Seminar aus Softwareentwicklung (Inside Java and .NET)

Seminararbeit

Die Architektur der Java-VM

von Christian Schwarzbauer (0056391)

Inhaltsverzeichnis

1. Einleitung

Jede Software benötigt Hardware (in unserem Fall einen Computer), um ausgeführt werden zu können.

Bei „herkömmlicher“ Softwareentwicklung wird aus dem Quelltext des Programmierers durch einen Compiler Maschinencode erzeugt, der auf den Prozessor des Computers abgestimmt ist. Auf unterschiedlichen Prozessoren (d.h. Prozessoren mit unterschiedlichen Befehlssätzen) kann also dasselbe Programm nicht ausgeführt werden, es ist also *plattformabhängig*.

Mit der Programmiersprache Java wurde ein anderer Weg eingeschlagen. Anstatt Maschinencode für einen Prozessor zu erzeugen, wird durch den Java-Compiler so genannter *Byte-Code* für eine abstrakte virtuelle Maschine – die so genannte *Java Virtual Machine* (kurz JVM) – erzeugt, die nur durch eine abstrakte Spezifikation bekannt ist. Diese JVM kann auf den unterschiedlichsten Plattformen implementiert werden und somit kann jedes Java-Programm auf diesen Plattformen ausgeführt werden. Dadurch wird *Plattformunabhängigkeit* erreicht.

In dieser Seminararbeit werden wir einen Blick hinter die Architektur der Java-VM machen und auch teilweise auf Möglichkeiten für die Implementierung eingehen.

1.1. Der Begriff der Java-VM

Wenn man von der bzw. einer Java Virtual Machine spricht, muss man sich im Klaren darüber sein, dass dieser Begriff 3 unterschiedliche Bedeutungen haben kann:

- die *abstrakte Computer-Spezifikation* der JVM
- eine *konkrete Implementierung* der JVM
- eine *Laufzeit-Instanz* der JVM

Die **abstrakte Spezifikation** der JVM ist das Konzept, das hinter der JVM steckt. Diese Spezifikation – zu finden unter **[Yel99]** – beschreibt detailgenau, aus welchen Teilen eine JVM zumindest bestehen muss und welchen Befehlssatz sie kennen und verstehen muss. Außerdem wird das `class`-Dateiformat genau beschrieben, das jede JVM verstehen muss. Es werden allerdings keine Details der Implementierung vorgegeben, über diese muss sich jeder Programmierer einer JVM seine eigenen Gedanken machen.

Um ein Java-Programm auf einer Plattform (z.B. MS Windows XP auf einem Intel Pentium 4) zum Laufen zu bringen, benötigt man eine **konkrete Implementierung** einer JVM. Diese liest die `class`-Dateien des Java-Programms aus und bringt das Programm zur Ausführung. Implementierungen der JVM gibt es zum Beispiel von Sun (**die** Standard-JVM, für mehrere Plattformen erhältlich), Microsoft und IBM. Eine Implementierung kann sowohl in Soft- als auch in Hardware realisiert werden.

Allerdings benötigt jedes Java-Programm eine eigene **Laufzeit-Instanz** der JVM, um ausgeführt zu werden. Das heißt, wenn auf einem Computer 3 Java-Programme parallel ausgeführt werden, so werden auch gleichzeitig 3 Instanzen der JVM ausgeführt, für jedes Programm eine.

Wir werden uns in dieser Arbeit in erster Linie auf die Spezifikation der JVM konzentrieren.

1.2. Anmerkung zur Namensgebung

In Büchern, im Internet und generell in der Literatur findet man in der Regel die Begriffe *Klasse* und *Typ* mit (annähernd) derselben Bedeutung versehen.

Der Begriff **Klasse** wird meist im Zusammenhang mit class-Dateien, mit dem Laden von Klassen und generell im Zusammenhang mit Objektorientierung verwendet.

Der Begriff **Typ** hingegen wird meist verwendet, wenn es um den Datentyp von Variablen geht oder wenn explizit darauf hingewiesen werden soll, dass es sich um Klassen oder Interfaces handeln kann.

Ich persönlich verwende den Ausdruck Klasse häufiger, da ich viel mehr an ihn gewohnt bin und ich glaube, dass man sich von einer Klasse einfacher eine Vorstellung machen kann, als von einem Typ. Ich werde daher in dieser Arbeit meist den Begriff Klasse verwenden, außer ich möchte wirklich darauf hinweisen, dass an einer Stelle sowohl Klassen als auch Interfaces zum Einsatz kommen können.

1.3. Anmerkung zu Fachausdrücken

Die Fachausdrücke, die in der Spezifikation der Java-VM gebraucht werden, sind natürlich allesamt in Englisch, aber nichtsdestotrotz in der Regel einfach zu verstehen. Mir persönlich sind die englischen Ausdrücke lieber, da diese in der Regel sehr aussagekräftig sind und sich teilweise nur umständlich ins Deutsche übersetzen lassen.

Ich werde daher in dieser Arbeit meist die englischen Ausdrücke verwenden, nur selten deutsche Übersetzungen.

2. Überblick über die Architektur der Java-VM

In der Spezifikation der Java-VM **[Yel99]** wird das Verhalten einer JVM durch Teilsysteme, Speicherbereiche, Datentypen und einem Befehlssatz beschrieben. Durch diese Komponenten wird die Architektur einer JVM beschrieben. Allerdings soll diese Beschreibung den Implementierer nicht einengen, sondern ihm vielmehr eine plausible Vorstellung davon geben, wie sich die virtuelle Maschine nach außen verhalten muss.

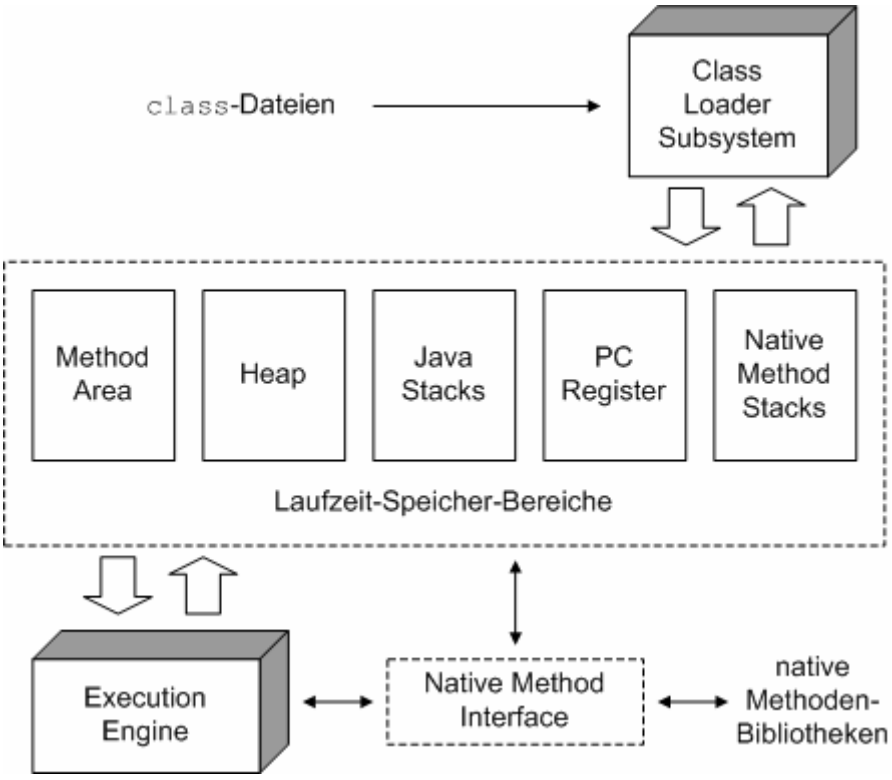

Abbildung 1: Die interne Architektur der Java-VM

In Abbildung 1 wird die Architektur der JVM gezeigt, wie sie in der Spezifikation beschrieben wird, mitsamt den wichtigsten Teilsystemen und Speicherbereichen.

Auf all diese Teilsysteme und Speicherbereiche werde ich in dieser Arbeit näher eingehen. Für noch detailliertere Beschreibungen verweise ich allerdings auf die JVM-Spezifikation **[Yel99]**.

3. Datentypen der Java-VM

Wie auch in der Programmiersprache Java, werden in der Java Virtual Machine unterschiedliche *Datentypen* verwendet, um bestimmte Operationen darauf anzuwenden. Sowohl die Datentypen als auch die darauf erlaubten Operationen werden in der JVM Spezifikation genau vorgeschrieben.

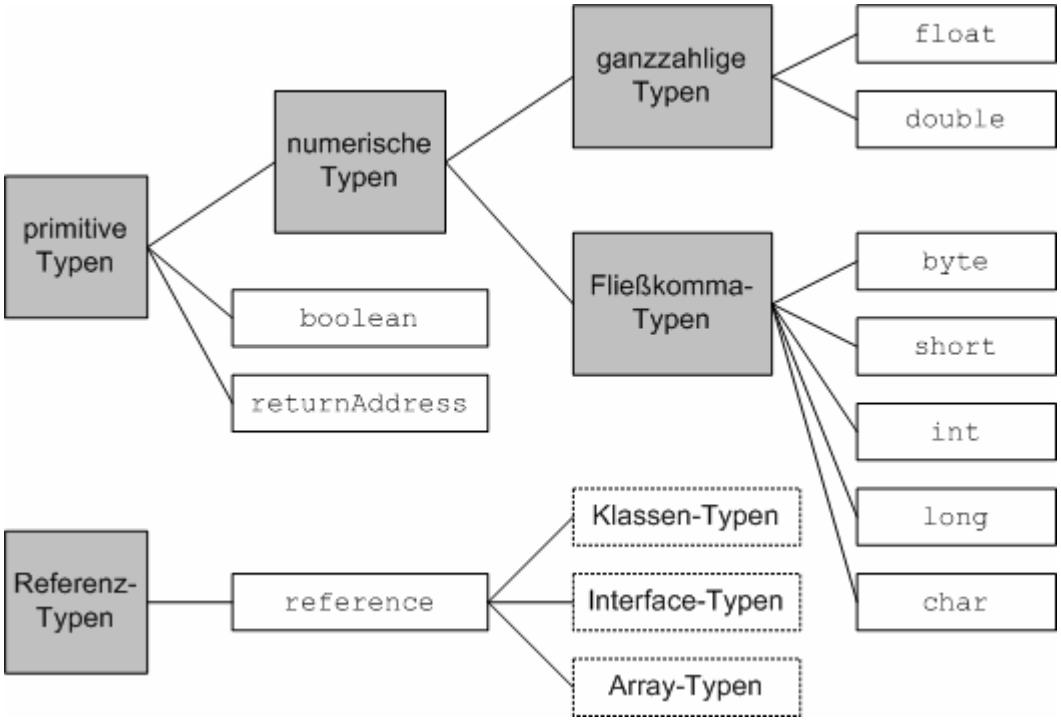

Abbildung 2: Datentypen der Java-VM

Abbildung 2 enthält alle Datentypen der JVM, unterteilt in mehrere Familien von ähnlichen Datentypen.

Die Datentypen können zunächst in *primitive Typen* – diese enthalten *primitive Werte* – und *Referenz-Typen* – diese enthalten *Referenz-Werte* – unterteilt werden. Während primitive Werte den Inhalt einer Variablen selbst enthalten – zum Beispiel Zahlen – verweisen Referenz-Werte auf den Inhalt einer Variablen. Referenz-Werte sind also selbst keine Objekte, verweisen allerdings auf Objekte.

Alle primitiven Typen, die in Java verwendet werden können, werden auch in der Java-VM unterstützt. Allerdings wird der Datentyp `boolean` zwar von der JVM unterstützt, doch gibt es keine Befehle, die ihn verwenden. Wenn ein Java-Programm in Byte-Code übersetzt wird, dann werden Variablen vom Typ `boolean` durch `int`s oder `byte`s dargestellt. Der Wert `false` wird dann durch den Zahlenwert 0 und der Wert `true` durch den Zahlenwert 1 dargestellt.

Alle aus Java bekannten primitiven Datentypen außer `boolean` werden unter dem Begriff *numerische Typen* zusammengefasst. Diese können wiederum in die *ganzzahligen Typen* – `byte`, `short`, `int`, `long` und `char` – und die

Fließkomma-Typen – `float` und `double` – unterteilt werden. Auf die Wertebereiche dieser Typen komme ich gleich noch zu sprechen.

Wie dem aufmerksamen Beobachter wahrscheinlich schon aufgefallen ist, gibt es in Abbildung 2 einen primitiven Datentyp, der in Java nicht erlaubt ist: `returnAddress`. Dieser Datentyp wird von der Java-VM nur intern benutzt und dient dazu, `finally`-Blöcke zu implementieren. Auf diesen Datentyp werde ich nicht näher eingehen, da er für das Verständnis der JVM-Architektur ziemlich unerheblich ist.

Der Referenz-Typ der Java-VM wird mit `reference` bezeichnet und stellt die Grundlage für 3 verschiedene Typen von Referenzen dar: *Klassen-Typen*, *Interface-Typen* und *Array-Typen*. All diese Typen enthalten Werte, die Referenzen auf Objekte sind. Der Unterschied ist: Klassen-Typen enthalten Referenzen auf Instanzen von Klassen, Interface-Typen enthalten Referenzen auf Klassen-Instanzen, die ein Interface implementieren, und Array-Typen enthalten Referenzen auf Arrays – die in der Java Virtual Machine ebenfalls durch Objekte dargestellt werden, darauf komme ich in einem anderen Kapitel zurück.

3.1. Wertebereiche der Datentypen

Die Spezifikation der Java Virtual Machine schreibt genau vor, welche Wertebereiche die einzelnen Datentypen haben müssen, jedoch macht sie keine Angaben über deren Größe in Bits. Diese Entscheidung bleibt also dem Implementierer überlassen, nur für die korrekten Wertebereiche muss er Sorge tragen.

Datentyp	Wertebereich
`byte`	8-bit 2er-Komplement Ganzzahl mit Vorzeichen (-2^7 bis $2^7 - 1$, inklusive)
`short`	16-bit 2er-Komplement Ganzzahl mit Vorzeichen (-2^{15} bis $2^{15} - 1$, inklusive)
`int`	32-bit 2er-Komplement Ganzzahl mit Vorzeichen (-2^{31} bis $2^{31} - 1$, inklusive)
`long`	64-bit 2er-Komplement Ganzzahl mit Vorzeichen (-2^{63} bis $2^{63} - 1$, inklusive)
`char`	16-bit 2er-Komplement Ganzzahl ohne Vorzeichen (0 bis $2^{16} - 1$, inklusive)
`float`	32-bit IEEE 754 Fließkommazahl mit einfacher Genauigkeit
`double`	64-bit IEEE 754 Fließkommazahl mit doppelter Genauigkeit

`returnAddress`	Adresse eines Befehls in derselben Methode
`reference`	Referenz auf ein Objekt am Heap oder `null`

Tabelle 1: Wertebereiche der Datentypen der Java-VM

Natürlich finden wir hier keinen Eintrag zum Datentyp `boolean`, da dieser in der Java-VM immer durch `int`s und `byte`s dargestellt wird.

3.2. Wort-Größe

In der Spezifikation der Java Virtual Machine wird die Größe eines *Wortes* häufig gebraucht, vor allem für die Beschreibung der Laufzeit-Speicherbereiche der JVM. Die Spezifikation gibt keine genaue Auskunft darüber, wie groß ein Wort sein muss, nur müssen folgende Vorgaben erfüllt sein:

- ein Wort muss groß genug sein, um einen Wert vom Typ `byte`, `short`, `int`, `char`, `float`, `returnAddress` oder `reference` zu beinhalten
- zwei Wörter müssen groß genug sein, um einen Wert vom Typ `long` oder `double` zu beinhalten

Daraus resultiert, dass ein Wort zumindest 32 Bits groß sein muss, der Implementierer kann aber davon abgesehen jede beliebige Größe wählen, je nachdem welche ihm am passendsten erscheint. Meistens wird die Wort-Größe der Ziel-Plattform angepasst und entspricht dort der Größe eines nativen Zeigers, da dies meist am effizientesten zu implementieren ist.

4. Das Class Loader Subsystem

Dieses Teilsystem der Java Virtual Machine übernimmt die Verantwortung für das Finden und Laden von Klassen, die für die Ausführung eines Java-Programms benötigt werden.

Wir werden uns im Folgenden die genauen Aufgaben eines *Class Loaders* sowie die beiden unterschiedlichen Typen von Class Loadern anschauen.

4.1. Aufgaben eines Class Loaders

Wie bereits erwähnt, ist das Class Loader Subsystem in erster Linie für das Finden einzelner Klassen und das Laden der darin enthaltenen Informationen zuständig. Darüber hinaus muss es sich allerdings auch um die Verifikation der Korrektheit der geladenen Klassen, um das Reservieren und Initialisieren von Speicher und um die *Resolution* von *symbolischen Referenzen* kümmern.

Die meisten werden sich nun allerdings nichts unter Resolution und symbolischen Referenzen vorstellen können, daher zunächst zu diesen beiden Begriffen.

4.1.1. Symbolische Referenzen und Resolution

Wenn die binären Informationen einer Klasse gelesen und im Speicher abgelegt werden, so werden zunächst alle Referenzen auf andere Objekte durch einfache (allerdings voll qualifizierte) Zeichenketten dargestellt – zum Beispiel „`java.lang.Class`“. Diese Referenzen werden **symbolische Referenzen** genannt.

Wenn nun diese symbolischen Referenzen durch „echte“ Referenzen auf Objekte im Speicher ersetzt werden, so wird dieser Vorgang **Resolution** genannt.

4.1.2. Ablauf eines Ladevorgangs (Laden, Linken, Initialisieren)

Der Vorgang des Ladens einer Klasse mitsamt allem was dazugehört läuft immer nach folgendem Schema ab:

1. Loading: zunächst wird nach den binären Daten einer Klasse gesucht und sobald diese gefunden werden, werden sie importiert
2. Linking: wird in *Verifikation*, *Vorbereitung* und *Resolution* unterteilt:
   a. Verification: hier wird die Korrektheit der Klasseninformationen verifiziert
   b. Preparation: hier wird Speicher für Klassenvariablen reserviert und mit Startwerten initialisiert
   c. Resolution: hier werden die symbolischen Referenzen der Klasse durch direkte ersetzt (optional)
3. Initialization: schließlich werden alle Klassenvariablen mit ihren zugehörigen Startwerten initialisiert

4.2. Die 2 verschiedenen Arten von Class Loadern

Die Java Virtual Machine unterscheidet 2 verschiedene Arten von Class Loadern: den *Bootstrap Class Loader* und *User-Defined Class Loader*. Die Unterschiede zwischen diesen beiden Typen von Class Loadern werden wir uns nun im Folgenden anschauen.

4.2.1. Der Bootstrap Class Loader

Jede Java-VM muss zumindest imstande sein, Klasseninformationen aus binären Dateien zu lesen, die gemäß dem Format der `class`-Dateien laut Spezifikation aufgebaut sind. Es können noch beliebige andere Möglichkeiten zum Laden von Klasseninformationen von einer JVM zur Verfügung gestellt werden, aber das Auffinden und Auslesen von `class`-Dateien ist in der Spezifikation vorgeschrieben.

Aus diesem Grund muss jede Java-VM einen *Bootstrap Class Loader* haben, der Teil der JVM ist und `class`-Dateien auslesen kann. Wie der Bootstrap Class Loader Klassen auffindet, ist in der Spezifikation nicht vorgeschrieben, diese Entscheidung wird dem Implementierer überlassen.

Eine Möglichkeit, Klassen aufzufinden, wird durch Suns Java HotSpot VM realisiert. Diese JVM durchsucht Verzeichnisse, die in einer globalen Umgebungsvariablen namens `CLASSPATH` gespeichert sind, bzw. Verzeichnisse, die per Kommandozeile übergeben werden. Wenn es eine Datei mit dem Namen der Klasse und der Endung „`.class`“ findet, so werden die darin enthaltenen Informationen importiert. Allerdings müssen sich Klassen gemäß ihrem Package-Namen in den entsprechenden Unterverzeichnissen befinden, um gefunden zu werden.

Bis zur Version 1.2 der Java-VM von Sun suchte der Bootstrap Class Loader sowohl im Verzeichnis der System-Klassen (Klassen der Java-API) als auch im `CLASSPATH`. Ab Version 1.2 sucht der Bootstrap Class Loader nur noch im Verzeichnis der System-Klassen. Für das Durchsuchen des `CLASSPATH` stellt die Java-VM von Sun nun einen eigenen Class Loader zur Verfügung, den so genannten *System Class Loader*. Dieser ist allerdings ein *User-Defined Class Loader*, der automatisch beim Starten einer JVM-Instanz erzeugt wird. Somit wären wir beim nächsten Thema:

4.2.2. User-Defined Class Loaders

Ein *User-Defined Class Loader* wird vom Programmierer entworfen (daher der Name) und ist daher nicht Teil der Java-VM, sondern Teil einer Java-Applikation. Ein benutzerdefinierter Class Loader wird immer von der Klasse `java.lang.ClassLoader` abgeleitet und erhält mittels folgender Methoden Zugriff auf das Class Loader Subsystem der Java Virtual Machine:

```
Class defineClass(...);              // 2 Überladungen
Class findSystemClass(String name);
void  resolveClass(Class c);
```

Mit den beiden überladenen Methoden `defineClass()` kann der Programmierer eine neue Klasse erzeugen, die die von ihm übergebenen binären Daten enthalten soll. Jede JVM-Implementierung muss sicherstellen, dass mit diesen beiden Methoden neue Klassen in der *Method Area* (dazu später) erzeugt werden können.

Mit der Methode `findSystemClass()` kann der Programmierer den Bootstrap bzw. System Class Loader anweisen, eine bestimmte Klasse zu laden – falls dies noch nicht geschehen ist – und eine Referenz auf das zugehörige `Class` Objekt anfordern. Wenn die Klasse nicht geladen werden kann, dann wird eine `ClassNotFoundException` geworfen, um dies dem Programmierer anzuzeigen. Jede JVM-Implementierung muss sicherstellen, dass mit dieser Methode der Bootstrap bzw. System Class Loader aufgerufen werden kann.

Die Methode `resolveClass()` dient dazu, eine bestimmte Klasse zu linken – falls dies noch nicht geschehen ist. Die Methode `defineClass()` dient ja nur dazu, Klassen zu laden, aber diese müssen deswegen noch nicht gelinkt oder initialisiert sein. Jede JVM-Implementierung muss sicherstellen, dass diese Methode veranlassen kann, dass Klassen gelinkt werden.

4.3. Name Spaces

Jeder Class Loader verwaltet seinen eigenen *Name Space*, in dem alle Klassen liegen, die dieser Class Loader geladen hat. Daher kann eine Java-Applikation ein und dieselbe voll qualifizierte Klasse mit verschiedenen Class Loadern laden. Daraus resultiert, dass auf der Ebene der Java Virtual Machine eine voll qualifizierte Klassenbezeichnung nicht unbedingt eindeutig sein muss. Es ist unter Umständen nötig, auch den Class Loader zu kennen, der die Klasse geladen hat.

Aus diesem Grund speichert die JVM zu den Daten für jede Klasse auch den Class Loader, der die jeweilige Klasse geladen hat. Wenn die JVM nun eine symbolische Referenz von Klasse A auf eine andere Klasse B auflösen muss, dann verlangt sie diese andere Klasse B von demselben Class Loader, der die Klasse A bereits geladen hat.

Genauer will ich auf das Konzept der Name Spaces gar nicht eingehen, ich wollte sie nur der Vollständigkeit halber erwähnen.

5. Die Method Area

Alle Informationen über Typen und deren Methoden und Felder, sogar die Werte statischer Variablen, werden innerhalb der Java Virtual Machine in der *Method Area* gespeichert. Nachdem ein Class Loader eine Klasse gefunden und die binären Daten ausgelesen hat, übergibt er diese der JVM und diese speichert die Informationen in der Method Area. Wie die Informationen in der Method Area gespeichert werden oder wie die Method Area im Speicher aussehen soll, wird in der Spezifikation nicht vorgegeben, diese Entscheidung muss der Implementierer treffen (wie schon so oft).

Wenn die Java-VM Informationen über Typen, Methoden oder Felder benötigt, durchsucht sie die Method Area nach diesen Informationen. Aus diesem Grund sollten Implementierer für die Informationen der Method Area Datenstrukturen verwenden, die sehr schnell durchsucht werden können. Bei der Wahl der Datenstrukturen und der Organisation sind dem Implementierer keine Grenzen gesetzt, er kann entscheiden was für seine Zwecke am besten geeignet ist.

Als nächstes werden wir uns ansehen, welche Informationen zumindest in der Method Area gespeichert werden müssen.

5.1. Basis-Typ-Informationen

Für jeden Typ, der in die Method Area geladen wird, muss die Java Virtual Machine zumindest folgende Basis-Informationen speichern:

- den voll qualifizierten Namen des Typs
- den voll qualifizierten Namen der direkten Super-Klasse (außer der Typ ist ein Interface oder die Klasse `java.lang.Object`)
- ist der Typ eine Klasse oder ein Interface
- die Modifizierer des Typs (`public`, `abstract`, `final`)
- eine geordnete Liste der voll qualifizierten Namen aller direkten Super-Interfaces

5.2. Erweiterte Typ-Informationen

Zusätzlich zu den Basis-Informationen für jeden Typ müssen auch folgende erweiterte Informationen gespeichert werden:

- der Constant Pool für diesen Typ (dazu gleich mehr)
- Felder-Informationen
- Methoden-Informationen
- alle statischen Variablen des Typs, außer Konstanten
- eine Referenz auf ein Objekt des Typs ClassLoader
- eine Referenz auf ein Objekt des Typs Class

Diese Daten werde ich nun näher erklären.

5.2.1. Der Constant Pool

Für jeden Typ in der Method Area muss die Java-VM einen so genannten *Constant Pool* anlegen. Dieser stellt eine geordnete Liste aller Konstanten dar, die in diesem Typ verwendet werden, das heißt alle Literale (String-, Integer- und Fließkomma-Konstanten) und symbolische Referenzen auf Typen, Felder und Methoden. Der Constant Pool ist wie ein Array angelegt, wird also über Indizes angesprochen.

Die Verwendung des Constant Pools werde ich später in diesem Kapitel noch an einem Beispiel verdeutlichen.

5.2.2. Felder-Informationen

Für jedes Feld, das in einem Typ deklariert wird, müssen folgende Informationen gespeichert werden:

- der Name des Feldes
- der Typ des Feldes
- die Modifizierer des Feldes (`public`, `private`, `protected`, `static`, `final`, `volatile`, `transient`)

Außerdem muss die Reihenfolge, in der die Felder deklariert werden, festgehalten werden.

5.2.3. Methoden-Informationen

Für jede Methode, die in einem Typ deklariert wird, müssen folgende Informationen gespeichert werden:

- der Name der Methode
- der Rückgabewert der Methode (oder `void`)
- Anzahl und Typen der Parameter (in Reihenfolge der Deklaration)
- die Modifizierer der Methode (`public`, `private`, `protected`, `static`, `final`, `synchronized`, `native`, `abstract`)

Genau wie bei Feldern muss die Reihenfolge, in der die Methoden deklariert werden, festgehalten werden.

Für Methoden, die weder abstrakt noch nativ sind, müssen außerdem folgende Informationen gespeichert werden:

- der Byte-Code der Methode
- die Größe des *Operanden Stack* und die Anzahl der *lokalen Variablen* des *Stack Frames* der Methode
- eine *Exception Table*

Auf die Begriffe *Operanden Stack*, *lokale Variablen* und *Stack Frame* werde ich im Kapitel über den *Java Stack* noch zurückkommen.

Die *Exception Table* werde ich nicht näher erklären, da sie nicht wirklich in den Bereich dieser Arbeit fällt.

5.2.4. Statische Variablen

Da statische Variablen (Klassenvariablen) zu einer Klasse gehören, und nicht zu den Instanzen einer Klasse, gehören sie logisch gesehen zu den Typ-Informationen dazu und werden daher auch in der Method Area gespeichert. Bevor also eine Klasse von der Java-VM verwendet wird, muss die JVM Speicher für alle nicht finalen (!) statischen Variablen reservieren.

Statische Variablen, die als final deklariert werden – also Konstanten – werden allerdings anders behandelt. Wie bereits erwähnt, werden diese im Constant Pool des Typs gespeichert. Somit werden zwar sowohl finale als auch nicht-finale statische Variablen in der Method Area gespeichert, nur an unterschiedlichen Positionen.

5.2.5. Referenz auf ein Objekt des Typs `ClassLoader`

Die Java-VM muss für jeden geladenen Typ festhalten, ob dieser vom Boostrap oder von einem User-Defined Class Loader geladen wurde. Wenn nun ein Typ von einem User-Defined Class Loader geladen wurde, so muss in der Method Area eine Referenz auf diesen Class Loader gespeichert werden.

Wie bereits erwähnt, fordert die Java-VM referenzierte Typen immer von demselben Class Loader an, der den Typ mit dieser Referenz geladen hat. Dazu benutzt sie diese Referenz auf den Class Loader. Außerdem kann mithilfe dieser Referenz ein Typ eindeutig über alle Name Spaces hinweg identifiziert werden.

5.2.6. Referenz auf ein Objekt des Typs `Class`

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

Java-Applikationen können Referenzen auf `Class`-Objekte anfordern und so Informationen über Klassen herausfinden. Mit folgender statischer Methode der Klasse `java.lang.Class` kann eine Applikation über den voll qualifizierten Namen eine Referenz für eine beliebige Klasse anfordern:

```
static Class forName(String className);
```

Diese Methode liefert eine Referenz auf die angegebene Klasse zurück, falls diese geladen werden konnte oder bereits geladen war und wirft eine `ClassNotFoundException`, wenn die Klasse nicht geladen werden konnte.

Eine andere Möglichkeit, eine `Class`-Referenz zu erhalten, ist, die Methode `getClass()` auf einem beliebigen Objekt anzuwenden, da jedes Objekt diese Methode von `java.lang.Object` erbt.

```
final Class getClass();
```

Wenn man nun die gewünschte `Class`-Referenz in den Händen hält (nicht ganz wörtlich zu nehmen), kann man z.B. mit folgenden Methoden ganz einfach Informationen über den jeweiligen Typ in Erfahrung bringen:

```
String      getName();
Class       getSuperClass();
boolean     isInterface();
Class[]     getInterfaces();
ClassLoader getClassLoader();
```

Dem aufmerksamen Beobachter fällt an dieser Stelle wahrscheinlich auf, dass die Java-Applikation durch die Methoden von `java.lang.Class` Zugriff auf die in der Method Area gespeicherten Informationen erhält.

Auf genauere Erklärungen zu diesen Methoden werde ich hier verzichten, da die Namen der Methoden für sich sprechen sollten. Interessierten Lesern empfehle ich an dieser Stelle den Blick in die Java-API Dokumentation!

5.3. Method Tables

Natürlich sollten alle Informationen in der Method Area so schnell wie möglich zugänglich sein. Dies gilt im Besonderen auch für die einzelnen Methoden von nicht-abstrakten Klassen, da diese in der Regel recht häufig aufgerufen werden.

Eine Möglichkeit, einfachen und schnellen Zugriff auf die Methoden einer Klasse zu gewährleisten sind **Method Tables**. Method Tables sind Arrays, die zu jeder Methode, die auf der Klasse aufgerufen werden kann – also auch Methoden, die von Super-Klassen vererbt wurden – eine direkte Referenz zur entsprechenden Instanz in der Method Area besitzen.

Wenn nun die Java-VM bei jedem Laden eines Typs so eine Method Table erzeugt, kann sie beim Aufruf einer Methode viel schneller die entsprechenden Informationen und den zugehörigen Byte-Code auffinden.

Die Implementierung von Method Tables ist nur eine Möglichkeit, die Effizienz zu steigern, allerdings auf Kosten des Speicherplatzes, der durch die Tabelle belegt wird. Welcher Aspekt nun für seine Zwecke wichtiger ist, muss der Implementierer entscheiden.

5.4. Beispiel zur Benutzung der Method Area

Anhand eines kleinen Beispiels möchte ich verdeutlichen, wie die Java Virtual Machine die Informationen der Method Area verwendet. Wir haben folgende Klassen gegeben:

```
class Car {
  private int nOfDoors = 4;
  void drive() {}
}
class Example {
  public static void main(String[] args) {
    Car car = new Car();
    car.drive();
  }
}
```

Im Folgenden möchte ich beschreiben, wie eine Implementierung der Java-VM die erste Instruktion der Methode `main()` der Klasse `Example` abarbeiten könnte:

Um die Java-Applikation zu starten, muss der JVM zunächst der Name der Klasse mit der `main()` Methode übergeben werden – bei Suns JVM passiert dies zum Beispiel über Kommandozeilen-Parameter. Die JVM, genauer gesagt das Class Loader Subsystem der JVM, sucht daraufhin die Datei „`Example.class`“, liest die binären Daten der Klasse `Example` aus und speichert diese in der Method Area.

Die JVM erzeugt nun einen Constant Pool für die Klasse `Example` in der Method Area und speichert einen Zeiger darauf bei den Typ-Informationen.

Als nächstes beginnt die JVM mit der Ausführung der Methode `main()`, indem sie den zugehörigen Byte-Code interpretiert. Wie dem aufmerksamen Leser nicht entgangen sein dürfte, sind bei dieser Java-VM zum Zeitpunkt der Ausführung der Applikation noch nicht alle Klassen geladen, die in der Applikation verwendet werden (`Car`). Tatsächlich werden bei den meisten Implementierungen der JVM Klassen nur dann geladen, wenn sie wirklich benötigt werden.

Die erste Instruktion von `main()` weist die Java-VM an, genügend Speicher für die Klasse an der Stelle 1 im Constant Pool zu reservieren. Die JVM benutzt nun den Zeiger auf den Constant Pool von `Example`, liest den ersten Eintrag und findet eine symbolische Referenz auf die Klasse `Car`. Daraufhin durchsucht die JVM die Method Area, ob die Klasse `Car` bereits geladen wurde.

Sobald die Java-VM bemerkt, dass die Klasse `Car` noch nicht geladen wurde, sucht sie die Datei „`Car.class`“ und importiert deren binäre Daten in die Method Area. Spätestens an dieser Stelle sollte dem Implementierer klar werden, dass er für die Suche nach Typ-Informationen in der Method Area eine möglichst effiziente Datenstruktur verwenden sollte, da die Suche nach Typ-Informationen zu einem voll qualifizierten Namen – z.B. „`java.lang.Object`“ – recht häufig vorkommt.

Als nächstes ersetzt die Java-VM den Eintrag an der Stelle 1 im Constant Pool von `Example` – im Moment die Zeichenkette „`Car`“ – durch einen Zeiger auf die Typ-Informationen zur Klasse `Car`. Wenn die Java-VM wieder einmal den ersten Eintrag dieses Constant Pools benutzen sollte, so muss sie nicht wieder die gesamte Method Area nach der Klasse `Car` durchsuchen, sondern kann gleich diesen Zeiger verwenden, um zu den Typ-Informationen zu gelangen. Dieser Vorgang des Ersetzens von symbolischen Referenzen im Constant Pool durch direkte Referenzen wird **Constant Pool Resolution** genannt.

Nun ist die Java Virtual Machine bereit, Speicher für die Klasse `Car` zu reservieren. Die JVM benutzt also den eben installierten Zeiger vom Constant Pool zu den Typ-Informationen von `Car`, um herauszufinden, wie viel Speicher auf dem Heap für ein Objekt dieses Typs benötigt wird. An dieser Stelle ist anzumerken, dass die tatsächliche Größe auf dem Heap von der Implementierung der Java-VM abhängt, denn wie die Objekte auf dem Heap gespeichert werden, muss der Implementierer entscheiden (wieder einmal). Aber dazu komme ich später noch.

Sobald die JVM die Größe dieses Objekts festgestellt hat, reserviert sie den benötigten Speicher auf dem Heap und initialisiert die Instanz-Variable `nOfDoors` auf den Default-Wert 0 (!). Falls die Super-Klasse von `Car`, also die Klasse `Object` ebenfalls Instanz-Variablen besitzt, so werden diese ebenfalls auf Default-Werte (0, 0.0, `null`) gesetzt.

Die erste Instruktion von `main()` wird damit abgeschlossen, dass eine Referenz auf das eben erzeugte `Car`-Objekt auf den Stack gelegt wird. Eine spätere Instruktion wird diese Referenz benutzen, um die Variable `nOfDoors` auf ihren Initialisierungs-Wert 4 zu setzen und eine noch spätere Instruktion wird mit dieser Referenz die Method `drive()` auf diesem Objekt aufrufen.

6. Der Heap

Sobald in einer Java-Applikation eine Objekt-Instanz oder ein Array erzeugt wird, reserviert die Java-VM den dafür benötigten Speicher auf dem *Heap*. Es gibt genau einen Heap in jeder JVM-Instanz, daher benutzen mehrere Threads einer Java-Applikation auch denselben Heap. Aus diesem Grund muss die Java Virtual Machine auch dafür sorgen, dass der Zugriff von mehreren Threads auf den Heap synchronisiert erfolgt.

Während die Java-VM zwar Instruktionen für das Reservieren von Speicher für Objekte auf dem Heap bereitstellt, gibt es allerdings keine Instruktionen, die Speicher wieder freigeben könnten. Der Programmierer kann also Objekte nicht explizit freigeben. Die JVM allein ist verantwortlich dafür, herauszufinden, welche Objekte nicht mehr benötigt werden, und ob bzw. wann der zugehörige Speicher freigegeben werden soll. Zu diesem Zweck wird meist ein *Garbage Collector* eingesetzt.

6.1. Garbage Collection

Die Aufgabe eines *Garbage Collectors* besteht darin, herauszufinden, welche Objekte in einer Applikation nicht mehr benötigt – also nicht mehr referenziert – werden und deren Speicher freizugeben. In weiterer Folge kann ein Garbage Collector auch Objekte verschieben, um Fragmentierung möglichst zu verhindern.

In der JVM-Spezifikation wird nicht zwingend ein Garbage Collector vorgeschrieben. Es wird nur verlangt, dass eine Java-VM ihren Heap selbst verwaltet, wie sie das macht, bleibt dem Implementierer überlassen. Eine Implementation könnte zum Beispiel gar keine Aufräumarbeiten machen und wenn der verfügbare Speicher zu Ende ist, eine entsprechende `Exception` werfen. Die Spezifikation sagt nur aus, dass Java-Applikationen Speicher anfordern, aber nicht mehr freigeben werden. Wie der Implementierer mit dieser Tatsache umgeht, ist ihm überlassen.

Der Implementierer kann, wenn er einen Garbage Collector implementiert, jeden ihm passenden Algorithmus zur Speicherbereinigung einsetzen, der Kreativität sind hier keine Grenzen gesetzt. Ich werde darauf nicht weiter eingehen, wenn sich jemand mehr für Garbage Collection interessiert, empfehle ich ihm das Buch **[Jon97]**, die **LVA Systemsoftware von Prof. Mössenböck** sowie die Suche im Internet.

6.2. Objekt-Darstellung am Heap

In der Spezifikation der Java-VM werden keine Angaben darüber gemacht, wie Objekte am Heap dargestellt werden sollen. Diese Überlegungen werden dem Implementierer überlassen.

Zu jedem Objekt müssen natürlich in erster Linie einmal seine Instanz-Daten gespeichert werden, also alle Instanz-Variablen der Klasse und all ihrer Super-Klassen. Diese Daten müssen von einer Objekt-Referenz aus schnell erreichbar sein. Außerdem müssen die Typ-Informationen zu einem Objekt

von einer solchen Referenz aus erreichbar sein. Aus diesem Grund wird meistens zu jedem Objekt ein Zeiger auf die Typ-Informationen in der Method Area gespeichert.

Der Zugriff auf die Typ-Informationen von einer Referenz aus muss aus mehreren Gründen möglich sein:

- wenn eine Java-Applikation eine Cast-Operation auf einem Objekt ausführt, muss die JVM überprüfen, ob der Typ, in den gecastet werden soll, dem Typ des Objekts oder einem der Super-Typen des Objekts entspricht
- wenn eine Java-Applikation eine `instanceof`-Operation durchführt, muss ebenfalls der Typ eines Objekts festgestellt werden können
- wenn eine Instanz-Methode aufgerufen wird, muss die JVM dynamische Bindung durchführen, also die Methode aufrufen, die zur tatsächlichen Klasse des Objekts gehört, und nicht zum Typ der Referenz

Im Klartext werden die Typ-Informationen immer benötigt, wenn die JVM den dynamischen Typ eines Objekts feststellen muss, und der statische Typ (der Typ der Referenz) nicht ausreicht.

Im Folgenden werde ich ein paar Möglichkeiten vorstellen, wie man die Darstellung der Objekte am Heap realisieren könnte:

6.2.1. Möglichkeit 1: Zweiteilung des Heaps

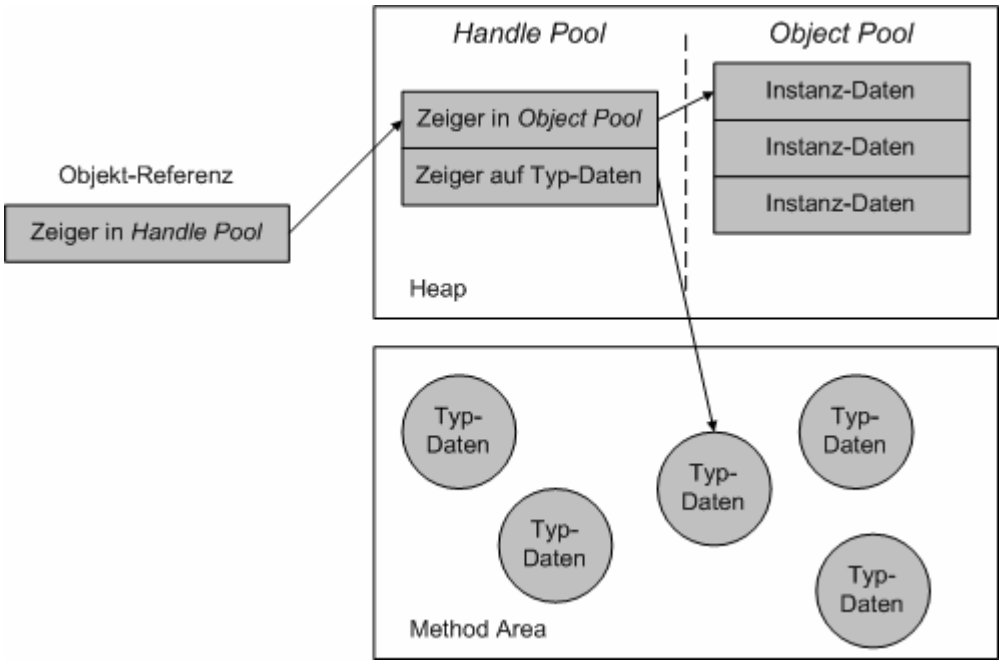

Abbildung 3: Aufteilung des Heaps in Handle und Object Pool

Eine Möglichkeit, Objekte auf dem Heap darzustellen, besteht darin, den Heap wie in Abbildung 3 in zwei Teile aufzuteilen: einen **Handle Pool** und einen **Object Pool**.

Eine Objekt-Referenz stellt einen Zeiger auf einen Eintrag im Handle Pool dar, der 2 Komponenten hat: einen Zeiger in den Object Pool und einen Zeiger zu den Typ-Informationen in der Method Area. Somit können die Typ-Informationen zu einem Objekt einfach erreicht werden.

Im Object Pool befinden sich die Instanz-Daten zu jedem Objekt, also sozusagen befinden sich die Objekte selbst an dieser Position im Heap.

☺ Der Vorteil dieser Darstellung liegt darin, dass Objekte im Object Pool einfach verschoben werden können, um die Fragmentierung möglichst niedrig zu halten. In diesem Fall braucht nur ein Zeiger (der Zeiger aus dem Handle Pool) umgeändert zu werden.

☹ Der Nachteil liegt natürlich darin, dass jeder Zugriff auf die Instanz-Daten eines Objekts erfordert, dass 2 Zeiger dereferenziert werden. Dieser Aufwand kostet natürlich zusätzliche Rechenzeit.

6.2.2. Möglichkeit 2: Alle Daten an einer Stelle

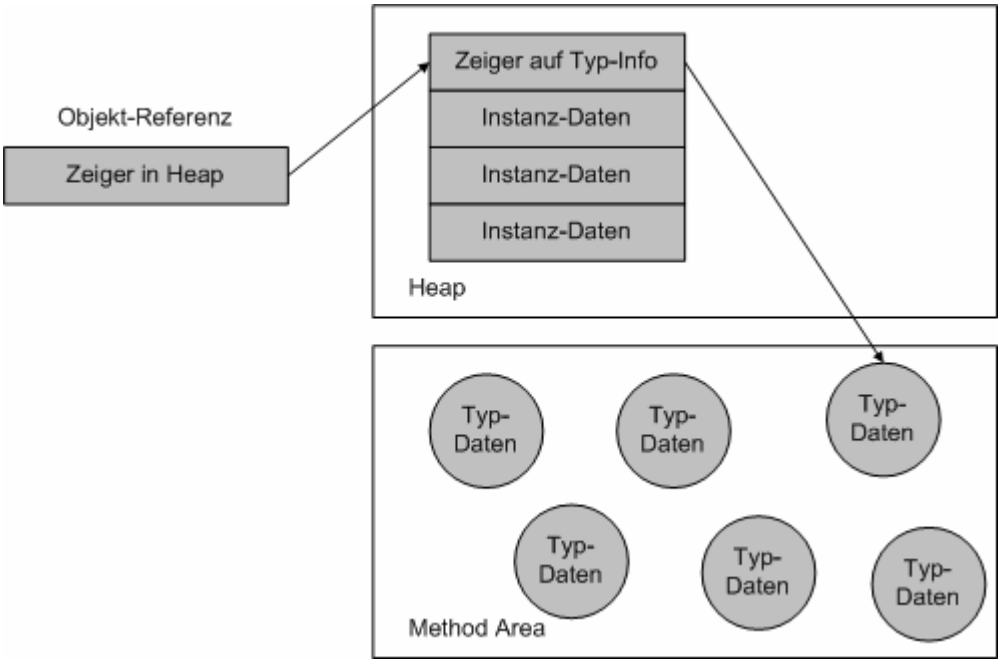

Abbildung 4: Alle Objekt-Daten an einer Stelle

Eine andere Möglichkeit der Objekt-Darstellung ist, wie in Abbildung 4 gezeigt, alle Objekt-Daten, also Instanz-Daten und den Zeiger auf die Typ-Informationen, an einer Stelle im Heap zu vereinen und Objekt-Referenzen direkt auf diese Daten zu verweisen.

☺ Der Vorteil dieser Darstellung liegt natürlich darin, dass nur ein Zeiger dereferenziert werden muss, wenn man auf die Instanz-Daten zugreifen will.

☹ Der Nachteil liegt allerdings darin, dass Objekte im Heap bei dieser Darstellung sehr aufwendig zu verschieben sind, da alle Referenzen

auf ein Objekt in allen Speicher-Bereichen geändert werden müssen, wenn ein Objekt im Heap verschoben wird.

6.2.3. Möglichkeit 3: Darstellung mit Method Tables

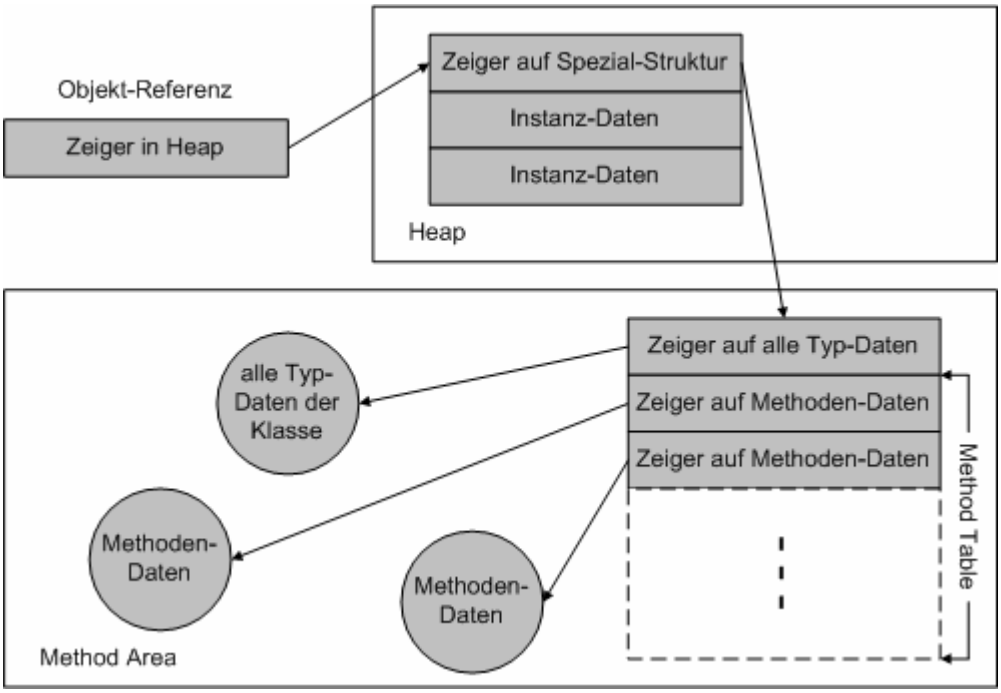

Abbildung 5: Objekt-Darstellung mit Method Tables

Wie bereits zuvor erwähnt, können Method Tables dazu beitragen, dass Java-Applikationen weitaus performanter ausgeführt werden. Abbildung 5 zeigt, wie man zum Beispiel die Objekt-Darstellung am Heap mit Method Tables realisieren könnte.

Diese Darstellung ähnelt der zuletzt vorgestellten Realisierung sehr, mit dem Unterschied, dass in diesem Fall auch die Method Table sehr schnell erreichbar ist und somit Methoden sehr schnell und einfach aufgerufen werden können.

6.3. Array-Darstellung am Heap

In Java werden Arrays wie Objekte behandelt – genauer gesagt sind Arrays in Java sogar Objekte – sie werden daher auch am Heap gespeichert und genauso wie bei Objekten bleibt es dem Implementierer überlassen, in welcher Weise er sie auf dem Heap speichert.

Wie jedes Objekt hat auch ein Array eine zu seinem Typ zugehörige Instanz vom Typ `Class`. Zwei Arrays haben genau dann denselben Typ, wenn sie denselben Datentyp und dieselbe Dimension haben. Zum Beispiel haben ein Array mit 100 Elementen vom Datentyp `int` und ein Array mit 50 Elementen vom Datentyp `int` denselben Typ. Die Länge bzw. die Längen der einzelnen

Dimensionen eines Arrays sind für den Typ nicht wichtig, denn die Länge wird als Teil der Instanz-Daten gespeichert.

Der Typ-Name eines Arrays setzt sich zusammen aus einer geöffneten eckigen Klammer „`[`“ für jede Dimension und einem Buchstaben bzw. einer Zeichenkette für den Datentyp:

- primitive Datentypen werden durch einen einzelnen (Groß-)Buchstaben repräsentiert, z. B.: `I` für `int`, `B` für `byte`, …

  ein 3-dim. `int`-Array wird also durch „`[[[I`“ gekennzeichnet

- Referenz-Typen werden durch ein „`L`“ und den voll qualifizierten Namen des Typs repräsentiert, z. B.: `Ljava.lang.Object`

  ein 2-dim. `Object`-Array wird also durch „`[[Ljava.lang.Object`“ gekennzeichnet

Mehr-dimensionale Arrays werden von der Java Virtual Machine als Arrays von Arrays dargestellt. Ein 2-dimensionales `byte`-Array, würde als 1-dimensionales Array aus Referenzen auf mehrere 1-dimensionale Arrays aus `byte`s dargestellt werden. Abbildung 6 zeigt, wie so ein Array auf dem Heap dargestellt werden könnte.

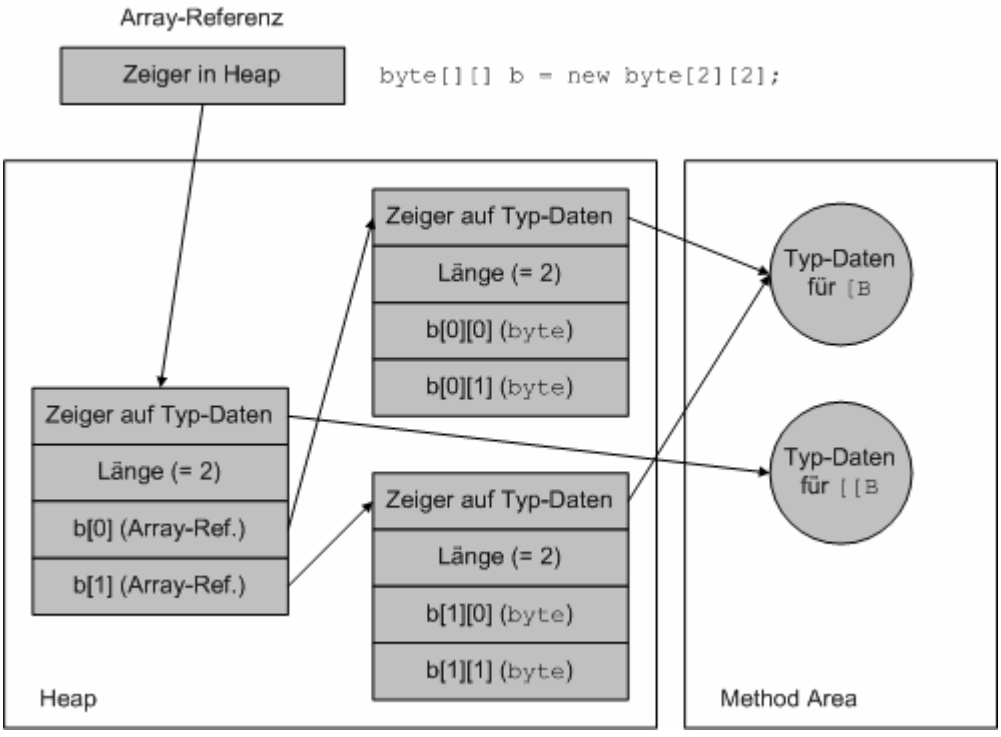

Abbildung 6: Mögliche Darstellung eines Arrays am Heap

Wie man in der Abbildung sieht, müssen für jedes Array folgende Informationen gespeichert werden:

- die Länge des Arrays

- die Daten des Arrays
- eine Referenz auf die Typ-Informationen des Arrays

Durch eine Referenz auf ein Array muss die Java-VM in der Lage sein, die Länge des Arrays herauszufinden, Informationen über den Typ des Arrays einzuholen und die Daten des Arrays über Indizes abzufragen oder festzulegen (nachdem sie überprüft hat, ob der Index gültig ist). Außerdem muss sie in der Lage sein, Methoden der Klasse `Object` aufzurufen, da diese die direkte Super-Klasse jedes Arrays ist.

7. Der Program Counter (PC)

Jeder Thread einer Java-Applikation hat einen so genannten Program Counter – das bedeutet übersetzt soviel wie „Befehlszähler“. Dieser wird erzeugt, sobald ein Thread gestartet wird.

Der PC ist genau ein Wort groß, und enthält die Adresse der Instruktion, die vom zugehörigen Thread gerade ausgeführt wird. Eine Adresse kann angegeben werden durch:

- einen nativen Zeiger oder
- einen Offset vom Beginn des Byte-Codes einer Methode

Wenn der zugehörige Thread gerade eine native Methode ausführt, also keine Java-Methode, so ist der Inhalt des Program Counters undefiniert.

8. Der Java Stack

Sobald ein neuer Thread gestartet wird, erzeugt die Java Virtual Machine einen eigenen *Java Stack* für diesen Thread. Auf einem solchen Java Stack wird der Zustand eines Threads in einzelnen *Stack Frames* gespeichert. Die einzigen beiden Operationen, die eine JVM auf dem Java Stack durchführt sind:

- das Stellen eines Stack Frames auf die Spitze des Stacks (`push`)
- das Holen eines Stack Frames von der Spitze des Stacks (`pop`)

Zu jeder Methode, die von einem Thread aufgerufen wird, gibt es einen Stack Frame. Zu jedem Zeitpunkt gibt es einen aktuellen Stack Frame, dieser liegt auf dem Java Stack ganz oben und gehört zur aktuell ausgeführten Methode eines Threads.

Sobald ein Thread eine Methode aufruft, erzeugt die JVM einen neuen Stack Frame und legt diesen auf die Spitze des Java Stacks (`push`). Dieser Frame wird also zum aktuellen Frame. Die eben aufgerufene (aktuelle) Methode benutzt diesen Stack Frame zum Speichern der Parameter, der lokalen Variablen, für Zwischenberechnungen und weitere Daten (dazu gleich mehr).

Es gibt für eine Methode 2 Möglichkeiten, wie sie beendet wird:

- durch normales Zurückkehren (*normal completion*)
- durch das Werfen einer Exception (*abrupt completion*)

Wenn eine Methode beendet wird, so oder so, wird der aktuelle Stack Frame verworfen, da er nicht mehr benötigt wird, und der vorherige Stack Frame wird aktuell.

Jeder Thread hat seinen eigenen Java Stack und er allein hat darauf Zugriff, es kann also kein anderer Thread, versehentlich oder nicht, darauf zugreifen. Aus diesem Grund braucht sich ein Java-Programmierer auch keine Gedanken darüber zu machen, lokale Variablen zu synchronisieren, wenn er mit mehreren Threads arbeitet.

Diese Stack Frames wollen wir uns nun ein wenig genauer ansehen.

8.1. Stack Frames

Jeder Stack Frame besteht aus 3 Teilen:

- den *lokalen Variablen*,
- dem *Operanden Stack* und
- den *Frame Daten*

Der Speicherbedarf der lokalen Variablen und des Operanden Stacks (angegeben in Wörtern) hängt von der jeweiligen Methode ab und wird bei der Kompilation in den `class`-Dateien gespeichert. Der Speicherbedarf der Frame Daten hängt von der Implementierung der Java Virtual Machine ab.

Wenn die JVM nun eine neue Methode aufruft, ermittelt sie den Speicherbedarf der lokalen Variablen und des Operanden Stacks, erzeugt

einen Stack Frame entsprechender Größe und legt diesen auf die Spitze des Java Stacks.

8.1.1. Bemerkungen zu den Datentypen

In den nächsten Kapiteln werden alle Größenangaben über Speicherbereiche in Wörtern gemacht. Dabei gelten folgende Grundregeln:

- Werte vom Typ `byte`, `short` oder `char` werden in den Typ `int` konvertiert, da dieser meist ein Wort groß ist (aber nie größer)
- Werte vom Typ `int`, `float`, `reference` oder `returnAddress` benötigen ein Wort Speicherplatz
- Werte vom Typ `long` oder `double` benötigen zwei Wörter Speicherplatz

8.1.2. Die lokalen Variablen

Wie der Name schon sagt, werden in diesem Teil eines Stack Frames die lokalen Variablen einer Methode gespeichert. Allerdings werden zusätzlich auch die Parameter gespeichert, die der Methode beim Aufruf übergeben wurden. Außerdem muss darauf geachtet werden, dass zuerst die Parameter gespeichert werden, und das in der Reihenfolge der Deklaration, und dann erst die lokalen Variablen.

Die lokalen Variablen sind wie ein 0-basiertes Array von Wörtern organisiert und Instruktionen, die auf lokale Variablen oder Methoden-Parameter zugreifen, greifen auf diese über den entsprechenden Index zu.

Datentypen, die zwei Wörter Speicherplatz benötigen, belegen also auch zwei Einträge in diesem Array. Angesprochen werden sie allerdings nur über den ersten Index.

8.1.3. Beispiel zu den lokalen Variablen

Die Verwendung der lokalen Variablen möchte ich anhand eines kleinen Beispiels anschaulicher erklären:

```
class Example {
  public static int staticMethod(int i, double d,
    Object o, String str, byte b, long l) {
    return 0;
  }
  public int instanceMethod(Class cl, long l,
    short s, boolean b, char c, float f) {
    return 0
  }
}
```

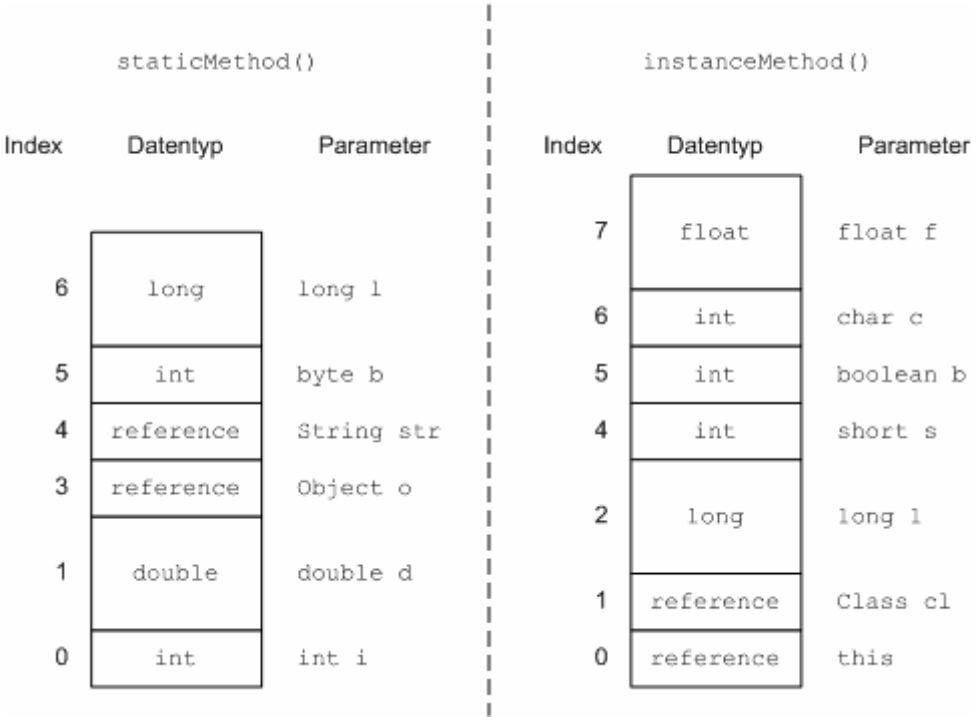

Abbildung 7: Methoden-Parameter auf dem Java Stack (lokale Variablen)

Anhand dieses kleinen Beispiels kann man schon einige Besonderheiten erkennen:

- Instanz-Methoden erhalten als ersten Eintrag in die lokalen Variablen eine Referenz, die auf das Objekt zeigt, auf dem die Methode aufgerufen wurde (`this`). Statische Methoden erhalten natürlich keine solche Referenz.
- Wie bereits zuvor erwähnt, werden die Datentypen `byte`, `short`, `char` und `boolean` in den Datentyp `int` konvertiert. Während dies bei `boolean` innerhalb der JVM immer der Fall ist, passiert dies bei den anderen Datentypen nur hier (und auf dem Operanden Stack). Während sich diese auf dem Stack Frame befinden, werden sie als `int`s behandelt, erst wenn sie wieder in die Method Area oder auf den Heap gelangen, werden sie zurückkonvertiert.
- Außerdem erkennt man, dass Objekte immer als Referenzen übergeben werden. Es kann sich nie ein Objekt bei den lokalen Variablen oder auf dem Operanden Stack befinden, nur Objekt-Referenzen.

Eine kleine Bemerkung noch zu den lokalen Variablen: während es zwingend vorgeschrieben ist, dass die Methoden-Parameter in der Reihenfolge der Deklaration in den lokalen Variablen gespeichert werden, können die lokalen Variablen einer Methode in beliebiger Reihenfolge gespeichert werden.

Es könnte theoretisch auch ein Platz in den lokalen Variablen für 2 lokale Variablen in der Methode verwendet werden, wenn sich deren

Gültigkeitsbereiche nicht überschneiden (zum Beispiel 2 nicht verschachtelte Schleifen, mit je einer lokalen Variablendeklaration).

8.1.4. Der Operanden Stack

Wie auch die lokalen Variablen, ist der Operanden Stack als Array von Wörtern organisiert. Nur zum Unterschied zu den lokalen Variablen, besteht hier kein wahlfreier Zugriff, sondern nur über `push` und `pop`, wie bei einem Stack üblich. Es können also nur Werte auf die Spitze gelegt und von dort wieder geholt werden.

Die Java Virtual Machine ist eine *Stack-Maschine*, das heißt sie besitzt keine Register (wie zum Beispiel Intel Prozessoren). In weiterer Folge bedeutet das, dass sich Instruktionen meistens ihre Operanden vom Operanden Stack holen. In seltenen Fällen können Operanden als *Opcode* im Byte-Code oder im Constant Pool vorkommen, aber der Befehlssatz der Java-VM ist in erster Linie auf den Operanden Stack ausgelegt.

Der Operanden Stack wird von der Java-VM für Berechnungen aller Art benutzt. Viele Byte-Code Instruktionen holen sich Werte vom Operanden Stack (`pop`), arbeiten mit diesen und stellen das Ergebnis wieder auf den Operanden Stack (`push`).

Ein Beispiel dafür wäre die Instruktion `fadd`. Diese holt 2 `float`s vom Operanden Stack, addiert diese und stellt die Summe wieder als `float` auf den Operanden Stack. Dazu gleich ein Beispiel:

8.1.5. Beispiel zum Operanden Stack

```
fload_0
fload_1
fadd
fstore_2
```

Diese Folge von Byte-Codes macht folgendes: durch `fload_0` und `fload_1` werden die beiden `float`-Werte an den ersten beiden Positionen (0 und 1) der lokalen Variablen auf den Operanden Stack geladen. Daraufhin werden durch `fadd` zwei `float`-Werte vom Operanden Stack geholt, diese addiert und das Ergebnis wieder als `float`-Wert auf den Operanden Stack gelegt. Und schließlich wird durch `fstore_2` der `float`-Wert auf dem Operanden Stack (das Ergebnis der Addition) in den lokalen Variablen an der Position 2 gespeichert.

In Abbildung 8 wird gezeigt, wie sich der Stack verändert, während diese Instruktionen ausgeführt werden:

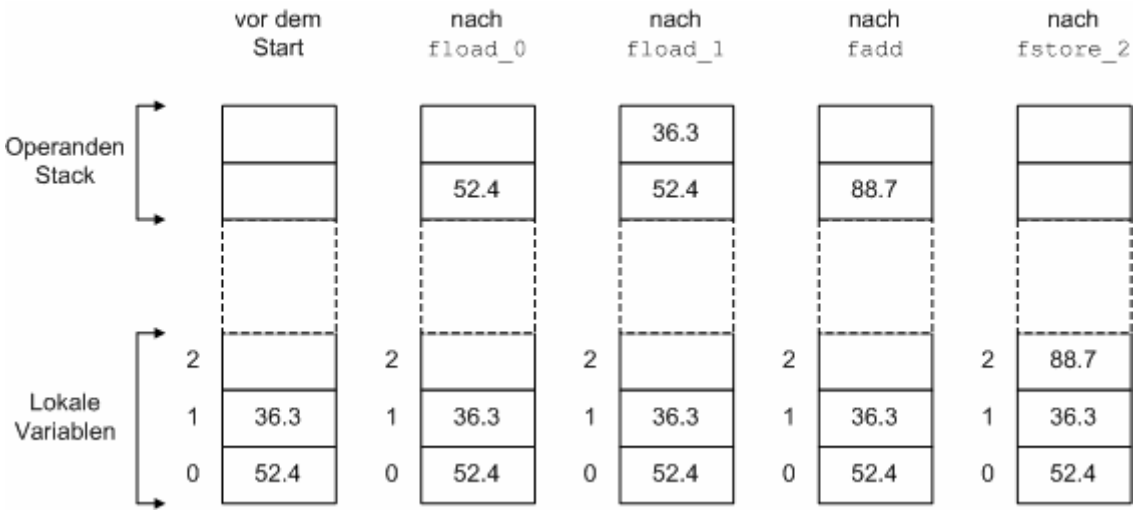

Abbildung 8: Addition zweier lokaler Variablen

8.1.6. Die Frame Daten

Zusätzlich zu den lokalen Variablen und dem Operanden Stack, enthält ein Stack Frame auch Daten, um die Constant Pool Resolution, normales Methodenende und Behandeln von Exceptions zu unterstützen. Diese Daten werden in den **Frame Daten** gespeichert.

Viele Instruktionen des JVM-Befehlssatzes verweisen auf Einträge im Constant Pool der aktuellen Klasse. Zum Beispiel holen manche Instruktionen konstante Werte aus dem Constant Pool und legen diese auf den Operanden Stack, während andere Instruktionen Constant Pool Einträge benutzen, um Klassen oder Arrays zu erzeugen, auf Felder zuzugreifen oder Methoden aufzurufen.

Sobald die Java Virtual Machine auf eine solche Instruktion trifft, benutzt sie einen entsprechenden Zeiger in den Frame Daten, um auf den aktuellen Constant Pool zuzugreifen. Wenn nun eine benötigte Referenz im Constant Pool noch symbolisch ist, muss die Java-VM zusätzlich noch Constant Pool Resolution durchführen.

Abgesehen davon müssen die Frame Daten die Java-VM außerdem beim normalen oder abrupten Ende einer Methode unterstützen:

- wenn eine Methode normal beendet wird, muss die JVM den vorhergehenden Stack Frame wiederherstellen, den PC auf die korrekte Instruktion setzen (eine Instruktion nach dem Aufruf der Methode) und eventuell den Rückgabewert der Funktion auf den Operanden Stack dieses vorhergehenden Stack Frames legen. Die Frame Daten halten die dazu notwendigen Informationen für die JVM bereit.
- wenn eine Methode abrupt beendet wird (also durch eine Exception), benutzt die Java-VM die *Exception Table*, auf die in den Frame Daten verwiesen wird, um festzustellen, ob die Exception durch einen passenden `catch`-Block abgefangen wird:

- wenn die Exception abgefangen wird, wird die Ausführung an der ersten Instruktion des `catch`-Blocks fortgesetzt.
- wenn die Exception nicht abgefangen wird, dann wird der aktuelle Stack Frame verworfen, der vorherige wiederhergestellt (mithilfe der Frame Daten) und dieselbe Exception im Kontext dieses Stack Frames neu geworfen. Der Vorgang wiederholt sich daraufhin.

8.2. Mögliche Implementierungen des Java Stacks

Wie bei den meisten Teilen der Java-VM, werden auch bei der Repräsentation des Java Stacks im Speicher dem Implementierer die meisten Entscheidungen überlassen. Anhand von folgendem Beispiel wollen wir uns 2 mögliche Implementierungs-Varianten ansehen:

```
class Example {
  public static void printSum(){
    float result = addFloats(34.2, 65.5);
    System.out.println(result);
  }
  public static float addFloats(float f1, float f2) {
    return f1 + f2;
  }
}
```

Bei diesem Beispiel wird eine Methode `addFloats()` aufgerufen, die zwei `float`-Werte addiert und das Ergebnis als `float` zurückliefert. Die Methode `printSum()`, in der die Addition der Zahlen 34.2 und 65.5 durchgeführt wird, gibt schließlich das Ergebnis auf dem Bildschirm aus. Wir wollen uns nun ansehen, wie der zugehörige Java Stack aussehen könnte.

8.2.1. Möglichkeit 1: Stack Frames hängen nicht zusammen

Der Java Stack muss nicht unbedingt als zusammenhängender Speicherbereich implementiert werden. Die einzelnen Stack Frames könnten zum Beispiel einzeln auf dem Heap angefordert werden und wenn sie nicht mehr referenziert werden, vom Garbage Collector – sofern vorhanden – eingesammelt werden.

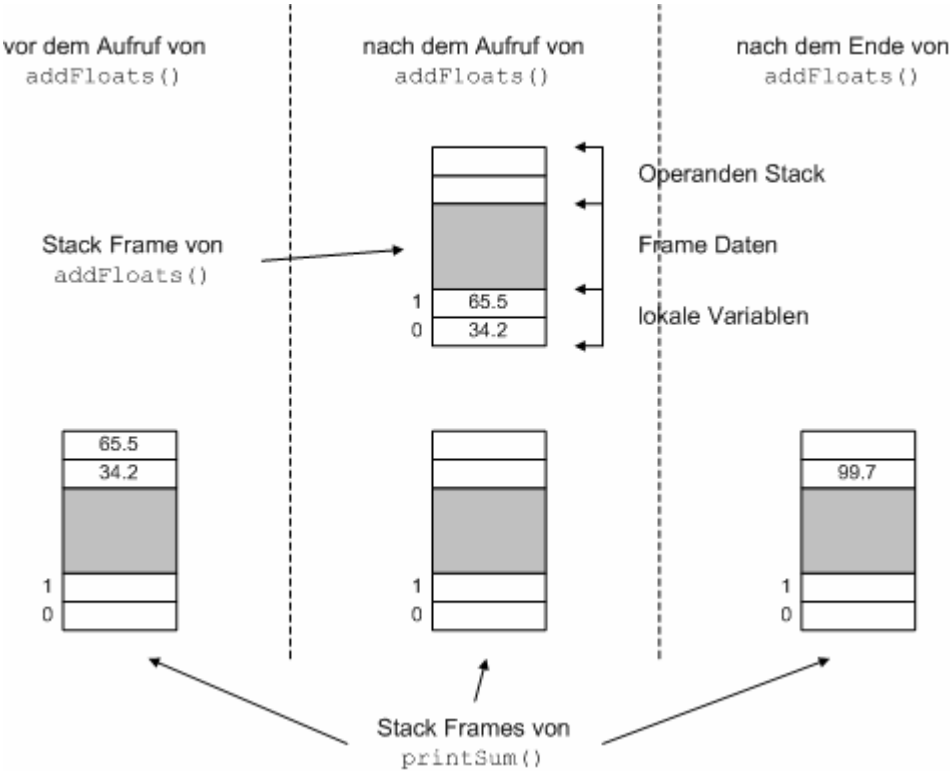

Abbildung 9: nicht zusammenhängende Stack Frames

In Abbildung 9 wird dieser Implementierungs-Ansatz dargestellt.

Zunächst wird die Methode `printSum()` ausgeführt. Um die Methode `addFloats()` nun aufzurufen, werden die beiden Werte 34.2 und 65.5 auf den Operanden Stack des aktuellen Stack Frames gebracht. Daraufhin wird die Methode `addFloats()` aufgerufen.

Die Instruktion zum Aufruf der Methode `addFloats()` verweist auf einen Eintrag im Constant Pool. Die Java-VM betrachtet diesen Eintrag und führt Resolution durch, falls dies notwendig ist. Man beachte, dass, obwohl diese beiden Methode Teil derselben Klasse sind, die Methode `printSum()` den Eintrag im Constant Pool benutzt, um die Methode `addFloats()` zu identifizieren. Und genauso wie bei Referenzen auf die Felder und Methoden anderer Klassen, können die Referenzen auf Felder und Methoden derselben Klasse anfangs symbolisch sein und in direkte transformiert werden müssen.

Jedenfalls zeigt der Eintrag im Constant Pool auf die Methoden-Informationen von `addFloats()` in der Method Area und die Java-VM holt sich von dort die Informationen über die Größe der lokalen Variablen und des Operanden Stacks der Methode `addFloats()`. Die JVM reserviert daraufhin genügend Speicher für den neuen Stack Frame vom Heap und holt die beiden Parameter (34.2 und 65.5) vom Operanden Stack von `printSum()` und legt sie in den lokalen Variablen von `addFloats()` an den Positionen 0 und 1 ab.

Nun werden die Byte-Code Instruktionen der Methode `addFloats()` ausgeführt und somit die beiden Zahlen 34.2 und 65.5 addiert. Sobald die Methode beendet wird, wird der Rückgabewert (99.7) auf den Operanden Stack von `addFloats()` gelegt. Die Java-VM benutzt die Frame Daten, um den vorherigen Stack Frame – den von `printSum()` – ausfindig zu machen, holt den Rückgabewert vom Operanden Stack von `addFloats()` und legt diesen auf den Operanden Stack von `printSum()`.

Zuletzt wird der Speicher des Stack Frames von `addFloats()` freigegeben – also nicht mehr referenziert in diesem Fall – und der Stack Frame von `printSum()` wird wieder zum aktuellen Stack Frame. Die Ausführung setzt nun an der Stelle nach dem Methoden-Aufruf von `addFloats()` fort.

8.2.2. Möglichkeit 2: Zusammenhängender Java Stack

Natürlich kann sich der Implementierer auch dafür entscheiden, den Stack als zusammenhängenden Speicherbereich zu realisieren – so wie die intuitive Vorstellung eines Stacks aussieht. Dieser Ansatz bringt außerdem noch ein paar Vorteile mit sich, auf die ich gleich anhand folgender Abbildung näher eingehen werde.

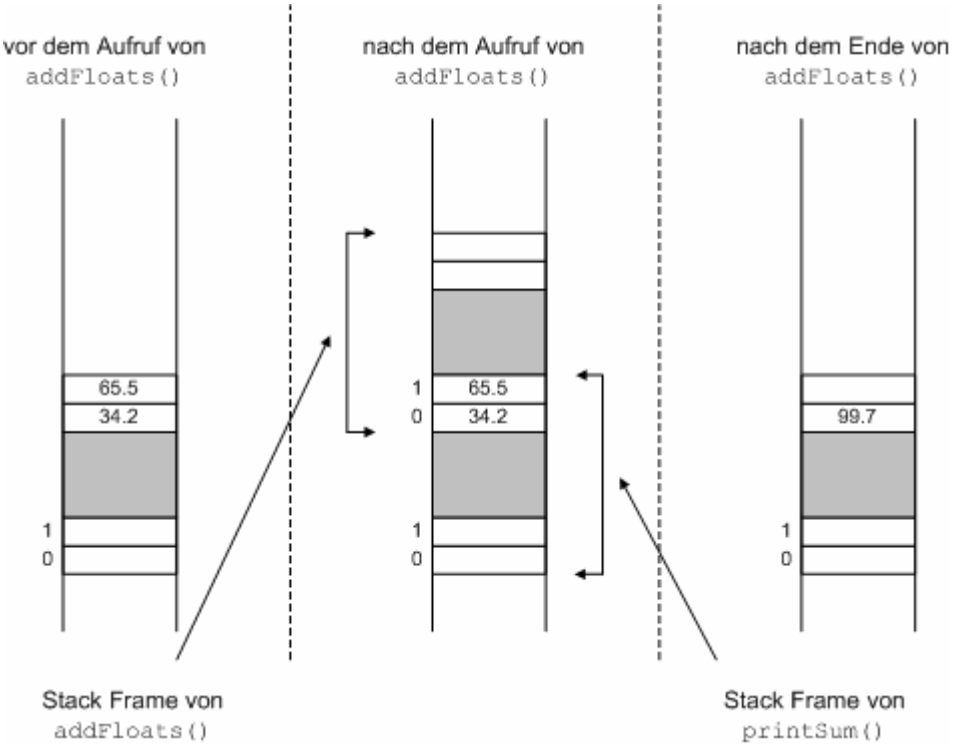

Abbildung 10: zusammenhängende Stack Frames

Bei einem Methoden-Aufruf werden zuerst die Parameter-Werte auf den Operanden Stack des alten Stack Frames gelegt und daraufhin werden diese in die lokalen Variablen des neuen Stack Frames kopiert. Wenn nun der Java Stack als zusammenhängender Speicherbereich realisiert wird,

dann können die Stack Frames einander überlappen und der Teil des alten Operanden Stack, der die Parameter enthält kann somit gleich als die lokalen Variablen des neuen Stack Frames verwendet werden.

In unserem Beispiel (Abbildung 10) wird der gesamte Operanden Stack von `printSum()` als die lokalen Variablen von `addFloats()` verwendet, da er keine weiteren Daten außer den Parametern enthält.

Dieser Ansatz spart sowohl Speicher, da der Platz für die Parameter nicht doppelt benötigt wird (in altem und neuem Stack Frame), als auch Zeit, da die Java-VM die übergebenen Parameter nicht von einem Frame zum nächsten kopieren muss.

In Abbildung 10 wird auch das Konzept des Stacks deutlicher dargestellt als zuvor. Der aktuelle Stack Frame liegt auf der Spitze des Java Stacks, auf die darunter liegenden Stack Frames ist kein Zugriff möglich. Und genauso verhält es sich mit dem Operanden Stack. Der aktuelle Operanden Stack befindet sich auf der Spitze des Stacks, und dieser ist auch der einzige, auf dem Operationen durchgeführt werden dürfen.

9. Native Method Stacks

Bisher haben wir immer nur über Java-Methoden gesprochen und wie diese behandelt werden. Wenn aber ein Thread nun eine *native* Methode ausführt, gelten alle Bestimmungen und Einschränkungen der Java Virtual Machine nicht mehr. Eine native Methode kann voraussichtlich auf die Datenstrukturen der Java-VM zugreifen – hängt vom *Native Method Interface* ab, dazu später – aber auch alle möglichen anderen Sachen machen, zum Beispiel auf Register des Prozessors zugreifen oder eigene Datenstrukturen erzeugen.

Wenn ein Implementierer ein Native Method Interface realisiert – ist in der JVM-Spezifikation allerdings nicht zwingend gefordert – also den Aufruf von nativen Methoden durch Java Threads zulässt, so wird er hierfür *Native Method Stacks* implementieren. Diese werden von den nativen Methoden benutzt, so wie Java Stacks von Java Methoden benutzt werden. Natürlich nicht unbedingt in derselben Art und Weise, diese hängt vom Native Method Interface ab.

Wenn nun eine native Methode aufgerufen wird, verlässt der Thread den Java Stack, die native Methode wird direkt aufgerufen und somit ein Native Method Stack betreten. Wie dieser Native Method Stack nun tatsächlich aufgebaut ist, hängt von der Implementierung des Native Method Interfaces ab, also wieder einmal vom Implementierer der Java-VM.

Während ein Native Method Interface natürlich den Aufruf von nativen Methoden aus Java Methoden gestattet, ist meist auch der umgekehrte Fall erlaubt, also der Aufruf von Java-Methoden von nativen Methoden aus. Dazu wollen wir uns ein Beispiel anschauen, dass diesen Sachverhalt hoffentlich verdeutlichen kann:

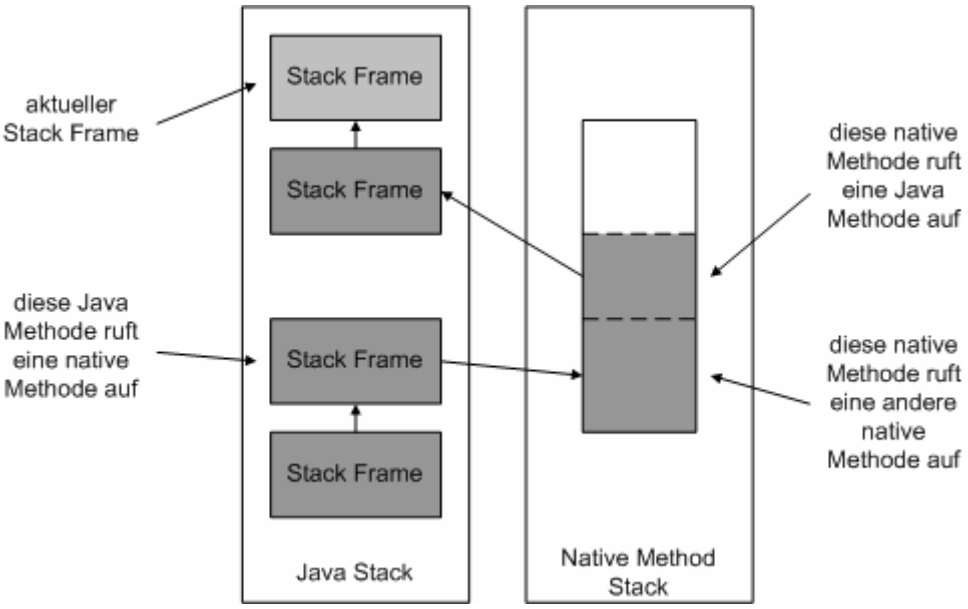

Abbildung 11: Java Stacks und Native Method Stacks

In diesem Beispiel führt ein Thread zuerst 2 Java Methoden aus, die dementsprechend 2 Stack Frames in einem Java Stack beanspruchen. Die 2.

Methode ruft eine native Methode auf, daher wird der Java Stack verlassen und die Java-VM benutzt weiters einen Native Method Stack. Diese native Methode ruft wiederum eine native Methode auf, diese allerdings ruft eine Java Methode auf (durch das Native Method Interface). Der Java Stack wird also fortgesetzt, es wird noch eine Java Methode aufgerufen und deren Frame ist der aktuelle Stack Frame.

So könnten Native Method Stacks zum Beispiel realisiert werden, dem Implementierer sind hierbei allerdings kaum Grenzen gesetzt.

10. Die Execution Engine

Das Herzstück jeder Java Virtual Machine ist die *Execution Engine*, die für das Ausführen der Java Byte-Codes zuständig ist. In der Spezifikation der Java-VM wird das Verhalten der Execution Engine anhand des Befehlssatzes der JVM definiert.

Für jede *Instruktion* wird genau bestimmt, *was* die Execution Engine machen soll, wenn sie diese Instruktion ausführt, jedoch wird in der Regel nichts darüber ausgesagt, *wie* sie das machen soll. Eine Execution Engine kann Byte-Codes interpretieren, nach Bedarf in nativen Code kompilieren, zu Beginn den gesamten Quelltext kompilieren, … dem Implementierer sind hier keine Grenzen gesetzt.

Ähnlich wie beim Begriff der Java-VM gibt es auch beim Begriff der Execution Engine 3 unterschiedliche Bedeutungen, zwischen denen man unterscheiden muss:

- die *abstrakte Spezifikation* der Execution Engine
- eine *konkrete Implementierung* der Execution Engine
- eine *Laufzeit-Instanz* der Execution Engine

Die **abstrakte Spezifikation** definiert das Verhalten einer Execution Engine, indem sie für jede einzelne Instruktion der Java-VM das Verhalten der Execution Engine vorgibt.

Konkrete Implementierungen sind entweder in Software, Hardware oder einer Kombination davon realisiert.

Und **Laufzeit-Instanzen** werden durch Threads dargestellt. Jeder Thread, der von einer Java-Applikation gestartet wird, stellt eine Instanz der Execution Engine dar und führt während seiner Lebenszeit immer Java Byte-Codes oder native Methoden aus.

10.1. Der Befehlssatz der Java-VM (Instruction Set)

Der Byte-Code einer Methode besteht aus einer Sequenz von *Instruktionen* für die Java Virtual Machine. Jede dieser Instruktionen besteht aus einem *Opcode*, gefolgt von keinem, einem oder mehreren *Operanden*.

Opcodes geben der Java-VM an, welche Operation ausgeführt werden soll, **Operanden** stellen die für diese Operation nötigen Informationen zur Verfügung. Ob bzw. wie viele Operanden eine Instruktion hat, wird durch den Opcode festgelegt, genauso wie der Typ der Operanden.

Eine Instanz einer Execution Engine (ein Thread) führt immer genau eine Instruktion zu einem Zeitpunkt aus. Die Execution Engine holt sich einen Opcode und eventuell die zugehörigen Operanden und führt die dadurch spezifizierte Operation aus. Dieser Vorgang wird wiederholt, bis der Thread (normal oder abrupt) beendet wird.

Genauer werde ich hier nicht auf den Befehlssatz der JVM eingehen, ich verweise zu diesem Thema auf die Lektüre von **[Yel99]**.

10.2. Techniken der Execution Engine

Im Laufe der Zeit haben sich verschiedene Techniken entwickelt, wie Java Byte-Codes von einer Execution Engine ausgeführt werden können. Die Spezifikation macht ja bekanntlich keine Aussagen darüber, wie Byte-Codes ausgeführt werden sollen, dem Implementierer werden hier alle Möglichkeiten offen gelassen.

Die wichtigsten heute verwendeten Techniken werden wir uns im Folgenden kurz anschauen.

10.2.1. Interpretation

Die erste – und auch einfachste – Technik, die von Java Virtual Machines zur Ausführung von Byte-Codes benutz wurde, war die *Interpretation*. Bei der Interpretation wird einfach eine Instruktion aus dem Byte-Code geholt und diese ausgeführt. Es erfolgt keine Kompilation in nativen Code und somit auch keine (oder zumindest kaum) Optimierung.

Diese Technik wird heutzutage von aktuellen Java-VMs nicht mehr durchgeführt, da sie einfach viel zu langsam für heutige Ansprüche ist.

10.2.2. Just-In-Time Kompilation

Bei dieser – schon fortgeschrittenen – Technik werden Methoden erst dann in nativen Code übersetzt, wenn sie wirklich benötigt, also aufgerufen werden. Bei einem erneuten Aufruf derselben Methode liegt diese dann schon in nativen Code vor. Es lassen sich bei dieser Methode bereits einige Optimierungen durchführen, da dem Compiler zur Laufzeit bereits Informationen über das Verhalten des Codes vorliegen.

Diese Technik wird heutzutage recht häufig verwendet, da sie relativ einfach zu implementieren ist und dennoch Möglichkeiten für Optimierungen bietet.

10.2.3. Adaptive Optimization

Bei dieser – sehr fortgeschrittenen – Technik, werden *Interpretation* und *JIT Kompilation* vereinigt, um maximale Performanz zu erzielen. Zunächst werden Instruktionen durch Interpretation ausgeführt, allerdings beobachtet die Java-VM das Verhalten des Codes.

Studien zufolge wird in den meisten Programmen 80 – 90 % der Ausführungszeit in 10 – 20 % des Codes verbracht. Indem die Java Virtual Machine nun die Ausführung des Codes beobachtet, kann sie diese „Hot Spots“ auffinden.

Wenn eine JVM zu dem Schluss gekommen ist, dass eine Methode in einem solchen „Hot Spot“ liegt, dann wird diese Methode (im Hintergrund) von der JVM mit starker Optimierung in nativen Code kompiliert. Dadurch, dass nur „Hot Spots“ kompiliert werden, erhält die JVM mehr Zeit, um Optimierungen durchzuführen.

Diese Technik wird heutzutage oft verwendet, auch von Suns HotSpot VM.

11. Das Native Method Interface

Wie bereits zuvor erwähnt, müssen Implementierungen der Java Virtual Machine nicht unbedingt ein *Native Method Interface* haben. Allerdings können Java-Applikationen dann natürlich auch keine nativen Methoden aufrufen.

Suns Native Method Interface in der HotSpot VM nennt sich Java Native Interface (JNI). Dieses ist in erster Linie auf Portabilität ausgerichtet, sodass jede Implementierung der Java-VM mit jeder beliebigen Objekt-Darstellung und jedem beliebigen Garbage Collection Algorithmus diese Schnittstelle unterstützen kann. Implementierer müssen sich natürlich nicht an das JNI halten, sie können stattdessen oder zusätzlich auch ihre eigenen Native Method Interfaces entwerfen, nur geht damit auch die Kompatibilität der nativen Methoden zu JNI-kompatiblen Java-VMs verloren.

Eine native Methode muss zu einem gewissen Grad mit dem internen Zustand der Java-VM interagieren können, um sinnvolle Arbeit leisten zu können. Möglichkeiten der Interaktion wären zum Beispiel:

- das Übergeben und Zurückgeben von Daten
- der Zugriff auf Instanz-Variablen und Aufruf von Objekt-Methoden
- der Zugriff auf statische Variablen und Aufruf von statischen Methoden
- der Zugriff auf Arrays
- das Sperren eines Objekts auf dem Heap für exklusiven Zugriff
- das Erstellen neuer Instanzen auf dem Heap
- das Laden neuer Klassen
- das Werfen neuer Exceptions
- das Abfangen von Exceptions, geworfen durch aufgerufene Java Methoden
- dem Garbage Collector mitteilen, dass ein Objekt nicht mehr benötigt wird

Wenn man sich diese Liste anschaut, wird man sich bereits denken können, dass die Implementierung eines Native Method Interfaces zu vielen Komplikationen führen kann. Vor allem muss dafür gesorgt werden, dass der Garbage Collector keine Objekte löscht, die noch von nativen Methoden benötigt werden. Und wenn Objekte vom Garbage Collector auf dem Heap bewegt werden, so muss dafür gesorgt werden, dass auch die Referenzen in den nativen Methoden geändert werden.

12. Fazit

Die Architektur der Java Virtual Machine ist eine sehr interessante Angelegenheit, wenn man sich eine Zeit damit beschäftigt hat. Erst dann versteht man den ganzen Aufwand und die Zeit, die benötigt werden, um so eine Java-VM zu implementieren.

Es ist ganz interessant, zu überlegen, wie man selbst so eine Java-VM implementieren würde – wenn man die Zeit dazu hätte natürlich. Welche Techniken man verwenden würde und welche Datenstrukturen. Man könnte auf jeden Fall sehr viel daraus lernen!

Abschließend möchte ich noch ein paar Bemerkungen zu meiner Primärliteratur machen, die im nächsten – letzten – Kapitel aufgelistet wird.

[Sun02] Das White Paper von Sun zu ihrer Java HotSpot VM beschreibt die Features ihrer JVM, allerdings relativ wenig über die internen Algorithmen und Datenstrukturen. Es ist so was wie eine „Verherrlichung“ der Java-VM von Sun, was sie nicht alles kann, wie toll sie nicht ist, …
Nichtsdestotrotz erfährt man sehr viel über die Möglichkeiten der Java HotSpot VM und doch auch ein paar Einzelheiten über interne Details.

[Ven99] **Das** Buch über die Java Virtual Machine. Bill Venners versteht es wie kein anderer, die Interna der Java-VM einfach und einleuchtend zu erklären und fundiert dieses Wissen zusätzlich durch zahlreiche interessante Beispiele. Alle Details über die Java-VM werden erklärt und dazu auch noch einige Möglichkeiten der Implementierung angeschnitten.
Jedem, der sich für die inneren Angelegenheiten der Java Virtual Machine interessiert, möchte ich dieses Buch nahe bringen. Absolut empfehlenswert.

[Yel99] Die Spezifikation ist natürlich das Referenz-Werk für alle Fragen zur Java Virtual Machine. Es werden zwar alle Details bezüglich der JVM erörtert, jedoch ist es auf Dauer ziemlich anstrengend, den doch sehr trockenen Schreibstil auszuhalten.
Als Referenz und für genaue Details ist die Spezifikation recht gut zu gebrauchen, wenn man sich allerdings wirklich interessiert und die Java-VM verstehen will, gibt es nur ein Buch: das von Bill Venners.

13. Literatur

13.1. Primärliteratur

[Sun02] The Java HotSpot Virtual Machine 1.4.1 White Paper
Sun Microsystems 2002

http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf

[Ven99] Bill Venners: Inside the Java 2 Virtual Machine
McGraw Hill 1999

http://www.artima.com/insidejvm/ed2/

[Yel99] Frank Yellin, Tim Lindholm: The Java Virtual Machine Specification
Addison Wesley 1999

http://java.sun.com/docs/books/vmspec/

13.2. Sekundärliteratur

[Gos00] James Gosling, Bill Joy, u.a.: The Java Language Specification
Addison Wesley 2000

http://java.sun.com/docs/books/jls/

[Jon97] Richard Jones, Rafael Lins: Garbage Collection
John Wiley & Sons 1997