# Java Speicherverwaltung

Tobia Ippolito

18. Juli 2020

# Inhaltsverzeichnis

# 1 Java: Hinter den Kulissen

## 1.1 Einführung

In dieser Arbeit geht es um die Details, wie eine Java-Applikation ausgeführt wird beziehungsweise was während der Ausführung passiert. Also ein Blick hinter die Kulissen von Java.
Wir werden sehen, wie ein Java-Code verständlich gemacht wird und was im Speicher genauer passiert.
Hierzu wird im zweiten Chapter genaueres zur Java Virtuel Machine stehen und den Weg vom Code zum Programm.
In den nächsten drei Kapiteln, gehe auf die genaue Funktionsweise der JVM ein. Unteranderem werden wir sehen, wie Java alles nötige ladet und wie die Speicherverwaltung funktioniert.

Ich finde dieses Thema sehr spannend und sehe es als ein notwendiges Basis-Wissen eines Java-Programmierers an.
Natürlich kann man auch ohne dieses Wissen hervorragende Software produzieren, aber mit dem hier stehendem Wissen, kann man sich vieles besser herleiten und vor allem verstehen, was man da eigentlich im Detail macht. Für die Suche nach Fehlern ist das sehr Wertvoll.
Mir hat es jedenfalls geholfen, aber entscheiden Sie lieber selbst, wie interessant und hilfreich Sie diese Arbeit finden.

Um dieses theoretische Wissen zu untermauern, habe ich mehrere kleine Projekte gemacht. Man kann unter anderem sehen, für was dieses Wissen genutzt werden kann und außerdem lernt man die Grenzen von Java kennen.

## 1.2 Java Durchführungverlauf (JVM)

So bevor wir mit der Speicherverwaltung und den anderen Funktionsweisen der JVM durchstarten können, müssen wir erst einmal verstehen was die JVM ist.
Hierzu fangen wir ganz von Beginn an, wir haben also gerade ein hypothetisches Programm in der Hochsprache Java geschrieben. Die Datei trägt die Endung .java und damit kann eine Siliziummaschine noch nichts wirkliches anfangen. Sie versteht nur 1 und 0. Durch digitale Logik noch etwas mehr.
Hierzu muss man auch die Architektur von Computern verstehen. Diese sind in verschiedenen Ebenen aufgebaut. Jede Ebene baut auf die darunterliegende Ebene auf und abstrahiert ihre Inhalte. Die unterste ist eben die digitale Logik und recht weit oben lässt sich die objektorientierte Programmiersprache Java finden. Man muss die geschrieben Inhalte dieser Sprache also immer weiter herunterbrechen, bis der Rechner dies verstehen kann.
Hierzu gibt es zwei Methoden, um menschenfreundlichen Code in maschinenfreundlichen Code umzuwandeln. Der Code kann kompiliert oder interpretiert werden.
Beim Kompilieren wird der Quellcode auf Fehler in der Syntax überprüft und schließlich optimiert und zu Maschinencode transformiert. Dieser kann dann ausgeführt werden. Hierbei gibt es das Problem, dass die Ausführung von System zu System unterschiedlich sein kann und somit speziell für ein System kompiliert werden muss. Diese Inkompatibilität ist nicht gerade schön, aber meist performanter.
Beim Interpretieren wird der Quellcode direkt von einem Interpreter durchgeführt und besitzt den Vorteil, dass der Quellcode auf jedem System (mit dem Interpreter) ausgeführt werden kann. Dabei geht er Zeile für Zeile durch und ist dabei langsamer als das Kompilieren.
Java will nicht in eine Schublade gesteckt werden und Vorteile beider Varianten vereinen. Dazu wird unsere eben erwähnte .java Datei in speziellen Java-Byte-Code kompiliert. Der Code wird hierbei auf Fehler überprüft und optimiert. Dieser Java-Byte-Code ist jedoch nicht wie üblicher kompilierter Code vom Computer ausführbar. Der kompilierte Code muss nämlich von einem Java-Interpreter interpretiert werden. Hier hat die Java Virtuel Machine ihren großen auftritt. Sie interpretiert den Java-Byte-Code und führt diesen Zeile für Zeile aus.
Aus diesem Grund sind Java-Programme so unabhängig und deswegen schreit Java auch meist so aufdringlich nach Updates (wie schon gesagt: der Interpreter muss nämlich auf dem System installiert sein, damit ein Java-Programm ausgeführt werden kann).

Nun wissen wir schon, was die JVM grob macht. Nun noch ein paar weitere Informationen über die JVM.
Die JVm ist meist in der Programmiersprache C oder in C++ geschrieben.
Die JVM besteht aus dem Klassenlader, der Speicherverwaltung und der Ausführungseinheit. Auf sie gehe ich später genauer ein.
Die JVM startet Java-Programme in ihrer eigenen virtuellen Umgebung (VM). Mit virtuellem Speichersystem, welches wir noch kennenlernen werden. Dabei laufen diese in eigenen Threads (Ausführungsstränge) und werden von Java verwaltet. Somit funktionieren Java-Programme auch bei Betriebssystemen, die kein Multithreading unterstützen. Außerdem muss sich der Programmierer nicht mit spezifischen Multithreading beschäftigen, da es immer über die JVM geht. Ein Nachteil ist hierbei, dass Fehler nun nicht einem spezifischen Thread zugesprochen werden können, sondern nur dem Programm im Ganzen. Schließlich sieht das Betriebssystem nur, dass das Programm einen Fehler erzeugt hat und hat keinen Einblick in die interne Verwaltung.
Zudem ist die JVM ein wichtiger Teil der Java Runtime Environment (JRE). Die JRE ist eben genau das was auf den Rechnern installiert werden muss, um Java-Programme abspielen zu können. Sie besteht, wie erwähnt, aus der JVM und zusätzlich aus der Klassenbibliothek von Java (Java-API).
Um Verwirrungen auszuschließen, gehe ich noch kurz auf die Begriffe 'JDK' und 'JFC' ein. 'JDK' steht für Java Development Kit und beinhaltet die JRE und zusätzliche Tools für die Entwicklung von Java Programmen. Beispielsweise einen Compiler (javac), damit der geschriebene Code noch in Java-Byte-Code umgewandelt werden kann.
Zudem trifft man häufig auf 'JFC' oder ähnliches. Hierbei handelt es sich häufig um Programmierschnittstellen. Also Programmbibliotheken. Bei 'JFC' handelt es sich beispielsweise um die Java Foundation Classes, welches ein GUI-Framework ist. Es beinhaltet javax.swing oder auch java.awt / Java2D.
Damit hätten wir die Grundlage von Java gelernt und können die JVM einordnen. Nun geht es ins Detail gehen.

## 1.3 Speicherbereiche (Runtime Data Area)

Die Java Virtuel Machine bietet verschiedene Speicherbereiche. Diese sind sehr wichtig für das Verständnis von seinen Programmen und Java selbst.
Um genau zu sein, gibt es 5 verschiedene Speicherbereiche. **Methoden** Bereich, **Heap** Speicher, **Stack** Speicher, **PC Register** und der **native Methoden Stack**.
Für einen Programmierer sind vor allem der Heap und der Stack sehr wichtig.

### 1.3.1 Heap

Der Heap ist ein sehr wichtiger Speicherbereich. Man kann auf ihn in der gesamten JVM zugreifen.
Im Heap werden Objekte und all ihre Daten (Variablen,...) gespeichert. Und wie wir wissen, geht es in Java immer um Objekte und somit kann man diesem Speicherbereich eine große Wichtigkeit zuschreiben.
Verwaltet wird dieser Speicherbereich von einem Speicher-Managment-System, namens **Garbage Collector**. Auf diesen werde ich im Kapitel 'Ausführungseinheit' eingehen.

### 1.3.2 Metaspace (Method Area)

Dieser Bereich ist nicht Teil des Heaps und speichert Metadaten von Klassen ab.
Darunter zählt der Code von Methoden, den Konstruktor-Code, lokale Daten einer Methode, Felder und Konstanten.
Außerdem werden hier auch die symbolischen Referenzen gespeichert (Konstanten Pool).
Dieser Speicherbereich ist in der ganzen JVM (allen Ausführungsstränge) erreichbar.

### 1.3.3 Stack

Der Stack beinhaltet lokale Variablen. Oder besser: hier sind die Referenzen auf den Heap/Objekte und primitive Variablen gespeichert.
Der Stack variiert in seiner Laufzeit aber ziemlich, da er nur die aktuelle Methode und dessen lokale Variablen enthält.
Zu beachten ist auch, dass jeder Ausführungsstrang einen eigenen Stack besitzt!
Außerdem genießt dieser Speicherbereich einen schnellen Zugriff.

### 1.3.4 Nativer Methoden Stapel

Hier werden Methoden (dessen lokale Variablen) gespeichert, welche nicht in Java programmiert wurden und nicht von der JVM interpretiert werden.

### 1.3.5 PC Register

Der 'Program Counter' Register zeigt auf die aktuelle Ausführung des Programms.
Dabei hat jeder Ausführungsstrang einen eigenen PC Register, welcher wie gesagt auf das aktuell Ausgeführte zeigt.
Falls in der Laufzeit ein neuer Thread erzeugt und gestartet wird, so wird ein neuer PC Register angelegt.

## 1.4 Klassenlader (Classloader)

Der Klassenlader macht das, was sein Name schon erahnen lässt. Er lädt java.class Dateien in den (virtuellen) Speicher (und ein bisschen mehr). Deswegen ist er chronologisch gesehen die erste Komponente der JVM, welche zum Einsatz kommt.
Dem Classloader ist es also zu verdanken, dass die JVM das File-System nicht kennen muss um trotzdem zu funktionieren.
Dabei gibt es aber verschiedene Arten von Klassenladern.
Der Classloader besteht dabei aus dem **Laden**, dem **Verbinden/Verknüpfen** und dem **Initialisieren** von Klassen.

### 1.4.1 Laden

Betrachten wir zu Beginn das **Laden**. Hierbei gibt es drei Typen. Den Bootstrap-Classloader, den Extention-Loader und den System-Classpath-Loader (Application-Loader).
Letzterer ist am einfachsten, denn die .class Dateien (kompilierte .java Dateien) lädt der System-Classpath-Loader in den virtuellen Speicher (also in die JVM). Diese Datei muss von Java interpretiert werden, wie wir das zu Beginn gelernt haben.
Kommen wir nun zum Bootstrap-Classloader. Dieser ist für das Laden der Kern-Klassen von Java (Klassenbibliothek) zuständig. Darunter befindet sich auch die rt.jar (rt für RunTime) Library.
Als Beispiel könnte man da System.out.println(String s); nehmen. Dieser Befehl arbeitet nahe am System und wird durch diesen Bootstrap-Classloader in den Speicher geladen.
Der Bootstrap-Classloader ist also ein Kernbestand des Klassenladers, aber er stellt auch die Grundlage für die anderen Klassenlader-Instanzen dar.
Der letzte Klassenlader ist der Extention-Classloader und wird meist aber nicht gebraucht. Er erweitert die Kernklassen, die durch den Bootstrap-Classloader geladen wurden und ladet die erweiterten Klassen in den Speicher. Diese Klassen befinden sich bei der Java Runtime Environment: /jre/lib/ext .
Wenn man beispielsweise mit einer Datenbank von Oracle arbeitet, werden die dafür benötigten Klassen hier geladen.

Diese Klassenlader sind Objekte (java.lang.ClassLoader) und stehen in einer Hierarchie. So ist der System-Classpath-Classoader der unterste Classloader und der Bootrap-Classloader der Oberste.
Ein Programm kommt meist nur mit dem System-Classpath-Classloader in Kontakt. Falls nun eine Klasse geladen werden soll, so gibt dieser die Anfrage weiter. So kommt es, dass der in der Hierarchie am höchsten Classloader immer zuerst geprüft wird. Somit wird auch sichergestellt, dass keine Klasse in der Klassenbibliothek nochmals existieren kann (2 Klassen mit den selben Namen).

### 1.4.2 Verknüpfen

Als nächsten betrachten wir das **Verknüpfen** des Klassenladers. Hierbei werden die geladenen Klassen zunächst geprüft. Es wird geschaut, ob sie den Standards entsprechen. Also ob beispielsweise die Struktur einer geladenen Klasse korrekt ist.
Anschließend kommt man in die Vorbereitungsphase. Hier werden die statischen und die normalen Variablen dem Speicher zugewiesen.

Die letzte Phase der Verknüpfung, macht das Auflösen der symbolischen Referenzen zu wirklichen Werten. Diese Anweisungen: anewarray, checkcast, getfield, getstatic, instanceof, invokedynamic, invokeinterface, invokespecial, invokestatic, invokevirtual, ldc, ldc_w, multianewarray, new, putfield, und putstatic, der JVM verweisen symbolisch auf einen Pool aus Konstanten während der Laufzeit. Um diese Anweisungen ausführen zu können, muss man die symbolische Referenz auflösen, um einen konkreten Wert zu erhalten. Genau das passiert in dieser Phase.

### 1.4.3 Initialisierung

Schlussendlich kommen wir zum letzten Part, der **Initialisierung**. Hier werden die Klassen nun initialisiert. Das passiert, indem statische Variablen und statische Methoden ausgeführt beziehungsweise ihrem Wert zugewiesen werden.
Nach der Initialisierung ist das Programm startklar.

## 1.5 Ausführungseinheit (Execution Engine)

Nun kommen wir zum letzten Teil der JVM. Wie wir ja wissen, interpretiert die JVM Java-Bytecode und hierfür gibt es diese Komponente.
Beinhalten tut diese eben einen **Interpreter**. Dieser liest class-Dateien und führt jeden Befehl nacheinander aus.
Dabei muss der Interpreter jeden Befehl neu interpretieren, egal ob er ihn schon mal interpretiert hat.
Deswegen hat diese Ausführungseinheit einen **JIT-Compiler** (Just-In-Time). Dieser übersetzt während der Laufzeit Java-Bytecode in Maschinencode, welcher von dem Betriebssystem verstanden werden kann (nativer Code).
Bei mehreren aufrufen einer Methode, kann dieser Compiler nachhelfen und verbessert so die Performance von Java-Programmen. Ansonsten müsste der Code immer neu übersetzt werden. Zudem verbessert der JIT-Compiler die Performance dieses Codes.
Dafür besitzt Java ein Interface (**JNI**) das die Verwendung von native Bibliotheken ermöglicht und diese können dann beispielsweise für den JIT-Compiler verwendet werden.
Zusätzlich enthält die Ausführungseinheit einen **Profiler**. Dieser wacht über sehr viele verschiedene Dinge. Durch ihn ist es möglich, einfach interne Daten zu erhalten. Diese können dann für Trouble Shooting und generelle Analysen verwendet werden.

Zu Letzt bietet die Ausführungseinheit eine Speicherverwaltungs-System namens Garbage Collector. Dabei werden wir lernen, wie der Heap in der Laufzeit aussieht und sich verändert.

### 1.5.1 Garbage Collector

Wie schon angesprochen kümmert sich der Garbage Collector um den Heap-Speicher. Er sorgt dafür, dass dieser Speicherbereich nur von 'lebenden' Objekten beherbergt ist.
Dabei war Java einer der ersten Programmiersprachen, die so ein Feature besaß. Der GC zählt immer noch zu einer der Effizientesten und lässt Java noch oben mit spielen.
Doch wie erkennt der GC ob ein Objekt noch aktiv ist?
Die einfache, aber sehr unperformante Variante ist die **Mark and Sweep Collection**. Dies geschieht über Referenzen und über sogenannten 'Garbage Collection Roots'. Diese Roots sind der Startpunkt des GC's.
Der GC geht diese ab und markiert dort, alle Objekte die noch referenziert sind. Wenn man keine Variablen mehr auf ein Objekt hat, so kann man ja auch nicht mehr mit dem Objekt interagieren und somit handelt es sich um Müll.
Die Roots die der GC durchgeht und dessen Objekte markiert werden, sind Class-Objekte, lebendige Threads, lokale Variablen und spezielle JVM-Objekte.
Somit markiert der GC alle erreichbaren Objekten. Alle nicht markierten Objekten werden gelöscht. Ohne Referenzierung kann man ja auch nicht mehr mit dem Objekt interagieren.

In modernen Java-Programmen ist es aber etwas komplizierter. Hier will man nicht immer alle Objekte überprüfen, deswegen gibt es verschiedene Generationen mit eigenen Bereichen.
Es gibt die **Young Generation** und die **Old Generation**.
Die Young Genaration besteht aus den Bereichen 'Eden', 'Survivor Space 1' und 'Survivor Space 2'. Die Old Generation besteht lediglich aus dem 'Tenured Space'.
Falls ein Objekt nun neu erstellt wird, so ist es im Bereich 'Eden' der Young Generation.
Eine **Minor Garbage Collection** wird dann ausgeführt, wenn es in Eden zu voll wird. Dann werden die Objekte, welche noch erreichbar sind (Referenzen) von Eden in den 'Survivor Space 1' gespeichert.
Alle Objekte in Eden können dann gelöscht werden und falls nun Eden wieder voll ist, so wird wieder eine Minor Garbage Collection durchgeführt. Die Objekte mit Referenzen darauf, werden nun in den anderen Survivor Space gebracht und Eden wird wieder geleert. Zudem werden die Objekte im Survivor Space 1 geprüft, ebenfalls in den anderen Survivor Space gebracht und dann geleert.
Objekte, welche lange in der Young Genrenation überleben, werden in die Old Generation in den Tenured Space befördert.
Dort werden sie seltener geprüft und zwar bei einem **Major Garbage Collection**.
Somit werden die langlebigen Objekte seltener geprüft und dies steigert die Performance.
Das Verschieben der Objekte hat auch den Vorteil, dass es immer einzelne Speicherblöcke gibt. Der Zugriff ist damit beschleunigt.

Nun konnten wir nähere Einblicke in die Speicherverwaltung des Heaps erhalten. Die ganze Wahrheit ist tatsächlich noch mal etwas komplizierter. Es gibt viele Faktoren die den Algorithmus des GC beeinflussen und somit ist er viel flexibler und situationsspezifischer.

# 2 Java Funktionsweise in der Praxis

In diesem Bereich werden weitere praktische Anwendungsmöglichkeiten der besprochenen Theorie gezeigt und zudem wird hier die Theorie praktisch gezeigt. So können wir sehen, dass die Theorie stimmt und noch ein paar Sachen dazu lernen.

## 2.1 Klassenlader relativ im Einsatz

## 2.2 Array und ArrayList

## 2.3 Threads

## 2.4 Wir sammeln Müll ein

### 2.4.1 Objekte zerstören

# 3 Quellen

**1)** Sachliteratur geschrieben von Kai Günster über fast alle Themen was Java anbelangt.
ISBN: 978-3-8362-4095-6, 'Einführung in Java'

Bewertung: Bei diesem Werk handelt es sich um ein tolles Werkzeug für Anfänger. Aber auch Fortgeschrittene können hieraus noch einiges lernen. Der Sachtext ist gut strukturiert. Außerdem wird hier viel von Erklärungen gehalten und das ist super. So versteht man was man da eigentlich macht. In meinem Projekt hat es an mehreren Stellen Verwendung gefunden.

Verwendung: Dieses Werk konnte ich an mehreren Stellen zur Rate ziehen. Bei allen Ungereimtheiten habe ich hier nachgeschlagen. Direkt entnommen habe ich Informationen über den Klassenlader und den Garbage Collector.

**2)** YouTube-Video über die JVM.
https://www.youtube.com/watch?v=QHIWkwxs0AI (stand 12.07.2020)

Bewertung: Das Video ist klar strukturiert und visualisiert sehr anschaulich die Themen. Jedoch musste ich oft noch externe Quellen verwenden, da er sich oftmals schwammig ausdrückt.

Verwendung: Funktionsweise der JVM.

**3)** YouTube-Video über die Funktionsweise der JVM.
https://www.youtube.com/watch?v=ZBJ0u9MaKtM (stand 12.07.2020)

Bewertung: In diesem Video wird die Funktionsweise der JVM strukturiert durchgegangen. Dabei wird das Medium 'Video' vollends ausgenutzt und der Inhalt grafisch dargestellt.

Verwendung: Funktionsweise der JVM.

**4)** Online Artikel von Oracle über den Classloader.
https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html (stand 12.07.2020)

Bewertung:Der Artikel ist sehr detailreich, aber auch schwer leserlich und unübersichtlich.

Verwendung: Funktionsweise der Classloader.

**5)** Online Artikel von Oracle über den die Speicherbereiche der JVM.
https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html (stand 13.07.2020)

Bewertung: Der Artikel ist sehr sachlich verfasst und prallt mit Details. Für ein gutes verständnis empfehle ich den Vergleich mit anderen Quellen.

Verwendung: Für die Speicherbereiche der JVM.

**6)** Ein Online-Artikel über die Speicherbereiche der JVM von 'Joe'.
https://javapapers.com/core-java/java-jvm-run-time-data-areas/ (stand 13.07.2020)

Bewertung: Der Artikel ist übersichtlich und verständlich gestaltet. Die Details lassen - aber zum Gewinn des Verständnisses - zu wünschen übrig.

Verwendung: Für die Speicherbereiche der JVM.

**7)** Ein Online-Artikel über die Architektur der JVM von Bill Venners.
https://www.artima.com/insidejvm/ed2/jvm2.html (stand 13.07.2020)

Bewertung: Dieser Artikel ist verständlich geschrieben und visualisiert dessen Inhalte ganz gut.

Verwendung: Ein klein bisschen für die Speicherbereiche der JVM.