FT-Ausarbeitung

Java Virtual Machine

Bestandteile und Funktionsweise

Alexander Rieppel

29. Mai 2014

5AHITT

Inhaltsverzeichnis

1 Einführung

Die Java Virtual Machine (JVM) wird innerhalb der Java Laufzeitumgebung (JRE) angeboten, welche für die Ausführung des Byte-Codes von Java-Programmen verantwortlich ist. Im Normalfall wird jedes gestartete Programm in seiner eigenen virtuellen Umgebung ausgeführt. Den restlichen Teil der JRE stellen die Java-Klassenbibliotheken dar. Die JVM dient als Schnittstelle zwischen Maschine und Betriebssystem und ist sowohl für Microsoft Windows, Linux, Mac OS X und andere Plattformen verfügbar.

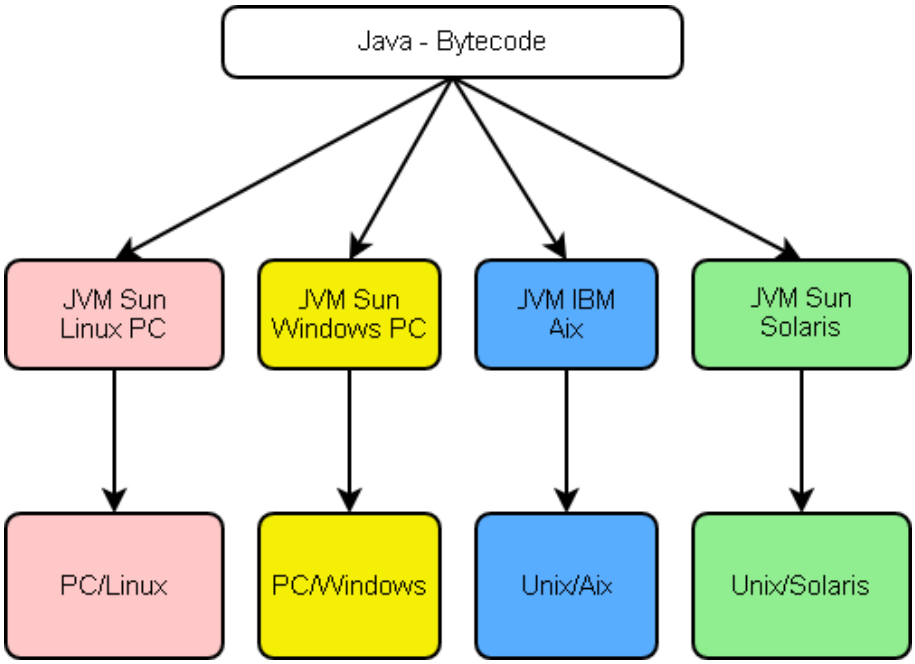

Abbildung 1.1: Java Virtual Machine als Schnittstelle

Die JVM besteht aus folgenden Teilen:

- Classloader
- Garbage Collector
- Execution Engine

1.1 Funktionsweise

Die Java Virtual Machine ist dazu da, um den Byte-Code der vom Java-Compiler erzeugt wurde, auf der jeweiligen Plattform auf der er ausgeführt werden soll ausführbar zu machen. Dazu werden die verschiedenen Byte-Code Dateien die erzeugt wurden (zu erkennen an Dateieindung "class") während der Laufzeit in die, vom jeweiligen Rechner verstandene Maschinensprache übersetzt. Die JVM arbeitet im Prinzip wie ein Interpreter, allerdings um einiges schneller, da nicht mehr auf Syntaxfehler geachtet werden muss.

1.2 Sicherheit

Die JVM ist neben der Plattformunabhängigkeit auch für ein gewisses Maß an Sicherheit verantwortlich. Sie überwacht während der Laufzeit die Ausführung der Programme und verhindert Pufferüberläufe, welche zu unvorhersehbarem Verhalten wie zum Beispiel ein Absturz des Programmes führen können. Diese Überwachung ist allerdings ziemlich einfach, da Pointer von Java nur implizit unterstützt werden.

1.3 Optimierungsverfahren

Um die Performance von Java-Programmen zu erhöhen, setzen die meisten Java-VMs sogenannte JIT-Compiler (Just-In-Time-Compiler oder JITC) ein, die unmittelbar während des Programmablaufs den Bytecode "genau zur rechten Zeit" in Maschinensprache übersetzen. Eine Weiterentwicklung ist der Hotspot-Optimizer, welcher dynamische Optimierung anbietet.

1.3.1 Dynamische Optimierung

Oft ist zum Zeitpunkt der Kompilierung nicht bekannt, welche konkreten Eingaben eine Software verarbeiten muss. Demnach muss die Software mit allen Arten von Eingaben zurechtkommen. Die Eingabe wird deshalb in Variablen gespeichert. Nach dem Start des Programms werden jedoch viele Variablen nicht mehr geändert. Folglich sind diese - von einem Zeitpunkt kurz nach dem Start an - Konstanten. Wird nun erst nach diesem Zeitpunkt die Software für die System-Architektur kompiliert (dies ist bei Java Hotspot der Fall), so können diese Konstanten berücksichtigt werden. Bestimmte Verzweigungen, die nur von solchen „Halbkonstanten“ abhängig sind, sind dann für immer eindeutig und stellen somit kein Risiko für eine falsche Sprungvorhersage dar. Ein solcher Programmcode kann also schneller ablaufen als zu früh compilierter Code.

1.4 Implementierung in Hardware

Ausführungen in Hardware sind Java-Prozessoren und Mikroprozessoren, die Java-Bytecode als Maschinensprache verwenden. Sie konnten sich gegen die schnelle Steigerung der Leistungsfähigkeit von Standard-PC und der JVM nicht durchsetzen.

1.5 Threads

Die Java VM schottet die in ihr laufenden Prozesse vom Betriebssystem ab (Green Threads). Sie bildet standardmäßig Java-Threads durch Threads des Betriebssystems ab und nur in Ausnahmefällen erfolgt das Thread-Management durch die Java VM. Somit ist es auch möglich, auf einem Betriebssystem, das kein Multithreading unterstützt, eine Java VM in vollem Funktionsumfang anzubieten.

Die Java VM hat stets volle und standardkonforme Kontrolle über die Java-Threads, d.h. der Programmierer braucht nicht die betriebssystemspezifischen Multi-Threading/Tasking/Processing-Eigenschaften zu berücksichtigen und kann sich stets auf die JRE verlassen. Nachteil ist, dass Probleme, die von einem Thread ausgehen, seitens des Betriebssystems dem gesamten Prozess zugeordnet werden. Gängige Betriebs-

systeme (wie zum Beispiel Linux, Windows) erlauben Kontrolle über diese „nativen“ Threads allenfalls über Software von Drittanbietern, wie beispielsweise die mit dem JDK mitgelieferte VisualVM. Standardwerkzeuge wie zum Beispiel der Windows Taskmanager zeigen solche Threads jedoch nicht an.

1.6 JVM kompatible Sprachen

Neben Java gibt es auch andere Sprachen, die als Programmiersprachen für JVM-Programme benutzt werden können. Unter anderem folgende Sprachen können auf einer JVM laufen:

- Clojure, ein Lisp-Dialekt
- Ceylon
- Erjang, ein Erlang-Dialekt für die JVM
- Free Pascal, das auch unter der JVM einen Großteil der Object-Pascal-Konstrukte unterstützt
- Groovy, eine dynamisch typisierte Programmiersprache
- JRuby, eine 100 % Ruby-kompatible Implementierung
- Jython (früher: JPython), ist eine reine Java-Implementierung der Programmiersprache Python
- Scala, eine Sprache, die Eigenschaften von Java mit funktionaler Programmierung vereint
- Kotlin, eine 2011 vorgestellte Sprache von JetBrains
- Jill, Kahlua (LUA)
- Rhino (JavaScript)
- Jacl (Tcl)
- Quercus (PHP)
- JavaFX

[Ull11][Wik][Ora]

2 Die Bestandteile der Java Virtual Machine

Wie zuvor bereits erwähnt besteht die JVM aus folgenden Teilen:

- Classloader
- Garbage Collector
- Execution Engine

Auf diese wird in den folgenden Punkten genauer eingegangen.

2.1 Classloader

2.1.1 Allgemeines

Der ClassLoader konstruiert Class-Objekte (java.lang.Class) aus Bytecode (byte-Array). Eine Klasse in Java wird durch seinen voll qualifizierten Namen UND dem ClassLoader (der ihn geladen hat) identifiziert. Nicht allein durch seinen voll qualifizierten Namen. Daher ist es möglich, dass mehrere Klassen mit dem selben Namen in einer VM gleichzeitig existieren, solange sie von verschiedenen ClassLoadern geladen wurden. Zwei Klassen, die mit verschiedenen ClassLoadern geladen wurden, sind von verschiedenen Typen (inkompatibel!), selbst wenn sie von ein und dem selben *.class-file geladen wurden. Dadurch können mehrere Klassen mit den gleichen Namen gleichzeitig existieren. Sie müssen lediglich durch andere ClassLoader geladen werden. In der Praxis instantiiert man einfach immer einen neuen ClassLoader. Somit können Namen "wiederverwendet"werden:

```
1 Class<?> autoClass = new MyClassLoader().loadClass("pkg.Auto
    ");
```

java.lang.ClassLoader ist eine abstrakte Klasse. Um selbst Klassen zur Laufzeit zu laden, muss ClassLoader erweitert werden. Wenn die JVM hochfährt, lädt der SystemClassLoader alle Klassen im ClassPath. Schreibt man z.B. "new Auto()"wird der SystemClassLoader diese Klasse laden. Mit der statischen Methode ClassLoader.getSystemClassLoader() kann eine Referenz auf den SystemClassLoader erhalten werden:

```
1 ClassLoader systemClassLoader = ClassLoader.
    getSystemClassLoader();
2 System.out.println(systemClassLoader); //"sun.misc.
    Launcher$AppClassLoader@11b86e7"
```

Liefert <Objekt>.getClass().getClassLoader() null zurück, so wurde diese Klasse vom SystemClassLoader geladen.

2.1.2 Delegation

Die ClassLoader verfolgen das Konzept der Delegation. Er hat immer einen Parent-Classloader. Wenn der ClassLoader eine Klasse laden soll, delegiert er die Aufgabe an seinen Parent. Wenn der Parent die Klasse nicht finden kann, versucht er selbst die Klasse zu laden. Der oberste Parent in der Delegationshirarchie ist der SystemClassLoader.

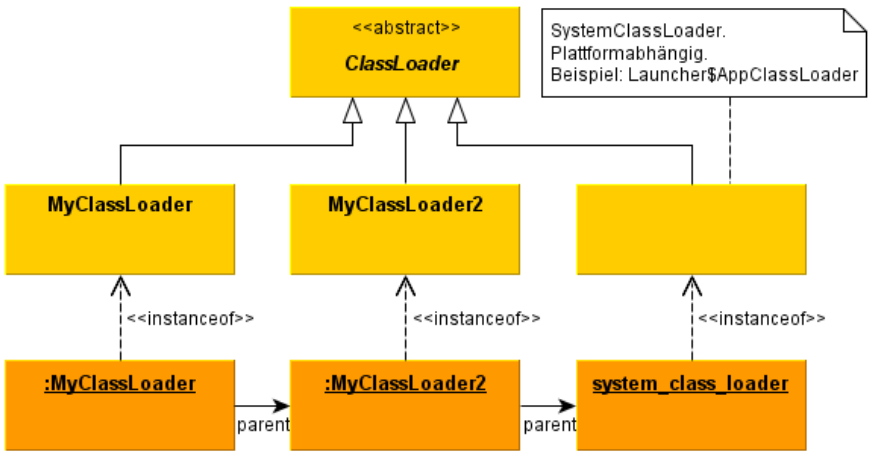

Abbildung 2.1: Vererbungshierarchie von ClassLoader und die Parentassoziation

2.1.3 API

Innerhalb des Classloader-Objektes existieren einige Methoden. Eine der wichtigsten stellt die loadClass() Methode dar. Mit dieser beginnt der Prozess des eigentlichen Ladens. Die JVM lädt dabei alle Klassen, in der sie loadClass() aufruft. Im folgender Grafik wird die Funktionsweise der Defaultimplementierung geschildert.

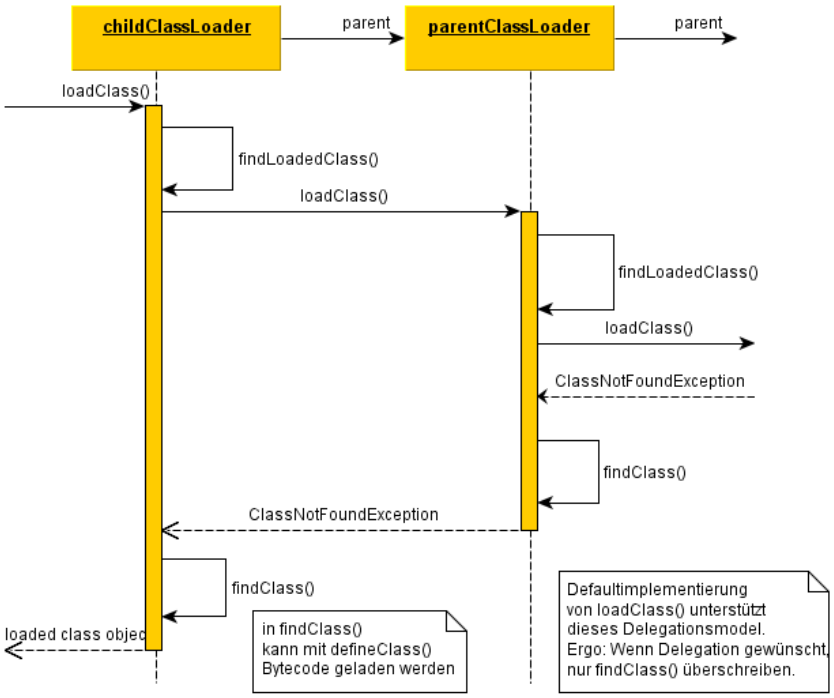

Abbildung 2.2: Funktionsweise von loadClass() als Sequenzdiagramm

1. Aufruf von findLoadedClass(), um zu checken, ob die Klasse nicht bereits vom ClassLoader selbst geladen wurde. ClassLoader merken sich die Klassen, die sie selbst geladen haben. Subklassen erben dieses Verhalten.

2. Wenn keine Klasse gefunden wurde, wird loadClass() auf dem Parent-ClassLoader aufgerufen. Die Defaultimplementierung von loadClass() unterstützt somit das Delegationsmodel.

3. Wenn wieder keine Klasse gefunden wurde, wird findClass() aufgerufen. An dieser Stelle sollte man sich in seiner ClassLoader-

Subklasse in den Loadingprozess einharken, wenn man das Delegationsmodell erhalten möchte. Mit der defineClass()-Methode kann hier eine Class aus Bytecode erzeugt werden.

4. Wenn einer dieser Schritte keine Class gefunden hat, wird eine ClassNotFoundException geworfen.

[Hau]

2.2 Garbage Collector

Garbage Collection beschreibt die automatische Speicherbereinigung um den Speicherbedarf eines Java-Programms (auch in anderen Sprachen) zu minimieren. Dies wird zur Laufzeit bewerkstelligt, wo versucht wird nicht benötigte Speicherbereiche automatisch zu erkennen und freizugeben. Der größte Vorteil von Garbage Collection ist die Vermeidung von Fehlern die bei einer manuellen Speicherverwaltung sehr leicht auftreten können. Dies geht allerdings auch mit einem gewissen Verwaltungsoverhead und oftmals nicht reproduzierbarem Verhalten der Umgebung und damit auch des Programms selbst ein her.

Die Spezifikation einer JVM schreibt vor, dass in jeder Implementierung einer JVM ein Garbage Collector enthalten sein muss. Die Effizienz dieses Garbage Collectors kann großen Einfluss auf die Performance des Java Programms haben.

2.2.1 Generational Garbage Collection

Alle uns bekannten, heutigen Garbage Collectoren in den virtuellen Maschinen für Java (von Sun) sind Generational Garbage Collectoren; dieser Ansatz hat sich in der Praxis als der beste Algorithmus für Garbage Collection in Java erwiesen. Die Idee der Generational Garbage Collection besteht darin, dass man den Heap-Speicher in verschiedene Bereiche einteilt, die jeweils Objekte unterschiedlicher Alterklassen enthalten. Mit speziellen auf das Alter der Objekte abgestimmten Algorithmen werden die jeweiligen Bereiche verwaltet und aufgeräumt.

Da man in Java (anders als beispielsweise in C++) keine Objekte

auf dem Stack anlegen kann, müssen alle Objekte auf dem Heap alloziert werden. Selbst Objekte, die nur lokal in einer Methode gebraucht werden, entstehen auf dem Heap. Die meisten dieser Objekte leben nicht lange: beim Verlassen der Methode werden sie bereits unerreichbar und sind damit "tot". Es gibt aber auch Heap-Objekte, die bis zum Ende der Laufzeit leben, weil sie für die gesamte Ablaufzeit der Applikation gebraucht werden, zum Beispiel final static Attribute einer Klasse. Es gibt sehr viele Java-Objekte, die nicht sehr alt wer-

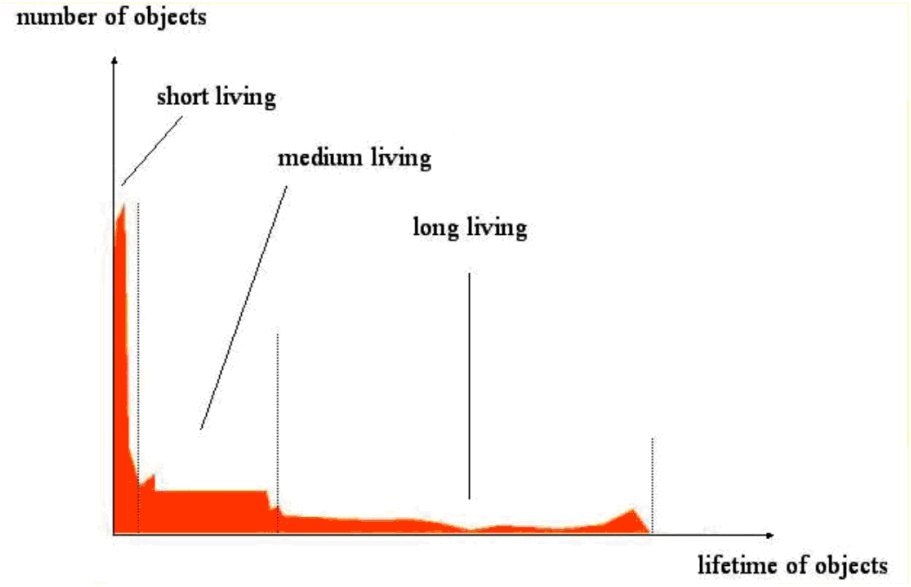

Abbildung 2.3: Typische Objekt-Population in Java

den, und es gibt vergleichsweise wenig Objekte, die sehr lange leben. Die Objekte mit kurzer Lebensdauer sind die, die nur innerhalb einer Methode oder manchmal nur innerhalb einer Expression gebraucht werden. Beispiele sind Iteratoren in einer Schleife oder StringBuilder fürs Zusammensetzen eines Strings. Die Objekte mit mittlerer Lebensdauer sind solche, die in nicht-stackbasierten Verarbeitungen benutzt werden. Ein Beispiel wäre der Conversational Session State einer Entity Bean. Er überlebt mehrere Methodenaufrufe und wird erst nach der Session nicht mehr gebraucht. Man beachte, die Zahl der Objekte mit mittlerer Lebensdauer ist deutlich geringer als die der kurzlebigen Objekte. Richtig alt werden nur ganz wenige Objekte. Das sind typischerweise Objekte, die schon beim Programmstart (oder per Lazy Evaluation etwas später) erzeugt werden und dann bis zum Ende der Anwendung in Gebrauch bleiben. Dazu gehören Thread Pools, Single-

tons und Objekte aus Frameworks, z.B. Servlet-Instanzen.

Um den unterschiedlichen Objekt-Lebenszeiten angemessen Rechnung zu tragen, werden die Objekte verschiedenen Generationen zugeteilt. Die Generationen sind separate Bereiche des Heaps, die unterschiedlich verwaltet werden. Sie haben unterschiedliche Allokatoren und sie werden unterschiedlich oft und mit jeweils eigenen Garbage-Collection-Algorithmen aufgeräumt werden. Im Wesentlichen unterscheidet man zwischen einer Young Generation, die häufig bereinigt wird, und einer Old Generation, die seltener bereinigt wird. Die Young Generation ist der Speicherbereich, in dem die neuen Objekte angelegt werden, z.B. wenn im Programm mit new Speicher angefordert wird. Die Old Generation ist der Bereich, in den die jungen Objekte ausgelagert werden, wenn sie ein gewisses Alter erreicht haben, d.h. wenn sie eine bestimmte Anzahl von Garbage Collections überlebt haben.

Warum ist es sinnvoll, dass die beiden Generationen unterschiedlich häufig und auch mit unterschiedlichen Algorithmen aufgeräumt werden? Dahinter steht die Beobachtung, dass von den jungen Objekten die meisten relativ schnell sterben, wie man in der Abbildung 1 gesehen hat, sodass in der Young Generation schnell viel Garbage entsteht. Da, wo viel Garbage entsteht, lohnt sich zügiges Aufräumen, weil dann rasch wieder freier Speicher zur Verfügung steht. Die Aufräumarbeiten auf die Young Generation zu beschränken, hat außerdem den Vorteil, dass nicht immer der gesamte Heap nach „toten“ Objekten durchforstet werden muss, sondern dass bereits das Aufräumen in einem Teilbereicht des Heaps signifikante Mengen an Speicher freigibt.

In der Old Generation lohnt sich das häufige Aufräumen nicht. Wenn ein Objekt erst einmal mittelalt geworden ist, dann wird es mit hoher Wahrscheinlichkeit auch noch älter. In der Old Generation findet man deshalb nicht so häufig „tote“ Objekte und damit weniger Potential für die Speicherfreigabe. Also macht man die Bereinigung der Old Generation wesentlich seltener.

Die Bereinigungen in der Young Generation werden als Minor Collections bezeichnet. Daneben gibt es die seltener durchgeführten Major oder Full Collections, bei denen sowohl die Young als auch die Old

Generation bereinigt wird. Diese Unterscheidung kann man bereits sehen, wenn man die virtuelle Maschine mit der Option –verbose:GC aufruft. Die virtuelle Maschine gibt dann Information über die Garbage Collection aus. Man kann sehen, dass eine Full Garbage Collection

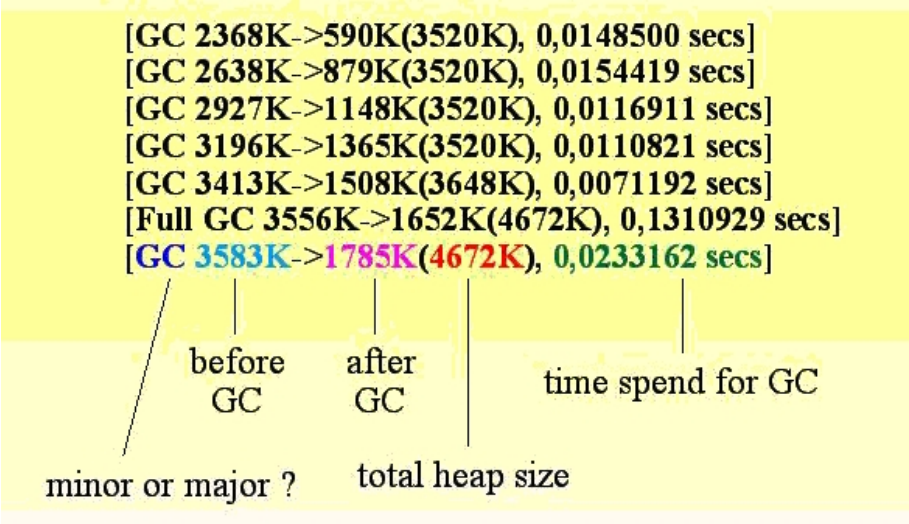

Abbildung 2.4: Garbage Collection Information per JVM-Option -verbose:GC

selten durchgeführt wird, deutlich länger dauert als die Minor Garbage Collections, aber keineswegs mehr Speicher freischaufelt als eine Minor Garbage Collection.

2.2.2 Die Heapaufteilung in einer JVM

Seit Java 1.3 ist der Speicher in 2 Generationen (young und old) und einen Sonderbereich (perm) eingeteilt. [Lan]

2.3 Execution Engine

[Ora]

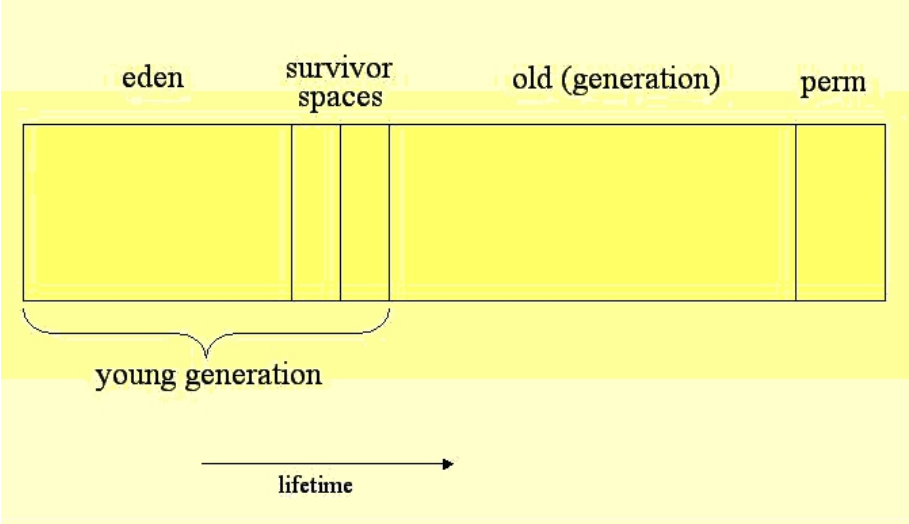

Abbildung 2.5: Heap-Aufteilung in einer JVM

Literaturverzeichnis

[Hau] Hauer, Philipp: *Classloader* - *http://www.philipphauer.de/study/se/classloader.php*

[Lan] Langer, Angelika: *Garbage Collector* - *http://www.angelikalanger.com/Articles/EffectiveJava/49.GC.GenerationalGC/49.GC.GenerationalGC.html*

[Ora] Oracle: *Java Virtual Machine* - *http://docs.oracle.com/javase/7/docs/technotes/guides/vm/index.html?intcmp=3170*

[Ull11] Ullenboom, Christian: *Java 7-Mehr als eine Insel.* Galileo Computing, 2011

[Wik] Wikipedia: *Java Virtual Machine* - *http://de.wikipedia.org/wiki/Java_Virtual_Machine*