3 Virtuelle Maschinen

Um zu wissen, wie man einen Motor frisiert, sollte man wissen, wie er funktioniert. Daher wollen wir uns in diesem Kapitel ein wenig mit Javas Virtueller Maschine beschäftigen.

Die Java VM ist eine abstrakte Maschine. Doch genau wie ein echter Prozessor hat sie einen Befehlssatz und manipuliert zur Laufzeit verschiedene Speicherbereiche. Sie setzt jedoch keinerlei spezifische Hardware voraus, sondern wird emuliert. Der emulierte Prozessortyp ist eine Kellermaschine mit mehreren Stacks.

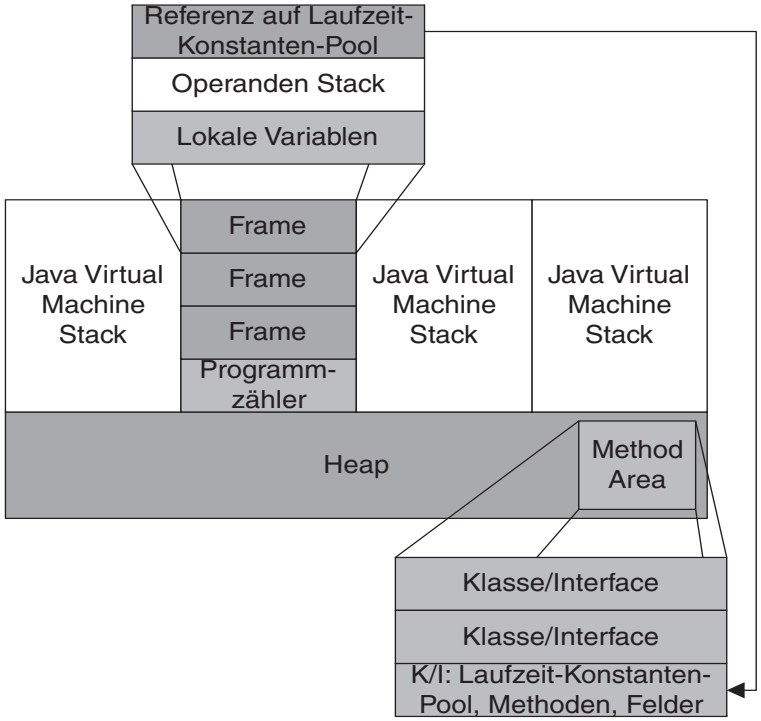

Abbildung 3.1: Schematischer Überblick über die Java VM

Die Grundbestandteile der VM sind jeweils ein Java Virtual Machine Stack pro Thread[1] sowie der von allen Threads geteilte Heap-Speicher (Abbildung 3.1). Im Heap werden sämtliche Objekt-Instanzen sowie Klassen und Interfaces gehalten. Letztere befinden sich dabei in einem speziellen Bereich namens Method-Area. Der Heap wird automatisch durch die Speicherbereinigung verwaltet.

Jeder Stack enthält eine Reihe von so genannten Frames. Während der Ausführung eines Java-Programms wird für jeden Methodenaufruf ein Frame angelegt, auf den Stack gelegt und nach Ausführung der Methode wieder heruntergenommen. Ein Frame enthält bzw. verweist auf alle wichtigen Daten, die zur Ausführung der Methode nötig sind. Zu jedem Stack existiert zudem ein Programmzähler, der auf die Adresse des gerade ausgeführten Codes zeigt.

Vereinfacht dargestellt wird beim Start der VM die zu startende Klasse geladen, ein Thread initialisiert und dessen Programmzähler auf den Beginn der statischen `main()`-Methode der Klasse gesetzt. Dann werden nacheinander die im Bytecode der `main()`-Methode enthaltenen VM-Befehle und somit das Programm ausgeführt.

Absichtlich beschreibt die Java-VM-Spezifikation kaum Implementierungsdetails und lässt VM-Herstellern viele Freiheiten. Die Hauptunterscheidungsmerkmale zwischen heute erhältlichen VMs liegen in der Art und Weise, wie der Java-Bytecode ausgeführt und wie der Heap verwaltet wird. Im Folgenden werden verschiedene Strategien für beides erläutert. Darüber hinaus werden wir kurz auf verschiedene Leistungstests eingehen, die bei der Entscheidung für oder gegen eine VM hilfreich sein können.

3.1 Bytecode-Ausführung

Java-Bytecode lässt sich auf verschiedene Weisen ausführen. Dazu gehören:

- Interpretieren
- Unmittelbar vor der Ausführung in Binärcode übersetzen (Just-in-Time, JIT) und dann den Binärcode ausführen
- Sofort nach dem Erstellen in Binärcode übersetzen und diesen später ausführen (Ahead-of-Time, AOT)
- Teilweise interpretieren und erst wenn notwendig dynamisch in Binärcode übersetzen (Dynamic-Adaptive-Compilation, DAC) und dann diesen ausführen
- Von einer in Silizium gegossenen VM ausführen lassen

Im Folgenden gehen wir kurz auf die einzelnen Möglichkeiten ein.

1 Für `native` Methodenaufrufe kann es zudem einen Native-Stack pro Thread geben.

3.1.1 Interpreter

In Javas frühen Tagen wurde Java ausschließlich interpretiert. Heute werden reine Interpreter für die Java 2 Standard Edition (J2SE) kaum noch verwendet. Der Grund liegt in ihrer Langsamkeit. Dennoch können Interpreter sehr nützlich sein, da sie bei Ausnahmen in Stacktraces gewöhnlich die Klasse, Methode und Zeilenzahl angeben. Das macht sie zu einem wertvollen Werkzeug für die Fehlersuche.

Ein weiterer Vorteil von Java-Interpretern ist ihr geringer Speicherverbrauch. Daher sind sie insbesondere für Kleingeräte wie PDA (Personal Digital Assistent) und Mobiltelefone interessant.

3.1.2 Just-in-Time-Compiler

Nachdem man merkte, dass mit interpretiertem Java nicht viel zu gewinnen war, kamen JIT-Compiler in Mode. JIT-Compiler übersetzen Bytecode unmittelbar vor der Ausführung in plattformspezifischen Binärcode. VMs mit JIT sind in der Regel sehr viel schneller als VMs, die nur über einen Interpreter verfügen. Jedoch führen JITs gegenüber Interpretern auch zu einem höheren Speicherverbrauch. Zudem ist die Qualität des erzeugten Binärcodes nicht mit der Qualität von Binärcode vergleichbar, der von einem optimierenden, statischen Compiler erzeugt wird. Dies liegt daran, dass die Übersetzung zur Laufzeit und somit sehr schnell erfolgen muss. Viel Zeit für langwierige Analysen und Optimierungen bleibt da nicht. Außerdem ist aufgrund der Übersetzung die Startzeit von VMs mit JITs in der Regel länger als die von VMs mit Interpretern.

Eine sehr erfolgreiche VM mit JIT-Technologie wird von IBM produziert.

- IBM Java VM: *http://www.ibm.com/java/*

3.1.3 Dynamisch angepasste Übersetzung

Suns aktuelle Java VMs werden mit HotSpot-Technologie ausgeliefert. Die Idee von HotSpot beruht auf dem 80-20-Prinzip. Dieses besagt, dass während 80% der Laufzeit eines Programms gewöhnlich 20% des Codes ausgeführt werden. Daraus folgt, dass es sich lohnt, genau jene 20% besonders schnell auszuführen, während Optimierungen in den restlichen 80% des Codes keinen großen Effekt haben.

Dementsprechend interpretiert HotSpot zunächst den Bytecode und analysiert zur Laufzeit, welche Programmteile in Binärcode übersetzt und optimiert werden sollten. Diese Teile werden dann übersetzt und mehr oder minder aggressiv optimiert. Im Englischen heißen die besonders kritischen Programmteile auch Hotspots – daher der Name. Das gesamte Verfahren heißt Dynamic-Adaptive-Compilation, kurz DAC.

Verfechter von DAC behaupten gerne, dass durch die adaptive Kompilierung bessere Resultate erzielt werden können als durch statisches Übersetzen. In der Theorie ist das

auch durchaus richtig. In der Praxis zeigt sich jedoch, dass außer der reinen Ausführung von Code noch viele andere Faktoren auf die Geschwindigkeit einer Java VM Einfluss haben. Insbesondere ist das die automatische Speicherbereinigung. Daher hinken Vergleiche mit statisch kompilierten Programmen ohne automatische Speicherbereinigung meist.

Zudem ist die adaptive, optimierende Übersetzung mit vorheriger Laufzeitanalyse ein recht aufwändiger Prozess, der sich nur für lang laufende Programme lohnt. Aus diesem Grund unterscheidet Sun die Client HotSpot VM von der Server HotSpot VM. Die Client-Version führt nur recht simple Optimierungen aus und übersetzt daher relativ schnell. Die Server-Version hingegen enthält einen volloptimierenden Compiler (Tabelle 3.1). Gemessen an JIT-Standards ist dieser Compiler langsam. Dies kann sich jedoch bei langer Laufzeit durch den aus Optimierungen resultierenden Zeitgewinn bezahlt machen.

Optimierung	**Beschreibung**	**JDK 1.3.1 Client**	**JDK 1.3.1 Server**
Range Check Elimination	Unter bestimmten Bedingungen entfällt der obligatorische Check, ob ein Array-Index gültig ist. Siehe *Kapitel 6.2.2 Teure Array-Zugriffe*	nein	ja
Null Check Elimination	Unter bestimmten Bedingungen entfällt der obligatorische Check, ob eine Array-Variable `null` ist. Siehe *Kapitel 6.2.2 Teure Array-Zugriffe*	nein	ja
Loop Unrolling	Mehrfaches Ausführen eines Schleifenkörpers ohne Überprüfung der Abbruchbedingung. Siehe *Kapitel 6.2.3 Loop Unrolling*	nein	ja
Instruction Scheduling	Neuordnung von Befehlen zur optimalen Ausführung auf einem bestimmten Prozessor	nein	ja (für UltraSPARC III)
Optimierung für Reflection API	Schnellere Ausführung von Methoden aus dem Reflection API (`java.lang.reflect`)	nein	ja
Dynamische Deoptimierung	Umkehroperation für eine Optimierung	nein	ja
Einfaches Inlining	Duplizieren und Einfügen von Methoden an Stellen, an denen diese ausgeführt werden. Der Overhead eines Methodenaufrufs entfällt somit. Einfaches Inlining trifft nur auf Klassenmethoden bzw. `final`-Methoden zu.	ja	nein
Vollständiges Inlining	Siehe Einfaches Inlining. Jedoch werden wesentlich mehr Methoden dupliziert und eingefügt.	nein	ja

Tabelle 3.1: Vergleich von Optimierungen der HotSpot Server und Client Compiler für Sun JDK 1.3.1

Optimierung	Beschreibung	JDK 1.3.1 Client	JDK 1.3.1 Server
Dead Code Elimination	Code, der nicht ausgeführt werden kann, wird entfernt.	nein	ja
Loop Invariant Hoisting	Variablen, die sich während einer Schleife nicht ändern, werden außerhalb der Schleife berechnet. Siehe *Kapitel 6.2.1 Loop Invariant Code Motion*	nein	ja
Common Subexpression Elimination	Die Ergebnisse einmal berechneter Ausdrücke werden wieder verwendet.	nein	ja
Constant Propagation	Konstanten werden durch ihre Werte ersetzt.	nein	ja
On Stack Replacement (OSR)	Während der Ausführung einer Methode kann interpretierter Code durch kompilierten ersetzt werden.	ja	ja

Tabelle 3.1: Vergleich von Optimierungen der HotSpot Server und Client Compiler für Sun JDK 1.3.1 (Fortsetzung)

Besonders interessant im Zusammenhang mit Java sind die Fähigkeiten *Inlining* und *Dynamische Deoptimierung*. Inlining ist eine Optimierung, bei der ein Methodenaufruf durch den Code der aufzurufenden Methode ersetzt wird. Dadurch wird der Verwaltungsaufwand für den Methodenaufruf, sprich das Anlegen und Beseitigen eines Frames, gespart. Während statische Compiler für Sprachen wie C relativ einfach Inlining anwenden können, ist das in Java nicht so einfach. Denn im Gegensatz zu C-Funktionsaufrufen sind in Java die meisten Methodenaufrufe *virtuell*, das heißt potenziell polymorph. Somit kann der Aufruf der Methode `doIt()` eines Objektes, das durch eine Referenz vom Typ `A` referenziert wird, durchaus zur Ausführung einer überschriebenen Version der Methode `doIt()` eines Objekts von `A`s Subtyp `B` führen. Listing 3.1 zeigt ein einfaches Beispiel für diesen Fall.

```
class A {
   public doIt() {
      System.out.println("doIt() von A");
   }
}
class B extends A {
   public doIt() {
      System.out.println("doIt() von B");
   }
   public static void main(String[] args) {
      A a = new B();
      a.doIt(); // führt doIt() der Klasse B aus!
   }
}
```

Listing 3.1: Beispiel für eine überschriebene Methode

Um die Sache noch ein wenig komplizierter zu machen, ist es in Java möglich, Klassen dynamisch zur Laufzeit nachzuladen. Somit ist das gefahrlose Inlining einer Methode nur möglich, wenn die einzureihende Methode entweder `final` oder `static` ist, da dann die Methode nicht von einer Unterklasse überschrieben werden kann. Die Fähigkeit `static` und `final` Methoden einzureihen wird in Tabelle 3.1 unter einfachem Inlining aufgeführt.

Mit vollständigem Inlining ist gemeint, dass auch Methoden, die nicht `final` oder `static` sind, eingeschoben werden können. Dies ist dann möglich, wenn keine Klasse geladen ist, die die fragliche Methode überschreibt. Da in Java aber auch zur Laufzeit noch Klassen geladen werden können, kann sich dieser Zustand ändern. Ist dies der Fall, so muss eine bereits eingereihte Methode evtl. wieder durch einen regulären Methodenaufruf ersetzt werden. Dies ist ein Beispiel für dynamische Deoptimierung. JIT-Compiler, die den Code in der Regel nur einmal übersetzen, sind zu solchen (De-)Optimierungen meist nicht in der Lage.

Neben dem gesparten Verwaltungsaufwand für Methodenaufrufe bietet vollständiges Inlining noch einen weiteren Vorteil: Die entstehenden längeren Code-Blöcke erlauben weitere Optimierungen.

Suns HotSpot-Technologie wurde von Hewlett Packard und Apple lizenziert. Daneben gibt es außerdem noch andere DAC-VMs. Eine davon ist die freie VM *JRockit* von Appeal Virtual Machines (BEA). Es handelt sich dabei um eine Java Laufzeitumgebung mit Fokus auf serverseitige Applikationen. Sun bemüht sich zudem, HotSpot auch für die J2ME-Plattform anzubieten.

- Sun J2SE: *http://java.sun.com/*
- Appeal Virtual Machines: *http://www.jrockit.com/*

3.1.4 Ahead-of-Time-Übersetzung

Anstatt Code erst zur Laufzeit in Binärcode zu übersetzen, kann man auch bereits beim Erstellen der Software Binärcode für die Zielplattform erzeugen. Dieser Ansatz wird auch als Ahead-of-Time-Übersetzung (AOT) bezeichnet.

AOT ermöglicht sehr gute Optimierungen und führt in der Regel zu robustem, schnellem Code. Dies liegt unter anderem daran, dass bereits vorhandene, ausgereifte Compiler zur Code-Generierung und -Optimierung genutzt werden können. So ist beispielsweise der freie GNU Java Compiler *GJC* lediglich ein Frontend zu anderen GNU Compilern. Dementsprechend wird Java von den GJC-Autoren auch nur als Teilmenge von C++ betrachtet.

Ein weiter Grund für die gute Performance von AOT sind eine im Vergleich zu JIT und DAC fast beliebig lange Code-Analyse-Phase, beinahe beliebiger Ressourcenverbrauch während der Übersetzung und die daraus resultierenden Optimierungen. Der kom-

merzielle Java Native Compiler *TowerJ* beispielsweise kann während des Kompilierens so genannte ThreadLocal-Objekte erkennen. ThreadLocals sind Objekte, die garantiert nur von einem Thread benutzt werden. Somit ist jegliche Synchronisation überflüssig, die Objekte können im schnelleren, threadspezifischen Stack statt im Heap gespeichert werden und unterliegen somit auch nicht der normalen Speicherbereinigung. Der erzeugte Code ist entsprechend schneller.

Zudem kann Binärcode nur schlecht dekompiliert werden und erschwert Reverse Engineering[2] erheblich. Somit ist gegenüber Bytecode ein besserer Schutz von geistigem Eigentum gewährleistet.

Alle AOT-Compiler haben jedoch mit Javas Fähigkeit zu kämpfen, Klassen dynamisch nachzuladen. Dies ist beispielsweise notwendig für Remote Method Invocation (RMI), Dynamische Proxies (`java.lang.reflect.Proxy`), Java Server Pages (JSP) etc. TowerJ löst dieses Problem, indem während der ersten Ausführung aufgezeichnet wird, welche Klassen nachgeladen und von einem eingebauten Interpreter ausgeführt werden mussten. In einem folgenden Übersetzungslauf werden diese Klassen dann ebenfalls in Binärcode übersetzt und optimiert.

Einen etwas anderen Weg beschreitet *Excelsiors JET*. JET verfügt über die Fähigkeit, zur Laufzeit nachgeladene Klassen mit einem JIT-Compiler zu übersetzen und als DLLs einzubinden. Dabei ist der JIT-Compiler selbst eine DLL und kann dementsprechend nach getaner Arbeit wieder aus dem Speicher entfernt werden.

Zusammenfassend lässt sich sagen, das AOT eine legitime Strategie ist, um schnelle und verlässliche Java-Programme für eine spezifische Zielplattform zu erzeugen. Die erzeugten Programme sind natürlich nicht portabel. Jedoch spricht nichts dagegen, neben verschiedenen nativen Versionen auch eine portable Bytecode-Version Ihrer Software auszuliefern.

Hier eine Auswahl von AOT-Compilern:

- TowerJ: *http://www.towerj.com/*
- NaturalBridge BulletTrain: *http://www.naturalbridge.com/*
- Excelsiors JET: *http://www.excelsior-usa.com/jet.html*
- Instantiations JOVE: *http://www.instantiations.com/jove/*
- GCJ ist noch in den frühen Phasen der Entwicklung: *http://gcc.gnu.org/java/*

2 Vorgang, bei dem aus Bytecode ein (visuelles) Modell erzeugt wird.

3.1.5 Java in Silizium

Natürlich ist man auch auf die Idee gekommen, aus der Java Virtuellen Maschine eine reale Java Maschine zu machen; mit anderen Worten, die VM in Silizium zu gießen. Dies ist insbesondere für den Mobiltelefon- und PDA-Markt interessant, der J2ME verwendet, und weniger für Geschäftsanwendungen auf Basis von J2SE oder J2EE. Der Vollständigkeit halber möchte ich kurz auf das Thema eingehen – für das Buch wird es im Weiteren nicht von Belang sein.

Schon 1996 hat Sun einen Prozessor-Kern namens *picoJava* spezifiziert. Dieser wurde auch von mehreren großen Firmen lizenziert, ein picoJava-Boom blieb jedoch aus. Keiner der Lizenznehmer hat jemals picoJava-basierte Chips verkauft.

Bereits zum Erscheinen der Spezifikation wurde der Ansatz kritisch beäugt. Wissenschaftler hatten schon für andere Sprachen wie LISP und Smalltalk Spezial-Prozessoren entwickelt, nur um zu entdecken, dass Software-Implementierungen auf RISC-Chips bessere Performance boten. Man zweifelte daran, dass Suns picoJava besser performte. Und tatsächlich stellte sich später heraus, dass picoJava weder schnell noch billig noch sparsam genug war, um im Markt für Mobiltelefone und PDAs mithalten zu können.

Stattdessen wurde in letzter Zeit ein etwas anderer Ansatz für Kleingeräte populär: Java-Beschleuniger. Dabei handelt es sich um Bausteine, die ähnlich wie Koprozessoren zusätzlich zum Hauptprozessor verwendet werden können. So lässt sich beispielsweise *Nazomis* Java-Koprozessor in bestehende Designs einbinden und erleichtert so Kleingeräte-Herstellern die Verwendung von Java unter Beibehaltung einer bereits vorhandenen Architektur.

Einen anderen Weg ging die Firma *ARM*. ARM hat seinen Chips den Java-VM-Befehlssatz schlicht als dritten Befehlssatz hinzugefügt. Ein einfaches Umschalten macht so aus dem herkömmlichen ARM-Chip eine Java VM.

- Suns picoJava: *http://www.sun.com/microelectronics/picoJava/*
- ARM: *http://www.arm.com/*
- Nazomi: *http://www.nazomi.com/*

3.2 Garbage Collection

Neben der Bytecode-Ausführung ist der andere entscheidende Aspekt für die Performance einer VM die Garbage Collection. Weder in der Java Sprachspezifikation [Gosling00] noch in der Java VM Spezifikation [Lindholm99] sind genaue Vorgaben für die Garbage Collection zu finden. Es steht den VM-Herstellern somit größtenteils frei, wie sie die Speicherverwaltung implementieren.

3.2.1 Objekt-Lebenszyklus

Um besser zu verstehen, was die Aufgabe der Speicherverwaltung ist, wollen wir uns den Lebenszyklus eines Objektes anschauen. Abbildung 3.2 gibt eine Übersicht.

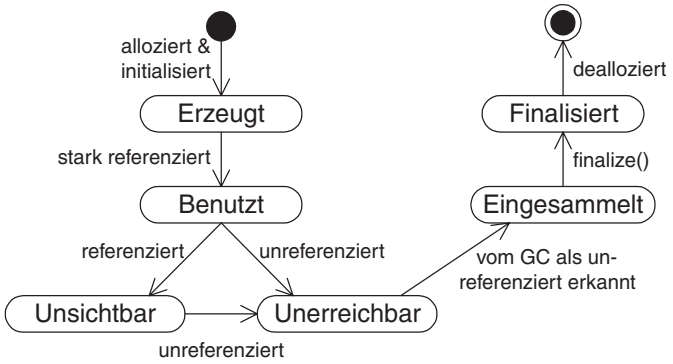

Abbildung 3.2: Lebenszyklus eines Objekts

Erzeugt

Es wurde Speicher für das Objekt alloziert und alle Konstruktoren sind ausgeführt worden. Das Objekt befindet sich also fertig initialisiert im Heap. Es wurde 'Erzeugt'.

Benutzt

Es existiert mindestens eine stark, vom Programm sichtbare Referenz auf das Objekt. Schwache, weiche und Phantom-Referenzen aus dem Paket `java.lang.ref` können zudem existieren, reichen aber nicht aus, um ein Objekt im Zustand 'Benutzt' zu halten.

Unsichtbar

Ein Objekt ist 'Unsichtbar', wenn keine starken Referenzen mehr existieren, die vom Programm benutzt werden könnten, trotzdem aber noch Referenzen vorhanden sind.

Nicht jedes Objekt durchläuft diesen Zustand. Er tritt aber zum Beispiel auf, wenn in einem Stackframe noch eine Referenz auf ein Objekt vorhanden ist, obwohl diese Referenz in einem abgeschlossenen Block deklariert und dieser Block bereits verlassen wurde. Listing 3.2 zeigt ein Beispiel.

```
public void do() {
   try {
      Integer i = new Integer(1);
      ...
   }
```

```
    catch (Exception e) {
       ...
    }
    while (true) {
       // äußerst lange Schleife
    }
}
```

Listing 3.2: In der while-Schleife ist i unsichtbar.

Das Objekt `i` wird in einem geschlossenen `try-catch`-Block alloziert. Eine effiziente VM-Implementierung wird die im Frame allozierte Referenz auf `i` jedoch nicht beim Verlassen des Blocks beseitigen, sondern erst, wenn der entsprechende Frame vom Stack genommen wird [Wilson00, S.196]. Daher ist `i` in der folgenden `while`-Schleife *sowohl* unsichtbar *als auch* referenziert und kann somit nicht vom Garbage Collector erfasst werden.

Unerreichbar

'Unerreichbar' sind jene Objekte, die nicht mehr von einer Objektbaumwurzel über Navigation zu erreichen sind. Zu den Objektbaumwurzeln gehören:

- Klassenvariablen (static)
- Temporäre Variablen auf dem Stack (lokale Methoden-Variablen)
- Besondere Referenzen von JNI-Code aus

Ist ein Objekt unerreichbar, so kann es vom Garbage Collector zu einem beliebigen späteren Zeitpunkt eingesammelt werden.

Eingesammelt

Ein Objekt ist dann 'Eingesammelt', wenn der Garbage Collector das Objekt als Garbage erkannt hat und es in die Warteschlange des Finalizer-Threads eingestellt hat. Ist die `finalize()`-Methode des Objekts nicht überschrieben, wird dieser Schritt übersprungen und das Objekt gelangt direkt in den Zustand 'Finalisiert'.

Finalisiert

'Finalisiert' ist ein Objekt, nachdem die `finalize()`-Methode aufgerufen wurde, sofern diese vorhanden ist bzw. überschrieben wurde. Beachten Sie, dass die `finalize()`-Methode meist in einem Extra-Thread, dem so genannten Finalizer-Thread ausgeführt wird. Falls die Threads Ihrer Applikation mit höherer Priorität laufen als der Finalizer-Thread und Sie die `finalize()`-Methode mit spezieller Aufräumlogik überschrieben haben, kann es sein, dass Sie die Garbage Collection blockieren, da der Finalizer-Thread nicht zum Zuge kommt und somit Objekte nicht deallociert werden können.

Daher ist es grundsätzlich besser, sich nicht auf den Finalizer zu verlassen, sondern stattdessen eigene Lebenszyklus-Methoden zu implementieren und diese kontrolliert aufzurufen. Beispielsweise sollten Sie immer die Methode `dispose()` eines `java.awt.Graphics`-Objektes aufrufen, wenn Sie es nicht mehr benötigen, da sonst erst der Finalizer wichtige Ressourcen freigibt.

Dealloziert

Ein Objekt ist 'Dealloziert', wenn es nach der Finalisierung immer noch unerreichbar war und somit beseitigt werden konnte. Wann dies geschieht, liegt im Ermessen der VM.

3.2.2 Garbage Collection-Algorithmen

Sie haben gesehen, dass Objekte erst dealloziert werden können, wenn sie finalisiert und unerreichbar sind. Wie die Unerreichbarkeit bestimmt und der verfügbare Speicher verwaltet wird, hängt vom verwendeten Gargabe Collector ab. Dieser lässt sich meist über VM-Parameter beeinflussen und optimieren. Deshalb wollen wir uns kurz mit verschiedenen Garbage Collection-Algorithmen auseinander setzen.

Kopierender Kollektor

Ein einfacher *Kopierender Kollektor* unterteilt den Speicher in zwei Hälften. Er alloziert so lange Objekte in der ersten Hälfte, bis diese voll ist. Dann besucht er, von den Objektbaumwurzeln ausgehend, alle lebendigen Objekte und kopiert sie in die zweite Speicherhälfte.[3] Die nicht mehr referenzierten Objekte werden gelöscht. Anschließend tauschen die Speicherhälften ihre Rollen. Im verbliebenen Speicher der zweiten Hälfte werden nun neue Objekte alloziert, bis diese voll ist. Dann werden die lebendigen Objekte wiederum in die erste Hälfte kopiert usw.

Kopierende Kollektoren führen zu sehr schnellen Allokationszeiten, da der Speicher nicht fragmentiert wird. Tatsächlich reicht es aus, einfach einen Zeiger auf die Speicherzelle nach dem zuletzt allozierten Objekt zu pflegen. Viel schneller kann man Speicher nicht allozieren. Dieser Komfort hat jedoch seinen Preis, denn der Speicherverbrauch einfacher kopierender Kollektoren ist doppelt so groß wie der anderer Kollektoren.[4] Zudem fällt die Interaktion mit dem Speicherverwaltungssystem des Betriebssystems eher ungünstig aus. In jedem Kollektionszyklus muss jede Speicherseite einer Hälfte geladen und komplett beschrieben werden. Falls der gesamte Heap nicht in den realen

3 Wegen dieses Verhaltens werden kopierende Kollektoren auch *Scavengers* (Aasfresser) genannt – sie nehmen, was noch zu gebrauchen ist.

4 Hier ist nur der simpelste Kopier-Algorithmus beschrieben. Natürlich ist es möglich, den Speicher in mehr als zwei Teile zu unterteilen, jedoch pro Kollektionszyklus nur zwei dieser Teile zu betrachten [Jones96, S.127] und somit den Speicherverbrauch zu reduzieren.

Speicher des Rechners passt, führt dies unweigerlich zu teuren Seitenfehlern. Ein weiterer Nachteil ist, dass alle Objekte ständig im Speicher umgruppiert werden, was wiederum zu schlechter Lokalität führen kann.

Somit eignen sich kopierende Kollektoren insbesondere für kleine Speicherbereiche mit kurzlebigen Objekten.

Mark & Sweep

Der *Mark-Sweep*-Algorithmus funktioniert folgendermaßen: Ausgehend von den Objektbaumwurzeln werden alle erreichbaren Objekte besucht und mit einem Bit markiert. Wurden so alle lebendigen Objekte markiert, werden alle nicht-markierten Objekte aus dem Speicher gekehrt.

Mark-Sweep-Kollektoren haben gegenüber kopierenden Kollektoren einen geringeren Speicherverbrauch. Dafür neigen sie jedoch dazu, den Speicher zu fragmentieren. Das bedeutet auch, das die Verwaltung des freien Speichers umständlich ist, was wiederum dazu führt, dass die Allokation von Speicher länger dauert. Zudem kann bedingt durch die Fragmentierung der Speicher nicht vollständig genutzt werden und es kann passieren, dass Objekte, die kurz nacheinander alloziert wurden, nicht nahe beieinander im Heap liegen. Diese schlechte Lokalität führt wiederum zu Seitenfehlern.

Der Mark-Sweep-Algorithmus selbst ist simpel und schnell, führt aber zu Problemen insbesondere bei Systemen mit virtuellem Speicher. Er ist die Grundlage für den *Mark-Compact*-Algorithmus.

Mark & Compact

Genau wie beim Mark-Sweep-Algorithmus werden zunächst alle lebendigen Objekte markiert. Anschließend werden die lebendigen Objekte so im Speicher verschoben, dass es nur noch zwei Bereiche gibt: einen durchgängig mit Objekten belegten Bereich und einen freien Bereich. Der Heap wurde kompaktiert. Das heißt zur Verwaltung des freien Speichers genügt ein einziger Zeiger, der auf das Ende des belegten Bereichs zeigt. Dies wiederum bedeutet schnelle Allokation, da nicht umständlich nach einem freien Stück Speicher der gewünschten Größe gefahndet werden muss.

Je nach verwendetem Algorithmus kann während des Kompaktierens die Ordnung der Objekte beibehalten werden. Dies führt zu guter Lokalität und weniger Seitenfehlern. Auch das hat jedoch seinen Preis. Während Mark-Sweep-Kollektoren nach der Markierungsphase nur einmal durch den Heap wandern, müssen Mark-Compact-Algorithmen den Heap meist zwei- oder dreimal durchkämmen. Die Kollektion dauert also länger, führt aber anschließend zu einem besseren Laufzeitverhalten.

Inkrementelle und nebenläufige Speicherbereinigung

Alle bisher vorgestellten Verfahren gehen jeweils davon aus, exklusiv auf den Heap zugreifen zu können. Das bedeutet, dass das Programm während der Kollektion gestoppt wird. Dies führt zu unangenehmen Pausen, in denen die Applikation nicht auf Eingaben reagiert. Diese Pausen sind für Echtzeitanwendungen (z.B. Audio-/ Video-Anwendungen, Maschinen-Steuerungssoftware) nicht akzeptabel. Die Pausen sind zudem umso länger, je größer der Heap ist.

Inkrementelle und nebenläufige Speicherbereinigung versuchen, die Pausen auf ein Minimum zu reduzieren bzw. ganz zu beseitigen. Im Fall der inkrementellen Speicherbereinigung wird jeweils nur ein kleiner Teil des Heaps von unreferenzierten Objekten gesäubert und dann die Ausführung des Programms fortgesetzt. Bei nebenläufiger Speicherbereinigung wird parallel zur Programmausführung jeweils ein Teil des Heaps gereinigt. Dabei treten gewöhnlich Synchronisationsprobleme auf, die zu einem gewissen Verwaltungsaufwand führen.

Inkrementelle oder nebenläufige Kollektoren reduzieren Pausen, führen ansonsten aber meist zu einer schlechteren Performance als andere Verfahren.

Generationen-Kollektoren

Generationen-Kollektoren gehen davon aus, dass einige Objekte länger leben als andere. Weiterhin wird angenommen, dass die meisten Objekte jung sterben. Dementsprechend wird der Heap in mehrere Bereiche – Generationen – unterteilt. Neue Objekte werden grundsätzlich in der jungen Generation alloziert. Ist kein Speicher mehr in diesem Bereich vorhanden, wird er mittels eines der oben geschilderten Algorithmen aufgeräumt. Objekte, die eine bestimmte Anzahl von Aufräumphasen überleben, werden in die ältere Generation verschoben. Dabei müssen alle Zeiger auf das Objekt entsprechend angepasst werden.

Die ältere Generation wird genau wie die jüngere bei Bedarf aufgeräumt. Grundsätzlich sind mehr als zwei Generationen denkbar.

Generationen optimieren die Speicherbereinigung insofern, als dass nicht immer der ganze Speicher aufgeräumt wird, sondern nur der Teil, der gerade vollgelaufen ist. Somit sind die Pausen, die auftreten, wenn das Programm zur Speicherbereinigung gestoppt werden muss, relativ geringer als bei nur einem großen zusammenhängenden Speicherbereich, der immer komplett gesäubert werden muss. Dabei macht man sich zunutze, dass ein kleiner Speicherteil gewöhnlich sehr schnell mit Objekten belegt ist, die zu einem großen Teil direkt wieder aus dem Speicher entfernt werden können.

3.2.3 Performance-Maße

Zum Beurteilen der Performance von Mechanismen zur automatischen Speicherbereinigung gibt es vier wichtige Maße:

- Durchsatz
- Pausen
- Speicherverbrauch
- Promptheit

Durchsatz bezeichnet den prozentualen Anteil der Laufzeit eines Programms, der nicht mit Speicherbereinigung oder Allokation verbracht wird. Dies ist eine wichtige Größe bei lang laufenden Programmen wie Servern.

Pausen sind Zeiten, in denen die Applikation nicht reagiert, da gerade der Speicher aufgeräumt wird. Insbesondere bei interaktiven oder Echtzeit-Programmen ist hier der Maximalwert eine interessante Größe zur Beurteilung der Speicherverwaltung.

Speicherverbrauch ist für Systeme mit keinem oder begrenztem virtuellen Speicher eine wichtige Größe.

Promptheit bezeichnet die Zeit, die nach dem Tod eines Objekts vergeht, bis es tatsächlich beseitigt ist.

Gewöhnlich gilt, dass ein guter Wert in einer Kategorie zu Lasten des Wertes einer anderen Kategorie geht. Es kann also keinen per se richtigen, sondern nur einen am besten zu den Anforderungen passenden Garbage Collector geben.

3.2.4 HotSpots Garbage Collection

Suns VM ist ein gutes Beispiel für einen Generationen-Kollektor. Der Heap ist in fünf Generationen unterteilt (Abbildung 3.3).

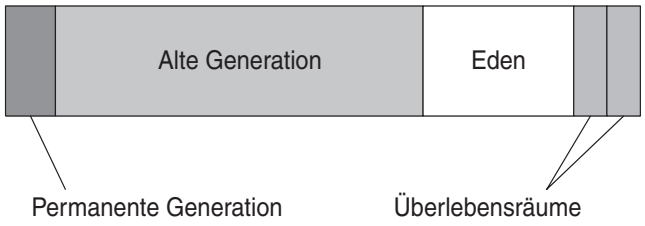

Abbildung 3.3: Aufteilung des Heaps der Sun HotSpot-VM in verschiedene Generationen

Objekte werden zunächst in *Eden* alloziert. Ist kein Platz mehr in Eden, greift ein kopierender Kollektor und verschiebt alle lebendigen Objekte aus Eden sowie einem der *Überlebensräume* in den anderen, leeren Überlebensraum. Wenn Objekte eine gewisse Zeit zwischen den beiden Überlebensräumen hin- und herkopiert wurden, werden sie befördert. Das heißt, sie werden in die *alte Generation* verschoben. In der Alten Generation wiederum greift ein Mark-Compact-Kollektor. Eine Besonderheit stellt die *permanente Generation* dar. Sie enthält Klassen- und Methoden-Objekte, die sehr selten – wenn überhaupt – deallociert werden müssen.

Alternativ zum standardmäßig verwendeten Mark-Compact-Algorithmus für die alte Generation kann man bei Bedarf mittels des VM-Parameters `-Xincgc` auch einen inkrementellen Algorithmus verwenden. Dieser verursacht zwar weniger Pausen, hat jedoch einen größeren Verwaltungsaufwand.

Die Größe des Heaps sowie der einzelnen Generationen lässt sich über VM-Parameter beeinflussen (Tabelle 3.2). Um festzustellen, welche Einstellung für Ihr Programm optimal ist, experimentieren Sie mit verschiedenen Werten und vergleichen Sie die Garbage Collection-Daten, die Sie durch den VM-Parameter `-verbose:gc` erhalten.

Der Durchsatz ist gewöhnlich am besten, wenn möglichst selten eine vollständige Speicherbereinigung inklusive der Alten Generation erforderlich ist. Dies ist der Fall, wenn Eden und die beiden Überlebensräume (*junge Generationen*) im Verhältnis zum Rest des Heaps sehr groß sind. Dies geht jedoch auf Kosten von Speicherverbrauch und Promptheit. Pausen wiederum können minimiert werden, indem die Jungen Generationen möglichst klein gehalten und inkrementelle Garbage Collection für die alte Generation benutzt wird.

Parameter	**Beschreibung**
-Xms<wert>	Minimale Heapgröße
-Xmx<wert>	Maximale Heapgröße
-Xminf<wert>	Prozentualer Anteil des Heaps, der nach einer vollständigen Speicherbereinigung mindestens frei sein sollte. Standard: 40
-Xmaxf<wert>	Prozentualer Anteil des Heaps, der nach einer vollständigen Speicherbereinigung höchstens frei sein sollte. Standard: 70
-XX:NewRatio=<wert>	Verhältnis der Größe der Neuen Generationen zur Alten Generation
-XX:NewSize=<wert>	Startgröße der Neuen Generationen
-XX:MaxNewSize=<wert>	Bestimmt die maximale Größe der Neuen Generationen
-XX:SurvivorRatio=<wert>	Verhältnis der Größe von Eden zu einem der Überlebensräume
-XX:MaxPermSize=<wert>	Maximale Größe der Permanenten Generation

Tabelle 3.2: Parameter zum Optimieren der HotSpot-Speicherverwaltung

3.3 Industrie-Benchmarks

Zur Beurteilung von Java VMs lohnt es sich, Leistungstests oder Benchmarks zu benutzen. Ein Benchmark ist eine definierte Referenz, die zu Vergleichszwecken herangezogen werden kann. Hiermit ist meist ein definiertes Testverfahren gemeint, das reproduzierbar die Leistung von Soft- oder Hardware misst. Oft wird das Ergebnis in einer einzelnen Maßzahl zusammengefasst.

Diese Labor-Benchmarks beschreiben per Definition nur eine Abbildung der Wirklichkeit. Aus diesem Grund definiert Eric Raymond einen Benchmark als »an inaccurate measure of computer performance« und zitiert in seinem Buch die alte Hacker-Weisheit:

> *In the computer industry, there are three kinds of lies: lies, damn lies, and benchmarks. [Raymond96].*

Nun gibt es glücklicherweise Benchmarks, die von Organisationen wie *SPEC* (Standard Performance Evaluation Corporation, *http://www.spec.org/*), einzelnen unparteiischen Firmen und Verlagen bzw. Magazinen kreiert wurden und somit sicherlich vertrauenswürdiger sind als beispielsweise ein reiner Intel-Benchmark zum Vergleich von Intel- und Motorola-Prozessoren.

Dennoch muss man sich bei jedem Benchmark fragen, was dieser misst, welche Aussage sich aus dem Ergebnis ableiten lässt und wie nützlich diese Aussage in Bezug auf das eigene Programm ist. Es nützt Ihnen wenig, eine VM zu verwenden, die hervorragend bzgl. gleichzeitiger TCP/IP-Verbindungen skaliert, wenn Sie tatsächlich numerische Berechnungen anstellen wollen und nie auch nur eine einzige TCP/IP-Verbindung aufbauen.

In diesem Abschnitt werden einige sehr verschiedene Benchmarks kurz vorgestellt. Hier sind die entsprechenden URLs:

- VolanoMark: *http://www.volano.com/benchmarks.html*
- SPEC JVM98: *http://www.spec.org/osg/jvm98/*
- SPEC JBB2000: *http://www.spec.org/osg/jbb2000/*
- jBYTEMark 0.9 (BYTE Magazine) scheint nicht mehr durch einen regulären Link erreichbar zu sein. Sie können jedoch mit einer Suchmaschine nach *jbyte.zip* suchen.
- ECperf: *http://java.sun.com/j2ee/ecperf/index.html*

3.3.1 VolanoMark

VolanoMark ist ein Benchmark, der entstand, um die Performance einer VM und ihrer Skalierbarkeit in Bezug auf TCP/IP-Verbindungen zu messen. Die Motivation für den letzteren Aspekt liegt in Javas Ein-Thread-pro-Verbindung-Modell begründet (siehe

Kapitel 10.5 Skalierbare Server). Dieses besagt, dass für jede Verbindung ein dedizierter Thread existieren muss. Wenn man also sehr viele Verbindungen gleichzeitig unterhalten will (und genau das macht der Volano-Chat-Server), ist es wichtig, eine VM zu benutzen, die sowohl viele Sockets als auch viele Threads performant unterstützt.

Während des Tests wird gemessen, wie viele Nachrichten Clients über einen Server verschicken können. Beim Performance-Test laufen sowohl Server als auch Clients auf demselben Rechner mit 200 gleichzeitigen Loopback-Verbindungen. Beim Skalierbarkeitstest simulieren die Clients von einem anderen Rechner aus eine ständig steigende Zahl gleichzeitiger Verbindungen. Gemessen wird die maximal mögliche Anzahl simultaner Verbindungen.

Seit JDK 1.4 bietet Java mit dem `java.nio`-Paket ein alternatives, asynchrones Ein-/Ausgabe-Modell, das nicht mehr zwingend einen Thread pro Verbindung vorschreibt. VolanoMark in seiner jetzigen Form könnte also entsprechend an Bedeutung verlieren.

Nichtsdestotrotz ist VolanoMark ein gerne zitierter Benchmark für Performance und Netzwerk-Skalierbarkeit. Ergebnisse werden regelmäßig auf der Volano-Website publiziert.

3.3.2 SPEC JVM98

SPEC JVM98 ist ein Client-Benchmark der SPEC zum Messen von Java-VM-Performance. Er besteht aus acht verschiedenen Tests, von denen fünf reale Anwendungen sind. Sieben der Tests dienen zum Messen von Daten, der achte verifiziert die korrekte Ausführung des Bytecodes:

- compress: Werkzeug zum (De-)Komprimieren von Dateien
- jess: Java Expertensystem
- db: ein kleines Datenmanagement-Programm
- javac: Suns Java-Compiler
- mpegaudio: ein MPEG-3-Dekoder
- mtrt: ein multithreaded Raytracer
- jack: ein Parser-Generator mit lexikalischer Analyse
- check: Überprüft Java VM und Java Features

JMV98 testet nicht AWT-, Netzwerk-, Datenbank- und Grafik-Performance. Zudem erschien JVM98 vor Java 2. Es werden also auch keine der neueren Java-Features wie schwache Referenzen oder dynamische Proxies getestet.

3.3.3 SPEC JBB2000

SPEC JBB2000 ist ein serverseitiger SPEC-Benchmark, der eine 3-Tier-Applikation simuliert. Die Hauptlast liegt dabei auf dem Mittel-Tier, der die Geschäftslogik enthält. Tier eins und drei enthalten die Benutzerschnittstelle und die Datenverwaltung. Der Test läuft vollständig in einer VM ab und benötigt keine weiteren Komponenten wie eine Datenbank oder einen Webbrowser. Im Test werden weder Enterprise Java Beans (EJB), Servlets noch JSP verwendet.

JBB2000 misst ausschließlich VM-Performance und Skalierbarkeit. Nicht gemessen werden AWT-, Netzwerk-, Ein-/Ausgabe- und Grafik-Leistung.

3.3.4 jBYTEMark

jBYTEMark ist ein ursprünglich in C geschriebener Benchmark, der vom BYTE-Magazin nach Java portiert wurde. Er enthält ausschließlich rechenintensive Algorithmen, versucht also gar nicht erst eine reale Applikation nachzuempfinden. Dem Benchmark-Code lässt sich zudem leicht ansehen, dass er aus der C-Welt stammt. OO-Performance lässt sich mit diesem Benchmark nicht messen.

jBYTEMark scheint von BYTE schon lange aufgegeben worden zu sein, ist aber immer noch für einen schnellen Numbercrunching-Vergleich zu gebrauchen. Wichtig ist zu betonen, dass dieser Benchmark keinen Gebrauch von Threads macht und weder AWT-, Netzwerk-, Ein-/Ausgabe- noch Grafik-Leistung misst.

3.3.5 ECperf

Anders als die zuvor aufgeführten Benchmarks dient *ECperf* zum Messen der Skalierbarkeit und Performance von J2EE-Servern (Java 2 Enterprise Edition) und nicht einer speziellen Java VM. Dabei werden hauptsächlich Speicherverwaltung, Verbindungs-Pooling, Passivierung/Aktivierung von EJBs und Caching des EJB-Containers getestet. Die Leistung der gewöhnlich nötigen Datenbank fließt angeblich kaum in den Benchmark ein.

ECperf simuliert eine reale Anwendung, die die Bereiche Herstellung, Supply-Chain-Management und Verkauf abbildet. Dies führt zur Nutzung und somit indirekt zur Bewertung der folgenden technischen Aspekte von J2EE-Anwendungen:

- Transaktionale Komponenten
- Verteilte Transaktionen
- Messaging und asynchrones Aufgabenmanagement
- Mehrere Service-Provider mit mehreren Websites
- Schnittstellen von und zu Altsystemen

- Sichere Datenübertragung
- Rollenbasierte Authentisierung
- Persistente Objekte

ECPerf wurde von SPEC übernommen und firmiert dort unter dem Namen *SPECjAppServer200X*.

3.4 Die richtige VM auswählen

Die VM zu wechseln ist eine der einfachsten und billigsten Methoden, die Performance Ihrer Software zu verbessern. Wenn Sie nicht gerade auf die VM eines bestimmten Herstellers angewiesen sind, sollte dies Ihr erster Schritt sein. Sie können eine grobe Vorauswahl anhand von publizierten Benchmarks vornehmen. Achten Sie dabei darauf, dass die Benchmarks Leistungsaspekte messen, die für Ihre Anwendung wichtig sind. Anders ausgedrückt: Die Endgeschwindigkeit eines Autos ist in den USA irrelevant. Wichtig ist die Beschleunigung bis zur erlaubten Höchstgeschwindigkeit.[5]

Viele publizierte Benchmarks messen die Leistung eines Ein-Prozessor-Systems. Wenn Sie wollen, dass Ihre VM mit der Prozessorzahl Ihres Mehrprozessor-Systems skaliert, verifizieren Sie, dass die VM dazu geeignet ist. VMs, die so genannte Green-Threads benutzen, skalieren nicht mit der Prozessorzahl, sondern nutzen immer nur einen Prozessor. Ist jedoch nur ein Prozessor vorhanden, brillieren Green-Threads, da sie in der Regel leichtgewichtiger sind als Native-Threads. Ein Beispiel hierfür ist das Linux Blackdown JDK.

Falls Ihr Programm Echtzeit-Kriterien[6] standhalten muss, wählen Sie eine VM mit inkrementellem oder nebenläufigem Garbage Collector. Inkrementelle Garbage Collection hält die maximalen Pausenzeiten kurz, führt jedoch sonst meist zu schlechterer Performance.

Wenn Ihre Anwendung keine Massenware ist, sondern nur in einer definierten Umgebung lauffähig sein muss, ziehen Sie AOT-Compiler in Betracht. »Write once, run anywhere« spielt für Sie keine Rolle, insbesondere nicht, wenn Sie über den Quellcode verfügen und jederzeit regulären Bytecode einsetzen können.

Nachdem Sie sich für eine VM entschiedenen haben, informieren Sie sich über Optimierungsmöglichkeiten. Fast alle VMs haben Parameter, mit denen Sie entscheidenden Einfluss auf die Leistung der VM nehmen können. Dazu gehören die minimale

5 In den meisten Staaten sind das 70 mph (etwa 113 km/h).

6 Kaum eine Java VM hält harten Echtzeitkriterien stand. Jedoch gibt es bzgl. des Echtzeitverhaltens natürlich bessere und schlechtere VMs.

und maximale Heapgröße, die Stackgröße, GC-Algorithmen und GC-Generationen-Größen, Thread-Modelle, JIT/DA-Compiler und vieles mehr. Insbesondere, wenn Sie wenig Einfluss auf den Quellcode haben, ist dies Ihr bester Ansatzpunkt.

Die Wahl der richtigen VM kann leicht über Erfolg und Misserfolg Ihres Projekts entscheiden. Es ist daher blauäugig, einfach die erstbeste VM zu benutzen. Zudem konkurrieren alle VM-Hersteller darum, die schnellste VM herzustellen. Es lohnt sich also, immer mal wieder eine neue oder andere VM auszuprobieren. Dank Javas Portabilität sollte dies keine allzu große Schwierigkeit darstellen.