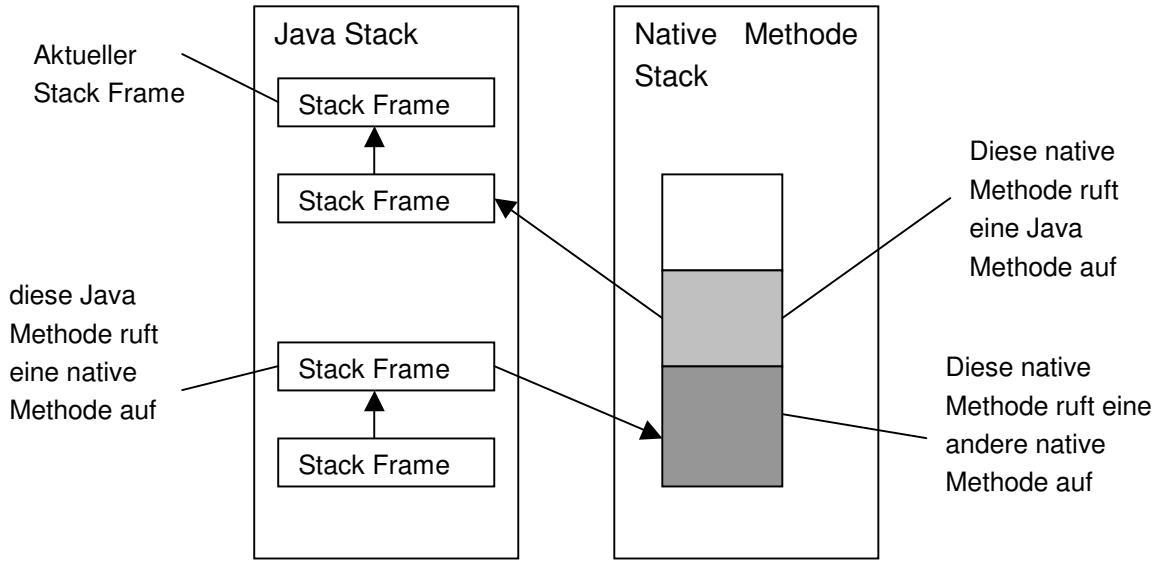

Abbildung 3 - Methodenaufrufe

Abbildung 3 zeigt wie ein Ablauf zwischen Java Stack und Native Methode Stack aussehen könnte. Am Anfang existiert auf dem Java Stack einen Stack Frame. Diese ruft einen weiteren Stack Frame auf, darauf hin wird auf dem Java Stack ein weiterer Stack Frame angelegt. Dieser wiederum ruft eine native Methode auf. Daher muss ein Wechsel auf den

Native Method Stack erfolgen. In diesem Fall wird aber kein Stack Frame auf dem Native Method Stack erzeugt, sondern der vorhandene benutzt. Wird von der Methode jetzt eine neue native Methode aufgerufen, dann wird der bisherige Stack Frame weitergeführt. Ruft die native Methode nun eine Java Methode auf, so wird auf dem Java Stack noch ein Stack Frame angelegt. Dieser ruft wieder eine Java Methode auf und erzeugt somit wieder einen Stack Frame.

3.3 Heap

Auf dem Heap der JVM werden neu erstellte Objekte abgelegt. Dieser ist nur einmal pro JVM vorhanden und wird gemeinsam von allen Threads genutzt. Daher muss die JVM dafür sorgen, dass der Zugriff von mehreren Threads synchronisiert wird.
Der Benutzer muss den benötigten Speicher nicht mehr freigeben, dies liegt daran, dass die JVM mit einer Freispeicherverwaltung arbeitet. Die Garbage Collection gibt den Speicher wieder frei, wenn er nicht mehr benötigt wird. Man kann die Garbage Collection bei Bedarf auch explizit aufrufen.
Ein Objekt besteht aus Instanzvariablen und Zeigern auf Klassendaten. Eine Instanzvariable besteht aus der Klasse des Objektes und all seinen Superklassen.
In Java ist ein Array ebenfalls ein Objekt und wird auf dem Heap angelegt.

3.4 Programm Counter

Für jeden Thread, der erzeugt wird, wird ein Programm Counter angelegt. Der Programm Counter enthält immer die Adresse des aktuell ausgeführten Befehls. Dabei ist der Programm Counter genau ein Wort groß. Wird eine native Methode ausgeführt, so ist der Wert des Programm Counters undefiniert.

3.5 Method Area

Die Method Area ist nur einmal je JVM vorhanden und wird von allen Threads gemeinsam genutzt. Der Class-Loader einer Klasse liest die Typinformationen aus und übergibt sie der JVM an die Method Area. Zu jeder geladenen Klasse gibt es Basis-Informationen und erweiterte Informationen. Die Basis-Informationen wären:

- Voll qualifizierter Name
- Superklasse
- Klasse oder Interface
- Modifiers
- Superinterfaces

Die erweiterten Informationen wären:

- Constant Pool für die Klasse, mit Konstanten der Klasse und symbolischen Referenzen zu Klassen, Feldern und Methoden
- Felderinformationen und Felderreihenfolge
- Methodeninformationen, Methodenreihenfolge und Bytecode
- Exception Tabe für jede Methode
- Statische Variablen
- Referenz auf Instanz von ClassLoader
- Referenz auf Instanz von Class

3.5.1 Constant Pool

Für jeden Typ in der Method Area legt die JVM einen Constant Pool an. Dieser wird als Liste aller Konstanten dargestellt, die in diesem Typ verwendet werden, dies bedeutet alle Literale und symbolische Referenzen auf Typen, Felder und Methoden. Der Constant Pool wird über Indizes angesprochen, da er in Form eines Felder angelegt ist.

3.5.2 Felderinformationen

Für jedes Feld, das in einem Typ deklariert wird, müssen Informationen wie Name, Typ und Modifier des Feldes gespeichert werden. Zusätzlich wird die Reihenfolge, in der die Felder deklariert worden sind, festgehalten.

3.5.3 Methoden Informationen

Für jede Methode, die in einem Typ gespeichert wird, müssen Informationen wie Name, Rückgabewert und Modifier der Methode, sowie Anzahl und Typ der Parameter gespeichert werden. Zusätzlich wird wie bei Felder auch noch die Reihenfolge festgehalten. Für jede Methode, die weder abstrakt noch nativ ist, müssen auch, Informationen wie der Bytecode der Methode, die Größe des Operanden Stack und die Anzahl der lokalen Variablen des Stack Frame der Methode, sowie eine Exception Table abgelegt werden.

3.5.4 Referenz auf Instanz von Class Loader

Die JVM muss für jeden Typ festhalten, ob dieser vom Bootstrap oder von einem User Defined Class Loader geladen wurde. Beim Laden von einem User Defined Class Loader muss in der Method Area eine Referenz auf diesen Class Loader gespeichert werden.

3.5.5 Referenz auf Instanz von Class

Für jeden geladenen Typ muss die JVM eine Instanz der Klasse java.lang.Class erzeugen und eine Referenz darauf in der Method Area festhalten.

3.5.6 Beispiel

```
class Auto {
      private int speed = 5;
      void drive(){}
}
class Fahrzeuge{
      public static void main(String[] args){
            Auto auto =  new Auto();
            Auto.drive();
      }
}
```

Der möglich Ablauf zur Ausführung dieser Klasse wäre:

1. finde und lese Datei Fahrzeuge class
2. Speichere Informationen in Method Area
3. erzeuge Zeiger zum Constant Pool von Fahrzeuge
4. arbeite Bytecode in Method Area für main() ab
    a. reserviere Speicher für Klasse an Stelle 1 im Constant Pool
        i. Constant Pool → symbolische Referenz auf Auto
        ii. wurde Auto bereits in Method Area geladen ? Nein !
        iii. lade Lava.class in Method Area
        iv. CP Eintrag 1 → Zeiger auf Klassendaten von Auto (CP Resolution)
        v. suche Objektgröße von Auto aus den Klassendaten
        vi. reserviere Speicher für Auto-Objekt auf dem Heap
    b. initialisiere Instanzvariablen auf Default-Werte (0, null)
    c. gib Referenz auf Auto-Objekt auf den Stack

4 Spezifikationen

4.1 CLASS-Datei

Der Java-Compiler erzeugt für jede Klasse eine class-Datei, in der Informationen über die Klasse und den Bytecode der Klasse gespeichert sind. Folgende Informationen werden angegeben:

- Beschreibung der abgeleiteten Oberklasse
- Implementierte Schnittstellen
- Anzahl, Name und Typbeschreibung der statischen Variablen und Instanz-Variablen der Klasse
- Anzahl, Name und Signaturen der statischen Methoden und Instanz-Methoden
- Zugriffsrechte
- Sichtbarkeit

Jede class-Datei enthält entweder genau eine Java-Klasse oder ein Java-Interface und kann als eine Bytefolge betrachtet werden. Die Struktur der class-Datei wird im folgenden dargestellt:

```
ClassFile {
    u4 magic;
    u4 version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count -1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
  }
```

Die Datentypen der class-Datei werden mit u2 und u4 dargestellt. Mit u2 ist ein zwei Byte und mit u4 eine vier Byte große Einheit gemeint.
Die einzelnen Objekte in der Datei haben folgende Bedeutung:

- magic
  Dient zur Identifikation und besitzt den Wert 0xCAFEBABE
- version
  Gibt die Version des Java Compilers an, der die class-Datei erzeugt hat
- constant_pool_count
  Gibt die Anzahl der Einträge im ConstantPool an
- constant_pool[]
  Eine Tabelle mit Konstanten, die Informationen enthalten (siehe Speicher)
- acces_flags
  Maske, in der die Sichtbarkeit und Benutzung dieser Klasse bzw. Interface kodiert ist.

- this_class
  Zeiger in den Constant Pool, Name der Klasse bzw. des Interfaces
- super_class
  Zeiger auf den Namen der Superklasse
- interfaces_count
  Anzahl der Interfaces der Klasse oder des Interfaces
- interfaces[]
  Zeiger der auf die Interfaces zeigt
- fields_count
  Anzahl der Klassenvariablen
- fields[]
  Klassenvariablen
- methods_count
  Anzahl der Methoden
- methods[]
  Elemente welche die Methoden repräsentieren
- attributes_count
  Anzahl von Attributen
- attributes[]
  Attribute (Zurzeit wird nur das Attribut mit dem Name der Quelldatei unterstützt, alle anderen werden überlesen)

4.1.1 Konstantenpool

Der Constant Pool repräsentiert die Symboltabelle einer class Datei. Auf folgende Art und Weise ist ein Eintrag im `constant_pool` eingetragen:

```
cp_info {
    u1 tag;
    u1 info[];
  }
```

Jeder Eintrag beginnt mit einem Indikator, der in `tag` abgelegt wird, dieser beinhaltet den Typ dieses Eintrags. Der Inhalt von `info` hängt von diesem Indikator ab. Die in Tabelle 1 dargestellten Werte sind dabei zulässig.

Typ	Wert
CONSTANT_Utf8	1
CONSTANT_Unicode	2
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_Class	7
CONSTANT_String	8
CONSTANT_Fieldref	9

CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_NameAndType	12

Tabelle 1 - Tag Eintrag im Konstantenpool

4.1.2 Felder

So sieht ein Eintrag in `fields` aus:

```
field_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
  }
```

Der Eintrag `access_flags` stellt die Zugriffsrechte und Eigenschaften dar. Dabei gib es die in Tabelle 2 aufgeführten Möglichkeiten.

Bit	Bedeutung
0	ACC_PUBLIC
1	ACC_PRIVATE
2	ACC_PROTECTED
3	ACC_STATIC
4	ACC_FINAL Bit
5	ACC_VOLATILE
6	ACC_TRANSIENT
7-15	ignoriert

Tabelle 2 - Access-Flags

Der Eintrag `name_index` ist der Index zu der Tabelle Constant Pool zu dem Feld CONSTANT_Utf8_info Item Name. Der Eintrag `descriptor_index` ist der Index zu der Tabelle Constant Pool zu dem Feld CONSTANT_Utf8_info Item Descriptor. Der Eintrag `attributes_count` gibt die Anzahl der Einträge in der Tabelle `attributes` an. Der Eintrag `attribute_info attributes[attributes_count]` ist die Tabelle von Attributen des Feldes.

4.1.3 Methoden

```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
  }
```

Die einzelnen Einträge sind genauso aufgebaut wie die Felder.

4.1.4 Attribute

So sehen Daten von Attributen aus:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
  }
```

Der Eintrag `attribute_name_index` ist der Index zu der Tabelle Constant Pool zu dem Feld CONSTANT_Utf8_info Item.
Es existieren Attribute, die von der JVM unbedingt erkannt werden müssen, weil diese für die Ausführung, Compilation und das Debugging wichtig sind. Folgende Attribute müssen erkannt werden:

- SourceFile Klassen
- ConstantValue Felder
- Code Methoden
- Exceptions Methoden
- InnerClasses Klassen
- Synthetic Klassen, Felder, Methoden
- LineNumberTable Attribut Code
- LocalVariableTable Attribut Code
- Deprecated Klassen, Felder, Methoden

4.2 Datentypen

Jede Maschine kennt primitive Datentypen wie BITS (AND, OR, EXOR, NOT), BYTES (Vergleich auf Identität, Shift, selektives EXOR) und WORD (Vergleich auf Identität, Shift, selektives EXOR), auf welchen verschiedene Operationen (Beschreibung hinter den Datentypen) ausgeführt werden können. Abbildung 4 und Tabelle 3 zeigen die verschiedenen Datentypen der JVM. In der Spezifikationen der JVM ist genau beschrieben, welche Datentypen existieren und welche Operationen darauf ausgeführt werden dürfen. Vorzeichenbehaftete Ganzzahlen sind im Zweierkomplement, Fließkommazahlen im IEE 754-Format und Zeichenwerte im Unicode-Format spezifiziert.

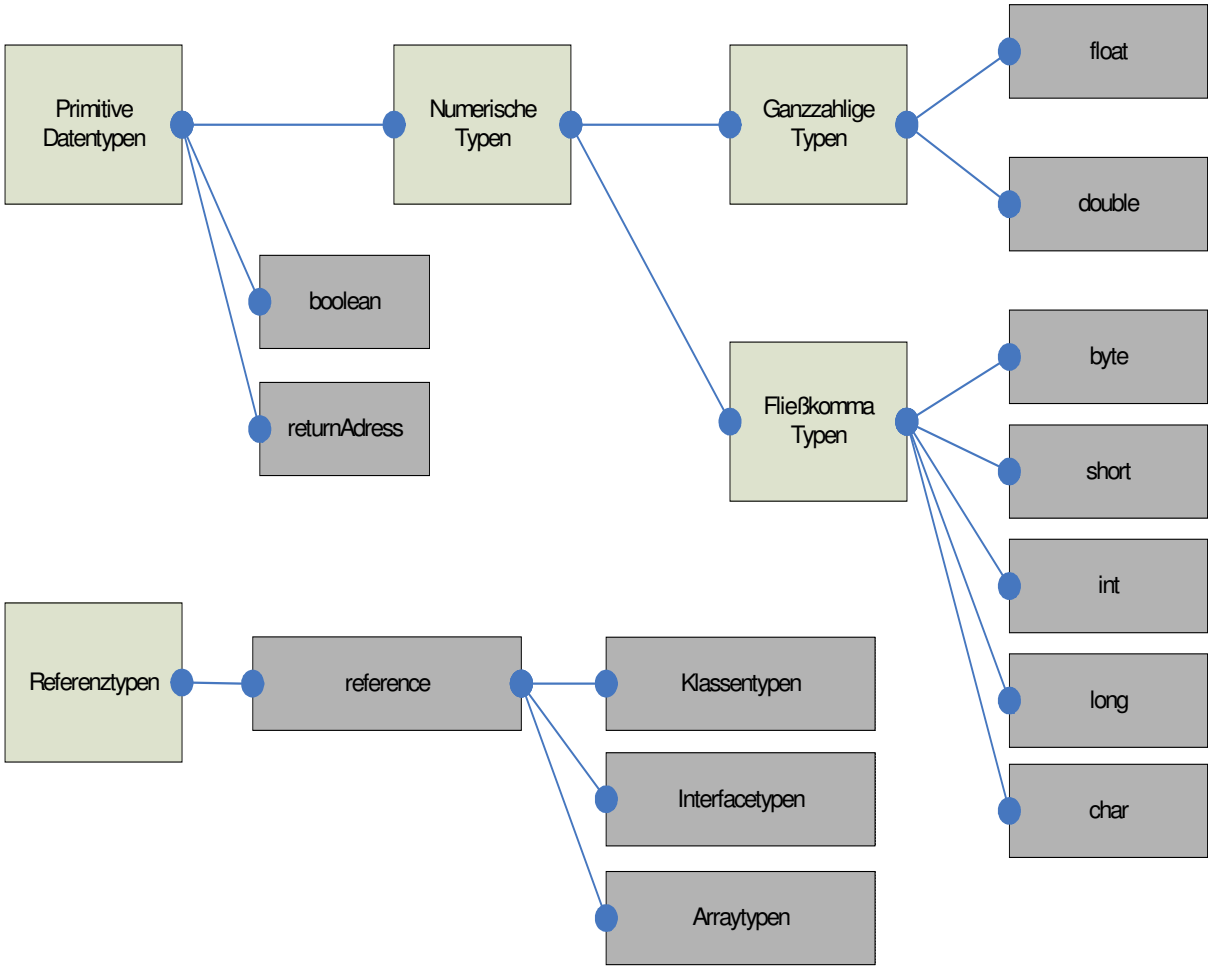

Abbildung 4 - Datentypen in Java

Allgemein können alle Datentypen der JVM in primitive Typen oder Referenz Typen unterteilt werden. Primitive Typen stellen den Inhalt einer Variable dar (z.B. Zahlen) und Referenz Typen verweisen auf den Inhalt einer Variable. Dadurch sind Referenz-Typen keine Objekte, sondern der „Zeiger“ auf ein Objekt im Heap.

Datentyp	Wertebereich	Kommentar
`byte`	8 bit - [-128 ; 127]	Ganzzahl mit Vorzeichen
`short`	16 bit - [-32768 ; 32767]	Ganzzahl mit Vorzeichen
`int`	32 bit - [-2^{31} ; 2^{31}-1]	Ganzzahl mit Vorzeichen
`long`	64 bit - [-2^{63} ; 2^{63}-1]	Ganzzahl mit Vorzeichen
`char`	16 bit - [0 ; 32767]	Nicht Vorzeichenbehaftet
`boolean`	8 oder 16 bit	
`float`	32 bit	Fließkommazahl mit einfacher Genauigkeit
`double`	64 bit	Fließkommazahl mit doppelter Genauigkeit
`returnAddress`	Adresse eines Befehls	
`reference`	Referenz auf ein Objekt im Heap oder Wert null	

Tabelle 3 - Gesamtübersicht Datentypen

4.2.1 Ordinale Typen

Unter ordinalen Zahlenwerten versteht man u.a. Ganze Zahlen, Natürliche Zahlen und Zeichen.
In Java sind sechs ordinale Datentypen in der Form von `byte`, `short`, `char`, `int`, `long` und `boolean` bekannt.
Sämtliche primitiven Typen werden von der JVM unterstützt, wobei es keine Befehle gibt, welche den Typ `boolean` verwenden. Variablen vom Typ `boolean` werden beim Kompilieren durch den Typ `int` oder `byte` ersetzt, wobei der Wert `false` durch den Zahlenwert `0` und der Wert `true` durch den Zahlenwert `1` dargestellt wird.
In den Speicherbereichen der JVM werden `long` Variablen in zwei Wörtern gespeichert, und sämtliche andere Variablen als `int` (ein Wort) gespeichert.

4.2.2 Gleitkomma Typen

Gleitkommazahlen werden von der JVM in den Typen `float` oder `double` gespeichert. Der `float` Typ ist 1 Wort und der `double` Typ ist 2 Wörter groß.
Für `float` und `double` Typen ist das Vorzeichen nicht angegeben, da die darstellbaren Zahlen im Wertebereich positiv wie negativ sein können.

4.2.3 Referenz Typen

Der Referenz Typ wird als `reference` bezeichnet und stellt 3 verschiedene Referenztypen dar: Klassentypen, Interfacetypen und Arraytypen. Diese drei Typen beinhalten Werte, welche eine Referenz auf ein Objekt darstellen, wobei Klassentypen nur Referenzen auf Klasseninstanzen, Interfacetypen nur Referenzen auf Klasseninstanzen, welche ein Interface implementieren, und Arraytypen nur Referenzen auf Arrays darstellen.
Der `reference` Typ ist ein Wort langer „Zeiger“ auf den Heap.

4.2.4 Sonstige Typen

Der JVM ist noch ein zusätzlicher Datentyp, die `returnAddress` bekannt, welcher allerdings nicht in der Programmiersprache verwendet wird. Dieser Typ ist notwendig, um finally Klauseln darstellen zu können und wird mittels einem 1 Wort langen Opcode innerhalb der aktuellen Methode dargestellt.

4.3 Lebenszyklus von Objekten und Klassen

Jede Klasse und jedes Objekt besitzt einen Lebenszyklus. Während des Programmablaufs wird ein Objekt einmal erzeugt, ändert dann seinen Zustand und wenn es nicht mehr benötigt wird, muss er zerstört werden. (Beispiel Auto: Ein Auto wird in der Fabrik produziert [Objekt erzeugen] – dieses Fahrzeug ändert im Betrieb seine Zustand z.B. der Tankinhalt verändert sich [Objekt ändert Zustand] – nach einer gewissen Zeit wird das Fahrzeug nicht mehr benötigt [Objekt wird gelöscht]). In Java wird ein Objekt automatisch durch den Garbage Collector zerstört, welcher nicht mehr benötigte Objekte aufsammelt und dann löscht.
Bei Klassen bedeutet das Erzeugen einer Instanz den Beginn des Lebenszyklusses. Wurde eine Klasse geladen ist es möglich von ihr Instanzen (Instanzierung) zu erzeugen und diese wieder zu löschen (Finalisierung). Die Dauer des Lebenszyklus hängt stark von der Arbeitsweise des Classloaders und der Garbage Collection ab.

5 Zusammenfassung

Die Architektur der Java Virtual Machine ist eine nicht exakt beschriebene Spezifikation, welche von dem jeweiligen Implementierer für ein bestimmtes Zielsystem interpretiert werden kann, da die Spezifikation keine detaillierte Beschreibung zur Realisierung der Bestandteile liefert.
Interessant ist die Tatsache, welcher Aufwand in Kauf genommen werden muss, damit eine JVM für ein beliebiges Zielsystem programmiert werden kann, da das Verhalten der JVM nach außen hin, zum Java Programm sich exakt wie in der Spezifikation angegeben, verhalten muss.
In dieser Ausarbeitung wurden die wichtigsten Bestandteile einer JVM beschrieben, ohne auf zu detaillierte Informationen einzugehen. Genauerer Informationen können aus [LY99] entnommen werden.

6 Quellenangaben

6.1 Literaturverzeichnis

- [LY99] Lindholm, Tim und Frank Yellin: *The Java Virtual Machine Specification Second Edition*. Addison-Wesley-Verlag, München (Deutschland), 1999.
- [Sch03] Schwarzbauer, Christian: Seminararbeit *Die Architektur der Java-VM.* Universität Linz (Österreich), 2003
- [Stum03] Stumptner, Reinhard: Seminararbeit *Dynamisches Laden und Binden in Java.* Universität Linz (Österreich), 2003.

6.2 Internetlinks

- [Jeck05] Scriptum zur Vorlesung Java
  www.jeckle.de/vorlesung/java/script.html, Abruf am 24.10.2005
- [Tue05] Die Java Virtual Machine
  http://www-ti.informatik.uni-tuebingen.de/~heim/lehre/proseminar_ws9798/jens/java/index.html, Abruf am 24.10.2005
- [Wiki05] Wikipedia
  http://de.wikipedia.org/wiki/Java_Virtual_Machine, Abruf am 24.10.2005
- [Lin05] PowerPoint-Präsentation zur Seminararbeit [Sch03]
  www.ssw.uni-linz.ac.at/Teaching/Lectures/Sem/2003/slides/Schwarzbauer.ppt, Abruf am 24.10.2005
- [Wed05] Die Architektur der Java-VM
  http://www.fh-wedel.de/~si/seminare/ws02/Ausarbeitung/2.jvm/javavm0.htm, Abruf am 24.10.2005
- [Self05] Tipps von selfhmtl.org
  http://aktuell.de.selfhtml.org/tippstricks/java/classloader/, Abruf am 24.10.2005