# Java Class Loader

Seminararbeit zur Lehrveranstaltung
**Verteilte und parallele Systeme 1**
bei Prof. Rudolf Berrendorf

Von Jan Seidel, Stefan Winarzki


Fachbereich Angewandte Informatik
Fachhochschule Bonn-Rhein-Sieg, St. Augustin

**Inhaltsverzeichnis:**

# 1. Einleitung

Diese schriftliche Ausarbeitung befasst sich mit dem Thema Java Class Loader. Unser Hauptaugenmerk liegt auf dem Ladevorgang selbst, jedoch werden auch sämtliche damit in Verbindung stehenden sicherheitsrelevanten Themen erläutert. Des weiteren wird, um zum Verständnis beizutragen, erläutert, wie man seinen eigenen benutzerdefinierten Class Loader erstellen kann. In einem weiteren Punkt werden einige Class Loader Klassen der Java API, speziell die abstrakte Klasse JAVA.LANG.CLASSLOADER, untersucht und deren Funktionen genau erklärt.

## 1.1 Motivation

Der Grund, warum man überhaupt etwas wie den Class Loader braucht ist, dass man den Benutzer von Java-Programmen vor der Ausführung schädlichen Codes schützen will. Die ungehinderte Ausführung solchen Codes könnte z.B. beliebigen Zugriff auf das lokale Dateisystem oder den Absturz der Java Virtual Machine zur Folge haben. Jedoch wird durch ein gut durchdachtes Sicherheitskonzept, in dem der Class Loader eine tragende Rolle spielt, weitgehend gewährleistet, dass so etwas nicht passieren kann.

## 1.2 Geschichtsrückblick

Der Grundgedanke von Java ist es, dass ausführbarer Code im Internet verfügbar ist und als Applet, welches von einem bestimmten Server geladen wird, abgerufen werden kann. Das Applet selbst läuft jedoch nur durch die Virtual Machine, die auf dem Client-Rechner installiert sein muss. Da also diese Applets frei im WWW zugänglich sein müssen, wurde von den Java-Entwicklern großer Wert auf Sicherheitsaspekte gelegt, denn es muss stets gewährleistet sein, dass ein Java-Programm keinen für den Client-Rechner schädlichen Code enthält.

Java 1.0: In der ersten Java Edition wurde das Sicherheitskonzept der sog. 'Sandbox' vorgestellt. Hierbei wurde der Code in zwei verschiedene Klassen aufgeteilt: lokal (local) und entfernt (remote). Der lokale Code hatte immer vollen Zugriff auf alle Ressourcen des Systems, auf dem er mittels der Virtual Machine ausgeführt wurde, während der entfernte Code stark in seinen Zugriffsrechten beschränkt wurde (Platzierung in der 'Sandbox' zur Begrenzung der Zugriffsrechte). Dieses Prinzip brachte dem Benutzer natürlich die Sicherheit, dass ein Java-Programm niemals unauthorisierte Operationen ausführen könnte, jedoch gab es keine Möglichkeit zur Feinabstimmung. Die starken Beschränkungen wirkten sich auf jeglichen entfernten Code aus, so dass man keine Möglichkeit hatte, zwischen vertrauenswürdigem und unsicherem Code unterschiedliche Vorgehensweisen zu vereinbaren. Zu diesem frühen Zeitpunkt war der Class Loader schon vorhanden, jedoch war er nur zum Laden von Applet Klassen im Hotjava Browser geplant. Seitdem wurden die Funktionen des Class Loaders kontinuierlich erweitert, wie z.B. das Laden von Programmen auf Serverseite (Servlets), Erweiterungen der Java Platform und Komponenten von JavaBeans.

Java 1.1: Der letzte angesprochene Nachteil von Java 1.0 wurde in der neuen Java-Version 1.1 verbessert. Programmentwickler hatten nun die Möglichkeit ihren Code mit einem Zertifikat zu signieren und ihn somit als vertrauenswürdig einstufen zu lassen. Solch ein signierter Code konnte nun, so es der Benutzer wollte, mit dem

lokalen Code verglichen und im folgenden als lokaler Code behandelt werden. Somit hatte dieser natürlich alle damit verbundenen Zugriffsrechte auf Systemressourcen. In den Java-Versionen 1.0 und 1.1 gab es noch einige Sicherheitslücken, wie zum Beispiel die Folgende: Durch nicht korrekt implementierte Class Loader konnte das Sicherheitsmerkmal 'Typesafety' der JVM außer Kraft gesetzt werden. Typesafety bedeutet, dass ein Programm nicht auf unzulässige Speicherbereiche zugreifen kann. Genauer gesagt gehört jeder Bereich des Speichers zu irgendeinem Java Objekt und jedes Objekt hat wiederum eine Klasse. D.h., dass ein Objekt keine ungültigen Operationen, die ein anderes Objekt betreffen, ausführen kann, sofern sie nicht explizit erlaubt wurden. Daraus folgten jedoch keine unmittelbaren Sicherheitslücken, denn 'untrusted' Code wie z. B. ein Applet konnte zwar geladen werden, aber es hatte nicht die Erlaubnis, neue Class Loaders zu erstellen.

Java 1.2: In der nächsten Java-Version 1.2 wurde das Sicherheitssystem von Grund auf erneuert. Nun wurde nicht mehr zwischen lokalem und entferntem Code unterschieden und das 'Sandbox'-Modell wurde erneuert. Es war von nun an möglich, die Zugriffsrechte und Sicherheitsbestimmungen individueller zu gestalten. Dieses Konzept wurde als wesentlich sinnvoller erachtet und bildet die Grundlage zur Entwicklung sicherer, verteilter Java-Programme.

## 1.3 Grundfunktionen

Der Kern der Java Security Architecture besteht aus:

- Class Loader
- Byte Code Verifier
- Security Manager
- Access Controller
- Permissions
- Policies
- Protection Domains

Der Class Loader stellt die zentrale Stütze der Sicherheitsarchitektur dar. Er lädt den Byte-Code eines ankommenden Datenstroms und ruft den 'Byte Code Verifier' auf, um überprüfen zu lassen, ob dieser fehlerfrei ist. Danach startet der Class Loader den 'Security Manager' und 'Access Controller', um die Zugriffsrechte des Java-Programms auf Systemressourcen zu bestätigen. Wurde dieser Vorgang beendet, wird die geladene Klasse zum Code Prozessor weitergeleitet, wo sie ausgeführt wird. Auf verschiedene Elemente und Methoden der Java Security Architecture wird in Kapitel 5 noch ausführlich eingegangen.

Es gibt verschiedene Arten von Class Loader:

- Der Bootstrap Class Loader ist ein Teil der Java Virtual Machine und ist dafür verantwortlich, dass die 'Kernel Classes' der Java API korrekt geladen werden. Er hat noch einige weitere Aufgaben wie z.B. das Laden von Array-Klassen, worauf wir aber in Kapitel 2 genauer eingehen.
- Benutzerdefinierte Class Loader können explizit vom Programmentwickler durch das Class Loader Interface implementiert werden. I.d.R. handelt es sich bei diesen selbstdefinierten Class Loader um Subklassen der abstrakten Klasse JAVA.LANG.CLASSLOADER.

# 2. Erläuterung des Ladevorgangs

## 2.1 Allgemeines

Im folgenden wird sehr oft der Begriff Klasse verwendet. Damit ist in diesem Kontext „Klasse oder Interface“ gemeint. Schreiben wir ‚Class Loader’ so ist allgemein ein Klassenlader gemeint, benutzen wird jedoch die Schreibweise CLASSLOADER, so ist eine Instanz oder eine Instanz eines Nachkommen der Abstrakten Klasse CLASSLOADER gemeint.

Zunächst muss die zu ladende Klasse C mit dem Namen N gefunden werden. Dies geschieht meist durch eine Suche im lokalen Dateisystem, möglich wäre jedoch auch bspw. ein Netzwerk. Die Suche geht in folgender Reihenfolge von statten:

- Bootstrap Classes, Laufzeitklassen der Java Runtime Environment, z.B. rt.jar
- Installierte Erweiterungen, Klassen in .JAR-Dateien im Verzeichnis lib/ext der JRE
- Classpath, Klassen bzw. Klassen in .JAR-Dateien, die sich in Pfaden befinden, die durch die Systemvariable CLASSPATH festgelegt wurden

Nun wird eine sogenannte Method-Area der implementationsabhängigen, internen Repräsentation der zu ladenden Klasse erstellt. Bei der Method-Area handelt es sich um einen speziellen Speicherbereich, in dem bspw. Feld- und Methodendaten, sowie der Code von Konstruktoren und Methoden gespeichert wird. Die Method-Area wird beim Start der VM erstellt und ist Teil des ‚Heaps’. Gemeinsamer Speicher für alle VM-Threads. Threads sind nebenläufige Ausführungseinheiten innerhalb genau eines Prozesses, die sich entweder einen Prozessor teilen oder von mehreren parallel ausgeführt werden können. Das Speichern und Löschen von Objekten wird durch ein automatisches Speicherverwaltungssystem, den Garbage Collector, übernommen.

Die Erstellung der Klasse wird durch eine weitere Klasse ausgelöst, die durch ihren ‚Runtime Constant Pool’ (RCP) Referenzen auf diese Klasse besitzt. Ein Runtime Constant Pool ist die Repräsentation der CONSTANT_POOL-Tabelle in Form einer Klassendatei. Diese enthält einige Arten von Konstanten, z.B. zur Initialisierung verwendete Zahlen, Namen von externen Klassen und Methoden, die schon während des Compilens bekannt sind, oder Methoden- und Feldreferenzen, die erst zur Laufzeit berechnet werden können. Die Informationen des Runtime Constant Pools stammen aus der Method Area der JVM. Ein RCP für eine Klasse wird immer genau dann erstellt, wenn die Klasse durch die JVM erschaffen wird und sorgt dafür, dass der erzeugte Bytecode möglichst kurz gehalten wird.
Die Erstellung der Klasse kann auch dadurch ausgelöst werden, dass eine weitere Klasse Methoden in bestimmten Java Klassenbibliotheken, z. B. REFLECTION, ausführt. Das Java Standardpaket REFLECTION erlaubt es einem Programm, zur Laufzeit auf Namen und Eigenschaften des Programmes selber zuzugreifen. So kann man bspw. eine Methode aufrufen, deren Name zur Laufzeit als Text bekannt ist.

Außer Array-Klassen werden die meisten Klassen mittels der Class Loader erzeugt. Im Gegensatz dazu werden Array-Klassen durch die JVM selbst erschaffen, da sie keine externe Binärrepräsentation besitzen. Mit Array-Klasse ist folgendes gemeint:

MyClass[] array = new MyClass[10], also die Generierung eines Arrays eines nichtprimitiven Datentyps.

Es gibt zwei verschiedene Arten von Class Loader: Benutzerdefinierte Class Loader und den Bootstrap Class Loader, der Teil der JVM ist. Jeder benutzerdefinierte Class Loader ist eine Instanz einer Unterklasse der abstrakten Klasse CLASSLOADER. Eine kleinere Ausnahme bildet hier z.B. der Spezialfall RMICLASSLOADER, der in Kapitel 4 noch ausführlich erläutert wird. Class Loader werden in Anwendungen eingesetzt, um die Eigenschaften des dynamischen Ladens der JVM noch zu erweitern. In benutzerdefinierten Class Loader können z.B. benutzt werden, um Klassen zu erzeugen, die von vom Benutzer festgelegten Quellen geladen werden. (z.B. können Klassen über ein Netzwerk heruntergeladen, ‚on-the-fly'-generiert oder aus einer verschlüsselten Datei extrahiert werden.

Ein Class Loader L kann eine Klasse C erzeugen, indem er sie sofort selbst definiert oder die Anfrage an einen anderen Class Loader weitergibt.
Wenn L C erzeugt, verwendet man folgende Ausdrucksweise: L definiert C oder L ist der definierende Class Loader von C (Andere Schreibweise: <N, $L_d$>, wobei N den Namen der Klasse C kennzeichnet und $L_d$ der definierende Class Loader ist).
Wenn L C entweder durch sofortiges Definieren oder durch das Weiterleiten der Anfrage zu einem anderen Class Loader erzeugt, sagt man: L initialisiert C oder L ist der initialisierende Class Loader von C (Andere Schreibweise: $N^{L_i}$, wobei N den Namen der Klasse C kennzeichnet und $L_i$ der initialisierende Class Loader ist).

Während der Laufzeit wird eine Klasse nicht nur durch ihren Namen bestimmt, sondern durch ein Paar von Eigenschaften: der volle Name der Klasse und der definierende Class Loader. Jede Klasse gehört zu genau einem sog. ‚Runtime Package'. Dieses wird durch den Namen des Pakets und den definierenden Class Loader der Klasse bestimmt.

Die JVM benutzt jeweils eine von 3 Möglichkeiten, eine Klasse C, die durch ihren Namen N gekennzeichnet wird, zu erzeugen:
1) Betrifft alle außer Array-Klassen:
- Wenn die Klasse D, die die Erzeugung der Klasse C ausgelöst hatte, vom Bootstrap Class Loader definiert wurde, wird auch die Klasse C vom Bootstrap Class Loader initialisiert.
- Wenn die Klasse D, von einem benutzerdefinierten Class Loader definiert wurde, wird auch die Klasse C von einem benutzerdefinierten Class Loader initialisiert.

2.) Betrifft nur Array-Klassen:
- Array-Klassen werden ausschließlich durch die JVM erzeugt (nie durch einen Class Loader). Jedoch wird auch hier der definierende Class Loader der Klasse D während des Erzeugungsprozesses der Klasse C benötigt.

## 2.2 Klassen mit dem Bootstrap Class Loader laden

Im folgenden wird beschrieben, wie eine Klasse (keine Array-Klasse) C, die durch den Namen N gekennzeichnet wird, mittels des Bootstrap Class Loaders geladen wird.

Zunächst wird von der JVM überprüft, ob der Bootstrap Class Loader der initialisierende Class Loader der Klasse, die durch N beschrieben wird, ist. Ist dies der Fall, heißt das, dass die Klasse C schon erzeugt wurde und somit keine weitere Bearbeitung nötig ist. Ist der Bootstrap Class Loader nicht der initialisierende Class Loader, so wird eine der beiden folgenden Aktionen ausgeführt:

- Die JVM sucht nach einer Repräsentation von C. Anmerkung: Sollte die Suche erfolgreich sein, gibt es jedoch keinerlei Garantie, dass die gefundene Repräsentation gültig und fehlerfrei bzw. überhaupt die richtige Repräsentation von C ist. Typischerweise ist die gesuchte Klasse eine Datei in einem hierarchischen Dateisystem, d.h., dass der Dateiname im Pfad der Datei kodiert und somit unschwer erkennbar ist. Zu diesem Zeitpunkt muss, falls keine Repräsentation der Klasse C gefunden wird, eine Instanz von NOCLASSDEFFOUNDERROR oder einer ihrer Unterklassen geworfen werden. Wurde jedoch eine Repräsentation gefunden, versucht nun die JVM mittels des Bootstrap Class Loader eine Klasse, die durch N gekennzeichnet wird, von der Repräsentation abzuleiten. Dabei benutzt sie einen speziellen Algorithmus, der hier nicht weiter behandelt wird. Die abgeleitete Klasse ist C.

- Der Bootstrap Class Loader leitet den noch nicht bearbeiteten Ladevorgang von C zu einem benutzerdefinierten Class Loader L weiter, indem der Klassenname N als Argument an die Methode LOADCLASS von L übergeben wird. Daraus resultiert der Aufruf von C, während die JVM festhält, dass der Bootstrap Class Loader initialisierender Class Loader von C ist.

## 2.3 Klassen mit einem benutzerdefinierten Class Loader laden

Im folgenden wird beschrieben, wie eine Klasse (keine Array-Klasse) C mit dem Namen N mit Hilfe eines benutzerdefinierten Class Loader L erzeugt und somit auch geladen wird.

Zunächst wird von der JVM überprüft, ob der Class Loader L der initialisierende Class Loader der Klasse, die durch N beschrieben wird, ist. Ist dies der Fall, heißt das, dass die Klasse C schon erzeugt wurde und somit keine weitere Bearbeitung nötig ist. Wenn dem nicht so ist, wird die Methode LOADCLASS(N) des Class Loader L mit dem Argument N, dem Klassennamen, aufgerufen. Der Rückgabewert der Methode ist die erzeugte Klasse C. Die JVM merkt sich nun, dass der L der initialisierende Class Loader von C ist.

Wenn LOADCLASS(N) aufgerufen wird, muss L eine der beiden folgenden Aktionen ausführen, damit C erfolgreich geladen wird:

- L kann ein Byte-Array erstellen, das C als CLASSFILE-Struktur repräsentiert. Danach muss die Methode DEFINECLASS der Klasse CLASSLOADER aufgerufen werden. Durch DEFINECLASS versucht die JVM eine Klasse, deren Name N ist, mit dem Class Loader L von dem Byte-Array abzuleiten. Dabei wird wieder der schon kurz erwähnte Algorithmus angewandt.

- Der Class Loader L leitet den noch nicht behandelten Ladevorgang von C zu einem anderen Class Loader L' weiter, indem eine bestimmte Methode

(typischerweise LOADCLASS) von L' mit dem Argument N aufgerufen wird. Der Rückgabewert dieser Methode ist C.

Da (fast) jeder benutzerdefinierte Class Loader eine Instanz einer von JAVA.LANG.CLASSLOADER abgeleiteten Klasse ist, gehen wir nun weiter auf die Aufgaben und Methoden dieser wichtigen Klasse ein.

### 2.3.1 JAVA.LANG.CLASSLOADER
**[extends JAVA.LANG.OBJECT]**

Die abstrakte Klasse CLASSLOADER sucht bzw. erstellt Daten, die die Definition einer Klasse beschreiben. Typischerweise wandelt man den Namen der gesuchten Klasse in einen Dateinamen um und sucht im lokalen Dateisystem nach der .class-Datei. Das Ergebnis ist eine Instanz von CLASS.

In Kapitel 2.1 wurde bereits ausführlich beschrieben, dass jede Instanz einer Klasse CLASS eine Referenz auf den CLASSLOADER, der es definiert hat, enthält. Um sich den definierenden CLASSLOADER anzeigen zu lassen, benutzt das CLASS Objekt die Methode GETCLASSLOADER. Falls der Rückgabewert dieser Methode NULL ist, so ist bei den meisten Implementationen der Bootstrap Class Loader der definierende Class Loader des Objekts, das die Klasse repräsentiert. Ansonsten wird, wenn ein Security Manager eingesetzt wird, und wenn der Class Loader des Aufrufenden nicht der gleiche oder ein Vorfahre des angeforderten Class Loaders ist, die Methode CHECKPERMISSION des Security Managers, wobei die Erlaubnis RUNTIMEPERMISSION("GETCLASSLOADER") als Parameter übergeben wird. CHECKPERMISSION wirft eine SECURITYEXCEPTION, wenn die Instanz keine Erlaubnis hat, auf den Class Loader zuzugreifen.
Array-Klassen werden explizit durch die JVM erzeugt, jedoch benötigt man den definierenden Class Loader der Klasse des Elementtyps des Arrays. Hat man zum Beispiel eine Klasse MYCLASS und in einer anderen Klasse ein Feld MYCLASS[] MYARRAY, so wird MYARRAY[] zwar durch die JVM erzeugt, jedoch wird dabei der Class Loader von MYCLASS benötigt. Felder primitiven Datentyps haben keinen Class Loader.

Wie schon in Kapitel 2.1 angesprochen kann eine CLASSLOADER Instanz eine Anfrage nach einer Klasse einfach zu einem anderen CLASSLOADER weiterleiten. Jeder CLASSLOADER hat einen ‚Vater'. Die Ausnahme bildet hier der Bootstrap Class Loader, der keinen ‚Vater' hat, jedoch als ‚Vater' für andere CLASSLOADER fungieren kann. Wenn nun eine CLASSLOADER Instanz den Auftrag erhält eine Klasse mit dem Namen N zu laden, so wird sie diese Anfrage immer zuerst zu ihrem ‚Vater'-CLASSLOADER weiterleiten bevor sie selbst versucht die Klasse zu finden.

Eine zu ladende Klasse muss nicht immer im lokalen Dateisystem liegen. Es gibt bspw. die Möglichkeit sie über ein Netzwerk zu erhalten oder den Bytecode der Klasse dynamisch generieren zu lassen. Um nun solch eine Klasse zu laden braucht man einen benutzerdefinierten Class Loader, der den Bytecode folgendermaßen mit der Methode DEFINECLASS() lädt. An DEFINECLASS() wird ein Bytearray als Parameter übergeben und eine Instanz der Klasse CLASS zurückgegeben. Um nun neue Instanzen dieser neu definierten Klasse zu erzeugen, benutzt man die Methode NEWINSTANCE() von CLASS.

## 2.4 Erzeugung von Array-Klassen

Nun gehen wir kurz darauf ein, wie eine Array-Klasse C, die durch den Namen N gekennzeichnet ist, mit Hilfe des Class Loader L erzeugt wird. L kann nun entweder der Bootstrap Class Loader oder ein benutzerdefinierter Class Loader sein.

Wenn L schon als initialisierender Class Loader einer Array-Klasse des selben Komponententyps wie N eingetragen ist, so ist diese Klasse C, d.h. es ist keine Erschaffung einer neuen Array-Klasse nötig. Ansonsten werden folgende Schritte ausgeführt, um C zu erzeugen:

- Ist der Komponententyp von C ein Referenztyp, so wird L eingesetzt, um die Klasse zu laden.

- Die JVM erstellt die Array-Klasse mit dem vorgegebenen Komponententyp und der Anzahl der Dimensionen. Wenn der Komponententyp ein Referenztyp ist, so wird festgehalten, dass C durch den definierenden Class Loader des Komponententyps definiert wurde. Ansonsten wird vermerkt, dass C durch den Bootstrap Class Loader definiert wurde. In jedem Fall hält die JVM jedoch fest, dass L der initialisierende Class Loader von C ist. Wenn der Komponententyp ein Referenztyp ist, so wird der Zugriff auf die Array-Klasse durch die Zugriffseigenschaften des Komponententyps gesteuert. Ansonsten wird der Array-Klasse das Attribut PUBLIC zugeteilt.

## 3. Beschreibung des Aufbaus eines einfachen Class Loader

Ein benutzerdefinierter Class Loader muss, wie bereits erwähnt, immer einen bereits vorhandenen Class Loader erweitern und erbt die Methoden von JAVA.LANG.CLASSLOADER. Diese werden verwendet, um Klassen in die Virtual Machine zu laden. Der Ladeprozess besteht aus den Schritten des eigentlichen Ladens des Codes der Klasse, des Verifizierens, Vorbereitens, Auflösens und Initialisierens. Betrachten wir hier diese Schritte, wie sie in der Java API realisiert sind:

```
protected Class findClass(String name)
```

In dieser Klasse findet der eigentliche Ladeprozess statt. Die Methode nimmt den übergebenen Namen und sucht nach dieser Klasse. Wie und wo nach dieser Klasse gesucht wird, muss ebenfalls in dieser Methode definiert werden, deshalb muss sie überschrieben werden. Es gibt keine Vorschriften, von wo eine Klasse kommen muss, es ist möglich, sie aus einem Netzwerk zu laden, aus einer Datenbank, sie während der Laufzeit selbst zu erzeugen, etc. Wird die angegebene Datei nicht gefunden wird eine ClassNotFoundException zurückgegeben, die später weiterverarbeitet werden kann.

```
protected final Class defineClass (String name, byte[] b, …,
      ProtectionDomain pd)
```

Diese Klasse realisiert den letzten Schritt des Ladeprozesses. Sie bekommt einen Namen als String übergeben, der dem erwarteten Namen der Klasse entspricht. Er wird ohne die Endung .class angegeben, diese hängt der Class Loader automatisch an. Das Byte-Array ist der eigentliche Code der Klasse, dafür werden noch zwei Integer Werte übergeben, um Anfang und Ende des eigentlichen Codes in diesem Array zu kennzeichnen. Der Parameter ProtectionDomain ist optional (es existiert die gleiche Methode ohne diesen Parameter) und weist der Klasse eine Protection Domain zu (siehe unten). Wird er nicht übergeben wird die Standard- Protection Domain verwendet. Da diese Klasse final ist darf sie nicht überschrieben werden, das Umwandeln eines Byte-Arrays in eine Klasse ist also bei jedem Class Loader identisch.

```
protected final void resolveClass(Class c)
```

Hier wird das Linken vorgenommen, weshalb der Name der Klasse eigentlich schlecht gewählt ist. Linken bedeutet, die Klasse von ihrem ursprünglicher Bytefolge in die Laufzeitumgebung der Java Virtual Machine zu laden. Einziger Parameter ist die Klasse selber, die also vorher bereits durch `defineClass` erzeugt werden musste. Es wird nichts zurückgegeben, aber die Klasse befindet sich nach erfolgreichem Abarbeiten dieser Methode in der Java Virtual Machine. Auch diese Klasse ist nicht überschreibbar, also ist auch das Linken von Klassen bei jedem Class Loader identisch.

```
protected final Class findLoadedClass(String name)
```

Diese Methode gibt die Klasse mit dem Namen “name” zurück, wenn sie bereits durch diesen Class Loader geladen wurde. Dies sollte versucht werden, bevor die Klasse neu geladen wird, da, um Typenkonsistenz zu garantieren, immer die gleiche Referenz auf eine Klasse zurückgegeben werden muss. Als Konsequenz ergibt sich daraus, dass jeder Class Loader Referenzen auf alle jemals von ihm geladenen Klassen speichern muss, dies kann zu einem hohen Speicherbedarf führen. Diese

Klasse ist ebenfalls mit dem Schlüsselwort final versehen, kann also nicht überschrieben werden.

`protected Class loadClass(String name, boolean resolve)`
Dies ist die Klasse, die von der Java Virtual Machine aufgerufen wird, wenn eine Klasse mit dem Namen „name“ geladen werden soll, sie fasst alle bisher vorgestellten Methoden zusammen. Die Standard-Implementierung führt folgende Schritte aus:
1. `findLoadedClass(name):` Als Erstes wird also überprüft, ob die Klasse bereits vorher geladen wurde, wenn dies der Fall ist werden die Schritte 2 und 3 übersprungen.
2. Ruft die Methode `loadClass` beim Vorgänger des Class Loaders auf, delegiert das Laden der Klasse also nach oben, was als. Dies ist ein wichtiger Schritt um Typenkonsistenz zu garantieren. Ist kein Vorgänger vorhanden wird der eingebaute Class Loader der JVM benutzt.
3. `findClass(String name):` Wenn keiner der Vorgänger die Klasse laden können wird versucht, sie über findClass zu laden.

Wenn einer dieser Schritte erfolgreich war wurde also eine Referenz auf eine Klasse zurückgegeben. Wenn der resolve-Parameter wahr ist wird `resolveClass()` mit dieser Referenz aufgerufen, die Klasse wird also gelinked, anderenfalls wird die Referenz einfach so zurückgegeben. Diese Klasse ist nicht als final deklariert, es ist also möglich die vorgestellte Reihenfolge zu ändern, beispielsweise den zweiten Schritt wegzulassen oder erst nach dem dritten Schritt auszuführen. Dies ist jedoch keineswegs ratsam, da dies zu Inkonsistenzen der Typen führen kann. Folgend ein Beispiel, das solch eine Inkonsistenz verdeutlicht:

Angenommen, der Class Loader führt Schritt 3 ganz zu Anfang aus. Er bekommt jetzt von der Virtual Machine die Aufgabe, die Klasse „String“ zu laden. Also versucht er zuerst, sie in seinem Bereich zu lokalisieren (beispielsweise dem Verzeichnis /tmp). Wir gehen jetzt davon aus, dass sich in diesem Augenblick die Klasse nicht in diesem Verzeichnis befindet, der 1. Schritt ist also nicht erfolgreich und die Anfrage wird bis zum Bootstrap Class Loader nach oben delegiert, der java.lang.String zurückgibt. Während der Laufzeit wird jetzt von einem Benutzer die Datei String.class in das Verzeichnis /tmp geladen und die Virtual Machine möchte wieder eine Referenz auf die Klasse String auflösen lassen. Der Class Loader schaut als erstes wieder in „sein“ Verzeichnis /tmp, findet die Klasse jetzt dort und gibt sie an die Virtual Machine zurück. Die beiden String Klassen sind aber nicht identisch, doch von der VM nicht zu unterscheiden, da sie beide vom gleichen Class Loader geladen wurden, sich also im gleichen Namespace befinden (siehe unten). Jetzt kann es sein, dass auf der einen String Klasse Methoden ausgeführt werden sollen, die auf ihr gar nicht existieren, da eigentlich die andere String Klasse gemeint war. Dies würde zu einem schwerwiegenden Fehler und einem Terminieren des Programms führen.

Diese Beispiel soll also zeigen, dass die oben vorgestellte Reihenfolge des Ladens einer Klasse dringend eingehalten werden sollte, da sonst ein sicherer Ablauf des Programms nicht garantiert ist.

# 4. Beschreibung verschiedener Arten von Class Loader

Das nun folgende Kapitel befasst sich mit den wichtigsten Class Loader Klassen der Java API. Insbesondere wird erklärt welche Aufgaben und Einsatzgebiete die Klassen haben und welche Methoden sie einsetzen, um diese Aufgaben zu erfüllen.

## 4.1 JAVA.SECURITY.SECURECLASSLOADER
**[extends JAVA.LANG.CLASSLOADER]**

SECURECLASSLOADER erweitert die abstrakte Klasse CLASSLOADER und stellt einige weitere Funktionen zur Verfügung, um Klassen zu definieren, sie mit dem zugehörigen Quellcode in Verbindung zu bringen und deren Erlaubnisse korrekt zu verwalten. Falls ein SECURITYMANAGER vorhanden ist, wird sofort bei der Konstruktion des Class Loader die Methode CHECKCREATECLASSLOADER ausgeführt, um zu überprüfen, ob die Erzeugung dieses Class Loader überhaupt erlaubt ist. Ist die Konstruktion nicht erlaubt, so wird eine SECURITYEXCEPTION geworfen.

## 4.2 JAVA.NET.URLCLASSLOADER
**[extends JAVA.SECURITY.SECURECLASSLOADER]**

Dieser Class Loader wird benutzt, um Klassen, die bei einer bestimmten URL hinterlegt sind, zu laden. Dabei sucht er ausschließlich nach Verzeichnissen und .JAR-Dateien. Endet die URL mit einem /, so weiß der URLCLASSLOADER, dass es sich um ein Verzeichnis handelt, andernfalls stellt die URL aus seiner Sicht die Referenz zu einer .JAR-Datei dar. Der ACCESSCONTROLCONTEXT des Threads, der die Instanz von URLCLASSLOADER erzeugt hat, spielt hierbei eine wichtige Rolle bei der Zuweisung von Zugriffsrechten (mit der Methode CHECKPERMISSION(PERMISSION perm)) der Klassen auf Systemressourcen. Die geladenen Klassen haben standardmäßig nur Zugriff auf bestimmte URL's, die festgelegt wurden als der URLCLASSLOADER erzeugt wurde. Auch hier findet wieder die Überprüfung des SECURITYMANAGER (falls vorhanden) statt, ob der Class Loader erzeugt werden darf.

## 4.3 JAVA.RMI.SERVER.RMICLASSLOADER
**[extends JAVA.LANG.OBJECT]**

RMICLASSLOADER enthält einige statische Methoden, die das dynamische Laden von Klassen mit RMI unterstützen sollen. Z.B. Methoden, die die Lokalisierung von Klassen im Netzwerk sowie den Ladevorgang von entfernten Klassen unterstützen. Beispiele: LOADCLASS(), LOADPROXYCLASS(), GETCLASSLOADER(), GETCLASSANNOTATION()
Diese Methoden werden durch eine Instanz der Klasse RMICLASSLOADERSPI (Service Provider Interface) bereitgestellt. Wenn eine der Methoden aufgerufen wird, wird dieser Aufruf an die entsprechende Methode der Instanz des Service Provider Interfaces weitergeleitet.

# 5. Sicherheitsaspekte

## 5.1 Der Class Loader in der Java Sandbox

Bei einem Class Loader (CL) in Java muss ganz besonders auf Sicherheitsaspekte geachtet werden, da durch das Laden neuer Klassen gefährlicher oder fehlerhafter Code in die Virtual Machine gelangen kann. Der CL ist also genau die Instanz, die sicherstellen muss, dass nur als sicher eingestufte Klassen in die Laufzeitumgebung geladen werden.

Der Class Loader trägt auf drei Arten zur Java Sandbox („Sandkasten“) bei, die hier verdeutlicht werden sollen:

1. Er hält möglicherweise gefährlichen Code von der Beeinflussung korrekten und sicheren Codes ab. Dies wird über Namespaces gehandhabt, siehe hierzu Kapitel 5.1.1
2. Er verhindert das Verändern von sogenannten „trusted class libraries“, siehe Kapitel 5.1.2
3. Er teilt Code in Kategorien ein („Protection domains“), die dem Programm seine Ausführungsrechte und Benutzungsrechte zuweisen, siehe Kapitel 5.1.3

### 5.1.1 Class Loader und Namespaces

Ein Namespace ist eine Menge unabhängiger und eindeutiger Namen, ein Name pro geladener Klasse. Jeder Class Loader besitzt solch einen eigenen Namespace. Falls z.B. eine Klasse *Auto* in einen Namespace geladen wird, ist es danach unmöglich eine weitere Klasse mit dem Namen *Auto* in diesen Namespace zu laden. Es ist jedoch möglich, generell mehrere *Auto*-Klassen in eine Virtual Machine zu laden, solange sich diese in unterschiedlichen Namespaces befinden. Die sicherheitsrelevanten Aspekte ergeben sich daraus, dass Namespaces eine Art Schutzschild zwischen Klassen unterschiedlicher Namespaces legen. Klassen in identischen Namespaces können über die gewohnten Möglichkeiten miteinander kommunizieren, z.B. über mit dem Schlüsselwort *public* definierte Methoden. Sobald sich zwei Klassen aber in unterschiedlichen Namespaces befinden, also von unterschiedlichen Class Loadern geladen wurden, ist es für diese nicht einmal möglich, die Existenz der jeweils anderen festzustellen, solange der Programmierer dies nicht explizit ermöglicht.

### 5.1.2 Trusted class libraries

Trusted class libraries sind die Pakete, die von der Java Virtual Machine als definitiv sicher angesehen werden, dazu gehören unter anderem die Klassen der Java API. Diese Pakete werden von einem anderen Class Loader geladen als untrusted packages. Seit Java Version 1.2 hat jeder CL, außer dem „Bootstrap Class Loader“, einen Vorgänger-ClassLoader, den er mit *extends* angeben muss. Dieser „Bootstrap CL“ steht ganz oben in der Kette und ist nur für das Laden der Klassen der Java API zuständig. Daher kommt auch sein Name, bootstrap bedeutet soviel wie „Ladeprogramm“, der Kern der Java API wird nämlich benötigt, um die Virtual Machine zu initialisieren. Alle Klassen außer dieser API werden von weiteren, z.B. auch benutzerdefinierten, Class Loadern geladen, die alle in einer Kette mit dem Bootstrap CL verbunden sind. Anfangs steht am unteren Ende dieser Kette der

„System Class Loader“, der, wenn kein eigener CL definiert wurde, die Aufgabe übernimmt, die erste Klasse einer Applikation zu laden. Außerdem ist er üblicherweise der Vorgänger der selbstdefinierten Class Loader. Auch diese Aspekte dienen dem Sicherheitskonzept von Java, da mit ihnen verhindert wird, dass sich eine beliebige, möglicherweise gefährliche, Klasse als trusted class ausgibt. Wäre dies möglich, könnte diese Klasse die Sandbox-Barriere durchbrechen, da sie fälschlicherweise als sicher angesehen würde. Durch das Prinzip des Hochdelegierens, das bereits in Kapitel 3 vorgestellt wurde, ist es nicht möglich, trusted classes durch eigene zu ersetzen. Wenn ein Custom CL beispielsweise versuchen würde, eine eigenen *java.lang.String* Klasse zu laden, würde diese Anfrage als erstes bis zum Bootstrap CL nach oben geleitet. Dieser würde feststellen, dass das Paket *java.lang* zur Java API gehört und die Referenz auf diese Klasse zurückgeben. Auch ein anderes Fallbeispiel führt nicht zu dem gewollten, böswilligen, Erfolg führen. Nehmen wir an, ein Programm möchte die Datei *java.lang.Virus* laden, die die Virtual Machine angreifen soll. Analog zum ersten Beispiel würde auch diese Anfrage bis ganz nach oben delegiert werden, der Bootstrap CL würde feststellen, dass er zwar das Paket java.lang kennt, aber die Klasse nicht enthalten ist und würde zurückgeben, dass er die Klasse nicht laden kann. Da auch alle anderen übergeordneten Class Loader die Datei nicht in ihrem Bereich finden können, würde sie also, wie vom Angreifer gewünscht, von dem eigenen CL geladen. Da diese im Paket java.lang liegt könnte man jetzt intuitiv davon ausgehen, dass diese die gleichen Rechte hat, wie jede Klasse in diesem Paket, beispielsweise auf mit dem Schlüsselwort *protected* geschützte Methoden und Attribute zuzugreifen. Dies ist aber nicht der Fall, da die java.lang API Pakete von einem anderen CL geladen wurden, als die java.lang.Virus Klasse. Hier kommt der Begriff des „runtime packages“ ins Spiel, der bedeutet, dass sich zwei (oder mehrere) Klassen nur dann im gleichen „runtime package“ befinden, wenn sie den gleichen Paketnamen haben (was hier der Fall ist) und sie vom gleichen CL geladen wurden (was hier nicht der Fall ist). Da sich, um auf Paket-sichtbare Variablen und Methoden zugreifen zu können, die beiden Klassen im gleichen „runtime package“ befinden müssen ist es der hier beispielhaft beschriebenen java.lang.Virus Klasse also nicht möglich, die java.lang Klassen der API zu beeinflussen.

### 5.1.3 Protection domains

Das Prinzip der Protection domains beruht darauf, dass sich jede Klasse in Java in solch einer Protection domain befindet. Diese regelt, inwiefern diese Klasse auf Systemressourcen, wie Datei I/O und das Netzwerk zugreifen darf. Wenn Code geladen wird bekommt dieser von seinem Class Loader eine Protection domain zugewiesen, üblicherweise ist dies die Java Sandbox, wenn man beispielhaft einen AppletClass Loader betrachtet hat dieser jedoch weitere Einschränkungen. Da sich dieses Thema jedoch zu weit von dem Konzept der Java Class Loader entfernen würde, soll dieses hier nicht weiter vertieft werden, es sei nur gesagt, dass es möglich ist, Klassen unterschiedlichen Protection Domains zuzuweisen, die damit unterschiedliche Rechte haben.

## 5.2 Weitere Sicherungsmöglichkeiten

Die bisher vorgestellten Sicherheitsaspekte trugen automatisch dazu bei, geladene Klassen sicher zu behandeln, es gibt aber natürlich auch weitere Möglichkeiten, die Sicherheit der Virtual Machine auszubauen, diese müssen aber von dem

Programmierer, der einen eigenen Class Loader schreibt, explizit angegeben werden. Es kann beispielsweise nötig sein, sicherheitsrelevante Klassen nicht von vertrauensunwürdigen Klassen laden zu lassen. Dies kann man dadurch erreichen, dass in dem ClassLoader für die vertrauensunwürdigen Klassen (die z.B. aus dem Internet geladen werden) ein Test stattfindet, dass die angeforderte Klasse nicht zu einem „gesperrten“ Paket gehört, dies kann einfach durch einen Namensvergleich geschehen. Dies garantiert, dass keine „unsichere“ Klasse solch eine gesperrte Klasse laden kann, auch nicht während des dynamischen Linkings (siehe unten).
Bis jetzt haben wir nur betrachtet, wie bereits geladene Klassen in das Sicherheitssystem der Java Virtual Machine eingebunden werden. Bevor Klassendateien aber zu Klassen in Java werden müssen diese einige Überprüfungen über sich ergehen lassen, dies wird vom sogenannten „Class File Verifier“ erledigt, der aus vier Phasen besteht.

## 5.3 Class File Verifier

Der Class File Verifier (auch kurz Verifier genannt) stellt sicher, dass Klassendateien, die in die Java Virtual Machine geladen werden sollen, eine korrekte Struktur haben und korrekt mit anderen Klassen interagieren. Ist dies nicht der Fall wird eine Exception geworfen, die anzeigt, dass die Klasse nicht geladen wurde. Zwar enthalten die von gültigen Java Compilern erzeugten Klasse keine dieser Fehler, aber die Virtual Machine kann bei einer zu ladenden Klasse nicht feststellen, ob diese Klasse von einem korrekten Compiler erzeugt wurde oder nachträglich manipuliert wurde. Um also für Sicherheit zu garantieren muss jede Klassendatei, außer der Java API, die ja wie bereits beschrieben also trusted package angesehen wird, erst überprüft werden, bevor sie geladen wird. Eine Klasse liegt ursprünglich als Sequenz von Befehlen für die Virtual Machine vor, üblicherweise als Klassendatei (.class), es ist aber auch möglich, diese Befehlssequenzen dynamisch zu generieren. Sobald diese Sequenz von Befehlen erfolgreich geladen wurde, wird sie zu einer Klasse in der Java Laufzeitumgebung, die ausgeführt werden kann und mit anderen Klassen interagieren und kommunizieren kann.

Die Robustheit der Programme ist eines der Sicherheitsziele von Java, welches der Class File Verifier zu erreichen hilft. Wenn eine Methode einen Sprung-Befehl hinter das Ende der Methode, oder sogar der Klasse, beinhaltet könnte dies die ganze Java Virtual Machine abstürzen lassen, da dann eventuell auf ungültige Speicherbereiche zugegriffen würde, oder zugriffsgeschützte Bereiche anderer Klassen zugänglich machen. Aus diesem Grund stellt der Verifier unter anderem sicher, dass solche Sprungbefehle nicht in der zu ladenden Klasse vorkommen. Der Verifier erledigt den größten Teil seiner Arbeit bevor der Bytecode ausgeführt wird. So wird beispielsweise nicht bei jedem Sprungbefehl während der Programmausführung die Korrektheit des Zieladresse geprüft, sondern ein Mal beim Laden der Klasse der Bytecode analysiert und verifiziert. Dies wird in vier Phasen gemacht:

### 5.3.1 Phase 1: Strukturelle Überprüfung der Klassendatei

Eine Sequenz von Bytes, die zu einem Typ (einer Klasse) werden soll muss der Grundstruktur einer Java Klassendatei entsprechen. Diese interne Struktur wird vor dem eigentlichen Laden der Klasse überprüft um sicherzustellen, dass die Datei ohne Risiken geparsed werden kann. In dieser Phase finden viele Überprüfungen der Bytesequenz statt, beispielsweise muss jede Klassendatei mit den selben 4 Bytes

beginnen, nämlich mit der sogenannten „magic number“ 0xCAFEBABE, als Anspielung auf die Herkunft des Begriffes Java, der in den USA die umgangssprachliche Bezeichnung für Kaffee ist. Dies ist eine einfache Möglichkeit, beschädigte Dateien und solche, die nie als Java Klassendatei gedacht waren, zurückzuweisen. Außerdem speichert jede Klassendatei eine minimal und eine maximal verwendete Java Versionsnummer. Bei diesen muss überprüft werden, ob die lokale Virtual Machine diese unterstützt, ansonsten muss auch in diesem Fall die Klassen zurückgewiesen werden. Außerdem wird sichergestellt, dass an die Klassendatei keine Daten angehängt oder an einer anderen Stelle eingefügt wurden. Da die verwendeten Komponententypen und –längen die Gesamtgröße der Klassendatei bestimmen, kann über ein einfaches Abzählen geprüft werden, ob die Klasse die richtige Größe hat. Mit diesen und einigen weiteren Tests dieser Phase wird also sichergestellt, dass die eingelesen Bytes syntaktisch zu dem Java Klassenformat passen und in implementationsspezifische Datenstrukturen geparsed werden können. Dieser Schritt trägt auch zur Robustheit der Virtual Machine bei, indem Dateien, die die VM angreifen könnten, gar nicht erst an diese weitergegeben werden. Die nun folgenden Phasen betrachten nicht mehr die rohen Binärdaten, sondern die Datenstrukturen in den Methoden der Klasse.

### 5.3.2 Phase 2: Semantische Überprüfungen

Diese Phase muss nicht den Bytecode der Klasse betrachten und auch keine anderen Typen laden. Es wird die „Wohlgeformtheit“ verschiedener Komponenten überprüft. In der Klassendatei werden beispielsweise die Methodenbeschreibungen mit ihrer Signatur, also ihrem Return Typ, der Anzahl und der Typen der Parameter, als String gespeichert und dieser muss zu einer von dem Verifier vorgegebenen kontextfreien Grammatik passen. Dieser Test wird für alle Methoden der Klasse ausgeführt, beispielsweise würden Anweisungen im Methodenkopf zu einer Verletzung der Wohlgeformtheit führen und damit solche Methoden zurückgewiesen werden. Weitere in Java spezifizierte semantische Bedingungen werden in dieser Phase ebenfalls überprüft, so muss jede Klasse außer java.lang.Object eine Superklasse haben, finale Klassen dürfen nicht abgeleitet werden, Konstanten nicht verändert werden etc. Auch diese Regeln hätten bereits während des Kompilierens eingehalten werden müssen, aber auch hier kann der Verifier nicht wissen, ob die Klasse von einem gültigen und fehlerfreien Compiler erstellt wurde. Die nächste Phase ist die Umfangreichste und widmet sich dem Bytecode.

### 5.3.3 Phase 3: Bytecode Verifikation

In dieser Phase wird eine Analyse der Bytecode-Streams gemacht, aus denen eine Klasse besteht und die den Methoden der Klasse entsprechen. Dies setzt auf dem Bytecode- und Framework-Prinzip der Java Virtual Machine auf, welches hier kurz skizziert wird:

Jeder Bytecode-Stream besteht aus einer Folge sogenannter Opcodes, die alle ein Byte lang sind und die Maschinensprache der Virtual Machine bilden. Daraus lässt sich folgern, dass es maximal 255 Opcodes geben kann, momentan sind dies 204, von denen 3 für die Virtual Machine reserviert sind und nicht in Klassen vorkommen dürfen. Alle Opcodes und deren Bedeutung lassen sich in [1] nachschlagen. Diese Opcodes können für gewöhnlich in einem Zyklus der VM abgearbeitet werden, nur einige Aufwendigere, wie solche, die mit Fließkommatypen arbeiten, benötigen bis zu

drei Zyklen. Einige Opcodes benötigen noch Parameter mit denen sie arbeiten. Hier sei beispielsweise der Opcode für den Test eines Integer Wertes gegen Null beschrieben. Dieser lautet ifeq und hat den Code 0x99 (dezimal 153). Er testet, ob der oberste Stackwert der Methode den Wert 0 hat. Ist dies der Fall wird um s Bytes im Bytecode gesprungen, wobei s ein short-Wert ist, der aus den beiden dem Opcode als Parameter übergegebenen Bytewerten besteht. Dieses Sprungziel muss sich innerhalb der gleichen Methode befinden, sonst wird eine Exception geworfen. Dieses Ausführen aller Opcodes einer Methode, einer nach dem anderen, entspricht einem Thread der Virtual Machine. Jeder dieser Threads, also jede Bytecode-Folge, also jede Methode, hat seinen eigenen Java Stack, der wiederum aus vielen Frames besteht. Jedem Methodenaufruf wird solch ein Frame zugewiesen, in dem lokale Variablen, lokale Berechnungsergebnisse und einige weitere Daten gespeichert werden. Die lokalen Berechnungsergebnisse werden in einem besonderen Teil des Frames gespeichert, dem sogenannten Operanden-Stack. Von diesem Operanden-Stack nimmt beispielsweise der eben vorgestellte ifeq-Opcode den obersten Wert (der in diesem Fall ein Integer sein muss) und testet ihn gegen den Wert 0. Generell kann ein Opcode auf den Operanden-Stack und die lokalen Variablen einer Methode zugreifen.

Der Bytecode-Verifier arbeitet mit diesen Elementen der Virtual Machine. Er überprüft jeden verwendeten Opcode darauf, ob er gültige Parameterwerte übergeben bekommt und für diesen Opcode gültige Werte auf dem Operanden-Stack und in den lokalen Variablen liegen. Weiterhin stellt er sicher, dass auf lokale Variablen erst dann zugegriffen wird, wenn diese initialisiert wurden und ihnen nur gültige Werte zugewiesen werden. Mit diesen und weiteren Tests garantiert der Bytecode-Verifier, dass der untersuchte Stream aus Bytecodes sicher von der Virtual Machine verarbeitet werden kann.

Jedoch kann der Bytecode-Verifier nicht sicherstellen, ob ein Programm für alle Eingabewerte zu einem fest definierten Ergebnis kommt, da dies, wie im Halteproblem, einem grundlegenden Theorem der Informatik, bewiesen wurde, nicht möglich ist. Die Eigenschaft, ob ein Programm terminieren wird oder nicht ist unentscheidbar. Der Bytecode-Verifier kann also nicht feststellen, ob eine beliebige Folge von Bytecode sicher von der Virtual Machine ausgeführt werden kann. Dieses Problem wird so umgangen, dass gar nicht versucht wird, jedes beliebige Programm komplett zu durchlaufen. Der Verifier überprüft nur, dass der Bytecode einige fest definierte Regeln einhält. Ist dies der Fall wird der Bytestrom als sicher eingestuft, ansonsten wird er zurückgewiesen. Auf diese Weise umgeht der Bytecode-Verifier das Halteproblem, indem er nicht alle, sondern nur eine Teilmenge aller sicheren Bytecode-Folgen akzeptiert. Die eben erwähnten Regeln, anhand derer die Einstufung als sicher oder unsicher erfolgt sind so aufgebaut, dass jedes gültige Java Programm zu Bytecode kompiliert werden kann, der diese Regeln einhält. Auch andere Programme, die nicht als Java Quelltext geschrieben werden können durchlaufen diesen Verifier und werden an anderer Stelle abgelehnt (beispielsweise in Phase 2). Es gibt aber auch Programme, die zwar für die Virtual Machine sicher wären und terminieren würden, die diese Regeln aber nicht einhalten. Da diese Programme aber nicht als Java Quelltext ausdrückbar sind ist dies hier nicht von weiterer Bedeutung. Der Bytecode-Verifier akzeptiert also alle sicheren Java-Programme, er erfüllt also seine Aufgabe.

Die bis jetzt beschriebenen ersten drei Phasen des Class File Verifiers stellen also sicher, dass die zu ladende Klassendatei korrekt aufgebaut ist, sie die syntaktischen und semantischen Regeln von Java einhält und sie für die Virtual Machine sicheren Bytecode enthält. Falls eine dieser Bedingungen nicht erfüllt wird, wird eine Exception geworfen und die Klassendatei wird nicht ausgeführt.

### 5.3.4 Phase 4: Verifikation symbolischer Referenzen

Phase 4 findet statt, wenn die symbolischen Referenzen einer Klassendatei während des dynamischen Linkings aufgelöst werden. Es werden alle Verweise auf andere Klassen sowie auf Methoden und Attribute anderer Klassen verfolgt und überprüft. Da das Prinzip das „lazy loadings“ vorschreibt, dass solche Referenzen erst dann aufgelöst werden, wenn zum ersten Mal auf sie zugegriffen wird, findet ein Großteil dieser Verifikation erst während der Laufzeit der Java Applikation statt. Da sich diese symbolischen Referenzen üblicherweise auf andere Klassendateien beziehen (ausgenommen dem Fall, dass es sich um eine Referenz auf eine innere Klasse derselben Datei handelt) kann es notwendig sein, auch diese neu zu laden, wenn sie sich noch nicht in der Laufzeitumgebung der Virtual Machine befinden. In diesem Fall finden alle bereits beschriebenen Verifikationen auch für diese neu zu ladende Klasse statt. Die symbolischen Referenzen sind Strings, die aus dem Namen und eventuell anderen Informationen der referenzierten Typs bestehen, bei referenzierten Methoden beispielsweise noch aus dem Methodennamen und der Methodensignatur. Es wird überprüft, ob ein Typ mit diesem Namen besteht, er korrekt aufgerufen wird, die korrekte Methodensignatur benutzt wird, Zugriffsrechte auf diesen Typ bestehen, etc. Werden diese Vorschriften eingehalten wird die symbolische Referenz auf eine direkte Referenz auf den Typen aufgelöst, üblicherweise ein Pointer auf die referenzierte Klasse, Variable oder Methode. Dies wird, wie erwähnt, erst dann gemacht, wenn ein Opcode zum ersten Mal eine symbolische Referenz auf diesen Typ enthält. Die aufgelöste direkte Referenz (der Pointer) wird von der Java Virtual Machine gespeichert und bei folgendem Auftreten der gleichen symbolischen Referenz direkt wiederverwendet, damit nicht zwei Referenzen auf den gleichen Typ existieren, dies könnte zu Mehrdeutigkeiten und Problemen führen. Phase 4 stellt also sicher, dass die symbolischen Referenzen einer Klasse gültig sind und von der Klasse verwendet werden dürfen.

Nachdem diese vier Phasen abgearbeitet wurden, gilt eine Klasse als sicher und kann vom Class Loader als Bytecode an die Virtual Machine übergeben werden.

## 6. Schlusswort

Diese Seminarausarbeitung sollte die Wirkungsweise und die Prinzipien eines Class Loaders in Java verdeutlichen. Es wurde die Funktion eines Class Loaders erklärt, seine Einbettung in die Java Virtual Machine und sein Ablauf. Weiterhin wurde aufgezeigt, wie ein eigener Class Loader erstellt werden kann und welche unterschiedlichen Class Loader bereits vorhanden sind. Besonderer Wert wurde auch auf die Sicherheitsaspekte gelegt, da Class Loader dadurch, dass sie ausführbaren Code in die Virtual Machine einbringen, einen wesentlichen Angriffspunkt für bösartigen Code bieten. Es wurde aufgeführt, dass durch die verschiedenen Sicherheitsmechanismen eines Class Loaders diese Gefahr gering gehalten wird und Class Loader also eine bewährte Möglichkeit sind, Code von verschiedenen Quellen zu laden.

# 7. Quellenangabe


**[1] Java API:**
http://java.sun.com/j2se/1.4.1/docs/api/

**[2] JVM Spezifikation:**
http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

**[3] Uni Göttingen, JVM:**
http://www.num.math.uni-goettingen.de/Lehre/Lehrmaterial/Vorlesungen/Informatik/1998/skript/texte/VM.html

**[4] High Technologies Center, JCL History & Security Architecture:**
http://www.htc-cs.com/go.aspx?pageid=20

**[5] JavaWorld, Basics of JCL:**
http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html

**[6] Weiterer Bericht über Java Security von Sun:**
http://developer.java.sun.com/developer/onlineTraining/Security/Fundamentals/Security.html#secClassLoader

**[7] Artima.com, Security and the Class Loader Architecture:**
http://www.artima.com/underthehood/classloaders.html

**[8] Artima.com, Inside the Java Virtual Machine, Chapter 3: Security, Bill Venners:**
http://www.artima.com/insidejvm/ed2/security.html

**[9] Java Class Loading and Class Loaders – Lecture Nodes:**
http://www.cs.rit.edu/~ark/lectures/cl/index.html

**[10] JVM Security Overview:**
http://www.cs.ttu.edu/~cs5331/ns/modules/jvmsOverview/jvmOverview.html

**[11] Artima.com, Control Flow in the Java Virtual Machine:**
http://www.artima.com/underthehood/flow.html

**[12] Securing Java: The Class Loader Architecture:**
http://www.securingjava.com/chapter-two/chapter-two-7.html