Fachhochschule Köln, Campus Gummersbach
Fakultät für Informatik und Ingenieurwissenschaften

Institut für Informatik

Der Aufbau der Laufzeitstrukturen einer Java Virtual Machine

Diplomarbeit im Studiengang Allgemeine Informatik

eingereicht von: Daniel Klein (11027858)

eingereicht am: 11. Mai 2007

Betreuer: Prof. Dr. Erich Ehses - Fachhochschule Köln
Prof. Dr. Horst Stenzel - Fachhochschule Köln

Inhaltsverzeichnis

Abbildungsverzeichnis

Vorwort

Wie funktioniert eine Java Virtual Machine? – Diese Frage war es, die ich mir selber vor einiger Zeit stellte. Die Lektüre verschiedener Bücher war nicht wirklich zufriedenstellend, da diese immer nur den theoretischen Teil einer Java Virtual Machine zum Thema hatten. Konkret auf eine Implementierung eingegangen wurde dabei leider jedoch nie. Nachforschungen über existierende Implementierungen ergaben außerdem, dass diese in der Regel für den Anfang viel zu komplex oder fernab von Lesbarkeit und Verständlichkeit realisiert waren. Eine leicht verständliche, übersichtliche und gut dokumentierte Java Virtual Machine musste her und da eine solche nicht aufzutreiben war, lag der Entschluss nahe selber diese Lücke zu füllen, was ich mit dieser Diplomarbeit nun in Ansätzen realisiert zu haben hoffe.

Zwar konnte mit dem schriftlichen Teil dieser Arbeit nicht die gesamte Implementierung einer Java Virtual Machine abgedeckt werden, dennoch hoffe ich, einige übliche Fragen, wie sie beispielsweise nach der Lektüre von [Lindholm99] entstehen, beantwortet zu haben. Sollte Ihre Frage nicht dabei sein, dann hilft Ihnen hoffentlich ein Blick in die Quellen der Implementierung weiter.

Auch wenn nur mein Name auf dem Titel steht, so wäre diese Arbeit nicht möglich gewesen, wenn ich nicht von vielen Personen tatkräftig unterstützt worden wäre. Danken möchte ich deshalb meinem Betreuer Prof. Erich Ehses, der mir immer mit Rat & Tat zur Seite stand, und der immer den passenden Vorschlag hatte, wenn ich einmal nicht weiter wusste. Des weiteren bedanken möchte ich mich bei meinem Zweitprüfer Prof. Horst Stenzel, bei meinen Korrekturleserinnen und -lesern (besonders bei Annika Krisp), bei Bill Venners, der mir freundlicherweise eine seiner Privatkopien seines nicht mehr im

Handel erhältlichen Buches [Venners99] zukommen ließ, bei meiner Freundin Nicole dafür, dass sie mich ertragen und ermuntert hat und ich das Wohnzimmer praktisch für ein halbes Jahr vollständig in Beschlag nehmen durfte, sowie meinen Eltern, die mir zur Seite gestanden, mich immer unterstützt und an mich geglaubt haben.

Bad Neuenahr, im Mai 2007 Daniel Klein

Kapitel 1

Einleitung

Dieses Kapitel bietet eine kurze Einführung in diese Arbeit. Die Motivation des Autors wird erläutert und warum ein potenzieller Leser diese Arbeit lesen sollte. Die Anforderungen, die an das Projekt gestellt wurden, werden erläutert, und der Aufbau der Arbeit erklärt.

1.1 Motivation

> Oddly enough, I've yet to see a book that covers how to build a JVM; every book published so far focuses on the abstract JVM rather than how someone would implement one.
>
> – Godfrey Nolan[1]

Die Motivation, die Implementierung und Dokumentation von Teilen einer Java Virtual Machine (JVM) als Aufgabe für meine Diplomarbeit zu nehmen, entstand praktisch durch eigenes Interesse und aus der Not heraus. Bedingt durch das intensive Arbeiten mit der Programmiersprache Java stellte ich mir in zunehmendem Maße die Frage, wie die Dinge im Hintergrund wohl ablaufen. Als ich mich auf die Suche nach Antworten machte, stieß ich auf zwei Dinge, die mich beide jedoch nicht vollends zufrieden stellen konnten.

[1] Aus [Nolan04] Seite 17.

Als erstes waren da die Java Virtual Machines, deren Quellcode frei zur Verfügung stehen. Im Internet findet man zahlreiche Implementierungen in den unterschiedlichsten Sprachen, sogar Implementierungen in Java sind zu finden. Doch waren diese Implementierungen entweder aus wissenschaftlichen Projekten entstanden und waren sehr umfangreich oder sie kamen aus dem Hobbyumfeld und entsprachen nicht meinen Vorstellungen von lesbarem Quellcode. Gerade die Übersichtlichkeit und Lesbarkeit eines Programms sind aber essentiell um von ihm lernen zu können.

Das zweite Problem bestand darin, dass keine Dokumentation zu finden war, die die Implementierung einer Java Virtual Machine ausführlich erklärt. Zwar findet man umfangreiche Dokumentationen über die Theorie hinter den Implementierungen, Erläuterungen, besonders zu den Basisbestandteilen einer Implementierung, findet man jedoch nirgends.

Aufgrund der Gegebenheiten habe ich mich entschlossen, eine eigene, kompakte und vor allen Dingen möglichst verständliche Java Virtual Machine zu implementieren und diese teilweise im Rahmen der Diplomarbeit auch zu dokumentieren. Eine vollständige Dokumentation hätte den Zeitrahmen eindeutig gesprengt, weshalb ich mich auf die wesentlichen Grundlagen, die Datenstrukturen, ihre Erstellung und Beispiele für deren Verwendung konzentriert habe.

Aber was bringt dem Leser nun die Lektüre diese Arbeit? Warum sollte er sie lesen und den Quellcode studieren? Einfach aus Neugier heraus, wäre eine gut Antwort. Die Praxis bei der Betreuung von Programmierpraktika an der FH Köln hat gezeigt, dass ein reges Interesse an den Hintergründen zu der Funktionsweise einer JVM besteht. Und was liegt näher, als sich diese Fragen am Beispiel einer konkreten aber einfachen Implementierung anzuschauen, anstatt das aufkeimende Interesse durch pure Theorie zu ersticken?

Um dem Leser die Funktionsweise verschiedener Aufgaben der JVM näher zu bringen, werden diese basierend auf Java-Beispielen näher erläutert. Dies soll helfen, die alltägliche Programmierarbeit mit dem Blick hinter die Kulissen zu verbinden. Das neuerworbene Wissen kann gerade wenn es um Optimierungen von Programmen geht, äußerst hilfreich sein.

1.2 Anforderungen

Zu allererst sollte sich das Projekt zur Nutzung als Lernwerkzeug eignen und die damit verbundenen Voraussetzungen erfüllen. Gute Lesbarkeit des Quellcodes ist dabei genau so wichtig wie eine einfache und verständliche Implementierung. Die Dokumentation sollte ebenfalls leicht verständlich geschrieben sein und möglichst wenig Vorwissen voraussetzen.

Um die virtuelle Maschine (VM) auf allen gängigen PC-Plattformen (Windows, Linux, Mac) einsetzen zu können, wurde außerdem Wert darauf gelegt, sie möglichst portabel zu realisieren. Auch plant der Autor den zukünftigen Einsatz der VM im Embedded-Bereich, in dem wiederum ganz andere Prozessoren und Betriebssysteme zum Einsatz kommen. Gerade in diesem Bereich variieren zum Beispiel die Größen der in C verwendeten Datentypen von Compiler zu Compiler stark, was von Anfang an zu beachten war, da spätere Probleme dieser Art nur schwer aufzufinden und zu lösen sind.

1.3 Organisation der Arbeit

Diese Arbeit ist in zwei logische Teile unterteilt. Kapitel 2 und 3 behandeln die nötigen theoretischen Grundlagen, auf die die folgenden Kapitel aufbauen. Dies beginnt mit einer Übersicht über virtuelle Maschinen im Allgemeinen, geht auf Details der Java Virtual Machine ein und diskutiert verschiedene Implementierungsmöglichkeiten und dazu passende Algorithmen.

In den Kapiteln 4 bis 7 geht es dann an die Praxis. Kapitel 4 erläutert die Entscheidung für Programmiersprache und Tools, Kapitel 5 gibt eine Übersicht über die JVM-Implementierung, Kapitel 6 erläutert die konkrete Implementierung der Laufzeitstrukturen und Kapitel 7 zeigt einige Anwendungsbeispiele.

Abgeschlossen wird die Arbeit in Kapitel 8 mit einer Zusammenfassung, einem Ausblick auf die mögliche, zukünftige Verwendung und Weiterentwicklung, sowie dem Fazit des Autors. Im Anhang folgen einige Hinweise zur Nutzung der Implementierung.

Kapitel 2

Einführung in virtuelle Maschinen

Der Ausdruck „virtuelle Maschine“ ist ein recht unpräziser Oberbegriff, der in diesem Kapitel erläutert wird. Genauer unterscheidet man zwischen zwei Gruppen von virtuellen Maschinen, die kurz vorgestellt werden. Konkret wird dann auf die Java Virtual Machine[1] (JVM) eingegangen und verschiedene Implementierungsarten vorgestellt. Zum Schluss wird das Java Class File Format, das System der symbolischen Referenzen, und der Ladeprozess einer Klasse erläutert. Das alles sind essenzielle Grundlagen für eine JVM-Implementierung.

2.1 Übersicht virtueller Maschinen

Eine virtuelle Maschine stellt die Simulation einer Maschine in Software dar. Der Begriff „Maschine“ ist sehr allgemein gefasst, im Kontext dieser Arbeit wird daher von der Simulation von Computersystemen ausgegangen. Da die simulierte Maschine nicht echt ist, spricht man von einer „virtuellen“ Maschine.

[1] Der Begriff „Java Virtual Maschine“ stellt einen Eigennamen dar, weshalb er nicht ins Deutsche übersetzt wird.

Aus der Sicht eines Betriebssystems stellt ein Computersystem durch seine Hardware eine Sammlung von Diensten zur Verfügung, die das Betriebssystem nutzt. So bietet ein Computersystem beispielsweise Zugang zum Prozessor, dem RAM Speicher und zu I/O Komponenten (Steckkarten, Festplatten, etc) an. Das Betriebssystem verwaltet diese Komponenten und bietet Prozessen, die auf dem Betriebssystem laufen, mittels einer verallgemeinerten Programmierschnittstelle Zugang zu diesen Komponenten und delegiert diese bei Bedarf auch an einzelne Prozesse, falls mehrere auf die selbe Komponente zuzugreifen versuchen.

Ein Betriebssystem ist üblicherweise auf eine bestimmte Hardware abgestimmt, so dass es nicht ohne weiteres auf einem System mit anderer Hardware laufen kann. Steht passende Hardware nicht zur Verfügung, dann kommen so genannte „System Virtual Machines“ (SVMs) zum Einsatz. Sie übersetzen die Anfragen des Betriebssystems an die Hardware von Typ A in Anfragen für die Hardware von Typ B, führen diese aus und übersetzen die Antwort entsprechend zurück. Obwohl das Betriebssystem also auf einem ihm unbekannten System ausgeführt wird, funktioniert es dennoch problemlos, da die SVM die Übersetzungsarbeit leistet. Für das Betriebssystem sieht das fremde System inklusive der virtuellen Maschine wie ein bekanntes System aus.

Abbildung 2.1 zeigt ein solches System. Auf der linken Seite unten sieht man die Hardware, auf der das System läuft. Man spricht hier auch von dem Host-System. Darauf läuft die SVM, hier verallgemeinert „Virtualisierung“ genannt, worauf das Betriebssystem (hierbei spricht man dann von dem Gast-System) mit seinen Anwendungen ausgeführt wird. Auf der rechten Seite sieht man die Ansicht, wie das Gast-Betriebssystem und deren Anwendungen das System sehen, wobei für sie allerdings nicht wirklich ersichtlich ist, dass es sich hier um eine virtuelle Maschine handelt.

Bekannte Beispiele für SVMs sind FX!32, welches für x86 Prozessoren kompilierte Windows-Software auf einem System mit Alpha Prozessor ausführt, oder die Software-Schicht von Transmetas[2] Crusoe- und Efficeon Prozessor-Familien, welche es ermöglicht, für x86 Prozessoren kompilierten Code auf

[2]Siehe `http://www.transmeta.com/`

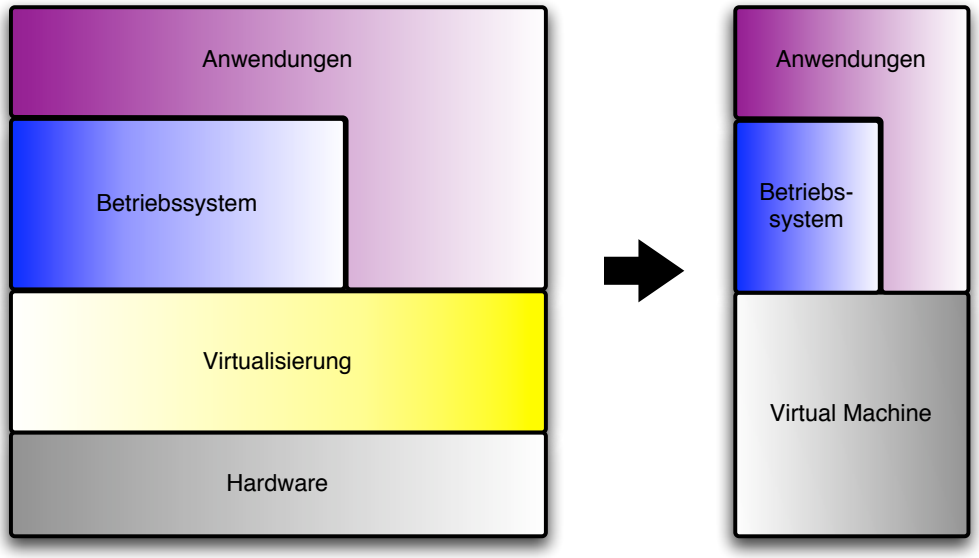


Abbildung 2.1: Eine „System Virtual Machine“ (Frei nach [Smith05])

den extrem stromsparenden Prozessoren des Herstellers mit anderer Architektur auszuführen.

Eine Unterkategorie der System Virtual Machines stellen die „Whole System Virtual Machines“ (WSVs) dar. In diesem Fall stellt nicht nur die Hardware das Host-System dar, sondern auch das entsprechende Betriebssystem. Die VM läuft als normaler Prozess auf dem Host-Betriebssystem, stellt aber in gewohnter Weise die Zwischenschicht für das Gast-Betriebssystem dar. Nur werden in diesem Fall die Anfragen des Gast-Systems nicht direkt auf ein anderes System übersetzt, sonders in Funktionsaufrufe des Host-Betriebssystems.

Diese WSVs erfreuen sich heutzutage sehr großer Beliebtheit, weil man auf einem System problemlos weitere Betriebssysteme, zum Beispiel zu Testzwecken, in VMs installieren kann, ohne das reale System zu beeinträchtigen. Bekannte WSVs sind zum Beispiel VMware[3], Virtual PC[4] oder Parallels[5], welche aktuelle PC Hardware abbilden. Immer beliebter werden aber auch

[3]Siehe `http://www.vmware.com`
[4]Siehe `http://www.microsoft.com/virtualpc`
[5]Siehe `http://www.parallels.com`

VMs, die alte Konsolen, wie zum Beispiel das SNES[6] oder den GameBoy[7], simulieren.

Abbildung 2.2 zeigt ein Beispiel für eine Anwendung einer solchen WSV. Auf dem Host-System wird Mac OS X betrieben, auf dem wie üblich Mac OS eigene Anwendungen ausgeführt werden. (Rechte Seite.) Eine dieser Anwendungen ist jedoch eine WSV, die in diesem Fall Windows XP mit seinen Anwendungen ausführt.

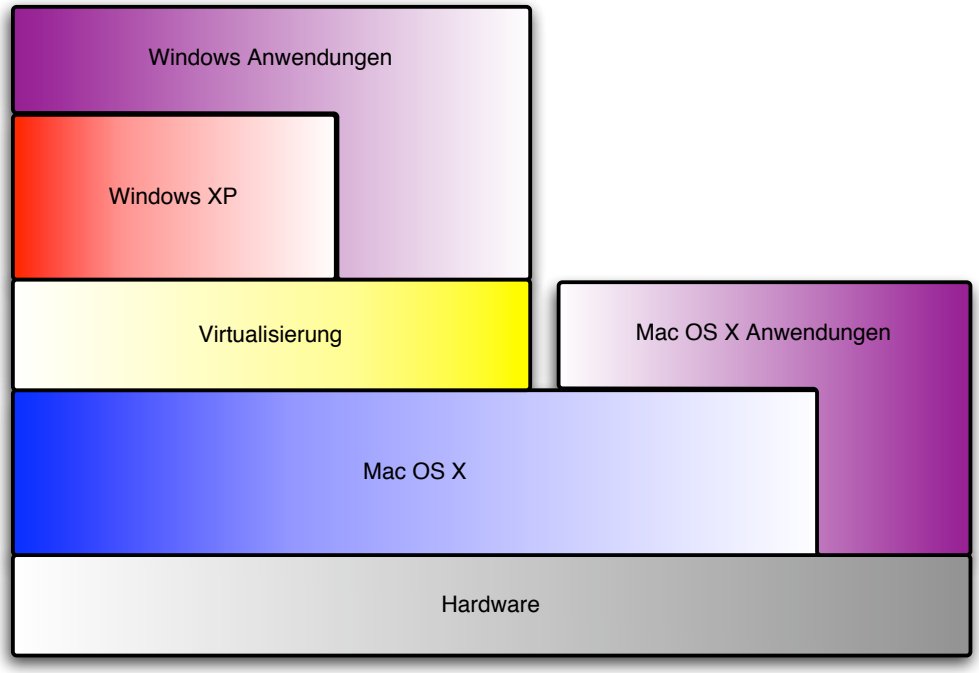


Abbildung 2.2: Beispiel einer „Whole System Virtual Machine“. Mac OS X bildet das Host-System, auf dem eine virtuelle Maschine läuft, und Windows XP bildet das Gastsystem, welches in der virtuellen Maschine läuft.

Neben den System Virtual Machines gibt es dann noch die Gruppe der „Process Virtual Machines“ (PVMs). Bei diesen läuft kein vollständiges Betriebssystem in der virtuellen Maschine, sondern lediglich ein Prozess. Dieser

[6]ZSNES (`http://www.zsnes.com/`) ist der bekannteste SNES Simulator, der auf vielen Systemen läuft.

[7]Der GameBoy Simulator JavaBoy (`http://www.millstone.demon.co.uk/download/javaboy/index.htm`) ist in Java geschrieben und läuft somit sogar auf allen Systemen, für die eine Java Virtual Machine existiert. Sogar auf manchen Mobiltelefonen läuft es.

Prozess wird von der VM ausgeführt und hat nichts mit den Prozessen des Host-Betriebssystems zu tun. Dies hat vor allen Dingen Vorteile im Sinne der Plattformunabhängigkeit. Anwendungen, die für eine virtuelle Maschine kompiliert sind, können potenziell auf jedem beliebigen System ausgeführt werden. Vorausgesetzt es existiert eine passende virtuelle Maschine für dieses System.

Abbildung 2.3 zeigt eine solche PVM in der schematischen Darstellung. Auf der linken Seite sieht man, wie die virtuelle Maschine, hier erneut verallgemeinert „Virtualisierung“ genannt, auf dem Host-System, und mit Unterstützung des Host-Betriebssystems, ausgeführt wird. In dieser läuft dann der Prozess, in dem eine Anwendung ausgeführt wird, welche für die virtuelle Maschine kompiliert wurde. Auf der rechten Seite sieht man die Ansicht aus der Sicht des Prozesses, für welchen das System einzig und allein aus der virtuellen Maschine besteht. Die darunter liegenden Schichten sind dem Prozess unbekannt und sollten ihn im Sinne der Portabilität der Anwendung auch nicht interessieren.

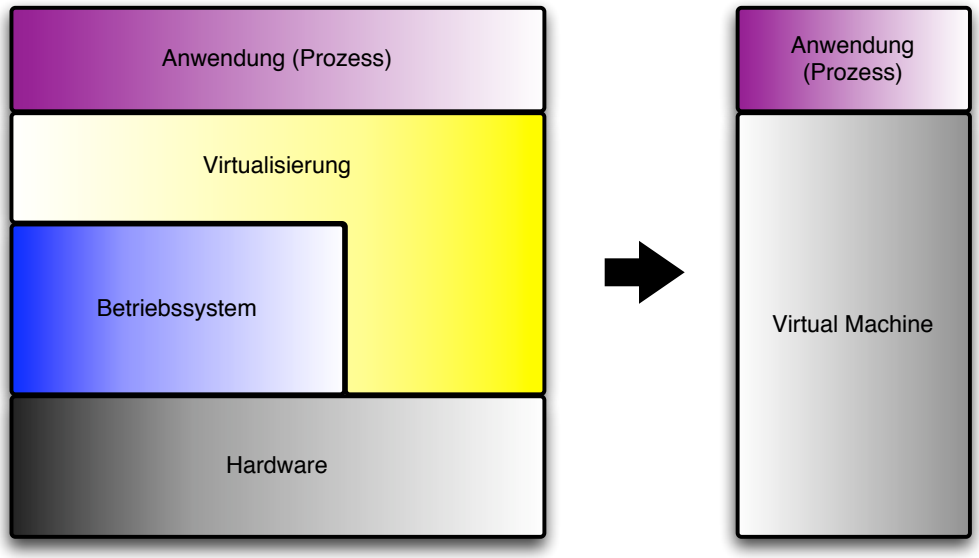


Abbildung 2.3: Eine „Process Virtual Machine“ (Frei nach [Smith05])

Process Virtual Machines werden heute bevorzugt als Ziel-Plattform für Hochsprachen benutzt. Java ist hier das bekannteste Beispiel. Der Compiler übersetzt den Java Quellcode in eine Zwischensprache (Bytecode genannt), welcher von der Java Virtual Machine ausgeführt wird. Weitere bekannte virtuelle Maschinen, die als Ziel-Plattform verwendet werden, sind zum Beispiel Microsofts .NET, die P-Code Virtual Machine oder die virtuelle Maschine von Smalltalk.

Aus Platzgründen konzentriert sich diese Arbeit im folgenden auf die Java Virtual Machine. Wer mehr zu den Grundlagen, oder zu anderen virtuellen Maschinen erfahren will, dem sei das äußerst umfangreiche [Smith05] empfohlen. Es behandelt alle Typen von virtuellen Maschinen und erläutert beispielhaft die Strukturen und den Aufbau verschiedener existierender System- und auch Project Virtual Machines.

2.2 Die Java Virtual Machine

Die Java Virtual Machine[8] ist eine so genannte „Stack Based“ virtual Machine. Das bedeutet, dass einfache Instruktionen, die sie ausführt, einzig und allein auf Daten auf dem Stack zugreifen können. Reelle Prozessoren arbeiten üblicherweise mit Registern, schnellen Speicherplätzen, die sich direkt im Prozessor befinden. Um bei der Spezifikation der JVM den kleinsten gemeinsamen Nenner von Prozessoren zu treffen, wurde deshalb auf die Nutzung von Registern, bis auf eine Ausnahme, verzichtet.

Bei der Nutzung von Registern in der Spezifikation von virtuellen Maschinen könnte es bei Portierungen zu Problemen kommen, und zwar dann, wenn der Prozessor des Systems, für das die virtuelle Maschine portiert wird, über weniger Register verfügt, als man für die Implementierung der virtuellen Maschine benötigen würde. Es käme also immer zu Problemen, wenn man eine VM auf einem System mit einem Prozessor laufen lassen müsste, der

[8]Da diese Arbeit im folgenden nur noch über JVMs spricht, werden die Begriffe „Java Virtual Machine“ und „Virtual Machine“ (oder auch Deutsch „virtuelle Maschine“) und deren Abkürzungen JVM und VM als gleichbedeutend für „Java Virtual Machine“ stehen, es sei denn, es ist aus dem Kontext heraus eindeutig anders zu verstehen.

selber weniger Register zur Verfügung stellt, als die VM benötigen würde. In solch einem Fall bleibt dann nur die langsame Simulation der VM Register im Speicher des Systems, was zu deutlichen Performancenachteilen führen kann.

Im Falle einer rein stackbasierten VM stellt sich das Gegenteil ein. Zwar ist die ursprüngliche Implementierung nicht schneller als eine Softwarelösung von Registern, jedoch lässt sich je nach genutztem Prozessor hier optimieren, so können zum Beispiel Teile des Stacks auf Prozessorregister abgebildet werden.

Neben dem Stack besteht eine JVM aus folgenden weiteren Komponenten:

Der Classloader ist zuständig für das Auffinden und Laden von Klassen. Diese Komponente ist direkt beim Start der JVM essentiell, da ohne eine initiale Instanz eines Classloaders keine weiteren Klassen geladen werden können.

Die Method Area[9] stellt einen globalen Speicherbereich dar, in dem die geladenen Klassen mit all ihren Laufzeitdaten und symbolischen sowie statischen Informationen gespeichert werden.

Im Heap werden die Instanzen von Klassen zur Laufzeit abgelegt.

Die Ausführungseinheit, oft auch einfach nur Interpreter genannt, ist das Herzstück, oder auch der theoretische „Prozessor“ der JVM. Hier wird der Bytecode ausgeführt.

Zu guter Letzt gibt es dann noch den so genannten Program Counter (PC), welcher die oben genannte Ausnahme für ein Register darstellt. Der Program Counter speichert die aktuelle Position, an dem die Anwendung, bzw. der Thread einer Anwendung, gerade ausgeführt wird. Er muss mindestens einmal pro Bytecode verändert werden, weshalb die Realisierung als Register angeraten wird. Natürlich lässt sich aber auch hier eine Lösung ohne Register implementieren, nur muss man dann entsprechende Performance-Verluste in Kauf nehmen.

[9]Der Name ist irreführend, da in diesem Bereich eben nicht nur die Methoden gespeichert werden. Jedoch wird dieser Begriff wohl traditionell in der Literatur für den beschriebenen Speicherbereich verwendet, weshalb der Autor dies hier auch beibehält.

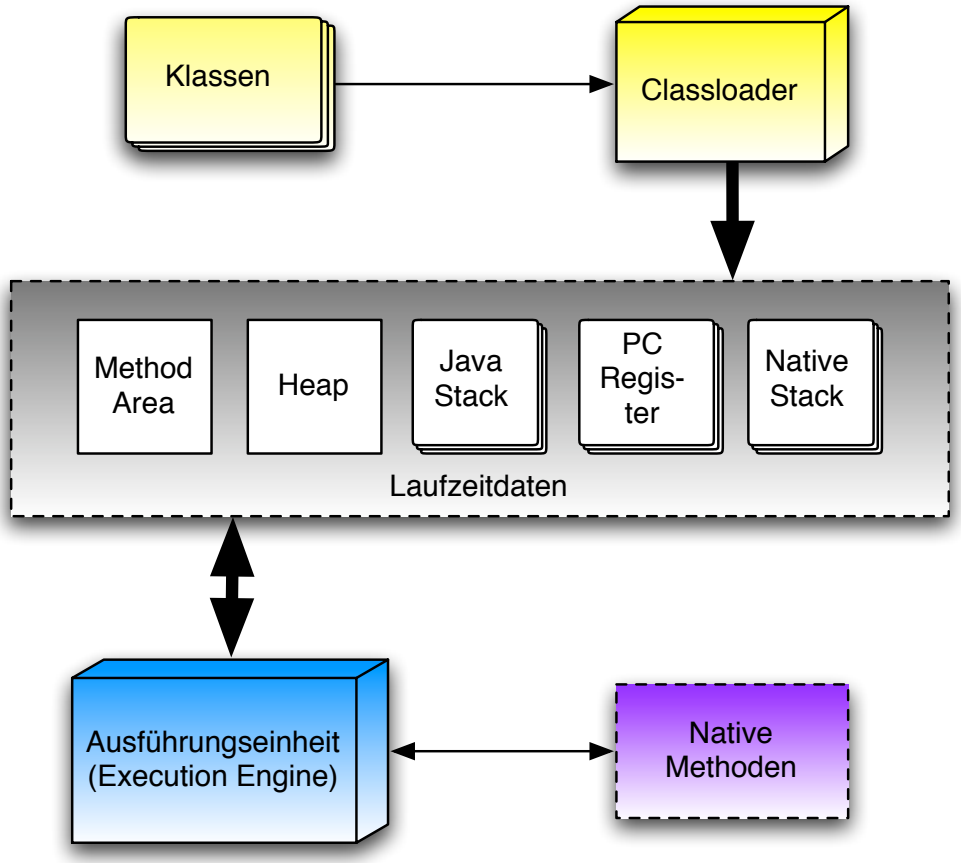


Abbildung 2.4: Die Architektur der Java Virtual Machine (Frei nach [Venners99])

Abbildung 2.4 zeigt die Zusammenhänge der einzelnen Komponenten der Java Virtual Machine.

Innerhalb einer JVM unterscheidet man zwei verschiedene Stacks. Der Native Stack ist der Stack, den die Programmiersprache (in der Regel C oder C++) verwendet, in dem die JVM implementiert ist. Der Java Stack wird von der Ausführungseinheit der JVM verwendet, um äquivalent zu einem native Stack auf ihm die Bytecodes einer Java Anwendung[10] auszuführen.

Da die JVM von Grund auf für Nebenläufigkeit entworfen worden ist, ist es kein Problem parallel mehrere Threads auszuführen, sofern die JVM Implementierung dies vorsieht. In Abbildung 2.4 kann man das mehrfache Vorhandensein von Native Stack, Java Stack und des Program Counter Registers erkennen, welche für jeden Thread separat angelegt werden. Potenziell können diese Thread vollständig unabhängig voneinander laufen, lediglich der Zugriff auf die Method Area und den Heap muss synchronisiert werden.

Im folgenden werden aus Platzgründen nur die Speicherstrukturen der JVM weiter behandelt. (Siehe Kapitel 3) Eine detaillierte, vollständige Dokumentation der JVM erhält man entweder in der Java Virtual Machine Specification (als Buch [Lindholm99] oder online `http://java.sun.com/docs/books/jvms/` verfügbar), oder in Bill Venners etwas ausführlicherem und weniger technisch geschriebenen Buch [Venners99][11].

[10]Die Java Virtual Machine ist von Natur aus nicht darauf beschränkt Java Code auszuführen. Im Prinzip ist sie „Turing Vollständig“, kann also jedes beliebige Programm ausführen. Der Java Compiler übersetzt den Java Code in von der JVM Spezifikation ([Lindholm99]) definierte Klassen. Beliebige andere Compiler können dies auch für andere Sprachen tun. Tatsächlich gibt es hier einige. Eine gut gepflegte Liste findet man beispielsweise hier: `http://www.robert-tolksdorf.de/vmlanguages.html`

[11]Leider ist das Buch schon seit einiger Zeit nicht mehr im Buchhandel erhältlich. Bill Venners war so freundlich mir auf Anfrage eine seiner eigenen „Backup“ Kopien zukommen zu lassen. Einige Kapitel hat er im Laufe der Zeit auf seiner Webseite online gestellt: `http://www.artima.com/insidejvm/ed2/`

2.3 Implementierungsarten der Java Virtual Machine

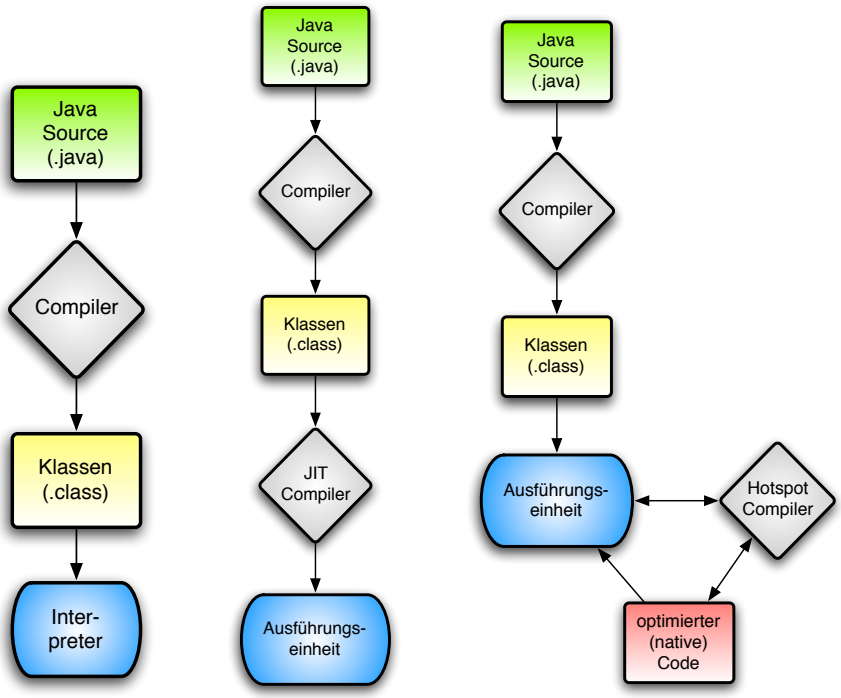
Die Java Virtual Machine Specification ([Lindholm99]) betont ausdrücklich, dass es unterschiedliche Möglichkeiten gibt, eine Java Virtual Machine zu implementieren[12]. Abbildung 2.5 zeigt die heute üblichen Varianten.

Abbildung 2.5(a) zeigt einen einfachen Interpreter. Der Java Quellcode wird mit dem Java Compiler (javac) in Klassen, das Binärformat der JVM, kompiliert. Die JVM lädt diese Klassen und führt sie aus. Dies passiert in einer Interpreter-Schleife Bytecode für Bytecode. Ausgiebige Optimierungen finden nicht statt, lediglich kleinere Tricks zur Beschleunigung der Auflösung von symbolischen Informationen und der Interpreter-Schleife werden angewandt.

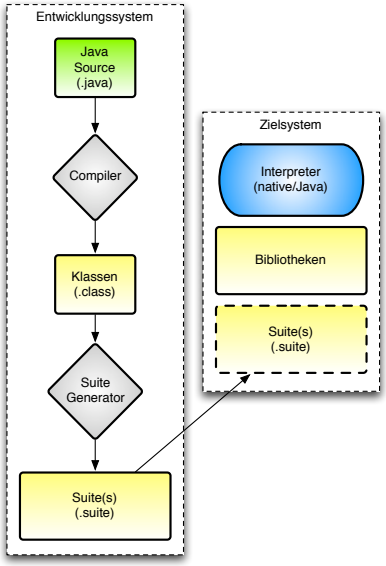
Abbildung 2.5(b) sieht Abbildung 2.5(a) sehr ähnlich, arbeitet jedoch nach einem vollkommen unterschiedlichen Konzept. Weiterhin wird der Java Quellcode mit dem Java Compiler in Klassen kompiliert. Dieses Format wird allerdings praktisch nur für den Transport eingesetzt. Werden die Klassen nun von einer JIT-JVM geladen, so übersetzt sie ein Just In Time Compiler (JIT) in das Binärformat des Zielsystems und führt es dort ohne Interpreter nativ aus. Eine solche JIT Implementierung ist deutlich schneller als ein klassischer Interpreter, jedoch gibt es auch Nachteile. So muss der JIT Compiler beim Starten der Anwendung möglichst schnell laufen, um die Startzeit nicht unnötig zu erhöhen, weshalb kann er sich nicht die Zeit für ausgiebige Optimierungen nehmen kann. Auch kann er beim Kompilieren nicht auf Laufzeitinformationen zurückgreifen, da ihm diese beim Start ebenfalls noch nicht zur Verfügung stehen.

Ein moderner Ansatz, der die Nachteile von Interpreter und JIT Compiler ausgleicht, ist Suns aktuelle Hybrid-JVM, welche nach ihrem Laufzeit-

[12]Dies ging letzten Endes sogar so weit, dass aus der zweiten Auflage ein vorher existierendes Kapitel wieder entfernt wurde, weil es ein potenzielles Implementierungsdetail erläuterte. Viele Entwickler verstanden jedoch scheinbar den Sinn dieses Kapitels nicht und nahmen die erläuterte Lösung als Voraussetzung an. Letzten Endes setze aber sogar Suns eigene Implementierung die in diesem Kapitel erwähnte Technik mittlerweile gar nicht mehr ein, so dass die Autoren es für besser befanden, das Kapitel zu entfernen.



(a) Einfacher Interpreter (b) JIT Compiler (c) Mixed-Mode JVM



(d) Split-VM Architektur

Abbildung 2.5: Die verschiedenen Implementierungsarten von JVMs.

optimierungssystem, dem Hotspot Compiler, auch kurz einfach nur Hotspot VM oder Mixed-Mode VM genannt wird. Diese Hybridlösung besteht, wie Abbildung 2.5(c) zeigt, daraus, dass die JVM erst einmal die Klassen unverändert lädt und in einem Interpreter-Modus (wie Abbildung 2.5(a) zeigt) interpretiert. Zusätzlich werden von einem mitlaufenden Profiler allerdings Laufzeitinformationen über den ausgeführten Code gesammelt. Diese Informationen werden an den parallel mitlaufenden Hotspot Compiler weitergereicht, welcher die „heißen Bereiche“ des Programms (daher auch der Name) dann ausgiebig optimiert.

Die Arbeit des Hotspot Compilers ist in den aktuellen Versionen von Suns JVM so umfangreich, dass man allein über diese eine Diplomarbeit schreiben könnte. Denn mit der einfachen Optimierung von Code ist es nicht getan. Sollte durch falsche Annahmen die Ausführung des optimierten Codes schneller statt langsamer werden, so kann das System beispielsweise ein so genanntes Rollback durchführen. Das bedeutet, dass anstatt der optimierten und im nativen Code des Systems vorliegenden Version des Programmteils nun wieder die unoptimierte Version erneut interpretativ ausgeführt wird. Die optimierte Version wird verworfen und eine angepasste neue erzeugt. Nach diesem System kann der Hotspot Compiler auch gewagtere Optimierungen durchführen, ohne dass das System dadurch Probleme bekommt.

Auch für andere Bereiche gibt es Auffanglösungen. Hat der Hotspot Compiler beispielsweise festgestellt, dass in einer Schleife trotz eines Exception-Handlers keine Exceptions geworfen werden, dann kann er diesen aus dem optimierten Code entfernen und stattdessen außen herum platzieren. Sollte dann doch einmal eine Exception ausgelöst werden, so greift das System auf die unoptimierte Bytecode-Variante des Codes zurück um die Exception zu verarbeiten, behält aber für zukünftige Nutzung die optimierte Version weiterhin bei. Es sei denn, das Ausführungsverhalten ändert sich längerfristig und es werden nun öfter Exceptions geworfen.

Abbildung 2.5(c) zeigt die schematische Darstellung einer Mixed-Mode JVM. Diese stellt im Desktop- und Server-Bereich momentan die beste und auch schnellste Lösung zur Verfügung. Negativ anzumerken ist jedoch der gesteigerte Speicherverbrauch des Systems. Immerhin müssen von allen op-

timierten Codeteilen beide, die optimierte und die unoptimierte Version, im Speicher vorgehalten werden.

Auf Systemen, bei denen die Mixed-Mode JVMs aus Speicherplatzgründen nicht in Frage kommen, zielt das neueste JVM Forschungsprojekt von Sun, die Squawk JVM. Systeme, die in diese Kategorie fallen, sind besonders Embedded Systeme, welche von Natur aus mit wenig Speicher auskommen müssen. Die schematische Darstellung eines solchen Systems zeigt Abbildung 2.5(d).

Der Ansatz sieht so aus, dass die kompilierten Klassen noch auf dem Entwicklungssystem von einem Suite Generator geladen, verifiziert und als Suites abgelegt werden. Suites entsprechen einem Speicherabbild von mehreren geladenen Klassen des Zielsystems. So können diese anschließend auf das Zielsystem übertragen und dort „in place“, also ohne geladen oder überprüft werden zu müssen, an Ort und Stelle ausgeführt werden. Durch die Vermeidung des Overheads von `.class`-Dateien, und durch die Verwendung eines effizienteren Bytecode Formats, verringert sich der Platzverbrauch von Suites auf 1/3 dessen, was die entsprechenden Klassen verbrauchen.

Auf dem Zielsystem wird der Code von einem klassischen Interpreter ausgeführt, welcher aus Performancegründen in C geschrieben ist und mit einem hochperformanten C Compiler kompiliert wurde. Das restliche Laufzeitsystem, bestehend aus Garbage Collector, Thread-Scheduler und Bibliotheken, ist vollständig in Java geschrieben, was den Entwicklungsaufwand im Vergleich zu einer größtenteils in C geschriebenen JVM deutlich verringert. Auch der Interpreter soll in zukünftigen Versionen in Java geschrieben und getestet werden, soll dann aber automatisiert von Java nach C übersetzt und wiederum mit einem hochoptimierenden C Compiler kompiliert werden um ein Maximum an Performance herauszuholen.

Die Homepage des Squawk-Projekts befindet sich unter [Squawk], die Implementierung wird in [Shaylor03] und [Simon06] beschrieben.

Je nach Art der Anwendung einer JVM kann also ein unterschiedliches System zur Ausführung von Java Anwendungen verwendet werden. Auch der Aufwand einer Implementierung, dessen Fehlerfreiheit, Sicherheit, und viele andere Merkmale, können also die Entscheidung für eine spezielle Implemen-

tierung beeinflussen. Für diese Arbeit hat sich der Autor für eine klassische Interpreter-Lösung entschieden. Seine Beweggründe für diese Entscheidung werden in Kapitel 5 genauer erläutert.

2.4 Das Java Class File Format

Eine `.class` Datei repräsentiert für die JVM eine Klasse mit all ihren Detailinformationen und Referenzen. Dies beginnt bei Flags und Ableitungsinformationen und endet bei den Bytecodes, die die Funktion einer Methode beschreiben. Das class Format ist als Datenstrom aus 8-Bit Wörtern (Bytes) zu interpretieren. Die Größenangaben von Elementen werden im Folgenden mit einem u oder einem s gepräfixt, je nachdem, ob sie ohne Vorzeichen (unsigned, u) oder mit Vorzeichen (signed, s) zu interpretieren sind. Danach folgt die Größe des Elements. Ein u2 beschreibt beispielsweise ein vorzeichenloses Element der Länge von 2 Bytes (16 Bit). Besteht ein Element aus mehreren Bytes, dann sind diese immer in Big-Endian Reihenfolge gespeichert, das höherwertige Byte steht im Datenstrom also vorn.

Abbildung 2.6 zeigt die Basis-Struktur des class Formats. Im folgenden werden die einzelnen Elemente erläutert.

magic

Die „Magic Number“ identifiziert das class Format. Es muss den Wert `0xCAFEBABE`[13] beinhalten.

minor_version, major_version

Diese Elemente beschreiben die Versionsnummer der class Datei. major_version beschreibt die Hauptversion, minor_version potenzielle Unterversionen der Hauptversion.

constant_pool_count

Anzahl der Einträge des Constant Pools plus eins. Die Differenz von

[13]Interessantes Detail am Rande: Das im hexadezimalen wunderschön als Wort zu erkennende CAFEBABE wird nicht nur bei den Java Class Files als Magic Number verwendet, sondern in der selben Art auch bei den ausführbaren Dateien des NeXT Betriebssystems. Sicherheitshalber sollte man also die Korrektheit weiterer Elemente überprüfen, bevor man sich sicher sein kann, wirklich eine class Datei vor sich zu haben.

```
struct ClassFile
{
   u4 magic;
   u2 minor_version;
   u2 major_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];
}
```

Abbildung 2.6: Die Struktur eines Java Class Files

eins entsteht dadurch, dass im ersten Eintrag (Index 0) des Constant Pools zur Laufzeit kein Eintrag gespeichert wird. Die folgende Liste wird dadurch jedoch nicht beeinflusst.

constant_pool[ ]
: Liste der Constant Pool Einträge, bestehend aus der `cp_info` Struktur.

access_flags
: Dieser Eintrag beinhaltet Flags für Zugriffsrechte und Eigenschaften.

this_class
: Der Wert dieses Eintrags ist ein Index in den Constant Pool der Klasse. Er weist dort auf eine `CONSTANT_Class_info` Struktur, welche die aktuelle Klasse oder das aktuelle Interface beschreibt.

super_class
: Der Wert dieses Eintrags verweist ebenfalls auf eine `CONSTANT_Class_-`

`info` Struktur, diese beschreibt aber die direkte Superklasse der aktuellen Klasse. Ist der Wert dieses Eintrags 0, dann muss die aktuelle Klasse die Klasse `java.lang.Object` beschreiben.

interfaces_count

Anzahl der folgenden Einträge für implementierte Interfaces dieser Klasse.

interfaces[ ]

Liste mit Indizes in den Constant Pool. Die Einträge müssen alle vom Typ `CONSTANT_Class_info` Struktur sein und beschreiben die Interfaces in der Definitionsreihenfolge (von links nach rechts im Quellcode).

fields_count

Anzahl der folgenden Einträge für Felder.

fields[ ]

Liste von `field_info` Strukturen, die die Felder in dieser Klasse oder in diesem Interface beschreibt. Dies schließt keine Felder mit ein, die von den Superklassen definiert werden, definiert aber ansonsten beide, Felder für Instanz-Variablen und Felder für Klassen-Variablen.

methods_count

Anzahl der folgenden Einträge für implementierte Methoden dieser Klasse oder dieses Interfaces.

methods[ ]

Jeder Eintrag dieser Liste beinhaltet eine `method_info` Struktur, welches jeweils eine Methode dieser Klasse oder dieses Interfaces beschreibt. Dies schließt nicht die Methoden von Superklassen mit ein. Die Methoden können von jedem Typ sein. Alle Methoden bis auf diese, die native oder abstract deklariert sind, bringen auch die Instruktionen mit, die ihre Funktionalität implementieren.

attributes_count

Anzahl der folgenden Attribute.

attributes[ ]

Jeder Eintrag dieser Liste enthält eine `attribute_info` Struktur, welches ein beliebiges Attribut beschreibt. Die JVM Specification definiert in Kapitel 4 verschiedene Typen, jedoch liegt es einem frei weitere zu definieren. JVMs müssen so implementiert sein, dass unbekannte Attribute ignoriert werden.

Im folgenden soll nun ein Einblick in einige ausgewählte Elemente des Formats gegeben werden. Aufgrund des Umfangs würde eine erschöpfende Erläuterung aller Details diese Arbeit sprengen, weshalb für weitere Informationen auf das Kapitel 4 der Java Virtual Machine Specification (JLS, [Lindholm99]) verwiesen wird.

Der Constant Pool[14] ist der Dreh- & Angelpunkt einer Klasse. In ihm werden die verschiedensten Informationen als konstante Symbole und Strukturen gespeichert, auf welche an anderer Stelle mit einem Index in den Constant Pool verwiesen wird, dies spart Wiederholungen und damit Speicherplatz. Die Anzahl der Strukturen ist zahlreich, weshalb diese hier nicht vollständig behandelt werden können. (Eine detaillierte Behandlung findet man jedoch im Kapitel 4.4 der JLS ([Lindholm99]).) Grundsätzlich sei jedoch angemerkt, dass es sich bei den Informationen im Constant Pool immer um konstante Werte handelt, die sich rein von der Spezifikation her nicht ändern. Später werden einige wenige Einträge im Zusammenhang mit Laufzeitoptimierungen behandelt, indem man die symbolischen Informationen durch direkte Zeiger ersetzt bzw. erweitert.

Die `field_info` Struktur (Abbildung 2.7) beschreibt die einzelnen Felder einer Klasse und besteht aus folgenden Elementen:

access_flags

Die Flags beschreiben auch hier wieder Zugriffsrechte und Eigenschaften.

name_index

Dieser Eintrag beinhaltet einen Index in den Constant Pool, welcher

[14]Die hier Englisch geschriebenen Begriffe werden im Sprachgebrauch als Eigenwörter verwendet, und deshalb vom Autor hier nicht übersetzt.

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

Abbildung 2.7: Die `field_info` Struktur

auf einen Eintrag vom Typ `CONSTANT_Utf8_info` verweist. Dieser beschreibt den Namen des Feldes in einer UTF8 kodierten Zeichenkette (String).

descriptor_index

Auch dieser Eintrag verweist auf einen `CONSTANT_Utf8_info` Eintrag im Constant Pool, jedoch beschreibt dieser einen Field Descriptor[15].

attribute_count, attribute_info

Ein Feld kann beliebig viele Attribute beinhalten. Auf jeden Fall von der JVM verarbeitet werden muss nur das `ConstantValue` Attribut, welches den Wert der Konstante beschreibt, wenn dieses Feld eine Konstante darstellt[16].

Methoden werden in einem `method_info` Eintrag beschrieben. (Abbildung 2.8.) Die ersten vier Einträge entsprechen denen der `field_info` Struktur. Die Struktur des `Code_attribute`s ist in Abbildung 2.9 gezeigt. Sie beinhaltet nicht nur den Bytecode (code array), sondern auch eine Fülle von weiteren Informationen, die zum Ausführen einer Methode benötigt werden. Dazu gehören die maximale Anzahl der Elemente auf dem Stack während

[15] Für die Erläuterung von Field Descriptoren siehe Kapitel 4.3 der JLS ([Lindholm99])

[16] Dieses Feature wird nur selten benutzt. Stattdessen fügt der Java Compiler Konstanten lieber direkt in den Bytecode Datenstrom ein. Dies ist auch der Grund, warum man ein ganzes Projekt neu kompilieren muss, wenn man eine öffentliche Konstante in einer Klasse ändert. Würden die anderen Klassen tatsächlich auf den Wert in dieser Klasse verweisen dann wäre dies nicht nötig. Vermutlich haben Performance-Überlegungen die Entwickler des Compilers dazu bewegt.

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

Abbildung 2.8: Die `method_info` Struktur

```
struct Code_attribute
{
   u2 attribute_name_index;
   u4 attribute_length;
   u2 max_stack;
   u2 max_locals;
   u4 code_length;
   u1 code[code_length];
   u2 exception_table_length;
   exception_table exception_table[exception_table_length];
   u2 attributes_count;
   attribute_info attributes[attributes_count];
}
```

Abbildung 2.9: Die `Code_attribute` Struktur

der Ausführung dieser Methode, die maximale Anzahl der lokalen Variablen und eine Tabelle mit Einträgen von Exception Handlern. Auch weitere Attribute dürfen hier erneut nicht fehlen. Sie beinhalten zum Beispiel Debugging-Informationen wie eine Zeilennummertabelle, eine Tabelle mit den Namen der lokalen Variablen[17] und verschiedene andere Dinge.

Wie die Daten der eingelesenen Klassen zur Laufzeit repräsentiert werden, behandelt Kapitel 6.

2.5 Symbolische Referenzen

Manche Informationen im Constant Pool werden als symbolische Referenzen gespeichert. Symbolisch bedeutet an dieser Stelle, dass dort nicht die eigentlich gewünschte Information zu finden ist, sondern ein Verweis an eine andere Stelle, der die gewünschte Information zur Laufzeit entnommen werden kann. Solche Informationen können zum Zeitpunkt der Erstellung der Klasse in der Regel nicht zweifelsfrei festgelegt werden, weil sie sich im Nachhinein noch ändern könnten, oder aber auch zum Erstellungszeitpunkt einfach noch nicht vorliegen.

Angewandt wird dies beispielsweise beim Verweis auf die Superklasse einer Klasse. Schaut man sich die Klasse `java.lang.String` an, so findet man im `super_index` Eintrag der Klasse einen Verweis auf einen Index im Constant Pool, in welchem eine `CONSTANT_Class_info` Struktur gespeichert ist, die eine symbolische Referenz auf die Superklasse enthält. In diesem Fall besteht die symbolische Information aus dem voll qualifizierten Klassennamen[18] „java/lang/Object“. Weitere Informationen muss die JVM also der

[17] Lokale Variablen werden von der JVM nicht mit Namen, sondern lediglich über eine Nummer angesprochen. Hierfür sieht sie auf dem Stack einen Bereich mit so genannten Slots vor, in dem die lokalen Variablen abgelegt werden. Die Namen werden nur für Debuggingzwecke verwendet und können je nach Einstellung des Compilers auch nicht eingefügt werden um das Dekompilieren zu erschweren, oder auch einfach nur um Platz zu sparen.

[18] Der voll qualifizierende Klassenname (Englisch: fully qualified class name) besteht aus dem Paket, in dem sich eine Klasse befindet, und dem Namen der Klasse. Die Trennung der einzelnen Bezeichner wird jedoch nicht wie in Java mit dem Punkt „.“, sondern mit dem Forward Slash „/“ vorgenommen. Die Klasse java.lang.Object lautet also dann java/lang/Object. Die JVM verwendet diese Struktur, da sie so die Trennzeichen nicht ersetzen muss, wenn sie in einem UNIX-Dateisystem im Classpath eine Klasse sucht.

Klasse Object entnehmen, die sie in der Method Area schon geladen vorfinden sollte[19]. Abbildung 2.10 zeigt das erwähnte Beispiel einer symbolischen Referenz zwischen zwei Klassen.

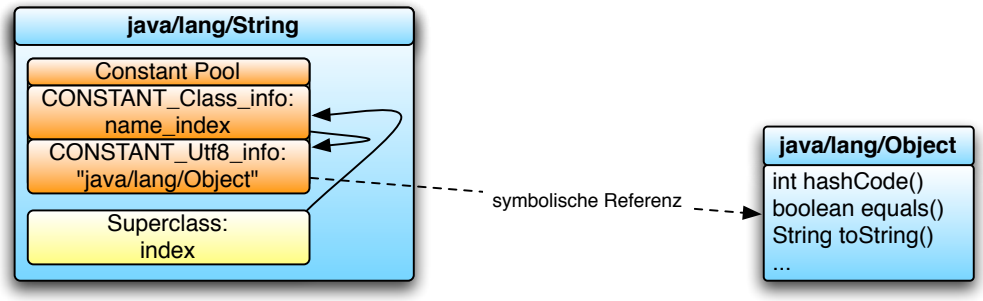


Abbildung 2.10: Symbolische Referenz zwischen Klasse und Superklasse

Neben dem Interpretieren des Bytecodes ist das Auflösen von symbolischen Referenzen eine der häufigsten Aufgaben der JVM. Zum Glück lässt sich dies recht einfach optimieren, worauf im Implementierungsteil (Kapitel 7.1) noch eingegangen wird.

2.6 Der Ladeprozess einer Klasse

Der Ladeprozess einer Klasse unterteilt sich in drei Bereiche, das Laden (Load), Linken (Link) und das Initialisieren (Initialize). Der Vorgang des Linkens wird dabei erneut unterteilt in die Verifikation (Verify), das Vorbereiten (Prepare) und das Auflösen (Resolve). (Abbildung 2.11 zeigt dies grafisch.)

Eine wichtige Unterscheidung besteht darin, dass der JVM relativ freie Hand gelassen wird wann sie das Laden und das Linken ausführt. Ziemlich genau festgelegt allerdings ist der Zeitpunkt wann die Initialisierung stattgefunden haben muss und der vollständige Prozess damit beendet zu sein

[19]Die Klasse Object ist hier ein Sonderfall, denn sie muss per Definition schon geladen sein, bevor die JVM mit dem Laden anderer Klassen beginnt. Sollte die JVM einem symbolischen Link folgen müssen und die entsprechende Klasse noch nicht geladen worden sein, so wird sie versuchen die entsprechende Klasse jetzt zu laden. Schlägt dies fehl, dann wirft sie eine ClassNotFoundException und beendet die Ausführung der Anwendung.

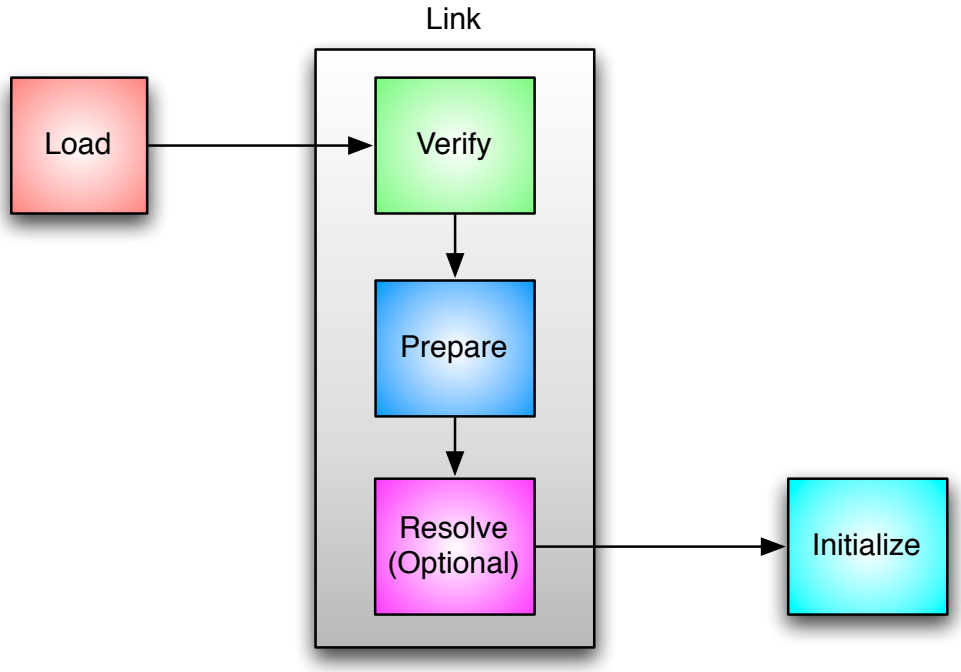


Abbildung 2.11: Ladeprozess einer Klasse

hat. Der Zeitpunkt, wo dies der Fall sein muss, ist der, wo eine Klasse das erste mal aktiv verwendet wird. Aktive Benutzung findet statt wenn eine Instanz der Klasse erzeugt wird, eine statische Methode der Klasse aufgerufen, einer statischen Klassenvariable (außer wenn sie `final` deklariert ist[20]) ein Wert zugewiesen, die Instanz einer Subklasse der Klasse erzeugt wird, oder die Klasse die initiale Klasse ist, also die anfänglich auszuführende `main()`-Methode enthält[21]. Alle übrigen Aktionen, die im Zusammenhang mit einer Klasse ausgeführt werden, folgen entweder implizit einem der oben genannten Fälle von aktiver Benutzung oder fallen in die Kategorie der passiven Benutzung und lösen deshalb nicht zwangsweise den Ladevorgang einer Klasse aus.

[20]Ist eine statische Klassenvariable `final` deklariert, dann betreibt der Compiler in der Regel so genanntes „inlining“. Anstatt dann den Wert zur Laufzeit aus der Variable zu laden, packt er den konstanten Wert direkt in den Bytecode-Datenstrom und spart sich so den entsprechenden Variablen-Zugriff.

[21]Ein weiterer Fall einer aktiven Benutzung wäre der Aufruf verschiedener Methoden des Reflection-Mechanismus, welcher im Umfang dieser JVM allerdings nicht enthalten ist und auch nicht zwangsweise vorhanden sein muss.

Mit dem Laden der Klasse bezeichnet man den Vorgang, in dem eine Klasse über den voll qualifizierten Namen lokalisiert und der Inhalt der entsprechenden `.class` Datei von der JVM eingelesen, verarbeitet und in die Laufzeitrepräsentation überführt wird. Bei vollständigen JVM Implementierungen schließt dies allerlei Varianten mit ein, wie zum Beispiel das Laden einer Klasse über ein Netzwerk, aus einer Archivdatei (ZIP, JAR, CAB), aus einer Datenbank und so weiter. Die verschiedenen Quellen werden in der Regel mit unterschiedlichen Classloadern bedient, was das System recht modular gestaltet. Einfachere JVM Implementierungen verwenden manchmal jedoch keinen Classloader-Mechanismus. Bei ihnen ist das Laden von einer Quelle, wie zum Beispiel aus Dateien, direkt in der JVM implementiert. Einen Ausnahmefall stellen die Array-Klassen dar, diese werden nicht von irgendwo geladen, sondern je nach Bedarf zur Laufzeit erzeugt.

Die Verifikation kontrolliert die geladenen Laufzeitinformationen der Klasse nach detaillierten Vorgaben auf Korrektheit und Stimmigkeit. Durch diesen Schritt spart man weitere Überprüfungsschritte während des Ausführens des Codes der Klasse, und man verhindert durch frühes Blockieren manipulierte oder defekte Klassen, welche die JVM zum Absturz oder in einen inkonsistenten Zustand bringen könnten. Kapitel 4.9 der JLS ([Lindholm99]) behandelt den Prozess der Verifikation im Detail.

Beim Vorgang des Linkens folgt nun die Vorbereitungs-Phase (Prepare). Diese beinhaltet das Bereitstellen von Speicherplatz für statische Klassenvariablen und deren Initialisierung mit den Standardwerten[22]. Zu diesem Zeitpunkt wird noch kein Code dieser Klasse ausgeführt und auch die eigentlichen Initialisierungswerte der Variablen werden noch nicht gesetzt. Dies folgt erst im Initialisierungs-Schritt. Der JVM steht es offen in diesem Schritt weitere, der Klasse zugehörige, Datenstrukturen anzulegen und zu füllen. Üblich ist hier zum Beispiel das Erzeugen einer Methoden-Tabelle (Method Table), oder das Errechnen und Cachen verschiedener Werte, wie zum Beispiel der Anzahl von Slots, die bei einem Methodenaufruf übergeben werden müssen.

[22] Alle Klassenvariablen werden mit den ihrem Typ entsprechenden 0-Werten 0, 0.0, `false` oder `null` initialisiert.

(Diese Information liegt nur implizit in Form des Descriptors der Methode vor und muss recht aufwändig extrahiert werden.)

Als abschließende Phase des Link-Vorgangs folgt die optionale Phase des Auflösens (Resolve). Dieser Vorgang ist dafür gedacht, die im vorigen Kapitel angesprochenen symbolischen Referenzen aufzulösen. Praktisch wird dieser Schritt jedoch eher nicht an dieser Stelle ausgeführt, sondern erst bei der ersten aktiven Benutzung. Dieses Vorgehen verkürzt die Ladezeiten und verhindert das unnötige Erzeugen niemals verwendeter Daten.

Es folgt der finale Vorgang der Initialisierung (Initialization). Diese umfasst das Ausführen von statischen Initialisierern (`static`-Blöcken in Java) und dem Code für die Initialisierung von statischen Variablen der Klasse. All dies wurde vom Compiler, der Reihenfolge nach wie es im Quellcode steht, in der `<clinit>`-Methode[23] zusammengefasst. Bevor diese jedoch ausgeführt wird, werden rekursiv alle Superklassen der Klasse initialisiert und deren `<clinit>`-Methoden aufgerufen.

Die Klasse ist nun fertig geladen, überprüft und vorbereitet und kann fortan eingesetzt werden.

[23]Dieser Bezeichner ist in durch die spitzen Klammern in Java als Name einer Methode nicht erlaubt und verhindert somit, dass er fälschlicherweise vom Programmierer und vom Compiler deklariert wird, was zu Kollisionen führen würde.

Kapitel 3

Datenstrukturen für virtuelle Maschinen

In diesem Kapitel geht es um generelle Datenstrukturen, die im Bereich von virtuellen Maschinen Anwendung finden. Behandelt werden Stacks und Heaps. Beide sind Strukturen, ohne die eine virtuelle Maschine kaum realisierbar wäre.

3.1 Stacks

Ein Stack[1] stellt vom Prinzip her eine einfache Datenstruktur dar, die nach dem LIFO[2]-Prinzip Daten speichert.

Man kann sich das vorstellen wie einen Stapel auf einem Schreibtisch. (Siehe Abbildung 3.1.) Die hereinkommende Post wird in der Reihenfolge des Eingangs auf den Stapel gelegt und eine nach der anderen von oben wieder aufgenommen, bearbeitet und abgeheftet. Beide Arbeiten, das Auflegen neuer Dokumente und das Abarbeiten existierender, kann in beliebiger Art und Weise abgewechselt werden[3].

[1] Dieser Begriff wird im Deutschen mit „Stapel“ übersetzt, was auch die Funktion korrekt wiedergibt. Jedoch hat sich auch im deutschsprachigen Bereich der Begriff Stack stark durchgesetzt, weshalb der Autor hier weiterhin den englischen Begriff verwenden wird.

[2] Last In First Out

[3] Es sei denn der Stapel wird irgendwann so groß, dass er umfällt.

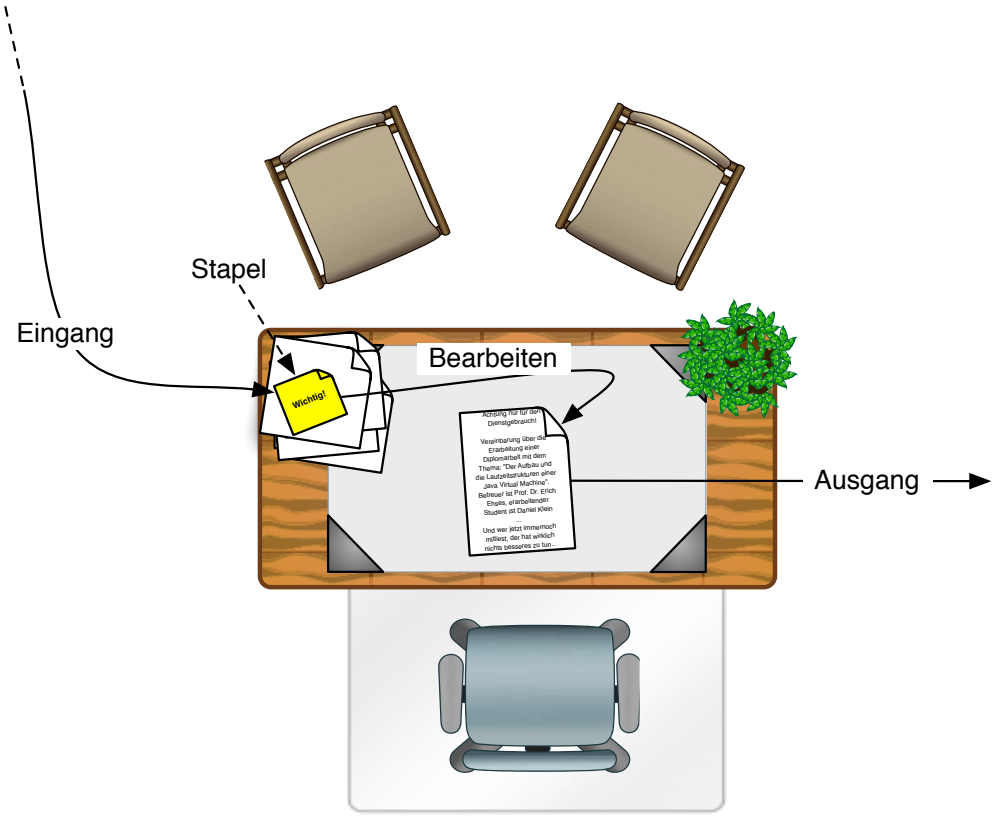


Abbildung 3.1: Beispiel für einen Stapel am Arbeitsplatz

Auch müssen die Dokumente, die auf den Stapel gelegt werden, nicht unbedingt die selbe Größe haben. So geht zum Beispiel auch ein kleiner Notizzettel im Stapel nicht verloren, wenn man anschließend wieder größere Dokumente darauf legt. Auch muss man sich auf einem Schreibtisch nicht auf einen einzigen Stapel beschränken, so lange denn der Platz auf dem Schreibtisch noch für weitere ausreicht.

Das Beispiel hinkt ein wenig, denn im Worst Case (der schlimmstmögliche Fall) würden die untersten Dokumente in einem Stapel nie bearbeitet werden. Eine FIFO[4]-Struktur wäre hier wohl die bessere Lösung, die Funktionsweise stellt sich über das Beispiel aber schön anschaulich dar, was für diesen Fall ausreichen soll.

[4]First In First Out, eine Warteschlange oder auch Queue.

Im Bereich der virtuellen Maschinen kommen nun analog zu diesem Beispiel auch zwei verschiedene Stacks zum Einsatz. Sie werden Method Stack und Operand Stack bezeichnet und im folgenden genauer erläutert[5]. Auch das Prinzip der unterschiedlich großen Elemente wird sich beim Method Stack wiederfinden. Man verwendet hier so genannte Stack Frames, deren Größe sich abhängig vom Kontext ergibt.

3.1.1 Operand Stack

Ein Operand Stack wird, wie der Name schon vermuten lässt, zum Speichern und Weiterreichen von Operanden verwendet. Bei der JVM sind Operanden entweder generische Datentypen, also Integer- und Fließkommazahlen, oder Referenzen auf Objekte. Der Operand Stack wird dazu benutzt, diese Operanden temporär zu speichern und um sie von einer Aktion zur nächsten weiter zu reichen. Wenn man zum Beispiel zwei Integerzahlen addieren möchte, so packt man die beiden Summanden auf den Operand Stack und führt den „iadd“ Opcode aus. Dieser nimmt die beiden Summanden vom Stack wieder herunter, addiert sie und schiebt nun seinerseits das Ergebnis auf den Stack, welches dann im Anschluss weiter verarbeitet wird. Abbildung 3.2 zeigt diesen Vorgang noch einmal grafisch.

Eine weitere Aufgabe kommt dem Oprand Stack bei Methodenaufrufen zu. Für einen solchen werden die zu übergebenden Parameter (für eine Java Methode von links nach rechts) der Reihenfolge nach auf dem Stack abgelegt und im Anschluss der Methodenaufruf ausgeführt. Die aufgerufene Methode nimmt die übergebenen Operanden wieder vom Stack herunter und nutzt sie. Ist die Ausführung der Methode beendet und muss ein Rückgabewert übergeben werden, dann wird dieser nun auf den Stack geschoben und die Programmausführung kehrt zur vorigen Methode zurück. Diese wiederum nimmt den Rückgabewert, wenn es einen gibt, vom Stack und nutzt ihn.

[5]Tatsächlich handelt es sich bei dem in Abbildung 3.1 gezeigten Beispiel um eine Art Operand Stack, da hier unterschiedliche Personen, zum Beispiel die Sekretärin und der Sachbearbeiter, jeweils auf den Stapel zugreifen. Würde der Sachbearbeiter die neuen Dokumente zuerst entgegennehmen und dann erst selber auf den Stapel legen, dann würde es sich um eine Art Method Stack handeln. Weiteres zu den beiden Stack-Arten folgt in den nächsten zwei Abschnitten.

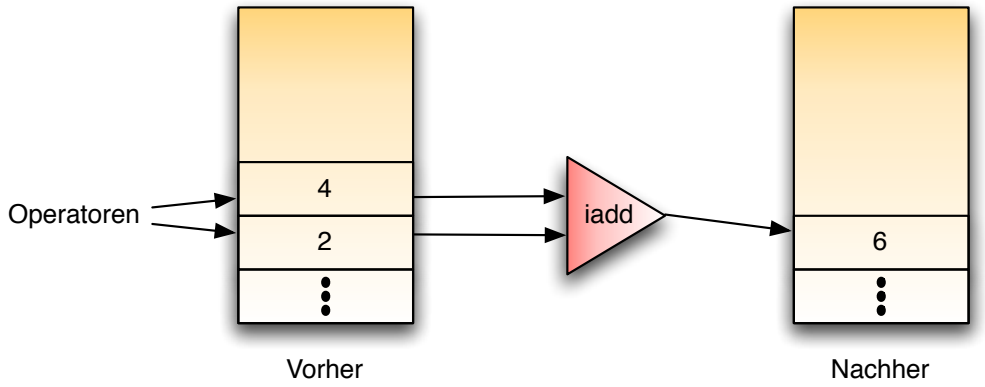


Abbildung 3.2: Verwendung eines Operand Stacks

3.1.2 Method Stack und Stack Frames

Der Method Stack ist ein etwas abstrakteres Gebilde. Auf ihm werden keine direkten Daten, sondern so genannte Stack Frames abgelegt. Method Stacks sind essenziell für die Ausführung moderner Programmiersprachen, denn auf ihnen wird der Ausführungszustand der aktuellen Methode mit all ihren Parametern und lokalen Variablen gespeichert, auch wenn weitere Methoden aufgerufen werden.

Der Aufbau eines Stack Frames hängt stark vom technischen Aufbau einer virtuellen Maschine ab. Bei der JVM beinhaltet er beispielsweise eine Liste der Parameter, die Anzahl der maximal auf dem Operand Stack liegenden Operanden, sowie den Program Counter (PC) und einen Zeiger zum vorherigen Stack Frame. Aufgrund der unterschiedlichen Anzahl der Parameter kann die Größe eines Stack Frames stark variieren.

Abbildung 3.3 zeigt die schematische Darstellung eines Method Stacks mit mehreren Stack Frames.

Wird von der aktuellen Methode nun eine weitere Methode aufgerufen, so speichert die JVM den Program Counter im aktuellen Stack Frame, nimmt die auf dem Operand Stack liegenden Parameter des Methodenaufrufs herunter und erzeugt einen neuen Stack Frame auf dem Method Stack. In diesem werden dann die Parameter des Methodenaufrufs abgelegt, der Zeiger auf den vorigen Stack Frame vermerkt und dann der Code der aufzurufenden

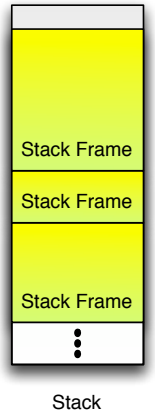


Abbildung 3.3: Schematischer Aufbau eines Method Stacks

Methode ausgeführt. Diese verrichtet ihre Arbeit, ruft vielleicht ihrerseits weitere Methoden auf und kehrt irgendwann nach Abarbeitung ihres Codes wieder zur aufrufenden Methode zurück. Ihr Stack Frame wird dann wieder abgebaut, der Program Counter der vorigen Methode wieder hergestellt und die Ausführung des Codes der vorigen Methode mit dem nächsten Opcode fortgesetzt.

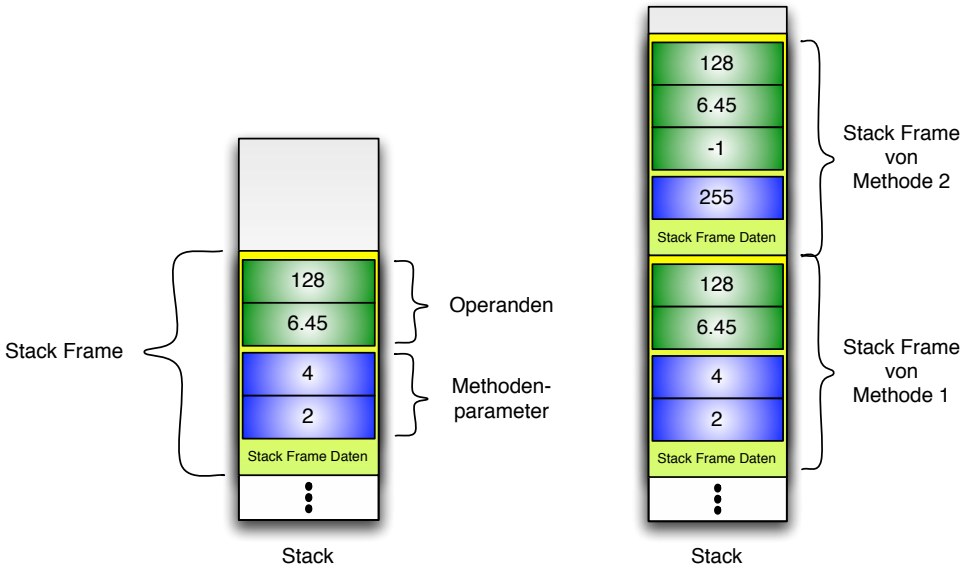
Auf diese Weise lässt sich heute problemlos Rekursion verwirklichen. Alte Programmiersprachen wie FORTRAN besitzen keinen Method Stack, weshalb man rekursive Methodenaufrufe höchstens durch Tricks realisieren konnte. Aus diesem Grund ist Rekursion in FORTRAN gar offiziell verboten.

3.1.3 Kombination auf einem Stack

Da sich Operand Stack und Method Stack immer nur nacheinander verändern liegt die Zusammenlegung der beiden auf der Hand. Wird eine Methode aufgerufen, dann wird wie bisher ein neuer Stack Frame aufgebaut und auf den Stack gelegt. Anschließend wird der selbe Stack dafür verwendet, Operanden auf ihm abzulegen. Wird während der Ausführung einer Methode eine weitere aufgerufen, so bleiben auf dem Stack liegende Operanden wo sie sind und der nächste Stack Frame wird einfach darauf gelegt. Nach Beendigung der Ausführung dieser Methode wird der entsprechende Stack Frame abgebaut

und der Stack befindet sich wieder in dem Zustand, in dem er sich befunden hat bevor die Methode aufgerufen wurde. Die noch auf dem Stack liegenden Operanden stehen wie vorher wieder zur Verfügung und das Ausführen der Operationen der Methode kann fortgesetzt werden.

Wenn man, wie zum Beispiel bei der JVM, die maximale Anzahl der auf dem Stack liegenden Operanden kennt, dann kann man die Operanden auch gleich noch in den Stack Frame mit einbeziehen und diesen entsprechend dimensionieren. Abbildung 3.4(a) zeigt eine schematische Darstellung dieser Variante. Die Stack Frame Daten beinhalten den Program Counter und den Zeiger auf den vorherigen Stack Frame, dann folgen erst die Parameter, die der entsprechenden Methode bei Aufruf übergeben worden sind und dann die Operanden, für die genügend Platz vorgesehen worden ist. Abbildung 3.4(b) zeigt den selben Stack, nachdem eine weitere Methode aufgerufen und dafür ein weiterer Stack Frame auf dem Stack aufgebaut wurde.



(a) Darstellung eines kombinierten Stacks

(b) Kombinierter Stack mit zwei Stack Frames

Abbildung 3.4: Die Kombination von Method- und Operand Stack

3.1.4 Realisierungen eines Stacks

In diesem Abschnitt werden nun verschiedene Implementierungsmöglichkeiten von Stacks gezeigt. Grundsätzlich unterscheidet man zwischen Realisierungen mit einem Array (bzw. einem Speicherbereich, je nach verwendeter Programmiersprache) und einer verketteten Liste.

Bei einer Array-Implementierung wird ein ausreichend großer Speicherbereich reserviert. Der Speicherbereich muss so groß sein, dass er die nötige Anzahl von Stack Frames und Operatoren aufnehmen kann, so dass er die maximale Verschachtelungstiefe eines Programms verkraften kann. Besonders bei der Verwendung von rekursiven Algorithmen kann es hier schnell zu Problemen kommen, da bei diesen die Verschachtelungstiefe von den Parametern der Anwendung abhängen kann und somit vorher nicht definitiv festzulegen ist. Sollte der für den Stack reservierte Speicherplatz nicht ausreichen, dann kann der entsprechende Thread (jeder Thread hat seinen eigenen Stack und ist dadurch unabhängig von den anderen) nur noch abgebrochen werden. Die JVM wirft in solch einem Fall einen StackOverflowError, was die Ausführung des entsprechenden Threads in der Regel beendet. Abbildung 3.5 zeigt den Augenblick eines solchen Überlaufs.

Bei der Realisierung eines Stacks mit Hilfe einer verketteten Liste kann es nicht zu den Problemen eines überfüllten Stacks kommen, da diese Realisierung von Natur aus kein solches Limit kennt. Jedes Mal wenn ein neuer Stack Frame für einen Methodenaufruf angelegt wird, wird für diesen neuer Speicher vom System allokiert. Dieser Speicherbereich muss so groß sein, dass er den neuen Stack Frame und die maximale Anzahl der zu einer Zeit auf dem Stack liegenden Operanden während der Ausführung der neuen Methode speichern kann. Die einzelnen Teile des Stacks werden gemäß dem Verfahren einer verketteten Liste miteinander verbunden. Eine einfache, rückwärts verkettete Liste reicht hier in der Regel aus, da man lediglich vom aktuellen Stack Frame aus den vorigen wieder finden können muss. Abbildung 3.6 zeigt einen solchen Stack, der mit einer verketteten Liste realisiert ist.

Das einzige Limit einer Stack Implementierung als verkettete Liste wäre das allgemeine Speicherlimit des Prozesses. Eine solche Implementierung in

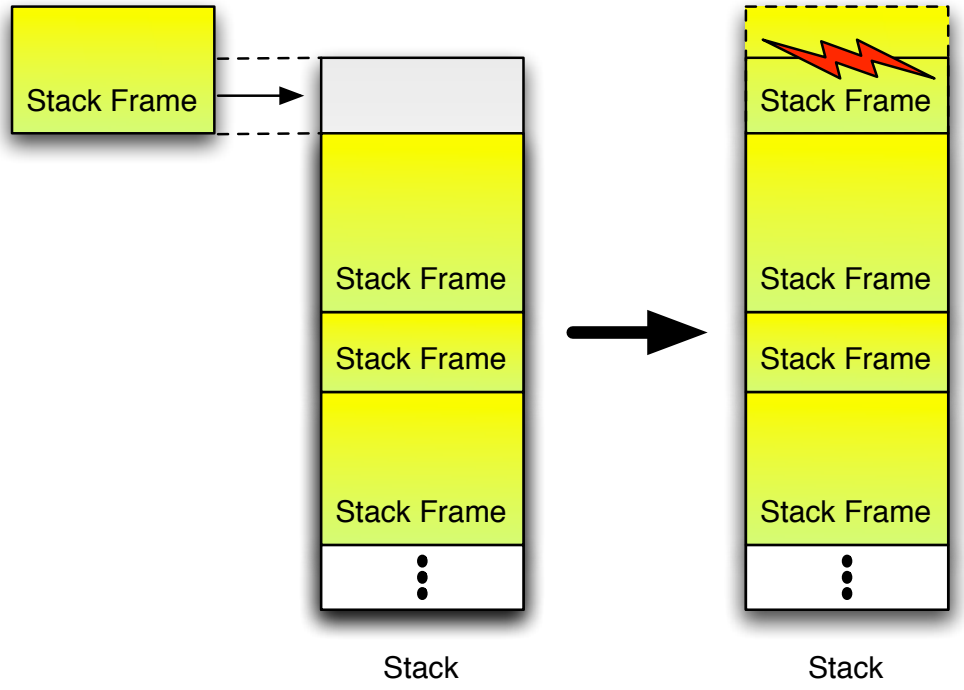


Abbildung 3.5: Überlauf einer Array-Stack Implementierung

einer JVM würde also so lange laufen, bis der gesamte Heap (siehe nächster Abschnitt) aufgebraucht wäre, so dass der JVM nichts anderes übrig bliebe als eine OutOfMemoryException zu werfen.

Von der Praxis her ist der theoretisch ganz vernünftig anmutende Ansatz mit der verketteten Liste eher nicht zu gebrauchen, da er eine eher bescheidene Performance aufweist. Das hat vor allem damit zu tun, dass mit jedem Methodenaufruf ein neues Speichersegment vom Speichermanager allokiert und beim Beenden einer Methode auch wieder freigegeben werden muss, was einen beträchtlichen Zeitaufwand darstellt. Die Array Lösung ist deutlich schneller, da dort nur ein Index (bzw. ein Zeiger) geändert werden muss um einen neuen Stack Frame anzulegen. – So lange denn noch Platz dazu ist.

Der dritte Implementierungsansatz ist eine Kombination aus den beiden zuvor genannten. Mit ihr werden die Performance-Probleme der verketteten Liste und das Problem der maximalen Größe bei der Array-Lösung elegant

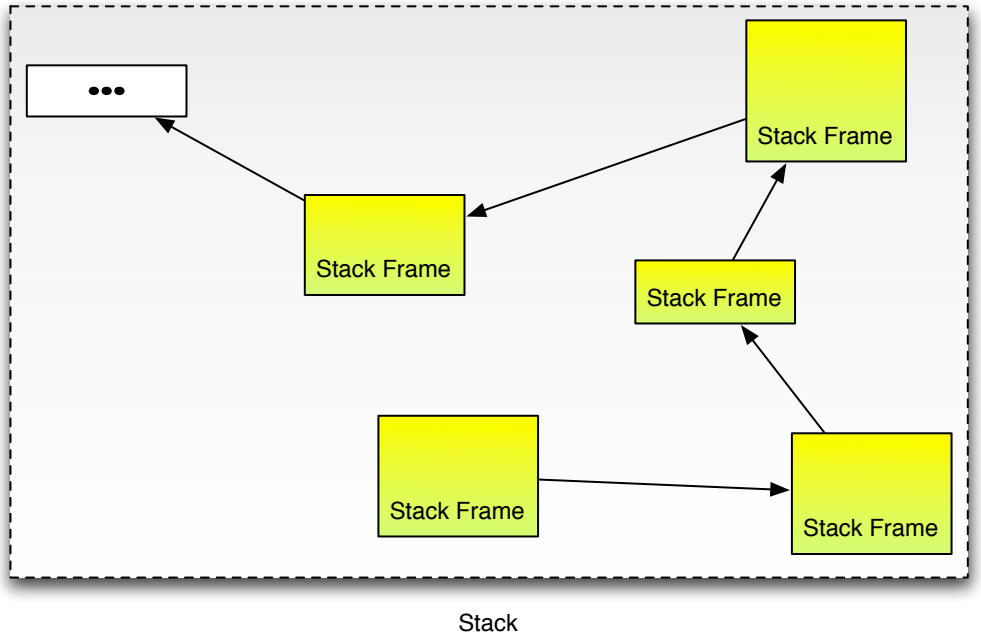


Abbildung 3.6: Implementierung eines Stacks mit verketteter Liste

umgangen, in dem man die beiden Techniken miteinander verbindet. Anstatt immer nur Speicher für einen einzigen Stack Frame zu allokieren, wird wie bei der Array-Lösung immer ein großer Speicherbereich angelegt. Auf diesem wird dann gemäß der Array-Lösung gearbeitet bis der Speicherbereich gefüllt ist. Anstatt jetzt jedoch eine Exception zu werfen wird einfach ein neuer großer Speicherbereich allokiert, dort ein Zeiger auf den vorherigen vermerkt und dann im neuen Speicherbereich fortgesetzt. Abbildung 3.7 zeigt die Darstellung eines solchen Stacks.

Bei diesem Ansatz muss beim Auf- und Abbau von Stack Frames zwar mehr Aufwand getrieben werden, da immer überprüft werden muss, ob man an die Grenzen eines Speicherbereichs stößt, jedoch wiegt dies die Nachteile der anderen Lösungen in der Regel auf. Dennoch trifft man auf den gängigen Computersystemen heutzutage eher die einfache Array-Lösung an. Zum Beispiel ist dies bei Suns JVM-Implementierung der Fall. Der Grund dieser Entscheidung liegt wohl in der maximierten Performance und der Tatsache, dass heutige Systeme über so viel Speicher verfügen, dass problemlos

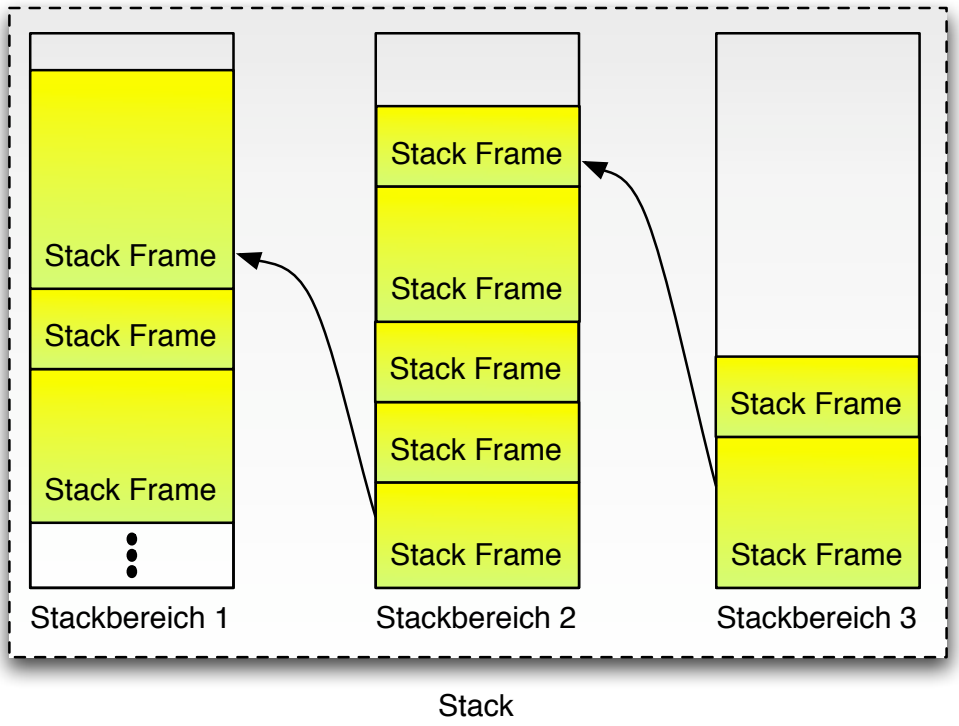


Abbildung 3.7: Kombination von Array und verketteter Liste

gleich von Anfang an ein genügend großer Speicherbereich für den Stack allokiert werden kann. Die Mischlösung wiederum wird eher im Embedded- und Mobile-Bereich eingesetzt, weil Speicher dort (noch) nicht im Überfluss vorhanden ist.

3.2 Heap

Der Heap im Bereich der virtuellen Maschinen und Programmiersprachen ist eine rein unsortierte Datenstruktur, die ein zentraler Sammelplatz für Speicherblöcke ist. (Ein Haufen von Speicherblöcken. Daher der Name.) Der Heap ist dabei eine rein passive Struktur. Er selber nimmt keinen Einfluss darauf was mit ihm passiert, sondern wird von außen heraus, entweder direkt von einer Anwendung oder seinen Bibliotheken, oder von einem

Speicherverwaltungs- oder Säuberungssystem (wie einem Garbage Collector) verwaltet.

Die hier diskutierte Struktur des Heaps ist übrigens nicht zu verwechseln mit den in der Algorithmik bekannten (binären) Heaps. Im Gegenteil, die beiden Heaps stellen vollkommen unterschiedliche Konzepte dar. So ist der binäre Heap eine sortierte, teilweise sogar selbst optimierende Datenstruktur, mit der effektiv, vergleichbar mit Baumstrukturen, Daten abgelegt werden können. Die Nutzung des selben Begriffs für die beiden Datenstrukturen rührt wohl daher, dass beide von der Struktur her tatsächlich an einen Haufen (Deutsch für Heap) erinnern. Trotz alledem soll die Diskussion hier auf den unsortierten Heap, so wie er bei virtuellen Maschinen verwendet wird, beschränkt bleiben. Details zu den binären Heaps findet der geneigte Leser beispielsweise in [Cormen04].

3.2.1 Heap mit klassischem Speichermanagement

In der objektorientierten Programmierung werden die Instanzen von Klassen (die Objekte) auf einem Heap angelegt. Ein Speichermanagement regelt das Vergeben und Freigeben von Speicher auf dem Heap und teilt der Anwendung die entsprechende Adresse (den Zeiger oder Pointer) eines Objekts auf dem Heap mit. Mit Hilfe dieses Zeigers kann die Anwendung nun beliebig auf das Objekt zugreifen. Was die Anwendung allerdings nicht weiß ist, wie es um das Objekt herum auf dem Heap aussieht. Es kann also sein, dass Objekte dicht an dicht direkt hintereinander gepackt sind, so dass zwischen ihnen kein Platz mehr ist, es kann aber auch sein, dass die Objekte ganz locker auf dem Heap verteilt sind. Grundsätzlich soll die Frage der Organisation des Heaps die Anwendung auch nicht weiter interessieren. Die Aufgabe des Speichermanagements besteht nämlich genau in dieser Abstraktion, so dass man nicht mehr, wie zum Beispiel bei maschinennaher Programmierung, den Speicher bis ins Detail selber planen und vergeben muss. Das einzige Aufgabe, bei der Verwendung eines Heaps mit einem klassischen Speichermanagement ist, dass man Objekte zwingend über das Speichermanagement wieder freigeben muss. Sollte man dies nicht tun und die Objekte einfach vergessen, so bleiben

diese als „Speicherleichen“ im Heap liegen, werden aber nie wieder angefasst. In solch einem Fall spricht man von einem Speicherleck oder englisch von einem Memory Leak.

Abbildung 3.8 zeigt einen solchen klassischen Heap mit normalen Zeigern.

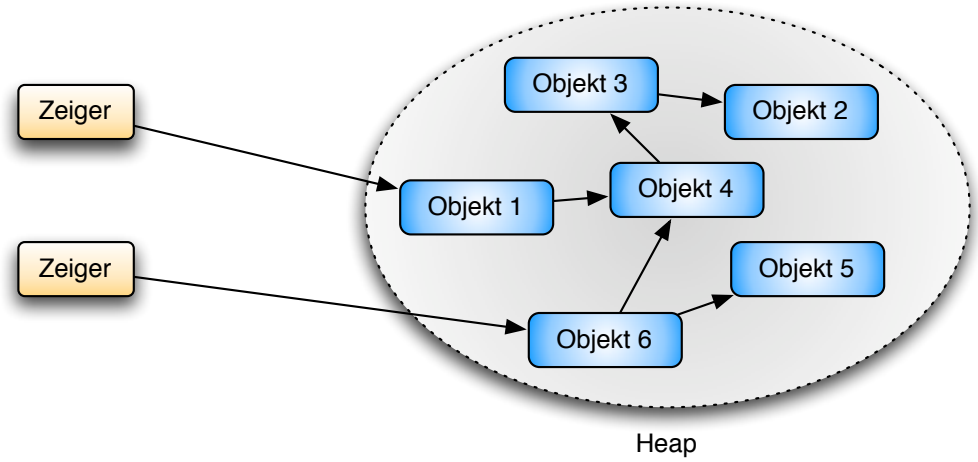


Abbildung 3.8: Beispiel eines Heaps mit Objekten

3.2.2 Referenzen statt Zeiger

Das Speichermanagement moderner Systeme gibt keine direkten Zeiger auf die Objekte des Heaps mehr heraus, sondern führt eine Zwischenschicht ein. Diese Zwischenschicht besteht aus einer Indextabelle, in der die Zeiger auf Objekte gespeichert werden. Anstatt die Zeiger direkt an die Anwendung weiterzureichen wird nun der Index in diese Tabelle an die Anwendung gereicht. In diesem Fall spricht man von einer Referenz.

Abbildung 3.9 zeigt einen solchen referenzierten Heap. Referenzen sind gestrichelt, Zeiger als vollständige Striche gezeichnet. Die Farben kennzeichnen der Übersichtlichkeit halber die Zusammengehörigkeiten zwischen Referenzen und Zeigern.

Der Vorteil dieser Lösung ist, dass man nun intern den Heap umorganisieren kann, ohne dass man Änderungen nach außen hin kommunizieren müsste.

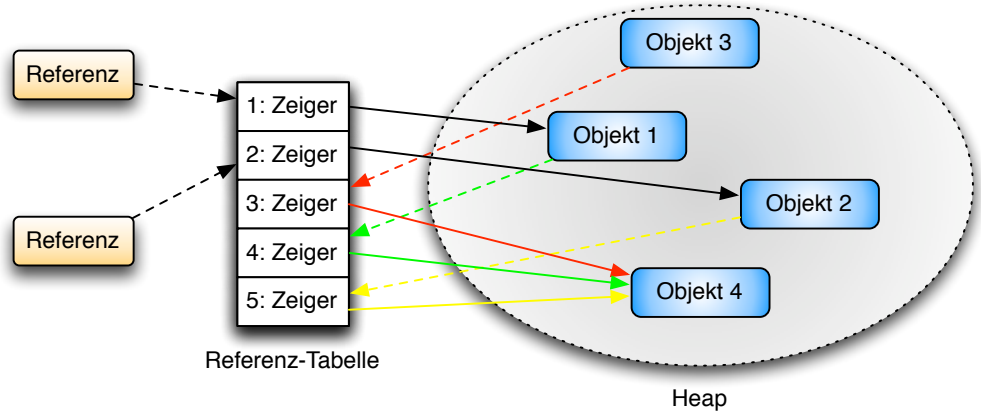


Abbildung 3.9: Beispiel eines Objekt-Heaps mit Referenzen

Sollte zum Beispiel ein Objekt angefordert werden, welches von der Gesamtsumme des freien Speichers auf dem Heap noch allokiert werden können müsste, dann kann es aufgrund von Fragmentierung dazu kommen, dass es trotzdem keinen durchgängigen Speicher in der gewünschten Größe mehr auf dem Heap gibt. Systemen mit einfachem Speichermanagement blieb jetzt nur noch die Möglichkeit den Heap zu vergrößern, was aber nicht immer möglich ist. Auf einem referenzierten Heap können Objekte verschoben und die zugehörigen Adressen in der Referenz-Tabelle problemlos aktualisiert werden, ohne dass dies die Anwendung beeinträchtigen würde.

Eine weitere, übliche Lösung, die mit referenzierten Heaps in der Regel auch noch eingeführt wird, ist die Garbage Collection. Die Anwendung braucht sich dann nicht mehr um die Freigabe von nicht mehr verwendeten Objekten zu kümmern, sondern der Garbage Collector erledigt dies. Dies hat den Vorteil, dass es im Prinzip nicht mehr zu vergessenen Objekten auf dem Heap kommen kann. Tatsächlich ist es in der Praxis ein klassisches Problem bei Systemen die keine Garbage Collection haben, dass es dort besonders bei lange laufenden Systemen immer wieder zu Problemen mit Speicherlecks kommt.

Referenzierte Heaps und Garbage Collection treten jedoch nicht immer paarweise auf. So gibt es auch (ineffizientere) Garbage Collection Systeme für

normale, verzeigerte Heaps und referenzierte Heaps ohne Garbage Collection sind ebenso vorstellbar.

Auch wenn hier bisher von einem Garbage Collection System gesprochen wurde, so gibt es jedoch nicht nur eine mögliche Implementierung dafür, im Gegenteil, es gibt Dutzende verschiedene Ansätze. Die Gängigsten werden in [Blunden03] beschrieben und das sehr ausführliche [Jones96] widmet sich ausschließlich dem Thema Garbage Collection.

Kapitel 4

Programmiersprache und verwendete Tools

In diesem Kapitel wird kurz erläutert, welcher Compiler und welche Tools für die Erarbeitung der Diplomarbeit eingesetzt wurden und warum die Entscheidung zu Gunsten dieser getroffen wurde.

4.1 Programmiersprache und Sprachstandard

Aufgrund der Vorgabe der Portabilität und um die Wiederverwendbarkeit im Embedded-Bereich sicherzustellen, musste die Programmiersprache und der Sprachstandards sehr sorgfältig ausgewählt werden.

Da die Implementierung der virtuellen Maschine klassisch maschinennah sein sollte, war die Wahl einer maschinennahen Sprache wie C oder C++ naheliegend. Höhere Sprachen wie Java und C# schieden daher von vornherein aus. Aufgrund der nötigen eigenen virtuellen Maschine, Laufzeitumgebung und Bibliotheken wäre die Verwendung einer in einer solchen Sparche realisierten VM im Embedded-Bereich außerdem nicht möglich gewesen, was jedoch ein Wunsch des Autors war. Auch die Wahl von C++ als Programmiersprache musste ausgeschlossen werden, denn auch hier sind die Entwicklungen im Embedded-Bereich noch nicht so weit fortgeschritten. Die meisten Compiler sind dort bisher reine C Compiler. Allein schon aufgrund der ge-

steigerten Platzanforderungen der Kompilate haben sich die objektorientierte Sprachen in diesem Bereich einfach noch nicht durchgesetzt.

Die Entscheidung für C als Programmiersprache stand demnach schnell fest, stellte sich nur noch die Frage nach dem zu verwendenden Standard. C ist immerhin nicht C. Einige verschieden alte Standards buhlen um das Interesse der Programmierer und Compilerbauer. Nach ausführlicher Konsultation von [Hook05] stand fest, dass nur ein älterer C-Standard überhaupt in Frage kommt. Immerhin unterstützen noch nicht einmal die großen Compiler-Pakete wie die GNU Compiler Collection oder Microsofts Visual Studio Compiler die neuesten Standards vollständig, auch wenn es diese schon seit einigen Jahren gibt. In die engere Auswahl kamen die Standards ANSI C89 und ANSI C99. Wiederum aufgrund von Einschränkungen im Embedded-Bereich wurde der älteste Standard, ANSI C89 – im folgenden einfach C89 genannt – gewählt. Gerade die Portierbarkeit auf andere Plattformen und die Verwendbarkeit mit verschiedenen Compilern ist so am besten gegeben.

Leider bedeutete die Wahl von C89 als Sprachstandard einige ungewohnte Einschränkungen. So kennt C89 weder einzeilige Kommentare, an die sich der Autor durch die Verwendung von Java und C++ in den letzten Jahren sehr gewöhnt hatte, noch kennt C89 die heute übliche Variablendeklaration innerhalb einer `for`-Schleife, so dass die Schleifenvariable immer vorher deklariert werden muss. Aber diese Einschränkungen waren nicht schwerwiegend genug, um von der Entscheidung für diesen Standard abzuweichen.

4.2 Compiler

Die Entscheidung für den C Compiler der GNU Compiler Collection (GCC) war schnell getroffen. Dieser unterstützt den gewählten Sprachstandard C89, steht für alle gängigen Betriebssysteme (Windows, Linux, Mac) zur Verfügung und kommt auch im Embedded-Bereich zur Anwendung. In diesem Umfang steht kein anderer plattformübergreifender Compiler zur Verfügung und da man bei der Nutzung verschiedener Compiler für die einzelnen Plattformen letzten Endes doch immer mit Kompatibilitätsproblemen zu rechnen hat, war eine plattformübergreifende Lösung eindeutig vorzuziehen.

4.3 Apple Mac OS X Entwicklertools

Die Wahl von Apples Entwicklungswerkzeugen für die Erarbeitung der Diplomarbeit lag nahe, da der Autor Macs einsetzt und weil sie zur kostenlosen Standardausrüstung von Apples Betriebssystem zählen. Auch verwenden sie die GCC als Basis, was glücklicherweise der Wahl des Compilers entsprach.

Durch das Vorhandensein von Macs mit zwei unterschiedlichen Architekturen (Intel und PowerPC Systeme) ergab sich außerdem die Möglichkeit, schon während der Entwicklung durch einfachen Wechsel der Rechner, aber bei weiterer Verwendung derselben Tools, die Lauffähigkeit der Entwicklung unter beiden Systemen zu testen. Dies war besonders deshalb von Vorteil, weil sich die Systeme durch unterschiedliche Endianess[1] unterscheiden, was eines der klassischen Probleme bei Portierungen darstellt.

[1]Mit „Endianess“ bezeichnet man die unterschiedliche Reihenfolge in der Bytes im Speicher eines Systems abgelegt werden. Man unterscheidet zwischen „Little Endian“ (niederwertiges Byte zuerst) und „Big Endian“ (höherwertiges Byte zuerst).

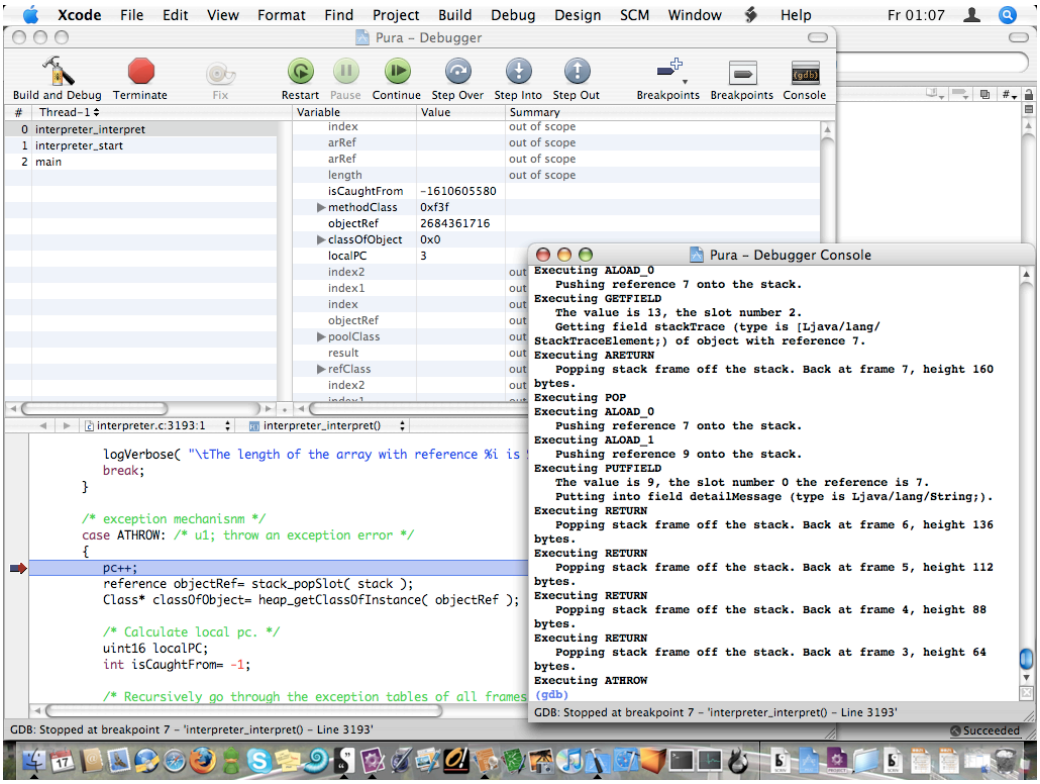


Abbildung 4.1: Xcode, die Entwicklungsumgebung der Mac OS X Entwicklertools

Kapitel 5

Übersicht über die Implementierung

Bevor in den folgenden Kapiteln auf Details der Implementierung der JVM eingegangen wird, soll nun ein Überblick über die Aufgabenstellung und die konkrete Implementierung der JVM gegeben werden.

5.1 Aufgabenstellung

Das grundsätzliche Ziel des praktischen Teils der Diplomarbeit war eine einfache, aber dennoch möglichst vollständige Implementierung einer Java Virtual Machine. Die Priorität wurde hierbei jedoch ausdrücklich auf den Schulungscharakter gelegt. Das heißt, gute Lesbarkeit des Codes ging eindeutig über Performance und die Verwendung möglichst einfacher Algorithmen sollte wenn möglich bevorzugt werden. Detaillierte Ausgaben zur Laufzeit sollen Einblick in die inneren Abläufe geben, was sehr zum weiteren Verständnis des Ablaufs eines Programms in der virtuellen Maschine beitragen kann.

Grundsätzlich wurde die JVM so geplant, dass sie praktisch als „drop-in replacement“[1] für Suns `java`-Kommando dienen kann. Ein Aufruf der Implementierung mit denselben Parametern sollte also nachweisbar dieselbe, oder

[1] Im direkten Austausch.

wo nicht problemlos möglich, eine ähnliche Text-Ausgabe auf der Konsole erzeugen[2].

Aufgrund ihres Umfangs ausdrücklich ausgeschlossen aus der Diplomarbeit waren der Verifier sowie die Garbage Collection. Auch wenn diese Bestandteile offiziell Teil einer JVM-Implementierung sind, so sind sie doch in ihrer Realisierung so aufwendig, dass jede einzelne vom Umfang her schon gut eine eigene Diplomarbeit werden könnte[3].

5.2 Implementierung

Nach der Wahl von C als Programmiersprache in Kapitel 4 wurde entschieden, die Implementierung in „objektorientiertem C“ zu realisieren. Konkret bedeutet dies, dass man in der funktionalen Programmiersprache C mit objektorientierten Konzepten arbeitet. So werden nötige Informationen in der Regel in einer Struktur[4] abgespeichert, ähnlich wie man sie bei objektorientierten Programmierpsrachen in einem Objekt ablegt und diese bei einem Funktionsaufruf als Parameter übergeben. Globale Variablen werden so kaum benötigt und die Implementierung wird modular, übersichtlich und leicht erweiterbar.

Die Ausführungseinheit der Implementierung wurde, hauptsächlich des Aufwands wegen, als klassischer Interpreter realisiert. Grundsätzlich wurde das System Singlethreaded implementiert, jedoch ist es von Anfang an so ausgelegt, dass es später problemlos auf Multithreading erweitert werden kann. (Mehrere Threads bedeuten separate Stacks für jeden Thread und Synchronisierung an den Zugriffsstellen auf allgemeine Bereiche wie Heap und Method Area.) Auch hier kam die Verwendung objektorientierter Konzepte

[2]Mathematische Berechnungen müssen selbstverständlich auch korrekt sein. Konkret geht es hier aber um Textausgaben, wie sie von Objekten aus der Standard-Klassenbibliothek erzeugt werden, wenn deren toString()-Methode benutzt wird.

[3]Vielleicht findet sich jemand, der sich mit seiner Diplomarbeit einem dieser Themen annimmt. Die vorliegende JVM Implementierung stellt der Autor dafür gerne zur Verfügung.

[4]In C ein `struct`.

der einfachen Erweiterbarkeit zu Gute. Die in C sonst übliche Verwendung von globalen Variablen hätte dies deutlich erschwert.

Die Implementierung ist weitestgehend typlos realisiert worden. Das heißt, dass die virtuelle Maschine selber nicht weiß, was für einen Typ eine Variable hat, sofern sie nicht aufgrund des Opcodes darauf schließen kann. Eine Ausnahme stellen hier jedoch die Arrays dar. Die Speicherung deren Typs ist zwangsweise nötig, da ansonsten zur Laufzeit keine Typprüfungen mit `instanceof` ausgeführt werden könnten.

Praktisch stellt diese typlose Realisierung kein Problem dar, da der Verifier die Klasse schon beim Laden ausgiebig inspizieren sollte. Für diese Arbeit wurde grundsätzlich von der Guthaftigkeit und Korrektheit von `.class` Dateien ausgegangen, da, wie oben schon erwähnt, kein Verifier vorgesehen ist. Würde man einen solchen nachrüsten, dann stünde der sicheren Verwendung der JVM jedoch nichts im Wege.

Probleme mit der typlosen Realisierung würden potenziell lediglich im Bereich des Reflection-Mechanismus heute üblicher JVMs entstehen. Dieser wäre ohne die Laufzeit-Typinformationen in der aus Suns Implementierung gewohnten Form nicht zu realisieren. Für kleinere JVMs, wie zum Beispiel den CLDC[5] JVM Versionen für Mobiltelefone, ist die Unterstützung des Reflection-Mechanismus aber sowieso nicht vorgesehen. Aus diesem Grund sah der Autor kein Argument gegen die (einfachere) Implementierung einer typlosen JVM gegeben, da diese von vorn herein nicht als Ersatz für Suns JVM vorgesehen war.

Alle Opcodes, mit Ausnahme der zwei Synchronisations-Opcodes, die nur für den Multithreading-Betrieb benötigt werden, wurden implementiert. Leider kann im Umfang dieser Arbeit auf diese Implementierung nicht weiter eingegangen werden. Dem geneigten Leser sei jedoch ein Blick in den Quellcode empfohlen.

Neben dem Interpreter, mit der Implementierung aller Opcodes, besteht das Projekt aus folgenden weiteren Modulen: Heap, Memory Management, Stack, Klasse, Klassenlader (Classloader), `native` Methoden und Logging.

[5]Connected Limited Device Configuration, eine JVM für Systeme mit stark eingeschränkten Resourcen.

Heap, Stack, das Handling von Klassen und deren Laufzeitstrukturen werden im folgenden behandelt. Für den Rest bleibt auch wieder nur der Verweis auf den beiliegenden Quellcode.

Um den geforderten, rudimentären Konsolenbetrieb der JVM gewährleisten zu können, mussten, außer der JVM selber, auch Teile der Klassenbibliothek implementiert werden. Ursprüngliche Planungen, die Bibliotheken von Suns JDK zu verwenden, wurden schnell wieder verworfen, da besonders in der Implementierung der Basisklassen des `java.lang` Pakets ausgiebig von nativen Methoden Gebrauch gemacht wird, wobei das Schema dahinter, ohne in den Quellcode von Suns JVM Einblick zu nehmen, nicht klar zu erkennen war. Eine Neuimplementierung mit eigener Logik und eigenen native Methoden war so schneller und problemloser zu realisieren. Besonders weil es sich hier nur um einige wenige Klassen und Basisfunktionen, hauptsächlich zur Ausgabe und Verarbeitung von Strings, handelte.

Kapitel 6

Implementierung der Laufzeitstrukturen

Dieses Kapitel behandelt die Laufzeitstrukturen der JVM Implementierung. Ausgehend vom in Kapitel 2.4 behandelten Java Class File Format wird die Laufzeitstruktur einer geladenen Klasse hergeleitet und erklärt. Weitere Strukturen ergeben sich aus den Datenstrukturen in Kapitel 3 und werden nun ebenfalls für diese konkrete Anwendung definiert.

6.1 Verwendete Datentypen

Bevor hier nun konkrete Datenstrukturen definiert werden, muss erst einmal das Vokabular, also die verwendeten Datentypen, definiert werden. Ausgegangen wird vom Java Class File Format von der in Kapitel 2.4 eingeführten Notation[1]. Diese und weitere, teilweise auch aus Portabilitätsgründen benötigte Datentypen, sind in der Datei `types.h` definiert und in Abbildung 6.1 aufgeführt.

Begonnen wird mit der Definition eines bool'schen Datentyps. Dieser wurde hier hauptsächlich zum besseren Verständnis definiert. Da es in C nicht wie in Java einen speziellen Datentypen für bool'sche Werte gibt, werden

[1] Zur Erinnerung: Präfix u oder s – u für unsigned/ohne Vorzeichen und s für signed/mit Vorzeichen – und eine Zahl, die die Länge des Datentyps angibt. Zum Beispiel u2 für einen Datentyp mit 16 Bit ohne Vorzeichen.

```
/* boolean */
typedef int boolean;
#define true 1
#define false 0

/* define NULL pointer if not present */
#ifndef NULL
#define NULL (void*)0
#endif

/* general types */
typedef unsigned char byte;

typedef unsigned char uint8;
typedef unsigned short uint16;
typedef unsigned int uint32;
typedef unsigned long long uint64;

typedef signed char  int8;
typedef signed short int16;
typedef signed int   int32;
typedef signed long long  int64;

/* Java class data types */
#define u1 byte
#define u2 uint16
#define u4 uint32

/* runtime engine data types */
#define slot uint32
#define reference uint32
```

Abbildung 6.1: Die Datentypen der JVM Implementierung

diese üblicherweise als Integer ausgedrückt. Ein Wert von 0 bedeutet dann false (falsch), ein Wert größer 0 true (wahr). Der Einfachheit halber wird hier ein `boolean` Datentyp äquivalent zu dem von Java definiert, der jedoch kompatibel mit dem ansonsten in C verwendeten System ist.

Sollte von den Bibliotheken des verwendeten Compilers kein `NULL` Zeiger definiert werden, dann wird es hier getan. In der Regel ist dies nicht nötig, es beugt aber potenziellen Problemen, besonders im Embedded-Bereich, vor. Die klassische Definition von `NULL` entspricht einem ungetypten Zeiger auf die Adresse 0, so wie es hier auch realisiert ist. Die Adresse 0 ist in der Regel keine gültige Adresse, da am untersten Ende des Speichers nicht verschiebbare Bereiche wie Ports zu finden sind und daher vom dynamischen Speichermanagement nicht belegt werden. So kann ein `NULL` Zeiger auch nicht fälschlicherweise mit einer gültigen Adresse verwechselt werden.

Es folgt die Definition von Datentypen jeder gängigen Größe mit und ohne Vorzeichen. Dies ist in C nötig, da die Standard-Datentypen von Compiler zu Compiler und Architektur zu Architektur unterschiedlich sein können. Im Kontext einer virtuellen Maschine muss man sich jedoch auf die Größe einer Variablen verlassen können. Verwendet man innerhalb der Anwendung nun die hier definierten Datentypen, dann braucht man lediglich lokal an dieser einen Stelle die passenden Datentypen des Compilers für einen selbst definierten Datentyp zu spezifizieren und schon arbeitet auch auf einem anderen System die Anwendung garantiert wieder fehlerfrei und ohne unerwartete Nebeneffekte. Klassisch unterschiedliche Größen haben zum Beispiel Integer (`int`) oder long Integer (`long` und `long long`). Im Zweifel müssen diese vor dem Kompilieren der JVM für eine bestimmte Architektur und mit einem bestimmten Compiler auf ihre Größe hin überprüft werden[2].

Nun folgen die schon erwähnten Präfix-Datentypen, welche mittels einfacher `#define`-Statements auf die dazu passenden generellen Datentypen umbenannt werden. So kommt es später zu keinen Typkonflikten zwischen den Präfix-Datentypen und ihren eigentlich gleichgroßen generellen Datentypen, was in manchen Fällen einen Cast nötig machen würde.

[2]Weitere detailliertere Informationen über die Probleme mit Datentypen bei portablen C Programmen enthält [Hook05].

Zum Abschluss folgen noch die zwei für die JVM wichtigsten Datentypen `slot` und `reference`. `slot` ist ein generischer Datentyp, den die JVM intern verwendet um jegliche Art von Variablen des auszuführenden Programms zu speichern. Auch Parameter, Operanden und vieles mehr werden in Slots gezählt und verwaltet. Ein Slot ist per Definition (siehe [Lindholm99] Kapitel 3.6.1) mindestens so groß, dass er alle Java-Datentypen außer `long` und `double` aufnehmen kann. Bei den heutigen Architekturen sind dies in der Regel 32 Bit (4 Byte). `long`s und `double`s sind 64 Bit (8 Byte) groß und werden grundsätzlich auf zwei Slots abgebildet, auch wenn zum Beispiel auf einer 64 Bit Maschine potenziell ein Slot ausreichen würde. Referenzen besitzen auch die Größe eines Slots, werden in dieser Implementierung aber der Verständlichkeit halber mit einem eigenen Datentypen von regulären Slots unterschieden, sofern im Code durch den Kontext bekannt ist, dass es sich tatsächlich um Referenzen handelt[3].

6.2 Vom Java Class File Format zur Laufzeitstruktur – Klassen

Da die Repräsentation von Klassen im Java Class File Format (siehe Kapitel 2.4) auch zur Laufzeit schon einer ziemlich geeigneten Struktur entspricht, wurde es als Grundlage für die Laufzeitstrukturen der JVM Implementierung genutzt. Lediglich nach dem Laden überflüssige Informationen werden weggelassen, symbolische Referenzen, wenn möglich und sinnvoll, aufgelöst und Arrays mit zur Kompilationszeit unbekannter Größe beim Laden dynamisch erzeugt und verzeigert[4]. Abbildung 6.2 zeigt die überarbeitete Hauptstruktur einer Klasse.

[3] Siehe auch die Erläuterungen zur tylosen JVM in Kapitel 5.

[4] In C lässt sich die Größe eines Arrays nur zur Kompilationszeit festlegen, wenn man dies als Bestandteil eines `struct`s verwenden will. Da zu diesem Zeitpunkt in unserem Fall die nötigen Informationen nicht zur Verfügung stehen, muss hier eine etwas umständlichere Lösung über die Reservierung von dynamischem Speicher und dem Speichern eines Zeigers auf diesen Speicher im `struct` realisiert werden.

```
struct Class
{
   /*u4 magic;*/
   /* u2 minor_version; */
   /* u2 major_version; */
   u2 constant_pool_count;
   cp_info** constant_pool; /* index 0 is always empty! */
   u2 access_flags;
   /* u2 this_class; */
   /* u2 super_class; */
   u2 interfaces_count;
   u2* interfaces;
   /* u2 fields_count; */
   /* field_info** fields; */
   u2 methods_count;
   method_info** methods;

   /* additional runtime data */
   u2 class_instance_variable_count;
   variable** class_instance_variable_table;

   u2 class_instance_variable_slot_count;
   int32* class_inctance_variable_slots;

   u2 instance_variable_count;
   variable** instance_variable_table;
   u2 instance_variable_slot_count;

   boolean isInitialized;
   const char* className;
   struct Class* superClass;
   const char* sourceFileName;
}
```

Abbildung 6.2: Repräsentation einer Klasse zur Laufzeit

Die auskommentierten Felder sind solche, die in der Struktur der `.class` Dateien vorhanden waren, hier aber nun nicht mehr benötigt werden. Konkret wurden sie aus den im folgenden erläuterten Gründen überflüssig:

magic, minor_version, major_version
: Diese Informationen werden nur zur Kontrolle beim Laden der Klasse benötigt und werden später nicht mehr gebraucht.

this_class, super_class
: Diese Indizes in den Constant Pool auf einen Eintrag vom Typ `CONSTANT_Class_info`, der die entsprechende Klasse beschreibt (symbolische Referenz), wurden direkt und indirekt ersetzt. Für die aktuelle Klasse (`this_class`) muss das Programm an der aktuellen Stelle schon einen Zugang zu den Klasseninformationen haben, sonst stünden die hier diskutierten Klasseninformationen nicht zur Verfügung. (Ansonsten könnte sich die JVM, wenn gewünscht, die Daten der entsprechenden Klasse einfach über die Method Area besorgen.) Anstatt der symbolischen Referenz auf die Superklasse wurde ein direkter Zeiger (`class*`) auf diese Klasse in der Variable `superClass` gespeichert.

fields_count, fields
: Das `fields`-Array enthält Klassenvariablen und Instanzvariablen ohne diese zu differenzieren. Beim Ladevorgang werden diese Informationen in zwei separate Arrays (`class_instance_variable_table` und `instance_variable_table`) getrennt.

Alle in Abbildung 6.2 unterhalb des Kommentars `additional runtime data` vermerkten Daten wurden den ursprünglichen Informationen der `.class` Datei hinzugefügt. Auch sie werden im folgenden erläutert:

class_instance_variable_*, instance_variable_*
: Wie oben schon erwähnt, wurden diese beiden Tabellen als Ersatz für das `fields`-Array eingeführt.

class_instance_variable_slot_count, class_inctance_variable_slots
: `class_instance_variable_slot_count` definiert die Anzahl der von

den Klassenvariablen dieser Klasse belegten Slots. Wie oben schon erwähnt, belegen alle Java Datentypen bis auf `long` und `double` einen, die übrigen jeweils zwei Slots. Im `class_inctance_variable_slots`-Array werden die Slots für die Klassenvariablen dieser Klasse und damit auch der Inhalt der entsprechenden Klassenvariablen gespeichert.

instance_variable_slot_count

Diese Variable speichert, äquivalent zum Eintrag für Klassenvariablen, die Anzahl der Slots für Instanzvariablen. Der Speicherplatz für diese Variablen befindet sich jedoch nicht in der Klasse, sondern in den entsprechenden Objekten auf dem Heap, welche Instanzen dieser Klasse darstellen. (Die Struktur von Objekten wird in Kapitel 6.5 behandelt.)

isInitialized

Hier vermerkt die JVM ob diese Klasse bereits initialisiert wurde. Dies ist nur einmal nötig, muss aber vor dem ersten Benutzen der Daten einer Klasse aus Java Code heraus durchgeführt worden sein.

```
struct Variable
{
   char* name;
   char* descriptor;
   u2 access_flags;
   uint32 slot_index;
}
```

Abbildung 6.3: Ein Variablen-Eintrag zur Laufzeit

Abbildung 6.3 zeigt die Struktur, die die Einträge der beiden Arrays `class_instance_variable_table` und `instance_variable_table` beinhalten. Auch sie wurde in abgewandelter Form von der ursprünglichen Struktur übernommen, die in einer `.class` Datei die `fields`-Einträge beschreibt. (Hier wurden die Indizes für Namen und Descriptor in den Constant Pool durch direkte Zeiger ersetzt und der Eintrag für den Slot Index hinzugefügt.)

name

Dieser Zeiger zeigt direkt auf den entsprechenden UTF-8 String im Constant Pool, der den Namen der hier beschriebenen Variablen enthält.

descriptor

Auch dies ist wieder ein Zeiger auf einen UTF-8 String im Constant Pool. Dieses Mal beinhaltet er den Descriptor der Variablen. Descriptoren beschreiben in einer textuellen Darstellung einen Datentypen[5].

access_flags

Diese Variable speichert verschiedene Flags. Diese geben unter anderem die Sichtbarkeit der Variablen an[6].

slot_index

Dieser Eintrag beschreibt die Position des Slots dieser Variablen im entsprechenden Daten-Array. Je nachdem, ob es sich hier um eine Klassen- oder Instanzvariable handelt, befindet sich der Eintrag in der oben erwähnten `class_instance_variable_table` oder im Objekt einer Instanz dieser Klasse auf dem Heap.

An den Einträgen des Methoden-Arrays wurden ebenfalls einige Änderungen vorgenommen. (Abbildung 6.4 zeigt die `method_info`-Struktur.) Erneut wurden Indizes durch Zeiger auf den Text im Constant Pool ersetzt (Name und Descriptor) und von den potenziell in der `.class` Datei existierenden Attributen nur diese nicht verworfen, für die es in der aktuellen Version der Implementierung Verwendung gab. Konkret war dies nur das `Code_attribute`, welches als nächstes behandelt wird.

Die Information, wie viele Slots die Parameter dieser Methode bei einem Methodenaufruf belegen, wird beim Laden der Klasse aus der Descriptor-Information gewonnen und in der `parameterSlotCount` Variable gespeichert. Diese Information zu gewinnen ist recht aufwändig, so dass es Methodenaufrufe unnötig ausbremsen würde, ändert sich jedoch zur Laufzeit nicht, weshalb sie problemlos vorab berechnet werden kann.

[5]Für eine Erläuterung von Descriptoren siehe Kapitel 4.3 der JLS ([Lindholm99])
[6]Für eine Tabelle aller Flags siehe Tabelle 4.4 in Kapitel 4.5 von [Lindholm99].

```
struct method_info
{
   u2 access_flags;
   /*u2 name_index;
   u2 descriptor_index;*/
   /*u2 attributes_count;*/
   /*attribute_info** attributes;*/
   Code_attribute* code;

   /* additional runtime info */
   char* name;
   char* descriptor;
   uint8 parameterSlotCount;
}
```

Abbildung 6.4: Struktur der Laufzeitinformationen für eine Methode

```
struct Code_attribute
{
   /*u2 attribute_name_index;*/
   /*u4 attribute_length;*/
   u2 max_stack;
   u2 max_locals;
   u4 code_length;
   u1* code;
   u2 exception_table_length;
   exception_table** exception_table_tab;
   u2 attributes_count;
   attribute_info* attributes;
}
```

Abbildung 6.5: Struktur des `Code_attribute`s einer Methode zur Laufzeit

An der `Code_attribute`-Struktur (siehe Abbildung 6.5) wurden nur minimale Änderungen vorgenommen. So wurden die beiden ersten Einträge ersatzlos gestrichen, da diese nur zur Identifikation des Attributs beim Ladevorgang benötigt werden. Potenzielle Attribute des Code Attributs, in denen eine Reihe von Debugginginformationen[7] zu finden sind, wurden vorerst einmal nicht verworfen, da der Autor davon ausgeht, in diesem Bereich noch Erweiterungen der JVM vorzunehmen. Zur Zeit werden diese Informationen jedoch nicht ausgewertet und könnten somit beim Laden ebenfalls verworfen werden.

6.3 Die Method Area

Die Method Area speichert zur Laufzeit geladene Klassen und stellt somit den globalen Sammelplatz für Klassen innerhalb der JVM dar. Die Implementierung wurde absichtlich so einfach wie möglich gehalten. Praktisch bedeutet dies, dass sie einzig und allein aus einem Array mit Zeigern auf `Class`-Strukturen besteht. Sucht die JVM nach einer als Textbeschreibung (symbolische Referenz für eine Klasse) gegebenen Klasse, dann wird diese Anfrage über Funktionen der Method Area ausgeführt. Mit einer linearen Suche über das Array wird festgestellt, ob die gesuchte Klasse schon geladen worden ist. Wird sie gefunden, dann wird ein Zeiger auf die entsprechende `Class`-Struktur zurückgegeben. Wurde sie nicht gefunden, dann wird versucht, sie zu laden. Konnte sie auch nicht geladen werden, dann stellt dies einen Fehler dar und eine `ClassNotFoundException` wird geworfen[8].

Aufgrund der linearen Suche durch das Array ist die Performance der Suche mit $O(n)$ im worst-case natürlich verbesserungswürdig. Da hier von Natur aus schon nach eindeutigen Bezeichnern gesucht wird, bietet sich der

[7] Hierzu zählen zum Beispiel die Zeileninformationen, also welcher Bytecode zu welcher Zeile im Quellcode gehört, und die Namen der lokalen Variablen, die innerhalb der JVM und nach dem Kompilieren durch den Compiler ansonsten verworfen werden, da die JVM diese intern nur noch durch Slot-Nummern adressiert.

[8] In der aktuellen Version der Implementierung wird dieser Fehler noch nicht als in Java Code fangbare Exception geworfen und stattdessen die Ausführung der JVM mit einer Fehlermeldung abgebrochen. Die Erweiterung der JVM, solche Exceptions auch korrekt zu werfen, ist jedoch geplant.

Gebrauch von Hashtabellen oder verschiedenen sortierten Baumstrukturen an, die die Suche nach einer Klasse bis auf eine amortisierte Laufzeit von $O(1)$ verbessern können.

6.4 Stack und Stack Frames

Der Stack wurde, wie in Kapitel 3.1 beschrieben, als kombinierter Method- und Operand-Stack mit einer festen, nicht zur Laufzeit erweiterbaren, Größe implementiert. Abbildung 6.6 zeigt die Struktur.

```
struct Stack
{
   StackFrame* currentFrame;
   slot* stackPointer; /* SP */
   uint32 frameCount;
   uint32 maxSize;
   byte* basePointer; /* BP */
}
```

Abbildung 6.6: Die Struktur des kombinierten Method- und Operand Stacks

Der `basePointer`, in der Literatur auch oft nur `BP` genannt, zeigt auf die Basis des Stacks und ändert sich zur Laufzeit nicht. `maxSize` speichert die Größe des Stacks, damit beim Erzeugen eines neues Stack Frames überprüft werden kann, ob das Maximum überschritten wird. `frameCount` hat nur einen informellen Wert und speichert die Anzahl der aktuell auf dem Stack befindlichen Stack Frames. Der Stack Pointer (`stackPointer`, in der Literatur auch nur `SP` genannt) zeigt auf das oberste Ende des Stacks. Wird ein neuer Operand auf den Stack geschoben, dann landet dieser exakt an der Position, auf die der Stack Pointer zeigt. Sollte ein neuer Stack Frame angelegt werden, dann beginnt dessen Struktur ebenfalls an der aktuellen Stack Pointer Position. `currentFrame` speichert einen Zeiger auf den zur Zeit obersten Stack Frame des Stacks. Aufgrund des `prevStackFrame` Zeigers der `StackFrame`-Struktur kann so jeder Stack Frame des Stacks erreicht werden.

```
struct StackFrame
{
   struct StackFrame* prevStackFrame; /* previous stack frame */
   Class* currentClass;
   method_info* methodInfo;
   byte* pc; /* program counter storage */
}
```

Abbildung 6.7: Struktur eines Stack Frames

Abbildung 6.7 zeigt die Struktur eines solchen Stack Frames. `prevStackFrame` zeigt, wie gerade schon erwähnt, auf den vorherigen Stack Frame auf dem Stack. `currentClass` verweist auf die „aktuelle Klasse“ dieses Stack Frames, und `methodInfo` zeigt auf diese Methode. Die `pc`-Variable dieser Struktur wird nur verwendet, wenn von dieser Methode aus weitere Methoden aufgerufen werden. Der Program Counter wird dann für die Dauer der Aufrufe in dieser Variable zwischengespeichert. Wird der Code dieser Methode ausgeführt, dann residiert er aus Geschwindigkeitsgründen in einer lokalen Variable direkt in der zentralen Interpreter-Funktion, von der er durch optimierende Compiler auch gleich in ein Prozessorregister verlagert werden kann. Dies ist wichtig, weil mit der Ausführung jedes Bytecodes der Program Counter mindestens einmal inkrementiert und damit sehr häufig verändert werden muss.

Hinter der `StackFrame`-Struktur folgen die Slots der Parameter der Methode in der Reihenfolge wie sie in Java deklariert sind. (Von links nach rechts. Das ist die Reihenfolge, in der sie vorher auch auf den Operand Stack geschoben worden sind.) Dann folgen die Slots der lokalen Variablen und dann wiederum die Operanden, die während des Ausführens des Codes der Methode auf dem Stack liegen können. Die Positionen der unterschiedlichen Teilbereiche berechnen sich allesamt relativ zur `StackFrame`-Struktur, auf welche beim obersten (aktuellen) Stack Frame durch den `currentFrame`-Zeiger der `Stack`-Struktur gezeigt wird. Bei tiefer im Stack liegenden Stack Frames zeigt der im darüber liegenden Stack Frame befindliche `prevStackFrame`-Zeiger den Weg.

Rein logisch entspricht die `StackFrame`-Struktur zusammen mit den Parametern, den lokalen Variablen und der maximalen Anzahl der Operanden erst dem gesamten Stack Frame, so wie er in Kapitel 3.1 beschrieben wurde. Bei der `StackFrame`-Struktur kann man also in diesem Fall nur von einem Stack Frame Sockel sprechen. Die übrigen Bereiche in die `StackFrame`-Struktur aufzunehmen wäre in C nicht praktikabel gewesen, da die Größe einer Struktur zum Zeitpunkt des Kompilierens festgelegt werden muss, diese aber pro Stack Frame unterschiedlich ausfallen kann.

Abbildung 6.8 zeigt dies in einer schematischen Darstellung. Die Angaben links neben dem Stack zeigen die Berechnung der Abstände der einzelnen Einheiten auf dem Stack jeweils relativ zu ihren Vorgängern.

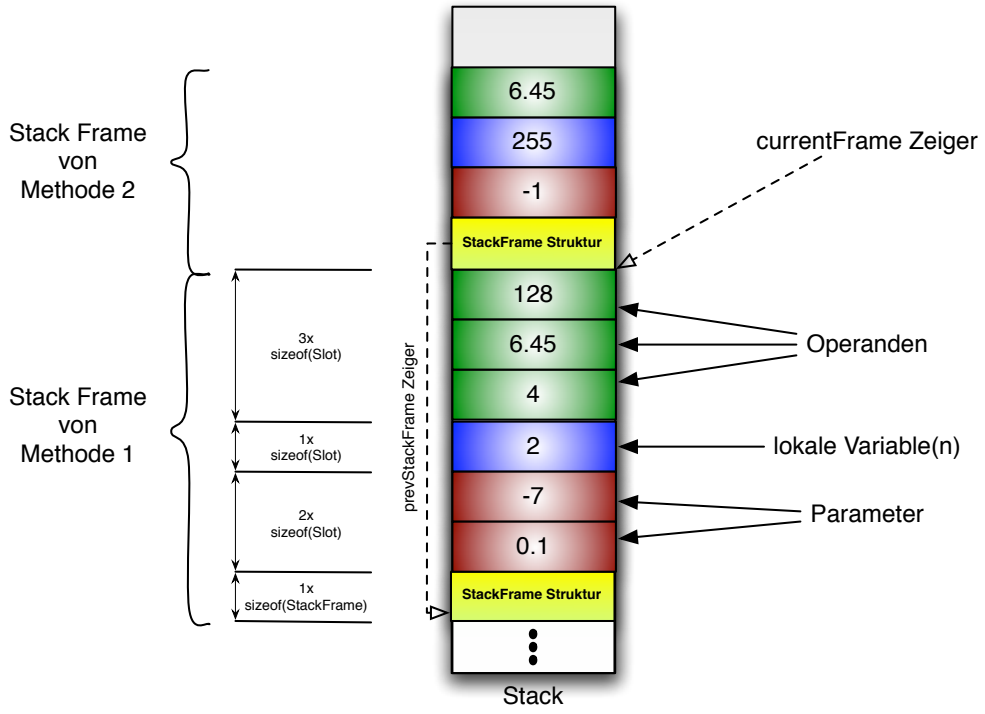


Abbildung 6.8: Schematische Darstellung der Stack-Implementierung

6.5 Heap, Objekte und Arrays

Der Heap wurde, wie in Kapitel 3.2 beschrieben, als referenzierter Heap, also als indirekter Heap mit einer Referenz-Tabelle ausgelegt. Diese besteht aus einem einfachen Array mit Zeigern auf `Object`-Strukturen. Der erste Eintrag (Index 0) wird nicht vergeben, da die Referenz mit dem Wert 0 als `null`-Referenz gewertet wird. Eine Besonderheit dieser Tabelle ist, dass sie bei Bedarf automatisch vergrößert wird. Wurde also der letzte freie Eintrag vergeben, dann wird bei der nächsten Anfrage die Länge des Arrays verdoppelt.

Die im Heap verwalteten Speicherblöcke liefert ein separates Speichermanagement-Modul, welches zur Zeit lediglich die Speicherverwaltungsfunktionen der C-Bibliothek (`malloc`, etc.) kapselt. Später kann dieses System aufgrund der Kapselung problemlos durch ein effizienteres und schnelleres ausgetauscht werden, welches sich beispielsweise auch um die Probleme der Fragmentierung kümmern kann. Aufgrund der Einfachheit wurde hier aber prinzipiell auf eine komplexe Implementierung verzichtet, da die Frage einer optimalen Speicherverwaltung nicht zur Kernthematik virtueller Maschinen gehört.

```
struct Object
{
   Class* cls;
   reference superInstance;
   /* boolean gcMarker; */
}
```

Abbildung 6.9: Struktur eines Stack Frames

Abbildung 6.9 zeigt die `Object`-Struktur, welche lediglich aus zwei Einträgen besteht. Nummer eins ist der Zeiger auf die Informationen der Klasse, wessen Instanz hier erzeugt worden ist (`cls`), und Nummer zwei ist eine Referenz auf die Instanz der Superklasse (`superInstance`), welche belegt ist, sofern die Instanz eine Super-Instanz hat. Was bei allen Klassen, außer bei der Java Basisklasse `Object` der Fall ist. Für die zukünftige Erweiterung ist

hier bereits eine bool'sche Variable für den Garbage Collection Algorithmus vorgesehen, welche aber noch nicht verwendet wird und deshalb noch auskommentiert ist.

Des Weiteren sind Objekte, wie schon die Stack Frames beim Stack, so aufgebaut, dass die eigentlichen Daten nicht in der `Object`-Struktur gespeichert sind, sondern dahinter. (Abbildung 6.10 zeigt dies grafisch.) So befinden sich hinter der `Object`-Struktur einfach eine Anzahl von Slots, auf welche die Instanzvariablen der hier instanziierten Klasse abgebildet sind. Die Zuordnung welche Variable in welchem Slot (oder bei `long`s und `double`s in welchen Slots) gespeichert wird, wurde schon beim Laden der Klasse festgelegt und in den `Variable`-Einträgen der Instanzvariablen-Tabelle der entsprechenden Klasse festgehalten. Darum benötigt die JVM im Objekt auch einen Zeiger auf die Klasseninformationen, damit sie Dinge wie die Position einer Variablen dort nachschlagen kann.

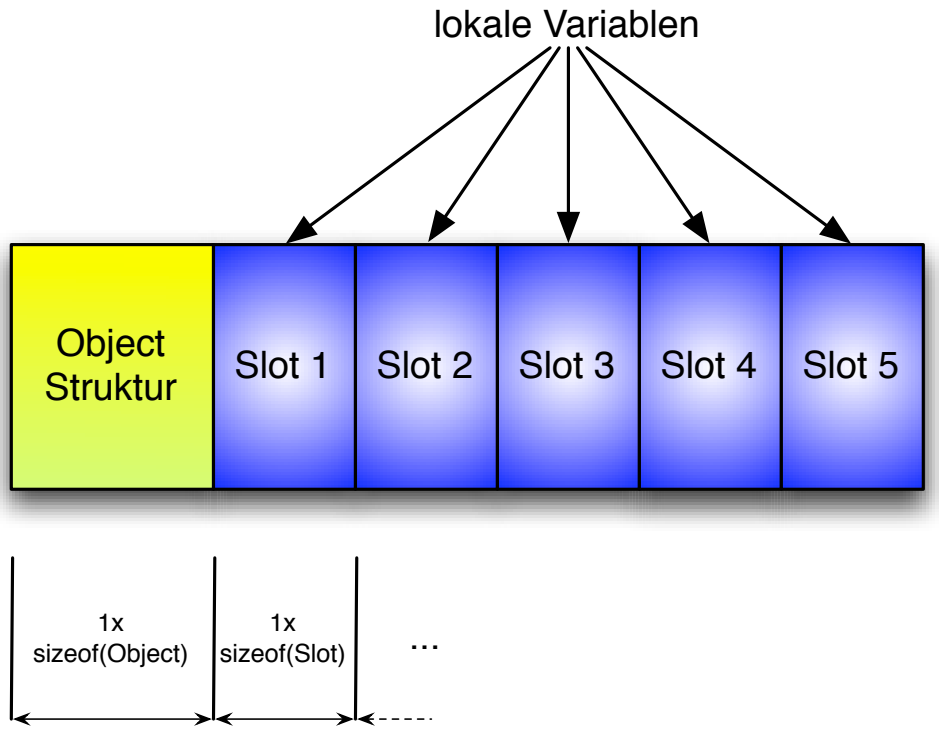


Abbildung 6.10: Schematische Darstellung eines Objekts auf dem Heap

Von der Verwaltung her wird auf dem Heap nicht zwischen normalen Objekten und Arrays unterschieden. Auch sie besitzen eine `Object`-Struktur, haben einen Zeiger auf ihre Klasseninformationen[9] und die Referenz auf die Instanz der Superklasse. Nur ist diese bei Arrays immer vom Typ `Object`, da man von Arrays nicht ableiten kann.

Abbildung 6.11 zeigt die schematische Darstellung eines Arrays auf dem Heap. Im Vergleich zu normalen Objekten (Abbildung 6.10) fällt auf, dass sich gleich hinter der `Object`-Struktur ein weiteres, festes Element befindet, und zwar die Länge des Arrays. Diese wurde nicht mit in die `Object`-Struktur integriert, weil sie bei normalen Objekten unbenutzt bliebe, und somit bei vielen der Objekte auf dem Heap Speicherplatz dafür verschwendet würde, obwohl diese gar keine Arrays darstellen und deshalb diese Variable gar nicht benötigen.

Hinter der Längenangabe folgen die einzelnen Elemente des Arrays, wobei die Größe der einzelnen Elemente hier vom Typ des Arrays abhängig ist. Elemente von Byte-Arrays (`byte[]`) haben eine Größe von einem Byte, Elemente von Char-Arrays (`char[]`) zwei Byte, und so weiter. Referenz-Arrays haben die (übliche) Elementgröße eines Slots, lediglich auf Boolean-Arrays trifft eine Ausnahme zu. Prinzipiell gibt es innerhalb der JVM nämlich gar keine Bool'schen Variablen. Der Compiler bildet diese bereits beim Kompilieren auf Integer (`int`s) ab. Tatsächlich erlaubt der Befehlssatz der JVM es aber Boolean-Arrays zu erzeugen, wobei auf diese jedoch mittels der Opcodes für Byte-Arrays zugegriffen wird, weshalb Boolean-Arrays in der Regel JVM-intern als Byte-Arrays realisiert werden. Auch hier ist dies so realisiert.

[9]Die Klasseninformationen der Arrays existieren nicht als `.class` Dateien, sondern werden zur Laufzeit von der JVM und je nach Bedarf erzeugt und dann aber wie jede andere Klasse auch in der Method Area abgelegt. Es gibt verschiedene Tyen von Array-Klassen. Eine für jeden Basisdatentyp, eine für Referenzen und jeweils eine für mehrdimensionale Arrays. Weitere Informationen hierzu findet man in Kapitel 2.15 in der JLS ([Lindholm99]).

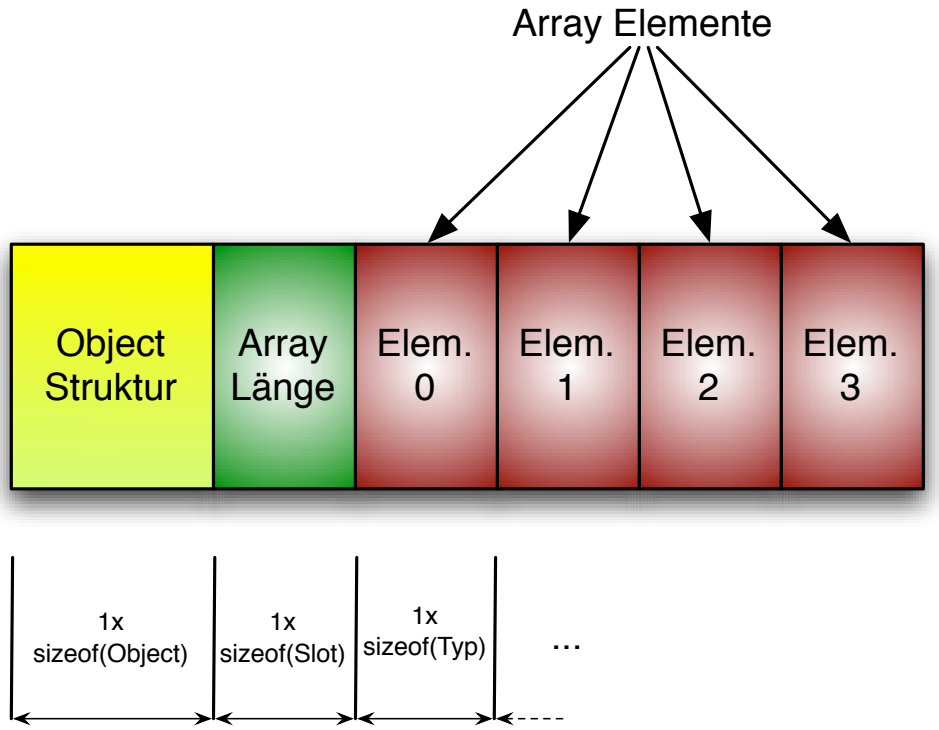


Abbildung 6.11: Schematische Darstellung eines Arrays auf dem Heap

Kapitel 7

Nutzung der Laufzeitstrukturen

Ging es in den bisherigen Kapiteln eher um Theorie und Strukturen einer JVM, so wird diese in diesem Kapitel nun aktiv. Anhand verschiedener Java-Beispiele werden einige Grundprinzipien der Ausführung von Code in der JVM erklärt. Behandelt werden grundlegende Dinge wie das Nutzen symbolischer Referenzen und das Laden, Linken und Initialisieren von Klassen und geht bis zur Verarbeitung von Variablen und dem Aufruf von Methoden.

Die Beispiele sind im Java Code absichtlich sehr kurz gehalten[1]. Das Problem dabei ist, dass schon das Ausführen einzelner Bytecodes eine Menge Arbeit innerhalb der JVM bedeuten kann und sogar dann, wenn nur die wesentlichen Tätigkeiten im Detail besprochen werden, nur gekürzt gezeigt werden kann, weil es sonst den Umfang dieser Arbeit sprengen würde.

Um die Einzelschritte besser erläutern zu können wird passend zum Java Code das Disassemblat des Teils der Klasse, also die einzelnen Instruktionen die die JVM ausführen wird, gezeigt. Grundsätzlich sind die Opcodes der JVM ohne Vorkenntnis schon recht gut zu verstehen. Der geneigte Leser kann die Details der verschiedenen Opcodes jedoch in der JLS ([Lindholm99]) in Kapitel 6 nachschlagen. Die Disassemblate von Klassen können jeder Zeit

[1] Aufgrund der nötigen Kürze der Beispielprogramme ist der Sinn oft fraglich bis unsinnig. Dies bittet der Autor zu ignorieren.

mit dem jedem Java SDK beiliegenden `javap` Kommandozeilentool selber erzeugt werden[2].

7.1 Das Auflösen symbolischer Referenzen – Ein Beispiel

Wie schon in Kapitel 2.5 erwähnt, ist eine der Hauptaufgaben der JVM das Auflösen symbolischer Referenzen. Wie häufig dies geschieht kann man an folgendem Beispiel sehen. Abbildung 7.1 zeigt ein äußerst kurzes Beispielprogramm, in welchem nicht mehr getan wird, als in der Variablen `s` die Referenz des konstanten Strings `Hallo Welt!` abzulegen.

```
public class SymbolicReference
{
   public static void main( String[] args )
   {
      String s= "Hallo Welt!";
   }
}
```

Abbildung 7.1: Java Beispielcode für das Auflösen symbolischer Referenzen

Schaut man sich nun das Disassemblat der Klasse (in wesentlichen Teilen) an, dann sind schon einige Informationen zusammengekommen. Abbildung 7.2 zeigt es.

Man sieht, dass die Ausgabe mit einigen allgemeinen Informationen wie der Klassendefinition, der Quelldatei und den Versionsnummern beginnt. Weiter geht es mit der Auflistung des Constant Pools der Beispiel-Klasse. Im Vergleich zur ursprünglichen Anwendung stehen hier sehr viel mehr Informationen als man annehmen würde. Aber jede auch nur irgendwie verwertbare konstante Information wird einzeln hier abgelegt. Und auch einige

[2]Der Autor empfiehlt jedem interessierten Leser mit dem `javap` Tool einmal eine Zeit lang zu „spielen“. (Optionen '-c' und '-verbose' benutzen!) Es ist erstaunlich, was für Konstrukte der Compiler an manchen Stellen erzeugt. Auch kann man so zum Beispiel wunderbar Performance-Engpässe im Bereich der String-Verarbeitung bei falscher Handhabung erkennen.

```
Daniel$ javap -c -verbose SymbolicReference
Compiled from "SymbolicReference.java"
public class SymbolicReference extends java.lang.Object
  SourceFile: "SymbolicReference.java"
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Method       #4.#13; //  java/lang/Object."<init>":()V
const #2 = String       #14;    //  Hallo Welt!
const #3 = class        #15;    //  SymbolicReference
const #4 = class        #16;    //  java/lang/Object
const #5 = Asciz        <init>;
const #6 = Asciz        ()V;
const #7 = Asciz        Code;
const #8 = Asciz        LineNumberTable;
const #9 = Asciz        main;
const #10 = Asciz       ([Ljava/lang/String;)V;
const #11 = Asciz       SourceFile;
const #12 = Asciz       SymbolicReference.java;
const #13 = NameAndType #5:#6;//  "<init>":()V
const #14 = Asciz       Hallo Welt!;
const #15 = Asciz       SymbolicReference;
const #16 = Asciz       java/lang/Object;

{
[...]

public static void main(java.lang.String[]);
  Code:
   Stack=1, Locals=2, Args_size=1
   0:   ldc     #2; //String Hallo Welt!
   2:   astore_1
   3:   return
}
```

Abbildung 7.2: Disassemblat des Quellcodes aus Abbildung 7.1

symbolische Referenzen kann man schon mit bloßem Auge erkennen. An all den Stellen, an denen man auf der rechten Seite des Gleichheitszeichens eine Zahl mit einer vorangestellten Raute (#) sehen kann, befindet sich eine symbolische Referenz, die auf einen anderen Eintrag innerhalb des selben Constant Pools verweist.

Betrachtet man beispielsweise Eintrag Nummer 4, dann sieht man die Information eines `CONSTANT_Class_info`-Eintrages dargestellt. Er beschreibt eine Klasse und die einzige hier relevante Information ist der Name der Klasse. Dieser befindet sich in Eintrag 16 des Constant Pools und lautet „java/lang/Object“. Der Verweis auf Eintrag Nummer 16 stellt eine symbolische Referenz dar. Damit aber noch nicht genug, denn auch der voll qualifizierte Name der Klasse, der hiermit nun bekannt ist, stellt wieder eine symbolische Referenz dar, die bei Bedarf weiter aufgelöst werden kann.

```
[...]
Executing method SymbolicReference.main([Ljava/lang/String;)V...
Executing LDC
Loading class [C
        Creating new String instance with text "Hallo Welt!",
        reference is 4.
        Pushing value 4 from constant pool index 2.
Executing ASTORE_1
        Popping reference 4 from the stack, storing it to slot 1.
Executing RETURN
[...]
```

Abbildung 7.3: Debuginformationen beim Ausführen des Beispiels aus Abbildung 7.1

Abbildung 7.3 zeigt wie das Beispielprogramm in der virtuellen Maschine ausgeführt wird. Aus Gründen der Übersichtlichkeit wurden alle für das Beispiel irrelevanten Informationen, wie etwa das Laden der Klasse, entfernt. Entsprechend beginnt das Listing mit dem Ausführen der `main()`-Methode. Der `LDC`-Opcode wird ausgeführt, was so viel bedeutet wie „Lade Konstante“. Wenn es in Java um Konstanten geht, dann dreht es sich in der Regel um den Constant Pool. Dies ist auch hier der Fall, denn Eintrag

Nummer 2 des Constant Pools soll geladen werden. Dieser jedoch stellt eine `CONSTANT_String_info`-Struktur dar und enthält weder ein String-Objekt, noch die entsprechende Zeichenkette[3] selber. Stattdessen findet man lediglich eine symbolische Referenz auf Constant Pool Eintrag Nummer 14, welche die eigentliche Zeichenkette enthält.

Per Definition muss nun die JVM eine Instanz der Klasse `String` mit dem gegebenen Text erzeugen, um eine Referenz auf diese für die weitere Verwendung auf den Stack schieben zu können. Dies wird mit einem kleinen Umweg nun getan, denn zuerst muss die Zeichenkette in ein `char`-Array gepackt werden. Um dies tun zu können, wird eine Klasse vom Typ `[C`, was einem Char-Array entspricht, erzeugt, in der Method Area abgelegt und dann benutzt. (Auch hier werden wieder symbolische Referenzen, zum Beispiel zum Auffinden der Klasse in der Method Area, verwendet.) Dann wird eine Char-Array Instanz erzeugt und die Zeichenkette hinein kopiert. Nun wird eine Instanz von `String` erzeugt (welche in diesem Beispiel schon geladen wurde) und dem Konstruktor das soeben erzeugte Char-Array übergeben. Das resultierende String-Objekt (die Referenz-Nummer lautet 4) wird nun für zukünftige Zugriffe im `CONSTANT_String_info` Eintrag (Nummer 2) des Constant Pools gespeichert und anschließend die Referenz auch auf den Stack geschoben. Es folgt noch das Ablegen der Referenz in der lokalen Variable Nummer 1 und das `RETURN`, welches das Ende des Beispiels darstellt.

Durch das Speichern der Referenz des String-Objekts im `CONSTANT_String_info` Eintrag steht nun bei folgenden Zugriffen auf die Text-Konstante eine direkt verwendbare Referenz zu dem entsprechenden `String`-Objekt zur Verfügung, was zukünftige Zugriffe deutlich erleichtert und das Auflösen der entsprechenden symbolischen Referenzen spart. Bei durchgehender Anwendung solcher Optimierungen lässt sich das Auflösen vieler symbolischer Referenzen sparen. Der erste Zugriff auf eine Konstante oder das erste Aufrufen einer statischen Methode ist dann zwar langsam, da die symbolischen

[3]Zeichenketten werden auch im Deutschen gerne als Strings bezeichnet, was der Autor bisher in der Regel auch getan hat. An der dieser Stelle jedoch besteht Verwechslungsgefahr zwischen der Klasse `String` und ihren Instanzen und Zeichenketten, weshalb der Autor hier den Begriff Zeichenkette verwendet wenn er eine Reihe von Buchstaben im Speicher meint.

Referenzen aufgelöst und echte Referenzen gespeichert werden müssen, jedoch amortisiert sich dieser Aufwand mit der Zeit recht schnell, wenn anstatt der symbolischen Referenzen immer gleich auf eine direkte Referenz zurückgegriffen werden kann.

Solche Optimierungen lassen sich zwar nur bei Konstanten vornehmen, wie man später beim Aufruf von Methoden noch sehen wird, zum Glück sind die meisten Informationen innerhalb der JVM aber unveränderlich. Sie wollen lediglich zur Laufzeit verbunden werden. Und gerade dieses Konzept, Komponenten erst zur Laufzeit zusammenzufügen, ist es, was Java und die Java Virtual Machine so flexibel macht. (Stichwort: Späte Bindung.)

7.2 Statische Klassenvariablen

In diesem Abschnitt soll das Laden und Speichern von statischen Klassenvariablen gezeigt werden. Abbildung 7.4 zeigt ein einfaches Java Beispiel. Es wird lediglich eine statische Klassenvariable `myVar` deklariert, diese wird implizit vom System mit 0 initialisiert und anschließend um eins erhöht.

```
public class StaticVars
{
   private static int myVar;

   public static void main(String[] args)
   {
      myVar++;
   }
}
```

Abbildung 7.4: Beispiel der Verwendung einer statischen Klassenvariable

Abbildung 7.5 zeigt das Disassemblat dieses Beispiels. In der `main`-Methode erkennt man gut die beiden `getstatic` und `putstatic` Befehle, welche benutzt werden um auf eine statische Klassenvariable zuzugreifen. Abbildung 7.6 zeigt wie das Beispiel von der JVM ausgeführt wird.

```
Daniel$ javap -c -verbose StaticVars
Compiled from "StaticVars.java"
public class StaticVars extends java.lang.Object
  SourceFile: "StaticVars.java"
  minor version: 0
  major version: 49
  Constant pool:
const #1 = Method       #4.#15; //  java/lang/Object."<init>":()V
const #2 = Field        #3.#16; //  StaticVars.myVar:I
const #3 = class        #17;    //  StaticVars
const #4 = class        #18;    //  java/lang/Object
const #5 = Asciz        myVar;
const #6 = Asciz        I;
const #7 = Asciz        <init>;
const #8 = Asciz        ()V;
const #9 = Asciz        Code;
const #10 = Asciz       LineNumberTable;
const #11 = Asciz       main;
const #12 = Asciz       ([Ljava/lang/String;)V;
const #13 = Asciz       SourceFile;
const #14 = Asciz       StaticVars.java;
const #15 = NameAndType #7:#8;//  "<init>":()V
const #16 = NameAndType #5:#6;//  myVar:I
const #17 = Asciz       StaticVars;
const #18 = Asciz       java/lang/Object;

{
[...]

public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=1, Args_size=1
   0:   getstatic       #2; //Field myVar:I
   3:   iconst_1
   4:   iadd
   5:   putstatic       #2; //Field myVar:I
   8:   return
}
```

Abbildung 7.5: Disassemblat des in Abbildung 7.4 gezeigten Beispiels

```
Daniel$ pura StaticVars
Pura Experimental Java Virtual Machine v0.13 - (c) 2007 Daniel Klein
[...]
Executing method StaticVars.main([Ljava/lang/String;)V...
Executing GETSTATIC
        The value is 0, the slot number 0.
        Getting static field StaticVars.myVar (type is I).
Executing ICONST_1
Executing IADD
        Adding 0 and 1, result is 1.
Executing PUTSTATIC
        The value is 1, the slot number 0.
        Putting into static field StaticVars.myVar (type is I).
Executing RETURN
        Popping stack frame off the stack. Back at frame 1,
        height 16 bytes.
Freeing up stack.

Execution finished. 4 Bytecodes executed.
```

Abbildung 7.6: Beispiel der Verwendung einer statischen Klassenvariable

Um den Inhalt der statischen Variable zu laden wird der Opcode `getstatic` ausgeführt. Wie man in Abbildung 7.5 sehen kann, bekommt der `getstatic`-Opcode den Constant-Pool-Index 2 übergeben, bei dem es sich um eine symbolische Referenz auf die Variable handelt. Der Eintrag des Constant Pools beinhaltet eine `CONSTANT_Field_info`-Struktur, welche wiederum auf die Einträge 3 und 16 verweist. Eintrag 3 beinhaltet eine `CONSTANT_Class_info`-Struktur, welche, wie schon besprochen, auf eine Klasse verweist. In diesem Fall wird auf die Klasse „StaticVars“, also unsere eigene Klasse, verwiesen. Eintrag Nummer 16 beinhaltet eine `CONSTANT_NameAndType_info`-Struktur, welche eine Klassenvariable beschreibt. Die erneuten Verweise an die Constant Pool Einträge 5 und 6 identifizieren die Klassenvariable eindeutig. Eintrag 5 beinhaltet den Namen, Eintrag 6 den Typ der Variablen als Zeichenkette. In unserem Fall lautet der Name `myVar` und der Typ `I`, was für den generischen Datentypen Integer (`int`) steht.

Mit Namen und Typ der Klassenvariablen kann man nun auf die Suche nach der gewünschten Variablen in der Liste der statischen Klassenvariablen gehen. Hierzu wird die beim Laden der Klasse angelegte `class_instance_variable_table`-Tabelle (siehe Kapitel 6.2) durchgesehen. Wurde die Variable gefunden, dann wird auch hier ein Zeiger zum entsprechenden Eintrag in die `class_instance_variable_table`-Tabelle in der `CONSTANT_Field_info`-Struktur in Constant Pool Eintrag Nummer 2 gespeichert, was das erneute Auflösen der symbolischen Referenz bei späteren Zugriffen auf die Variable spart.

Wurde die passende Variable ermittelt, so steht bei statischen Klassenvariablen auch gleich der Wert fest. Die `slot_index`-Variable der `Variable`-Struktur in der `class_instance_variable_table`-Tabelle speichert die Position des Slots, in dem in der `class_inctance_variable_slots`-Tabelle dieser Klasse der Wert der Variablen gespeichert ist.

Der gesuchte Wert 0 wurde erfolgreich gefunden und wird im Beispielprogramm auf den Stack geschoben. Es folgt die Konstante 1 (`ICONST_1`) und dann werden beide mittels Integer-Addition (`IADD`) addiert. Die beiden Operanden werden vom Stack herunter genommen und das Ergebnis 1 wiederum auf den Stack geschoben, welches nun mittels des `PUTSTATIC`-Opcodes

wieder in dieselbe statische Klassenvariable gespeichert wird. Erneut weist ein Index in den Constant Pool Eintrag Nummer 2, welcher die schon bekannte `CONSTANT_Field_info`-Struktur beinhaltet. Da der direkte Zeiger auf den Eintrag der Variablen in der `class_instance_variable_table`-Tabelle schon existiert, müssen keine symbolischen Referenzen mehr aufgelöst werden. Stattdessen kann direkt auf die Variable zugegriffen und der Wert im entsprechenden Slot der `class_inctance_variable_slots`-Tabelle abgelegt werden.

Einen Sonderfall bei den statischen Klassenvariablen stellen übrigens die in Java `static final` deklarierten Variablen dar. Diese werden vom Compiler in der Regel nicht als übliche statische Klassenvariablen genutzt, sondern via „Inlinig“ direkt am Ort der Verwendung, bei `public` deklarierten Variablen auch in anderen Klassen, direkt in den Bytecode-Datenstrom der entsprechenden Methode eingesetzt. Das spart die Zeit zum Nachschlagen des Wertes, kann aber zu Inkonsistenzen über Klassengrenzen hinweg führen. Benutzt man in einer Klasse A nämlich eine `public static final` Variable der Klasse B, und der Wert dieser Variable wird geändert nachdem Klasse A kompiliert worden ist, dann verwendet Klasse A weiterhin den alten, jetzt falschen, Wert der vom Compiler zum Zeitpunkt der Kompilation von Klasse A aus Klasse B kopiert worden ist.

```
public class StaticFinalVar
{
   public static final int WIDTH= 5;

   public static void main(String[] args)
   {
      int a= WIDTH;
   }
}
```

Abbildung 7.7: Beispiel der Verwendung einer `static final` Variable

Abbildung 7.7 zeigt ein vereinfachtes Beispiel mit nur einer Klasse aber demselben Effekt, und Abbildung 7.5 zeigt das Disassemblat dieses Beispiels. Wie man dort erkennen kann, wird in der `main()`-Methode nicht mittels

```
[...]
{
[...]
public static void main(java.lang.String[]);
  Code:
   Stack=1, Locals=2, Args_size=1
   0:   iconst_5
   1:   istore_1
   2:   return
}
```

Abbildung 7.8: Disassemblat des Beispiels aus Abbildung 7.7

`GETSTATIC` auf die Variable zugegriffen, sondern stattdessen gleich die Konstante 5 auf den Stack geschoben (`ICONST_5`), was dem Wert der Variablen entspricht, aber nicht mehr mit ihm zusammenhängt.

7.3 Instanzvariablen

Der Zugriff auf Instanzvariablen, auch Klassenvariablen der Instanz der Klasse genannt, verläuft für die JVM ähnlich wie der von statischen Klassenvariablen. Zur Unterscheidung werden unterschiedliche Opcodes benutzt, `getfield` zum Lesen des Wertes einer Variablen und `putfield` zum Schreiben des Wertes in eine Variable. Das Prozedere, wie der entsprechende Eintrag in der Variablentabelle der Klasse gefunden wird, entspricht dem der statischen Klassenvariablen. Lediglich befinden sich die Beschreibungen der Instanzvariablen in der Tabelle mit dem Namen `instance_variable_table`.

Der wichtigste Unterschied zwischen den beiden Variablen-Arten ist aber die Stelle, wo der Wert der Variablen gespeichert ist. Bei Instanzvariablen ist dieser Speicher nicht innerhalb der Laufzeitinformationen der Klasse zu finden, sondern in den Laufzeitdaten einer Instanz der entsprechenden Klasse auf dem Heap. Und genau eine Referenz auf diese muss den entsprechenden Opcodes auch zusätzlich übergeben werden. (Siehe Kapitel 6.5 für Infos über die Struktur von Objekten auf dem Heap.)

Der Rest funktioniert dann wieder wie bei den statischen Variablen. Der `Variable`-Eintrag der `instance_variable_table`-Tabelle enthält auch hier den Index in ein Array von Slots, und der passende Slot im Speicher der Klasseninstanz auf dem Heap beherbergt dann den gewünschten Wert.

Der Vollständigkeit halber soll auch hier ein kleines Beispiel gezeigt werden. Abbildung 7.9 zeigt das Beispiel aus dem vorigen Abschnitt mit der Verwendung einer Instanzvariablen anstatt einer Klassenvariablen. Um diese nutzen zu können muss vor der Verwendung eine Instanz der Klasse angelegt werden, was letzten Endes dann überhaupt erst den Speicherplatz für die Variable zur Verfügung stellt. Da man Instanzvariablen nicht aus einem statischen Kontext (i.e. einer `static` deklarierten Methode) heraus nutzen kann, wurde hier eine Methode `test()` hinzugefügt, welche in diesem Fall den eigentlichen Testcode beinhaltet. Wie üblich ist auch diese Variable automatisch mit 0 vorinitialisiert worden und wird einfach um eins erhöht, was das Laden und das Speichern der Variable nötig macht.

```
public class InstanceVar
{
   private int testVar;

   private void test()
   {
      testVar++;
   }

   public static void main(String[] args)
   {
      new InstanceVar().test();
   }
}
```

Abbildung 7.9: Beispiel der Verwendung einer Instanz-Klassenvariable

Abbildung 7.10 zeigt einen Teil des Disassemblats, wobei das Erzeugen der Klasseninstanz und das Aufrufen der Methode der Übersichtlichkeit halber außen vor gelassen wurden. (Methodenaufrufe werden jedoch in den folgenden Abschnitten noch diskutiert.)

```
[...]
private void test();
  Code:
   Stack=3, Locals=1, Args_size=1
   0:   aload_0
   1:   dup
   2:   getfield        #2; //Field testVar:I
   5:   iconst_1
   6:   iadd
   7:   putfield        #2; //Field testVar:I
   10:  return
[...]
```

Abbildung 7.10: Teil des Disassemblats des Beispiels aus Abbildung 7.9

Wie man an der angegebenen Anzahl der Argumente (`Args_size=1`) sehen kann, hat die von uns parameterlos definierte Methode vom Compiler einen Parameter bekommen. Dieser beinhaltet die `this` Referenz, also den Verweis auf die aktuell zu verwendende Instanz, durch die die Methode weiß mit welchem Satz von Instanzvariablen sie nun arbeiten soll. Beim Aufruf von nicht statischen Methoden übergibt die JVM die `this`-Referenz grundsätzlich als ersten Parameter, so dass dieser innerhalb der Methode auch immer als Wert in Slot 0 zu finden ist.

In diesem Sinne wird mit dem ersten Opcode der `test()`-Methode die `this`-Referenz auf den Stack geschoben und mit dem `dup`-Opcode gleich noch eine weitere Kopie („Duplikat“) auf dem Stack erzeugt. Nun folgt der Aufruf von `getfield`, der hier nicht noch einmal im Detail behandelt werden soll. Einer der beiden `this`-Referenzen wurde im Laufe der Abarbeitung des `getfield`-Opcodes vom Stack genommen, dafür aber der aktuelle Wert 0 auf den Stack geschoben. Es folgt die Konstante 1 (`iconst_1`), welche auch auf den Stack geschoben und dann die letzten beiden Werte, also 0 und 1, per Integer-Addition (`iadd`) addiert werden. Hierfür nimmt der `iadd`-Opcode erneut die entsprechenden Werte vom Stack, addiert diese, und schiebt das Ergebnis zurück auf den Stack.

Es liegt nun weiterhin eine `this`-Referenz, sowie das Ergebnis der Addition auf dem Stack und beide Informationen werden als Parameter für den `putfield`-Opcode verwendet um den Wert 1 (Ergebnis der Addition) wieder in die Instanzvariable zu schreiben.

7.4 Lokale Variablen

Um alle möglichen Arten von Variablen behandelt zu haben fehlen noch die lokalen Variablen. Der Zugriff auf diese ist denkbar einfach. Wie in Kapitel 3.1 und 6.4 behandelt, werden lokale Variablen auf dem Stack abgelegt und über einen Index adressiert. In Abbildung 7.10 sieht man zum Beispiel den Zugriff auf die lokale Variable mit dem Index 0, in der in diesem Beispiel die erwähnte `this`-Referenz gespeichert ist. Genau so verhält sich dies auch bei den übrigen lokalen Variablen.

Als Besonderheit zu erwähnen ist, dass es für den Zugriff auf die ersten vier lokalen Variablen dedizierte Opcodes (abhängig vom Typ zum Beispiel `iload_0` bis `iload_3`) gibt, wobei die übrigen lokalen Variablen ein wenig aufwändiger nur über einen generischen `iload`-Opcode adressiert werden können, wobei der Index der gewünschten Variablen im Bytecode-Datenstrom gleich hinter dem Opcode folgen muss[4]. Werden in einer Methode verschiedene lokale Variablen also sehr viel häufiger als andere verwendet, dann könnte es sinnvoll sein diese bei der Implementierung einer Methode als Erstes zu deklarieren. Dramatische Unterschiede kommen hierdurch jedoch nicht zu Stande[5].

[4]Für weitere Details hierzu siehe zum Beispiel JLS ([Lindholm99]), Seiten 274 und 275.

[5]Grundsätzlich sei auch hier davor gewarnt solche oft unnötigen Optimierungen im Code durchzuführen. Erstens verschlechtert sich somit die Lesbarkeit des Codes, und zweitens kann man sich nie sicher sein ob und wie ein Compiler die in Java existierenden Variablen überhaupt auf die Slots der lokalen Variablen auf dem Stack abbildet. (Das Verhalten jedes Java Compilers müsste neu daraufhin überprüft werden.) Theoretisch steht ihm, und auch dem Laufzeitsystem (zum Beispiel Suns Hotspot Compiler), nämlich nichts im Weg, eigene Optimierungen durchzuführen.

7.5 Statischer Methodenaufruf

In den folgenden zwei Abschnitten geht es abschließend um das Ausführen von Methoden. Diese auszuführen ist nicht gerade trivial, muss die JVM doch einen ziemlichen Aufwand treiben um einen Methodenaufruf erfolgreich vor- und nachzubereiten. Nichts desto trotz wäre die Programmierung von Computern ohne die Verwendung einer Art von Unterprogrammen (Methoden, Funktionen, Prozeduren) nicht das was es heute ist. Man würde ein gehöriges Maß an Bequemlichkeit verlieren und förmlich in die Steinzeit der Programmierung auf Maschinenebene zurückfallen.

Abbildung 7.11 zeigt ein minimalistisches Beispiel für die einfachste Art der Methodenaufrufe in Java, den Aufruf einer statischen Methode. Auch das Disassemblat des Beispiels (Abbildung 7.12) macht nicht gerade einen komplexen Eindruck. Aber wie schon beim Zugriff auf Klassenvariablen steckt der Teufel wie so üblich im Detail.

```
public class StaticMethod
{
   private static void test()
   {
      int a= 99;
   }

   public static void main(String[] args)
   {
      test();
   }
}
```

Abbildung 7.11: Aufruf einer statischen Methode

Betrachtet man die Implementierung der `main()`-Methode in Abbildung 7.12, dann sieht man den `invokestatic`-Opcode, der den Aufruf einer Methode einleitet. Mit diesem Opcode wird wie üblich ein Index in den Constant Pool der aktuellen Klasse, in diesem Fall Nummer 2, übergeben. Constant Pool Eintrag Nummer 2 enthält eine `CONSTANT_Method_info`-Struktur, wel-

```
Daniel$ javap -c -verbose -private StaticMethod
[...]
  Constant pool:
const #1 = Method       #4.#14; //  java/lang/Object."<init>":()V
const #2 = Method       #3.#15; //  StaticMethod.test:()V
const #3 = class        #16;    //  StaticMethod
const #4 = class        #17;    //  java/lang/Object
const #5 = Asciz        <init>;
const #6 = Asciz        ()V;
const #7 = Asciz        Code;
const #8 = Asciz        LineNumberTable;
const #9 = Asciz        test;
const #10 = Asciz       main;
const #11 = Asciz       ([Ljava/lang/String;)V;
const #12 = Asciz       SourceFile;
const #13 = Asciz       StaticMethod.java;
const #14 = NameAndType #5:#6;//  "<init>":()V
const #15 = NameAndType #9:#6;//  test:()V
const #16 = Asciz       StaticMethod;
const #17 = Asciz       java/lang/Object;

{
[...]
private static void test();
  Code:
   Stack=1, Locals=1, Args_size=0
   0:   bipush  99
   2:   istore_0
   3:   return

public static void main(java.lang.String[]);
  Code:
   Stack=0, Locals=1, Args_size=1
   0:   invokestatic    #2; //Method test:()V
   3:   return
}
```

Abbildung 7.12: Disassemblats des Beispiels aus Abbildung 7.11

che die aufzurufende Methode beschreibt. Diese Beschreibung besteht aus zwei Angaben, welche erneut als symbolische Referenzen mittels Indizes auf weitere Constant Pool Einträge gespeichert sind. Konkret handelt es sich um die Einträge Nummer 3 und 15. Eintrag 3 beschreibt in der gewohnten Art und Weise die Klasse, in der die Methode zu finden ist, Eintrag 15 enthält die ebenfalls schon bekannte `CONSTANT_NameAndType_info`-Struktur. Diese verweist ihrerseits wiederum auf zwei Einträge im Constant Pool, welche den Namen und (durch den Namen der Struktur hier nicht ganz eindeutig) den Descriptor der aufzurufenden Methode als Zeichenkette enthält. (Einträge 9 und 6.)

Mit Hilfe der Beschreibung der Klasse wird nun von der Method Area die gewünschte Klasse angefordert. Konnte diese erfolgreich abgerufen werden, wird deren `methods`-Tabelle linear nach der gewünschten Methode durchsucht. (Sollte sie dort nicht gefunden werden, dann wird eine MethodNotFoundException ausgelöst[6].)

Wurde die Methode gefunden, dann wird im Constant Pool Eintrag Nummer 2 ein Zeiger auf die entsprechende `method_info`-Struktur in der `methods`-Tabelle eingetragen, was das zukünftige Suchen nach der Methode erübrigt. Deshalb lohnt es sich auch nicht weiter die Suche in der Methoden-Liste effizienter zu implementieren. Der initialen Suche mit einer Laufzeit von $O(n)$ folgen nur noch Aufrufe mit einer Laufzeit $O(1)$. Die Dauer der initialen Suche amortisiert sich mit zunehmender Laufzeit zunehmends.

Als nächstes wird der Program Counter (PC) der aufrufenden Methode in ihrem Stack Frame gesichert und die Parameter des Methodenaufrufs vom Stack genommen und zwischengespeichert[7]. Dann wird ein neuer Stack

[6]Wie schon erwähnt, werden alle von der JVM erzeugten Fehler bisher noch nicht so implementiert, dass diese wirklich Exceptions werfen die in Java Code abgefangen werden können. Stattdessen wird momentan direkt von der JVM eine Fehlermeldung ausgegeben und die Ausführung abgebrochen. Die Implementierung des korrekten Verhaltens ist für die Zukunft allerdings noch vorgesehen.

[7]Tatsächlich werden die Parameter des Methodenaufrufs nicht zwischengespeichert, sondern gleich deren zukünftige Position auf dem Stack berechnet und diese dann dorthin verschoben. Dieses Verfahren spart einen Kopiervorgang und den relativ großen temporären Speicher. Besonders letzterer wäre ein Problem, denn entweder müsste dieser Speicher dauerhaft zur Verfügung stehen, oder für jeden Methodenaufruf müsste er angelegt und wieder freigegeben werden. Und da die Anzahl der möglichen Parameter (abhängig

Frame angelegt, in welchem Zeiger auf die `method_info`-Struktur der aufzurufenden Methode, sowie auf deren Klasse gespeichert werden. Die zwischengespeicherten Parameter werden nach dem Stack Frame wieder auf den Stack geschoben. Der Program Counter (PC) wird auf den Anfang des Codes der neuen Methode gesetzt und dort mit der Ausführung des ersten Opcodes begonnen.

Der letzte Opcode einer Methode muss immer aus der Gruppe der `RETURN`-Opcodes sein. Für jeden Rückgabetypen gibt es einen, wobei entweder ein oder zwei Slots auf dem Stack liegen, die zurückgegeben werden sollen. Wurde ein solcher Opcode verwendet, dann wird der oder die Slots, die den Rückgabewert enthalten, vom Stack genommen und zwischengespeichert. Im hier gezeigten Beispiel wird kein Wert von der Methode zurückgegeben, weshalb auch die typlose `RETURN` Variante verwendet wird.

Wurde das Ausführen der Methode beendet, dann werden die lokalen Variablen, die Parameter sowie der Stack Frame der nun abgeschlossenen Methode vom Stack heruntergenommen. (Operanden dieser Methode dürfen keine mehr darauf liegen.) Der Program Counter der vorigen (aufrufenden) Methode wird wiederhergestellt, vorhandene Rückgabewerte wieder auf den Stack geschoben und das Ausführen der entsprechenden Methode mit der nach dem Methodenaufruf-Opcode folgenden Instruktion fortgesetzt.

Abbildung 7.13 zeigt die Debuginformationen, die während der Ausführung des Beispiels ausgegeben werden. Aber auch hier lässt sich der dahinter steckende Aufwand praktisch nur erahnen.

Man sagt, statische Methoden werden fest (statisch) oder früh gebunden. Im Falle von Java also während der Kompilation. Die Entscheidung, welche Methode ausgeführt wird, wird nur aufgrund der gegebenen Klasse getroffen und kann sich zur Laufzeit nicht mehr ändern. Wie man im Vergleich zu den anderen Varianten noch sehen wird, ist dies die einfachste Art, Methoden aufzurufen.

---

von der Anzahl der Parameter die zwei Slots belegen) auf maximal 256 begrenzt ist, müsste diese Struktur 256 * sizeof(`Slot`) = 1024 Byte groß sein. In Embedded Systemen kann 1 KB kontinuierlicher Speicher schon eine große Verschwendung und aufgrund drohender Fragmentierung auch schon ein Problem darstellen. Bezüglich der Begrenzung auf max. 256 Parameter siehe JLS ([Lindholm99]), Kapitel 4.10.

```
Daniel$ pura StaticMethod
[...]
Executing method StaticMethod.main([Ljava/lang/String;)V...
Executing INVOKESTATIC
        Pushing new stack frame.
        Frame number 3, size 24 bytes, 0 parameter slots,
        stack is now 56 bytes high.
===> Executing method StaticMethod.test()V...
Executing BIPUSH
        Pushing byte value 99 onto the stack.
Executing ISTORE_0
        Storing integer 99 from the stack into slot 0.
Executing RETURN
        Popping stack frame off the stack.
        Back at frame 2, height 36 bytes.
Executing RETURN
        Popping stack frame off the stack.
        Back at frame 1, height 16 bytes.
Freeing up stack.

Execution finished. 4 Bytecodes executed.
```

Abbildung 7.13: Debuginformationen beim Ausführen des Beispiels aus Abbildung 7.11

7.6 Aufruf virtueller Methoden

Ganz im Gegensatz zu den früh gebundenen statischen Methoden werden die in Java am häufigsten verwendeten virtuellen Methoden dynamisch oder auch spät gebunden. Welche Methode aufgerufen wird entscheidet sich erst zur Laufzeit und hängt von der Klasse des Objekts ab, für welches die Methode aufgerufen werden soll. Außerdem muss die gesuchte Methode nicht unbedingt in der Klasse des Objekts zu finden sein, sondern sie kann sich durch Ableitungen bedingt auch in der oder den Superklassen der Klasse befinden. Die JVM geht hier systematisch rekursiv vor. Zuerst wird die Klasse des Objekts nach der Methode durchsucht. Wurde sie dort nicht gefunden ist deren Superklasse dran, dann deren Superklasse und so weiter. Entweder die Methode wird in einer der Klassen gefunden, oder die JVM erreicht die Klasse `Object` und wenn die Methode dort nicht gefunden wird, dann war die Suche erfolglos. Wurde die Methode gefunden, dann wird die Suche abgebrochen und die entsprechende Methode aufgerufen.

Dieses Vorgehen ergibt den gewohnten Effekt, dass durch diesen Mechanismus immer die unterste einer in der Ableitungshierarchie eventuell mehrfach überschriebenen Methode ausgeführt wird. Wurde diese nämlich entdeckt, so werden die Superklassen gar nicht weiter untersucht. Erzeugt man jedoch eine Instanz der Klasse, die die Superklasse der zuvor genannten Klasse ist, dann ist der Blickwinkel anders. Auch hier wird mit der Suche nach einer Methode dann in der Klasse der Instanz begonnen, was zur Folge hat, dass eine im vorigen Fall überschriebene Methode nun die Methode der Wahl ist.

Abbildung 7.14 zeigt solch ein Beispiel grafisch. Klasse A ist direkt von Object abgeleitet. Klasse B von A und Klasse C von B. Von allen drei Klassen (A, B, C) wurde jeweils eine Instanz erzeugt. Der grüne Pfeil zeigt, wie ein virtueller Methodenaufruf der Methode `test()` bei der Instanz der Klasse A aussieht. Der blaue Pfeil zeigt dies für die Instanz der Klasse B und der rote für die Instanz der Klasse C. Man sieht, dass jede Instanz als erstes ihre eigene Klasse ansteuert und von dort aus die Methode gesucht wird. Die Instanzen der Klassen A und B finden jeweils direkt eine Implementierung von `test()`,

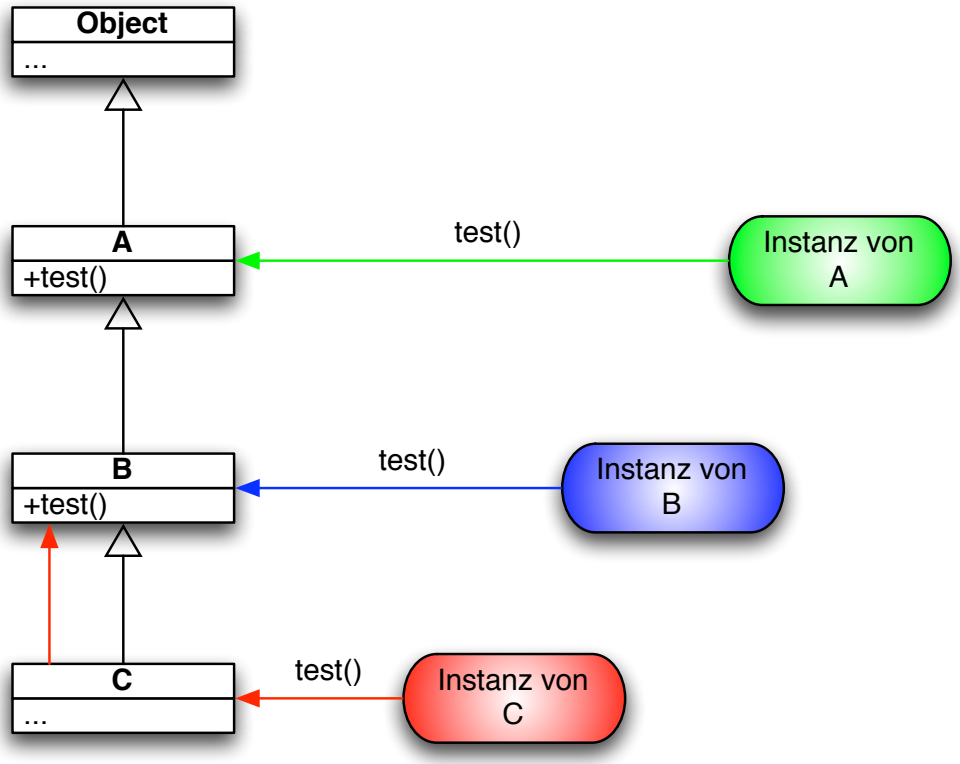


Abbildung 7.14: Verarbeitung verschiedener virtueller Methodenaufrufe

aber jeweils eine andere, wobei Klasse B in ihrer Ableitung die Methode der Klasse A überschrieben hat. Die Instanz der Klasse C findet in ihrer Klasse keine direkte Implementierung von `test()`, daher wird in ihrer Superklasse weitergesucht, wo sie nun auch eine Implementierung findet. Abbildung 7.15 zeigt eine entsprechende Beispiel-Anwendung in Java.

Abbildung 7.16 zeigt den wesentlichen Teil der `main()`-Methode des Beispiels. Die Indizes in den Constant Pool zeigen wie gehabt auf `CONSTANT_Method_info`-Strukturen. Es fällt jedoch auf, dass vor jedem Methodenaufruf eine andere Referenz[8] geladen wird. Hierbei handelt es sich um die Referenzen der entsprechenden Instanzen. Mit `aload_1` wird also die Referenz der Instanz der Klasse A geladen, mit `aload_2` die Referenz der Instanz von Klas-

[8]Das Präfix 'a' zum Beispiel von `aload_1` deutet an, dass es sich um eine Referenz handelt.

```
public class VirtualMethodCall
{
   private static class A
   {
      public void test() {}
   }

   private static class B extends A
   {
      public void test() {}
   }

   private static class C extends B {}

   public static void main(String[] args)
   {
      A a= new A();
      B b= new B();
      C c= new C();

      a.test();
      b.test();
      c.test();
   }
}
```

Abbildung 7.15: Beispiel für virtuelle Methodenaufrufe

```
   [...]
   27:  aload_1
   28:  invokevirtual   #8; //Method VirtualMethodCall$A.test:()V
   31:  aload_2
   32:  invokevirtual   #9; //Method VirtualMethodCall$B.test:()V
   35:  aload_3
   36:  invokevirtual   #10; //Method VirtualMethodCall$C.test:()V
   39:  return
}
```

Abbildung 7.16: Disassemblat des wesentlichen Teils der `main()`-Methode des Beispiels aus Abbildung 7.15

```
[...]
Executing ALOAD_1
        Pushing reference 4 onto the stack.
Executing INVOKEVIRTUAL
        Pushing new stack frame.
        Frame number 3, size 20 bytes, 1 parameter slots,
        stack is now 68 bytes high.
        ===> Executing method VirtualMethodCall$A.test()V...
Executing RETURN
        Popping stack frame off the stack.
        Back at frame 2, height 48 bytes.
Executing ALOAD_2
        Pushing reference 7 onto the stack.
Executing INVOKEVIRTUAL
        Pushing new stack frame.
        Frame number 3, size 20 bytes, 1 parameter slots,
        stack is now 68 bytes high.
        ===> Executing method VirtualMethodCall$B.test()V...
Executing RETURN
        Popping stack frame off the stack. Back at frame 2,
        height 48 bytes.
Executing ALOAD_3
        Pushing reference 11 onto the stack.
Executing INVOKEVIRTUAL
        Pushing new stack frame.
        Frame number 3, size 20 bytes, 1 parameter slots,
        stack is now 68 bytes high.
        ===> Executing method VirtualMethodCall$B.test()V...
Executing RETURN
        Popping stack frame off the stack. Back at frame 2,
        height 48 bytes.
Executing RETURN
        Popping stack frame off the stack. Back at frame 1,
        height 16 bytes.
Freeing up stack.

Execution finished. 66 Bytecodes executed.
```

Abbildung 7.17: Debuginformationen beim Ausführen des Beispiels aus Abbildung 7.15

se B und mit `aload_3` die Referenz der Instanz von Klasse C. Abbildung 7.17 zeigt, wie die drei Methoden aufgerufen werden. Die JVM gibt in den Debuginformationen auch Auskunft darüber, für welche Implementierung sie sich konkret entschieden hat. Wie man sieht, unterscheiden sich diese von den aus Abbildung 7.16 ersichtlichen, da in Klasse C keine Implementierung von `test()` gefunden wurde und daher die Implementierung in Klasse B gewählt wurde.

7.7 Spezielle und `private` Methoden

Ein Zwischending der beiden zuvor genannten statischen und virtuellen Methodenaufrufe sind die „speziellen“ Methodenaufrufe über den `INVOKESPECIAL`-Opcode. Auch hier wird die Referenz auf ein Objekt übergeben, jedoch ist die Methode statisch an den Typ dieser Referenz gebunden. Im Gegenteil zum virtuellen Methodenaufruf findet die Bindung also schon zur Kompilationszeit statt und wird nicht dynamisch zur Laufzeit aufgelöst. Realisiert werden mit diesem Verfahren die in Java mit `private` deklarierten Methodenaufrufe, sowie das Aufrufen von Konstruktoren und Methoden der Superklasse.

Bei `private` deklarierten Methoden ist der Verwendungsbereich naturgemäß stark eingeschränkt. Da private Methoden immer nur innerhalb der eigenen Klasse verwendet werden dürfen, werden sie auch immer (implizit oder explizit) über die `this`-Referenz der aktuellen Instanz der Klasse aufgerufen. Die `this`-Referenz besitzt automatisch den Typ der Instanz und ist damit unveränderlich an den Typ der Klasse gebunden. Die private Methode wird so auch nur innerhalb der entsprechenden Klasse gesucht. `this`-Referenzen mit anderem Typ können nicht vorkommen und auf anderen Weg lassen sich `private` deklarierte Methoden per Definition nicht aufrufen.

Beim Aufrufen von Methoden der Superklasse sieht es ähnlich aus. Möchte man eine von der Subklasse überschriebene Methode der Superklasse aus der Subklasse heraus aufrufen, so verwendet man äquivalent zur `this`-Referenz die `super`-Referenz. Die `super`-Referenz besitzt ebenfalls automatisch den Typ der dazugehörigen Klasse, vom Blick der Subklasse aus also den Typ

der Superklasse. Der Aufruf einer Methode über den `INVOKESPECIAL`-Opcode mit Hilfe der `super`-Referenz hat automatisch immer zur Folge, dass definitiv eine Methode der Superklasse aufgerufen wird, sofern sie von der Subklasse aus nach den Sichtbarkeitsregeln von Java sichtbar ist. Ein normaler, dynamisch aufgelöster virtueller Methodenaufruf würde hier nicht das gewünschte Ergebnis liefern, da über diesen unvermeidbar die neueste Implementierung der Methode in der Subklasse gefunden und aufgerufen würde.

Bei den Konstruktoren sieht es ein wenig anders aus. Sie sind in der Regel `public` deklariert und werden aus beliebigem Code heraus zum Initialisieren der neu erstellten Instanz einer Klasse aufgerufen. Aber auch diese Aufrufe sind statisch vom Typ der Referenz abhängig. So wird ein Konstruktor immer automatisch aufgerufen, nachdem der neue Speicherplatz für die Instanz einer Klasse im Heap angelegt wurde. Eine Trennung von der Reservierung des Speichers (`new`) und der Initialisierung (Aufruf des Konstruktors) ist nicht möglich und Konstruktoren können auch nicht manuell aufgerufen werden. (Außer innerhalb einer Klasse ein Konstruktur durch einen anderen.) Entsprechend kommt die Referenz, mit welcher der Konstruktor aufgerufen wird, immer unmittelbar vom Speichermanager der JVM und hat den zur Klasse der gerade erzeugten Instanz passenden Typ. Auch diese statische Bindung macht Sinn, denn in einem solchen Fall will man sich sicher sein, dass auch wirklich der frisch erzeugte Speicherbereich von dem dazugehörigen Konstruktor initialisiert wird und kein anderer, was fatale Folgen haben könnte. (Dort schon benutze Variablen würden wieder mit ihren Startwerten überschrieben, was Inkonsistenzen innerhalb einer Anwendung auslösen würde.)

Abbildung 7.18 zeigt ein Beispiel für die Verwendung des `INVOKESTATIC`-Opcodes. In Klasse `A` ist lediglich eine `public` deklarierte Methode `doTest()` implementiert. Klasse `B` überschreibt diese mit einer eigenen Implementierung und implementiert eine private Methode `privTest()`. In der Methode `test()`, welche von der `main()`-Methode heraus aufgerufen wird, werden alle drei Methoden aufgerufen.

Abbildung 7.19 zeigt die wesentlichen Teile des Disassemblats. Wie man an den Bytecodes der Methode `test()` erkennen kann, wird die private

```
public class A
{
   public void doTest()
   {
      System.out.println("This is class A!");
   }
}

public class B extends A
{
   private void privTest()
   {
      System.out.println( "Private Method!" );
   }

   public void doTest()
   {
      System.out.println("This is class B!");
   }

   public void test()
   {
      privTest();
      doTest();
      super.doTest();
   }

   public static void main(String[] args)
   {
      B b= new B();
      b.test();
   }
}
```

Abbildung 7.18: Beispiel für die Verwendung des `INVOKESPECIAL`-Methodenaufrufs

```
Daniel$ javap -c -verbose -private B
[...]
  Constant pool: // Auszug
const #6 = Method       #9.#31; //  B.privTest:()V
const #7 = Method       #9.#32; //  B.doTest:()V
const #8 = Method       #12.#32;        //  A.doTest:()V
const #9 = class        #33;    //  B
const #12 = class       #35;    //  A
const #31 = NameAndType #17:#14;//  privTest:()V
const #32 = NameAndType #18:#14;//  doTest:()V
const #33 = Asciz       B;
const #35 = Asciz       A;

{
[...]

public void test();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #6; //Method privTest:()V
   4:   aload_0
   5:   invokevirtual   #7; //Method doTest:()V
   8:   aload_0
   9:   invokespecial   #8; //Method A.doTest:()V
   12:  return

[...]
}
```

Abbildung 7.19: Wesentliche Teile des Disassemblats des Beispiels aus Abbildung 7.18

`privTest()` Methode mittels des `INVOKESPECIAL`-Opcodes aufgerufen. Eintrag 6 des Constant Pools enthält eine `CONSTANT_Method_info`-Struktur, welche wie erwartet auf Klasse `B` verweist, was dem Typ der (implizit) verwendeten `this`-Referenz entspricht. Der erste Aufruf von `doTest()` entspricht einem üblichen virtuellen Methodenaufruf und wird entsprechend durch den `INVOKEVIRTUAL`-Opcode ausgeführt. Beim zweiten Aufruf von `doTest()` wurde jedoch die `super`-Referenz verwendet, was wiederum in der Verwendung des `INVOKESPECIAL`-Opcodes resultiert. Im entsprechenden Constant Pool Eintrag 8 wird aber nun auf die Klasse `A` verwiesen, welche dem Typ der `super`-Referenz entspricht.

7.8 Methodenaufruf über ein Interface

Methodenaufrufe über Interfaces (Opcode `INVOKEINTERFACE`) sind im Prinzip das selbe wie virtuelle Methodenaufrufe (Opcode `INVOKEVIRTUAL`). Auch sie werden dynamisch zur Laufzeit gebunden, lediglich der Typ der verwendeten Referenz ist ein Interface, anstatt einer Klasse. Die JVM verwendet hier lediglich aus einem pragmatischen Grund zwei verschiedene Opcodes, weil Methodenaufrufe über Interfaces nicht in der Art optimiert werden können, wie dies bei normalen virtuellen Methodenaufrufen der Fall ist. So kann die symbolische Referenz beim virtuellen Methodenaufruf durch eine direkte Referenz (Zeiger) ersetzt werden, weil sich die bei einem virtuellen Methodenaufruf zu benutzende Methode zur Laufzeit nicht ändern kann. (Siehe auch Kapitel 7.6.) Bei Methodenaufrufen über Interfaces muss die symbolische Referenz immer neu aufgelöst werden, da sich hinter der Referenz mit dem Typ eines Interfaces jederzeit eine Instanz einer anderen Klasse befinden kann, welche auch das entsprechende Interface implementiert. Und diese Klasse würde ihre ganz eigene Implementierung der aufzurufenden Methode mitbringen.

Wie man schon ahnen kann, ist dieser Effekt für die Performance von Interface-Methodenaufrufen nicht gerade förderlich. Eine möglichst effiziente Implementierung zum Auflösen der entsprechenden symbolischen Referenz wäre wünschenswert. In der hier behandelten JVM Implementierung

wurde aus Gründen der Einfachheit absichtlich keine Optimierung vorgenommen, was aber auch in einer äußerst schlechten Laufzeit von Interface-Methodenaufrufen resultiert. Entsprechend des in Kapitel 7.5 und 7.6 beschriebenen Vorgehens muss die symbolische Referenz der auszuführenden Methode im Constant Pool nur einmal aufgelöst und damit nur einmal die entsprechende Methoden-Tabelle der Klasse(n) durchsucht werden, wobei im Worst-Case bei n Klassen in der Ableitungshierarchie einer Klasse mit jeweils m Einträgen in der Methodentabelle eine ungefähre Laufzeit von $n * m \equiv O(n^2)$ zu erwarten ist. Die Verwendung von Hashtabellen zum Auffinden der Methode in den Methodentabellen der Klassen würde den Vorgang deutlich auf eine Laufzeit von $O(n)$ beschleunigen. Lediglich die Anzahl der zu durchsuchenden Klassen würde dann noch einen Einfluss auf die Laufzeit haben, wobei dieser aber bei weitem nicht mehr so stark ins Gewicht fallen würde. Auch ist die Anzahl der Klassen in einer Ableitungshierarchie in der Regel recht begrenzt.

Kapitel 8

Abschluss

In diesem letzten Kapitel wird der Inhalt dieser Diplomarbeit noch einmal zusammengefasst, ein Ausblick auf zukünftige Planungen, Möglichkeiten und Entwicklungen wird gegeben und der Autor bewertet seine Arbeit und zieht ein Fazit.

8.1 Zusammenfassung

> Oddly enough, I've yet to see a book that covers how to build a JVM; every book published so far focuses on the abstract JVM rather than how someone would implement one.
>
> – Godfrey Nolan[1]

Die Idee zu dieser Diplomarbeit entstand aus einer allgemein bekannten Knappheit an praxisnahen Erläuterungen zu konkreten Implementierungen von Java Virtual Machines. Während Arbeiten wie [Lindholm99], [Venners99], [Meyer97] und auch [Smith05] den theoretischen Aspekt von Java Virtual Machines zur vollen Zufriedenheit erläutern, so existiert im Bereich der Erläuterungen zu konkreten Implementierungen eine große, klaffende Lücke. Mit dieser Arbeit sollte diese ansatzweise gestopft werden. Hierfür wurde die Grundfunktionalität einer Java Virtual Machine von Grund auf

[1] Aus [Nolan04] Seite 17.

und von Anfang an mit Ausrichtung auf Einfachheit und Lesbarkeit neu entwickelt. Da die Beschreibung einer vollständigen Java Virtual Machine den Umfang einer Diplomarbeit bei weitem sprengen würde, wurde sich in dieser Arbeit auf die Datenstrukturen und deren Verwendung konzentriert. Das Niveau der Erläuterungen wurde absichtlich niedrig angesetzt, so dass der interessierte Informatik-Student problemlos diese Arbeit lesen und verstehen kann. Leider konnte in manchen Detailfragen aus Platzgründen nur an die Java Virtual Machine Specification ([Lindholm99]) verwiesen werden, jedoch kann diese Arbeit auch für sich allein gelesen werden, wenn den Leser solche Details nicht interessieren.

Begonnen wird mit einer Übersicht über virtuelle Maschinen. Diese werden klassifiziert und Unterscheidungsmerkmale erläutert. Grob unterteilt man virtuelle Maschinen in System Virtual Machines und Process Virtual Machines, wovon Erstere eher ganze Systeme oder Betriebssysteme und zweitere nur einem Prozess, also im Prinzip einer Anwendung, ihre Dienste zur Verfügung stellen. Es folgen detaillierte Erläuterungen zur Java Virtual Machine und deren Aufbau. Grundsätzlich unterscheidet man hier die Module der Method Area, die Ausführungseinheit (Execution Engine), Stack und Heap, sowie den Classloader und die Möglichkeit zum Ausführen von nativen Methoden von Java Code aus. Es folgen die Erläuterungen verschiedener klassischer und moderner Implementierungsansätze von Java Virtual Machines. Erwähnenswert sind hier der klassische Interpreter, die Just-In-Time Compiler und der Hotspot Compiler der aktuellen Sun Implementierungen. Das Java Class File Format wird erläutert, welches zum Laden der Klassen und als Basis der Laufzeitstrukturen verwendet wird und der Begriff der symbolischen Referenz wird eingeführt, was das mit den Laufzeitstrukturen am häufigsten zu lösende Problem darstellt. Der Ladeprozess einer Klasse wird genauer beleuchtet und dessen einzelne Schritte Load, Link und Initialize erläutert.

Weiter geht es mit den für eine Java Virtual Machine nötigen Datenstrukturen. Im Wesentlichen handelt es sich hierbei um Operand- und Method-Stack, sowie den Heap, wobei man die beiden Stacks auch in einer Daten-

struktur vereinen kann. Für beide werden verschiedene Implementierungsansätze gezeigt und deren Vor- und Nachteile diskutiert.

Nach dem (leider nötigen) rein theoretischen Anfang geht es anschließend an die Praxis. In Kapitel 4 wird die Entscheidung der für die Implementierung gewählten Programmiersprache C, sowie für die verwendeten Entwicklertools (Apples Mac OS X Tools) erläutert. Im Anschluss folgt eine generelle Erläuterung über den für die eigene Java Virtual Machine (JVM) gewählten Implementierungsansatz und deren generellen Entscheidungen, die getroffen werden mussten. Grob zusammengefasst wurde entschieden, einen klassischen Interpreter zu implementieren, da dieser am einfachsten und am übersichtlichsten und damit auch am besten verständlich ist. Des Weiteren wurde die JVM als typlose JVM ausgelegt. Das bedeutet, dass die JVM ohne weitere Informationen nicht weiß, welchen Typ die Daten haben, die sie in den Variablen speichert. Durch die getypten Opcodes der JVM lässt sich dann bei Verwendung der Daten allerdings auf den Typ schließen. Lediglich bei Arrays musste eine Ausnahme von diesem System gemacht werden, da ansonsten der `instanceof`-Opcode nicht machbar gewesen wäre, der jedoch ein wichtiges Tool im Sinne der dynamischen, objektorientierten Programmierung darstellt.

In Kapitel 6 folgen Erläuterungen zu den in der Implementierung eingesetzten Datenstrukturen und wie diese aus dem ursprünglichen Java Class File Format hergeleitet und, wenn nötig, erweitert wurden. Grundsatz war auch hier die Einfachheit. Auf aufwändigere Lösungen wie Method Tables oder Hashtabellen wurde absichtlich verzichtet, an den passenden Stellen jedoch auf eventuell interessante Verbesserungen hingewiesen.

Zum Abschluss wird auf einige Fälle der Verwendung von Laufzeitstrukturen eingegangen, welche für Java Programmierer leicht verständlich anhand von Java Code-Beispielen erklärt werden. Diese werden disassembliert und Anweisung für Anweisung deren Ausführung von der JVM erläutert. Behandelt wird das Laden und Speichern der verschiedenen Variablen-Typen, sowie das Ausführen der unterschiedlichen Typen von Methodenaufrufen.

8.2 Ausblick

Das Potenzial der JVM-Implementierung ist bei weitem noch nicht ausgeschöpft. Wichtige Dinge wie der Garbage Collector und der Verifier fehlen noch. Auch die Unterstützung von Multithreading fehlt, auch wenn sie von der Struktur her schon vorgesehen ist. Von der JVM intern zu werfende Exceptions werden zur Zeit noch nicht korrekt geworfen[2]. Auch fehlt die für einige Opcodes definierte Behandlung des `Wide` Prefix-Opcodes bisher[3]. Der Einsatz von Hashtabellen oder Method Tables zur Beschleunigung von Methodenaufrufen wäre eine nützliche und interessante Erweiterung.

Im Zuge der Implementierung eines Garbage Collectors müssten Erweiterungen an den Datenstrukturen vorgenommen werden. Zum Beispiel könnten hier Bitfelder zur Kennzeichnung der vom Garbage Collector unter allen Variablen aufzufindenden Referenzen verwendet, oder in diesem Zusammenhang gleich ein vollständiges Typsystem innerhalb der JVM etabliert werden, so dass zur Laufzeit jeder Zeit der Typ einer Variable bekannt ist. Gerade bei letzterem stellt sich jedoch die Frage nach einer effizienten Lösung. So sind die Breiten der verwendeten Datentypen mit 32 und 64 Bit (einfacher und doppelter Slot) schon auf die zur Zeit üblichen Struktur- und Busbreiten moderner Prozessoren optimiert. Der Ort und der Umfang der zu speichernden Typinformationen müsste also gut überlegt sein.

[2]Zur Zeit werden diese Exceptions als normale Fehlermeldungen gehandhabt. Da diese meistens fatal sind, wird die Ausführung ohne die Möglichkeit der Intervention von Java Code, zum Beispiel durch try/catch-Blöcke, von der JVM abgebrochen. Eine Möglichkeit, dies nach den Vorgaben der Java Language Specification zu implementieren wäre die Verwendung der `longjmp()` und `setjmp()` Funktionen, die (inoffizieller) Bestandteil der meisten C-Bibliotheken sind.

[3]Der `Wide` Prefix-Opcode wird eingesetzt, wenn der übliche Platz für Daten (zum Beispiel Indizes) im Bytecode-Datenstrom nicht mehr ausreicht. Für solche Fälle gibt es dann die so genannten Wide-Varianten verschiedener Opcodes, bei denen mehr Platz für die Daten vorgesehen ist. Dieses Vorgehen verschwendet keinen unnötigen Speicher, da man meistens mit den kurzen Opcode-Varianten auskommt, es gibt jedoch die Möglichkeit für eine Erweiterung, um den manchmal eben etwas knapp bemessenen Einschränkungen der kurzen Varianten zu entkommen. Für diese Implementierung wurden die `Wide`-Opcodes nicht mit hoher Priorität behandelt, da das Ausführen von umfangreichen Anwendungen, bei denen der Einsatz der `Wide`-Opcodes dann nötig würde, zu Lernzwecken eher unüblich ist.

Anstatt des Verifiers könnte auch gleich die ab Java 6 nun vorgeschriebene Unterstützung von Stackmaps realisiert werden. Das Konzept der Stackmaps kommt aus dem Bereich der Embedded- und Mobile-Systeme und minimiert den Aufwand, den eine JVM zur Verifizierung von Klassen treiben muss. Es ersetzt den bisher üblichen Vorgang der Verifizierung einer Klasse noch während des Ladevorgangs (i.e. den Verifier) durch unkomplizierte einzelne Prüfungen mit Hilfe eines weiteren Stacks mit Typinformationen, sowie durch einen Teil den der Compiler übernimmt indem er die eigentlichen „Stack Maps“ (daher der Name) erzeugt und in den Klassen als Attribute speichert.

Um die Fehlerfreiheit und Funktionsfähigkeit der Implementierung sicherstellen zu können, müsste ein vollständiger Satz an Tests erstellt werden, der jeden Opcode mit allen seinen Varianten und auch die korrekte Funktion verschiedener Module überprüft. Eventuell könnte hier teilweise auch auf die existierenden Technology Compatibility Toolkits (TCK) von Sun zurückgegriffen werden, mit denen die Kompatibilität zu deren Standards getestet und nachgewiesen werden kann[4].

Um die Eignung der JVM-Implementierung als Lernwerkzeug weiter zu verbessern, wäre eine Erweiterung der zur Verfügung stehenden Informationen zur Laufzeit wünschenswert. Beschränken sich diese bisher nur auf das Laden von Klassen, das Abarbeiten von Opcodes und den Meldungen über Änderungen am Stack und beim Speicher, so könnte man hier weitere Informationen, wie zum Beispiel das Auflösen von symbolischen Informationen, zugänglich machen. Auch eine detaillierter einstellbare Logging-Funktionalität wäre in diesem Zusammenhang wünschenswert.

Ein etwas umfangreicherer Vorschlag für die Verwendung der JVM-Implementierung wäre der Einsatz in einem interaktiven Visualisierungswerkzeug. Mit diesem könnte man die einzelnen Strukturen der JVM zur Laufzeit betrachten, Stack und Heap mit ihren Zusammenhängen darstellen und Zugriffe auf Datenstrukturen nachvollziehen.

Auf der Seite der Dokumentation wäre eine möglichst vollständige Abdeckung der Implementierung wünschenswert. Dies schließt zum Beispiel

[4]Tatsächlich darf eine JVM-Implementierung nur dann den geschützten Namen „Java“ öffentlich tragen, wenn es das entsprechende TCK erfolgreich bestanden hat.

Erläuterungen über die Implementierung zum Auflösen einzelner Varianten von symbolischen Informationen mit ein, oder auch Erläuterungen zu den einzelnen Opcodes. Alleine schon der Umfang der Opcodes, ca. 250 sind definiert, lässt jedoch den erforderlichen Umfang dieser Arbeit erahnen.

Die vorhandene Klassenbibliothek stellt ebenfalls bisher nur einen Bruchteil der Bibliotheken der Java Standard Edition (JSE) dar. Eine vollständige Reimplementierung dieser vorzunehmen wäre absurd, jedoch würde die Erweiterung, zum Beispiel auf den Umfang der CLDC[5] Implementierung, Sinn machen und würde auch ein schönes Projekt für interessierte Studenten darstellen. Grundsätzlich wären im Bereich der nun vorgelegten Implementierung und Dokumentation einige potenzielle Diplom- und Projektarbeiten denkbar. Da der Autor die Quellen der Implementierung unter der GPL-Lizenz frei gibt, steht dem auch nichts im Weg.

Abschließend sei noch erwähnt, dass der Autor einen zukünftigen Einsatz der JVM-Implementierung im Embedded-Bereich plant. Hierfür wären jedoch einige Optimierungen, sowie verschiedene Änderungen an der Laufzeitstruktur nötig, damit Klassen und deren Code „in Place“, also ohne sie Laden zu müssen, aus dem für die JVM unveränderlichen Flash-Speicher genutzt und ausgeführt werden können.

8.3 Bewertung und Fazit

Der praktische Teil, also die Implementierung, verlief nach einer ausgiebigen Vorbereitungsphase sehr zufriedenstellend. Die Wahl der Programmiersprache und Tools stellte sich als passend heraus und die Arbeit ging gut von der Hand, so dass das Ziel, die Grundlagen einer JVM vollständig zu implementieren, eindeutig erreicht werden konnte. Früh stellte sich jedoch heraus, dass mein anfängliches Ansinnen, im schriftlichen Teil die gesamte Implementierung zu dokumentieren, aufgrund von Zeit- und Platzmangel im Umfang einer Diplomarbeit nicht zu realisieren war. Nach der rechtzeitigen

[5] Connected Limited Device Configuration, die Spezifikation einer eingeschränkten JVM und deren Bibliothek für mobile Endgeräte.

Einschränkung auf die Laufzeitstrukturen der Implementierung konnte aber auch in diesem Teil der Arbeit das gesteckte Ziel erreicht werden.

Die Erarbeitung dieser Diplomarbeit, der Ausflug in die hintersten Winkel der JVM Spezifikation, waren eine interessante Herausforderung, die ich gerne angenommen habe.

Anhang A

Hinweise zur JVM Implementierung

In diesem Anhang finden sich einige Informationen und Hinweise zu der behandelten JVM Implementierung. Da auch diese einen Namen haben muss, hat sich der Autor für den Namen „Pura“[1] entschieden.

A.1 Verfügbarkeit

Die Pura JVM steht in dem in dieser Arbeit beschriebenen Umfang fertig zum Testen zur Verfügung. Entweder wurden alle benötigten Komponenten auf der beiliegenden CD geliefert, alternativ stehen sie unter `http://www.hobsoft.de` zum freien Download zur Verfügung.

A.2 Lizenz

Die Pura JVM wird unter der GNU General Public License in der jeweils aktuellen Version (2 oder später) freigegeben. Die Lizenz kann unter [GPL] nachgelesen werden.

[1] Die Idee zum Namen der Java Virtual Machine Implementierung kam ihm beim freien Assoziieren mit einer frischen Tasse Kaffee der Art „Pura Arabica“ (100% Arabica) – der Lieblingssorte des Autors.

A.3 Bibliothek

Die folgenden Bibliotheksklassen wurden implementiert:

Packet java.lang

- ArrayIndexOutOfBoundsException
- Error
- Exception
- IllegalArgumentException
- IndexOutOfBoundsException
- Integer
- Long
- Object
- RuntimeException
- StackTraceElement
- String
- StringBuilder
- System
- Throwable

Packet java.io

- PrintStream

Die Implementierungen sind jedoch bei weitem nicht als vollständig zu bezeichnen. In der Regel wurde nur der Teil implementiert, mit dem einfache Konsolenanwendungen (Eingabe über Programm-Argumente, Ausgabe wie üblich über System.out) laufen gelassen werden können. Im Zweifel hilft ein Blick in den mitgelieferten Quellcode weiter.

A.4 Installation

Zur Installation ist nichts weiter zu beachten. Kopieren Sie einfach das gesamte Pura-Installationverzeichnis von CD oder aus dem Download-Archiv auf Ihre Festplatte und stellen Sie sicher, dass Sie das Verzeichnis <`Pura Installation`>`/bin` in Ihre `PATH` Evironment-Variable aufnehmen, damit sie Pura von einem beliebigen Verzeichnis heraus aufrufen können. Zusätzlich ist es empfehlenswert, die `PURA_CLASSPATH` Environment-Variable zu setzen. (Siehe Abschnitt A.7 weiter unten.)

A.5 Kompilieren von Pura

Grundsätzlich sollte sich Pura mit jedem ANSI C89 kompatiblen Compiler kompilieren lassen. Entwickelt und getestet wurde allerdings nur mit der GNU Compiler Collection ab Version 4. Ein Testlauf mit Windows unter Cygwin ergab, dass die dort noch verwendete Version 3 die `malloc_size()`-Funktion noch nicht unterstützt, welche in der aktuellen Version des Speichermanagers so benutzt wird, dass sie nicht auf Anhieb zu entfernen war. Um die Windows-Nutzer, die auf Cygwin angewiesen sind, nicht ganz auszuschließen, wurde eine Notlösung eingebaut. (Siehe auch Kommentar am Anfang der Datei `memoryManager.c`.) Leider macht diese die Speicherstatistik, welche mittels dem Parameter „-mem“ eingeschaltet werden kann, praktisch wertlos. Die verbleibenden Funktionen werden davon jedoch nicht beeinflusst.

Das Kompilieren von Pura kann direkt von der Kommandozeile (unter Windows aus der Cywin Bash Shell) aus erledigt werden. Wechseln Sie in das Quellcode-Verzeichnis (<`Pura Installation`>`/src`) und kompilieren

Sie die Quellen mit dem folgenden Befehl:

```
gcc -o pura *.c
```

Der Name der ausführbaren Datei lautet in diesem Fall „pura“. Die ausführbare Datei kann nun in ein beliebiges Verzeichnis verschoben werden.

A.6 Parameter von Pura

Ähnlich zur Benutzung des `java` Kommandozeilentools der üblichen Java SDKs und JDKs wird Pura nach folgendem Schema aufgerufen:

```
pura [parameters] <main class> [arguments]
```

Die folgende Liste zeigt die zur Verfügung stehenden Parameter, die Argumente können, wie gewohnt, je nach Anwendung übergeben werden.

-cp | -classpath <classpath>
: Gibt den Classpath an. Der Default-Classpath ist das lokale Verzeichnis ('.'). Dieser Parameter überschreibt die `PURA_CLASSPATH`-Environment Variable.

-silent
: Schaltet die Debug-Ausgaben von Pura ab.

-mem | -memory
: Zeigt Speichernutzungsinformationen zur Laufzeit und eine Übersicht am Ende der Ausführung.

-opcodestats
: Zeigt eine Statistik der verwendeten Opcodes am Ende der Ausführung.

-all
: Schaltet alle möglichen Ausgaben ein.

-stack <stack size>

Definiert die (zur Laufzeit unveränderliche) Größe des Java Stacks. Default ist 10 KB.

A.7 Evironment-Variable

Um das wiederholte Eintippen eines Classpaths zu vermeiden, kann die Environment Variable `PURA_CLASSPATH` verwendet werden. Der Syntax entspricht der üblichen Syntax von Environment-Variablen des benutzten Betriebssystems.

Nicht vergessen werden sollte die Einbeziehung des lokalen Verzeichnisses ('.') und der Pfad zu den Bibliotheksklassen. (<Pura Installation>/lib)

A.8 Verbose Debug-Ausgaben entfernen

Die detaillierten Debug-Ausgaben, die überall im Code in Form von `logVerbose()` Funktionsaufrufen präsent sind, sind für einen signifikanten Anteil der Laufzeit verantwortlich zu machen. Um Pura mit anderen, nicht so „gesprächigen", JVMs vergleichen zu können, können alle diese Funktionsaufrufe durch die Definition der `VERBOSE_LOGGING_DISABLED` Präprozessor-Direktive entfernt werden. In der Datei `puraGlobals.h` ist diese Direktive bereits definiert aber auskommentiert.

Anhang B

Inhalt der CD

Im Folgenden ist aufgeführt, was sich auf der beigefügten CD befindet. (Sollte keine CD beigefügt sein, so kann deren Inhalt auch über `http://www.hobsoft.de` heruntergeladen werden.)

bin
: Beinhaltet ausführbare Dateien von Pura für verschiedene Betriebssysteme.

src
: Enthält den Quellcode der Pura JVM.

lib
: Die kompilierten Bibliotheksklassen von Pura. Dieses Verzeichnis muss immer als Classpath angegeben, oder in der `PURA_CLASSPATH` Environment-Variable enthalten sein.

libSrc
: Der Quellcode der Bibliotheksklassen.

examples
: Einige Beispiele und Test-Anwendungen

examples/paper
: Beispiele aus Kapitel 7.

XcodeProject

Der Quellcode zusammen mit dem Xcode-Projekt fertig zum Benutzen.

doc

Beinhaltet die PDF-Version der Diplomarbeit.

Literaturverzeichnis

[Blunden02] Blunden, Bill: Virtual Machine Design and Implementation in C/C++; Wordware Publishing, 2002

[Blunden03] Blunden, Bill: Memory Management, Algorithms and Implementation in C/C++; Wordware Publishing, 2003

[Cormen04] Cormen, T. H., Leiserson, C. E., Rivest, R., Stein, C.: Algorithmen – Eine Einführung; Oldenbourg Wissenschaftsverlag, 2004

[Goldberg83] Goldberg, A., Robson, D.: Smalltalk-80, The Language and its Implementation; Addison Wesley, 1983

[GPL] GNU General Public License: `http://www.gnu.org/licenses/gpl.html`

[Hook05] Hook, Brian: Write Portable Code; No Starch Press, 2005

[Jones96] Jones, R., Lins, R.: Garbage Collection, Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, 1996

[Lindholm99] Lindholm T., Yellin, F.: The Java Virtual Machine Specification, Second Edition; Addison Wesley, 1999

[Meyer97] Meyer J., Downing, T.: Java Virtual Machine; O'Reilly, 1997

[Nolan04] Nolan, Godfrey: Decompiling Java; Apress, 2004

[Shaylor03] Shaylor, N., Simon, D., Bush, W. R.: A Java Virtual Machine Architechture for Very Small Devices. In Proceedings of Conference

of Language, Compiler and Tool Support for Embedded Systems (LCTES'03); ACM Press, 2003 (Online unter `http://research.sun.com/projects/squawk/docs/lctes03.pdf`)

[Sedgewick92] Sedgewick, Robert: Algorithmen in C++; Addison-Wesley, 1992

[Simon06] Simon, Doug u.a.: Java on the Bare Metal of Wireless Sensor Devices: The Squawk Java Virtual Machine. In Proceedings of the Second International Conference on Virtual Execution Environment (VEE'06); ACM Press, 2006 (Online unter `http://research.sun.com/projects/squawk/docs/vee06-squawk.pdf`)

[Smith05] Smith, J. E., Nair, R.: Virtual Machines, Versatile Platforms for Systems and Processes; Morgan Kaufmann, 2005

[Squawk] Homepage des Squawk-Projekts: `http://research.sun.com/projects/squawk/`

[Venners99] Venners, Bill: Inside the Java 2 Virtual Machine, Second Edition; McGraw-Hill, 1999

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum:

Unterschrift: