**Seminar aus Softwareentwicklung:** *Inside Java and .NET*

# Java Security

**Seminararbeit von:**

**Johanna Fitzinger**

***Abstract:***

Da die meisten Rechner mit einem Intranet oder dem Internet verbunden sind, wurde das Thema Rechnersicherheit in den letzten Jahrzehnten zunehmend wichtiger.
SUN hat dieses Problem schon frühzeitig erkannt, und die Java Architektur mit einem umfangreichen, eingebauten Security Modell ausgestattet, dass sich mit jeder Version der Java Plattform weiterentwickelt hat. In diesem Bericht möchte ich die wesentlichen Bestandteile des Java Security Modells vorstellen, und ihre Funktionsweise erläutern.

# Inhalt

**1. Einleitung ........ 3**

**2. Sprachmodell ........ 3**

**3. Das Sandbox Sicherheitsmodell ........ 4**

3.1. Konzept des Sandbox Sicherheitsmodells ........ 4
3.2. Entwicklung des Sandbox Modells ........ 4
3.3. Komponenten der Sandbox ........ 6

**4. Die Class Loader Architektur ........ 6**

**5. Der Class File Verifier ........ 8**

5.1. Der Bytecode Verifier ........ 10

**6. Der Security Manager ........ 12**

**7. Signing ........ 14**

7.1. Grundlagen ........ 14
7.2. Erstellung eines signierten Applets ........ 15

**8. Rechteverwaltung: ........ 16**

8.1 Policy File ........ 17
8.2 Protection Domains ........ 18
8.3 AccessController ........ 18

**9. Referenzen: ........ 20**

## *1. Einleitung*

Die Programmiersprache Java wurde als plattformunabhängige Sprache sowohl für stand-alone Anwendungen als auch für internetbasierte Anwendungen (Java-Applets) entwickelt. Diese Java-Applets sind in html-Seiten eingebettet und werden beim Aufrufen dieser Seiten im Browser automatisch gestartet. Da der Autor eines solchen Applets nicht von vornherein als bekannt und vertrauenswürdig angesehen werden kann, und der Datentransport über das Internet generell relativ unsicher ist, müssen die Zugriffsmöglichkeiten von Applets auf das Nutzer-System entsprechend eingeschränkt werden. Es muss gewährleistet werden, dass das Java-Applet auf dem Rechner weder Schaden anrichten noch sicherheitsrelevante Daten ausspionieren kann. Aus diesem Grund haben die Entwickler von Java ein spezielles Sicherheitsmodell entworfen. Dieses Sicherheitsmodell ist Thema meiner Ausarbeitung. Dabei möchte ich zunächst erläutern wie das Sprachmodell von Java Sicherheit unterstützt. Dann erkläre ich das Java Sicherheitsmodell, das so genannte Sandbox Security Model und seine Entwicklung.

## *2. Sprachmodell*

Schon bei der Entwicklung von Java wurde darauf geachtet, aus den Fehlern von anderen Programmiersprachen, wie z.B. C oder C++, zu lernen. Der Verzicht von Java auf C/C++ typische und sehr maschinennahe Sprachelemente wie Pointer, Zeigerarithmetik, ungeprüfte Typumwandlung , hat ein erhebliches Maß an Sicherheit gebracht hat. Zum einen sind Java-Programme erheblich stabiler als vergleichbare C/C++ Programme, zum anderen ist die Sprache vieler sprachspezifischer Missbrauchsfähigkeiten beraubt.

Im Detail sind folgende Aspekte des Java-Designs auf maximale Sicherheit ausgelegt:

- **Strenge Typisierung** und sichere Typumwandlung: Java ist streng typisiert. Die Typumwandlung wird in Java sowohl statisch als auch dynamisch geprüft, um sicherzustellen, dass der deklarierte Kompilierzeit-Typ eines Objekts mit einem eventuellen Laufzeit-Typen kompatibel ist, selbst wenn das Objekt in einen anderen Typ umgewandelt wird.
- **Keine Pointer:** Da Java keine Pointer verwendet, sondern Objektreferenzen, werden sämtliche Umwandlungen und Zugriffe vorab (d.h. vor der eigentlichen Laufzeit) bereits auf Zulässigkeit geprüft.
- **Sichtbarkeits-/Zugriffsbereiche:** private, protected, package, public
- **Gültigkeit von Variablen:** Variablen müssen initialisiert sein bevor sie verwendet werden können.
- **Final Entities:** In Java können Klassen, Variablen und Methoden als final deklariert werden, womit eine weitere Ableitung der Klasse, das Verändern von Variablenwerten und das Überschreiben von Methoden nicht mehr möglich ist.
- **Strenge Arraygrenzen:** Verhindern (versehentlichen) Zugriff auf benachbarte Entitäten.
- **Exceptions:** Ermöglichen definierte und kontrollierte Programmabbrüche

- **Garbage Collection**: Die automatische Speicherfreigabe schützt vor unbeabsichtigten Programmierfehlern. So könnte beispielsweise die Speicherfreigabe vergessen werden, oder es wird versehentlich mehrmals der gleiche Speicher freigegeben.
- **Virtuelle Maschine:** Verbirgt Besonderheiten und Gefahren der ausführenden Architektur und überwacht Programmausführung und Rechte.

# *3. Das Sandbox Sicherheitsmodell*

## 3.1. Konzept des Sandbox Sicherheitsmodells

Die Idee des Sandbox Sicherheitsmodells besteht darin, nicht vertrauenswürdige Programme in einer geschützten Umgebung, der sogenannten Sandbox, ablaufen zu lassen. Zugriff auf wichtige Systemressourcen wird dem Programm nicht gestattet. Die Sandbox verhindert also, dass das Programm Aktionen ausführen kann, die dem Nutzer-System Schaden zufügen könnten. Das Programm kann innerhalb der Grenzen der Sandbox  tun was immer er möchte, hat aber keine Erlaubnis Aktionen außerhalb der durch die Sandbox definierten Grenzen auszuführen.

## 3.2. Entwicklung des Sandbox Modells

Das Sandkasten Sicherheitsmodell des JDK 1.0 untersagte nicht vertrauenswürdigem Applet-Code unter anderem folgende Aktionen:

- Lesen oder Schreiben von der lokalen Platte
- Verbindung zu einem Host aufzubauen mit Ausnahme des Hosts von dem das Applet geladen wurde
- Einen neuen Prozess zu starten
- Eine neue DLL zu laden.
- Systemeigenschaften einsehen

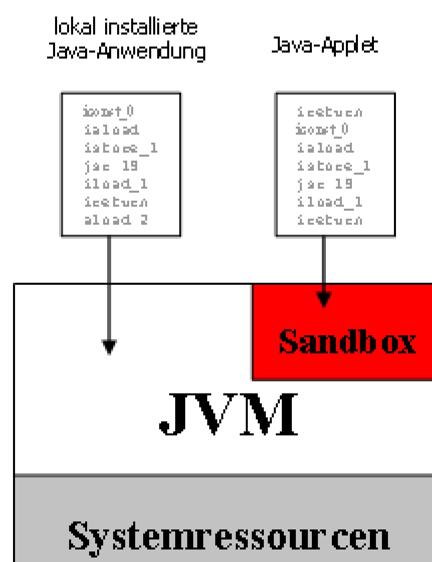



Abbildung 3.2.1.: Sicherheitsmodell unter JDK 1.0

Das Applet konnte also in diesem kreierten Sandkasten arbeiten, ohne ein Sicherheitsrisiko für die JVM oder das darunterliegende System darzustellen.
Man sah jedoch schnell ein, dass das Sandbox-Model des JDK 1.0 zu restriktive war, da durch die Einschränkungen der Sandbox viele Anwendungen nicht als Applet realisiert werden konnten.
Im JDK 1.1 wurde daher das sogenannte signierte Applet (*signed applet*) eingeführt. Ein signiertes Applet ist ein Applet verpackt in einem Java Archiv (JAR) und mit einem privaten Schlüssel (*private key*) signiert. Dieses Konzept ermöglichte eine Unterscheidung zwischen Applets, die von vertrauenswürdigen Quellen kamen und solchen die von nicht vertrauenswürdigen Quellen stammten. Signierte Applets die vom Benutzer als vertrauenswürdig erachtet wurden erhielten die gleichen Berechtigungen wie lokal geladene Applikationen Dieses Konzept war aber immer noch nicht zufriedenstellend, denn es erlaubte nur einem Applet den vollen Zugriff zu gewähren, oder es in dem vorgegebenen Sandkasten laufen zu lassen.

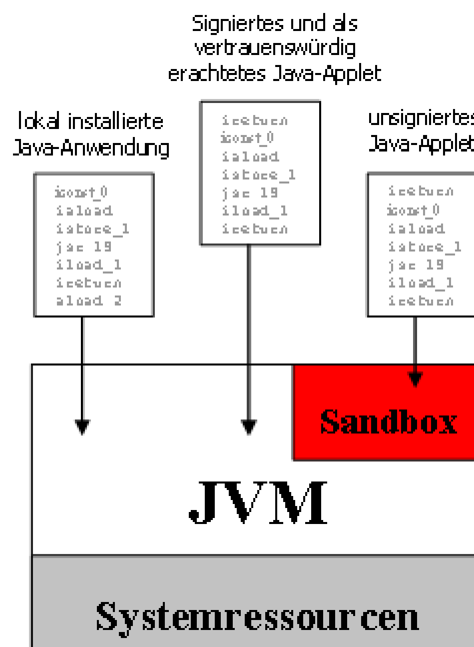



Abbildung 3.2.2.: Umgang mit signierten Applets unter Java 1.1

JDK 1.2 erlaubte erstmals nicht mehr nur zwischen vertrauenswürdigen und nicht vertrauenswürdigen Applets zu unterscheiden, sondern erlaubte, Code in verschiedene Sicherheitsstufen einzuteilen (fine-grained security), indem man Code definierte Rechte (Permissions) zuweist (siehe 8. Rechteverwaltung). Dem Code wird eine Protection Domain zugeordnet, die alle Rechte, die dem Code gewährt werden verwaltet. Einschränkungen können sowohl für Applets, wie auch für Applikationen definiert werden.

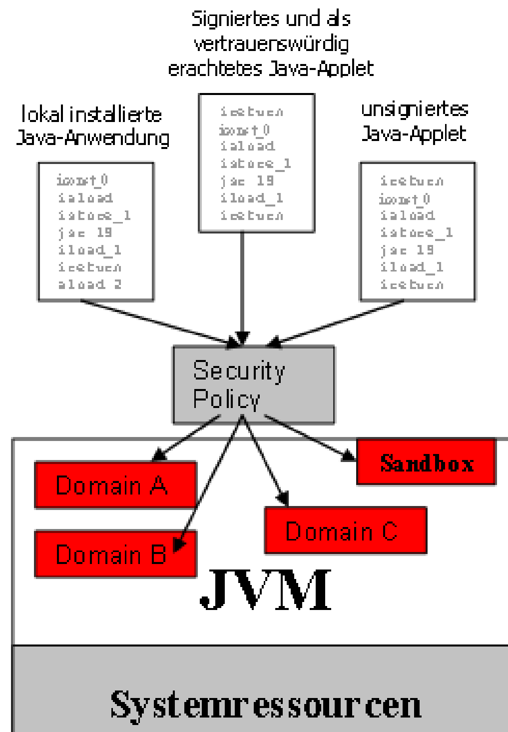



Abbildung 3.2.3.:Protection Domains in Java 2

### 3.3. Komponenten der Sandbox

Das Java Security Model umfasst jeden Aspekt der Java Architektur. Gäbe es Teile in der Java Architektur, deren Sicherheit nicht bedacht worden wären, könnten diese Teile ausgeforscht werden, um sie dazu zu benutzen, die Sandbox zu umgehen.

Um die Sandbox Architektur zu verstehen müssen verschiedene Teile der Java Architektur betrachtet werden.

Die Hauptbestandteile der Java Sandbox sind:
- Die Class Loader Architektur
- Der Class File Verifier
- Der Security Manager

Im Folgenden soll jede Komponente der Sandbox im Einzelnen betrachtet werden.

## *4. Die Class Loader Architektur*

Der Class Loader trägt auf drei Arten zur Java Sandbox bei, die hier verdeutlicht werden sollen: [VEN 99]

1. Er hält möglicherweise gefährlichen Code von der Beeinflussung korrekten und sicheren Codes ab. Dies wird über Namespaces gehandhabt.
2. Er verhindert das Verändern von sogenannten *trusted class libraries*

3. Er teilt Code in Kategorien ein (Protection Domains), die dem Programm seine Ausführungsrechte und Benutzungsrechte zuweisen.

**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 mehrere *Auto*-Klassen in eine Virtual Machine zu laden, solange sich diese in unterschiedlichen Namespaces befinden(von unterschiedlichen Class Loadern geladen werden).
Namespaces legen eine Art Schutzschild zwischen Klassen unterschiedlicher Namespaces 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.

**Trusted class libraries**

*Trusted class libraries* sind die Pakete, die von der Java Virtual Machine als definitiv sicher angesehen werden. Dazu gehören die Klassen der Core Java API.
In den meisten VM Implementierungen älteren Versionen war der eingebaute Standard (Primordial) Class Loader für das laden lokal verfügbarer Klassen verantwortlich. Seit Java Version 1.2 bilden die Class Loader eine Hierarchie. Die Class Loader wurden in einer Vater-Sohn Beziehung angeordnet. Der sogenannte *Bootstrap Class Loader* steht dabei in der Hierarchie an der Spitze. Dieser Class Loader ist nur für das Laden der Klassen der Core-Java API zuständig. Für das Laden anderer Klassen, wie z.B. die Klassen der ausgeführten Applikation, sind seit Version 1.2 benutzerdefinierte Class Loader verantwortlich. Wird eine Version 1.2 Virtuelle Maschine gestartet, werden also einer oder mehrere benutzerdefinierte Class Loader gestartet. (abhängig von der Java Plattform Implementierung).
Am unteren Ende dieser Kette ist der Default *System Class Loader*. Der Name System Class Loader bezeichnet üblicherweise den benutzerdefinierten Class Loader, der die erste Klasse der Applikation lädt.
Möchte also eine Applikation eine Klasse laden, so gibt der *System Class Loader* die Anfrage an seinen Vater Knoten weiter, der selbst die Anfrage wieder an seinen Vaterknoten weiterleitet. Der letzte Knoten in der Hierarchie ist der *Bootstrap Class Loader*. Dieser sucht die zu ladende Klasse in der Java API. Findet er die Klasse nicht, so gibt er die Anfrage wieder an seinen Sohn Class Loader weiter, der ebenfalls versucht die Klasse zu laden usw.
Kann der *Bootstrap Class Loader* jedoch die Klasse laden, so gibt er die Klasse an seine Sohn-Knoten weiter, die dann ihrerseits nicht mehr versuchen die Klasse zu laden.

Durch diesen Mechanismus wird verhindert, 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, ist es auch nicht möglich, trusted classes durch eigene Klassen zu überschreiben. Wenn ein Custom Class Loader beispielsweise versuchen

würde, eine eigenen *java.lang.String* Klasse zu laden, würde diese Anfrage als erstes bis zum *Bootstrap Class Loader* 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 Beispiel führt nicht zu dem gewollten Erfolg. Angenommen 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 Class Loader geladen. Da diese im Paket java.lang liegt könnte man jetzt 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 möglich, da die java.lang API Pakete von einem anderen Class Loader geladen wurden, als die java.lang.Virus Klasse. Hier kommt der Begriff des r*untime packages* ins Spiel, der bedeutet, dass sich zwei (oder mehrere) Klassen nur dann im gleichen *runtime package* befinden, wenn sie den gleichen Paketnamen haben und sie vom gleichen Class Loader geladen wurden. Da sich, um auf package-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.

**Protection Domains**

Wenn eine Klasse geladen wird, bekommt sie von ihrem Class Loader eine Protection Domain zugewiesen. Eine Protection Domain verwaltet alle Rechte, die einer bestimmten Klasse gewährt werden. Diese regelt also, inwiefern eine Klasse auf Systemressourcen, wie Datei I/O und das Netzwerk zugreifen darf (siehe Abschnitt 8.2).

## *5. Der Class File Verifier*

Ein korrekter Java Compiler sollte eigentlich nur Class Dateien generieren, die allen Anforderungen genügen, welche in der Spezifikation der JVM angegeben sind. Allerdings hat die JVM keine Möglichkeit zu überprüfen, ob eine Datei, die geladen wird, auch von einem korrekten Java-Compiler generiert wurde. Dies hat aber große Auswirkungen auf die Java Sicherheit. Würde der Bytecode vor der Ausführung nicht verifiziert werden, wäre es für einen ungültigen Bytecode beispielsweise möglich durch illegale Operationen die Virtuelle Maschine zum Absturz zu bringen. Weiters wäre es möglich durch eine inkorrekte Typumwandlung Zeiger umzusetzen und nicht erlaubte Operationen durchzuführen, oder Zugriff auf geheime Informationen zu erlangen.

Es muss also eine Verifikation des Bytecodes durchgeführt werden, mit der sichergestellt wird, dass der Class Datei vertraut werden kann.

Für diese Verifikation ist der Class File Verifier zuständig. Die Verifikation besteht aus vier Durchgängen.

### Pass 1

In Durchlauf 1 wird das grundlegende Format der Class Datei überprüft. Dies geschieht beim Lesen in den Java Interpreter.
Mit diesen grundlegenden Tests kann folgendes verifiziert werden:
- die ersten vier Bytes enthalten die korrekte magic Number 0xCAFEBABE
- alle ermittelten Attribute besitzen die korrekte Länge
- die Class Datei ist weder verkürzt, noch enthält sie leere Bytes am Anfang der Datei
- der Konstanten Pool enthält keine unbekannten Informationen

### Pass 2

Falls die Class Datei den ersten Durchlauf überlebt hat, wird in einem zweiten Durchlauf geprüft, ob alle Java Konzepte korrekt implementiert wurden. Es wird insbesondere überprüft, ob:
- finale Klassen keine Unterklassen haben
- alle Klassen eine Oberklasse haben, außer der Klasse Object
- der Konstanten Pool korrekt formatiert ist
- alle Datenfelder und Methodenaufrufe im Konstanten Pool auf gültige Namen und Typdeskriptoren verweisen.

Die Verifikation der Existenz referenzierter Datenfelder, Methoden und Klassen wird in dieser Phase nicht durchgeführt.

### Pass 3

Pass 3 wird als der Bytecode Verifier bezeichnet. In dieser Phase wird die aktuelle Class Datei analysiert. Dieser Prozess geschieht während dem Linken und ist sehr komplex. In Durchgang 3 wird für jede Methode eine Datenflussanalyse durchgeführt (siehe 5.1. Der Bytecode Verifier).

Diese Analyse garantiert insbesondere, dass unabhängig davon, wie man bei der Programmausführung zu einer Anweisung gelangt, folgende Aussagen zutreffen:
- dass der Operanden Stack immer die korrekte Größe hat
- Variablen wurden vor der Benutzung initialisiert
- Methoden werden mit korrekten Parametern aufgerufen.
- Felder bekommen nur Werte des korrekten Typs zugewiesen
- Instruktionen werden mit passenden Operandentypen aufgerufen
- Sprunganweisung springen immer an den Anfang einer Instruktion

### Pass 4

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.

Bei der Sun JVM Implementation wird beim ersten Mal, bei dem eine Anweisung einer Klasse referenziert wird, folgendes ausgeführt:

- die Definition der referenzierten Klasse wird geladen, falls dies nicht bereits geschehen ist.
- es werden Security Checks durchgeführt: darf die ausführende Klasse auf diese Klasse referenzieren.
- initialisiert die Klasse, falls dies nicht bereits geschehen ist.

Beim ersten Aufruf einer Methode oder dem Zugriff oder modifizieren eines Datenfeldes wird geprüft:

- ob die referenzierte Methode oder das referenzierte Datenfeld überhaupt in der gegebenen Klasse existiert.
- ob die referenzierte Methode oder das referenzierte Datenfeld eine korrekte Beschreibung besitzt.
- ob die gerade auszuführende Methode Zugriff auf die referenzierte Methode oder Datenfelder besitzt.

## 5.1. Der Bytecode Verifier

Wie oben erwähnt, handelt es sich bei der Bytecode Verifikation um die komplexeste Phase der Verifikation. Dabei wird zuerst der Bytecode selbst überprüft. Zuerst wird der Bytecode in einzelne Befehlsfolgen unterteilt, die dann darauf untersucht werden, ob alle Kontrollflussinstruktionen zum Beginn einer Instruktion springen, Referenzen auf lokale Variablen und den Konstantenpool korrekt sind und der Code nicht plötzlich abbricht.

Als nächstes findet eine Datenflussanalyse für jede Methode statt, die die Auswirkung jeder Instruktion auf den Stack simuliert.

Zuerst muss die Datenflussanalyse vorbereitet werden. Dafür wird die Situation vor der Ausführung der ersten Instruktion einer Methode betrachtet.
Das Local Variable Array wird initialisiert:

- Referenz auf this: Typ der Klasse
- Typen der Parameter, die an die Methode übergeben wurden

Dann wird der Operanden Stack initialisiert. Er ist zu Beginn leer. Vor der Datenflussanalyse wird die erste Instruktion als *changed* markiert, alle übrigen *unchanged.*

Schließlich wird die Datenflussanalyse durchgeführt. Dabei wird folgender Algorithmus ausgeführt:

1. Wähle eine Instruktion mit gesetztem *changed* Bit und lösche das Bit. Ist keine Instruktion mit gesetztem *changed* Bit mehr übrig, wurde die Methode erfolgreich verifiziert.

2. Modelliere den Effekt der Instruktion auf Operand Stack und Local Variable Array.
   - Falls die Instruktion Werte vom Operanden Stack benutzt, überprüfe ob genug Werte am Stack liegen und ob sie den korrekten Typ besitzen. Ist dies nicht der Fall schlägt die Verifikation fehl.
     Diese Überprüfung würde beispielsweise fehlschlagen, wenn es sich bei der aktuellen Instruktion um eine *iadd*-Anweisung handelt, und nur ein Wert am Operanden Stack liegt.
   - Falls die Instruktion lokale Variablen verwendet, überprüfe ob die lokale Variable den richtigen Typ hat. Andernfalls schlägt die Verifikation fehl.
   - Wird von der Instruktion ein Wert auf den Operanden Stack gelegt muss verifiziert werden ob der Operanden Stack diese Variable noch aufnehmen kann.
   - Ändert die Instruktion eine lokale Variable, so muss festgehalten werden, dass die Variable nun einen neuen Typ enthält.

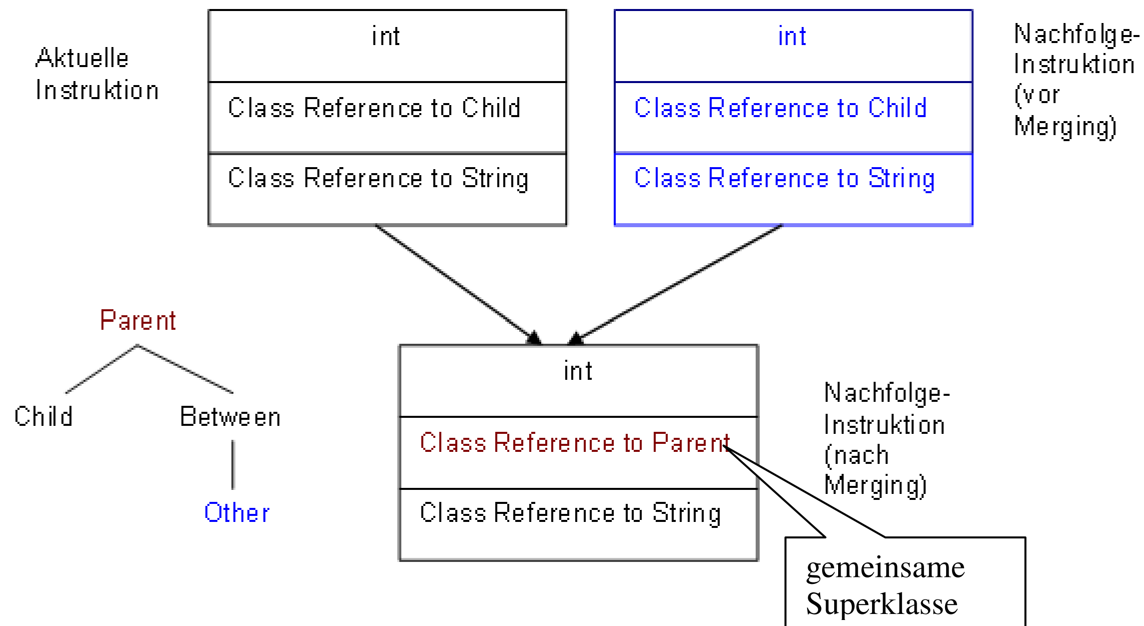
3. Finde die Menge ***N*** möglicher Nachfolge–Instruktionen: Nachfolge Instruktionen können sein:
   - Die im Codesegment folgende Instruktion, falls es sich bei der aktuellen Instruktion weder um *goto, return* oder *athrow*-Anweisung handelt.
   - Sprungziel eines unbedingten Sprungs
   - Bedingter Sprung: Es gibt 2 Nachfolgeinstruktionen: Das Sprungziel und die folgende Instruktion (diese wird ausgeführt falls die Bedingung nicht erfüllt ist)
   - Exception Handler: Nachfolgeinstruktion ist die erste Instruktion der Ausnahmebehandlungsroutine

4. Verschmelze den Operanden Stack und das Local Variable Array mit dem Operanden Stack und dem Local Variable aller Elemente von ***N***:
   - Falls die Nachfolge Instruktion zum ersten Mal besucht wird: Ordne den Operanden Stack und das Local Variable Array dieser Instruktion zu und setze das *changed* Bit der Operation.
   - Falls die Nachfolge-Instruktion schon einmal besucht wurde, so verschmelze den Operanden Stack und das Local Variable Array mit dem Oprendan Stack und dem Local Variable Array, dass dieser Instruktion zugewiesen wurde (ist eine Verschmelzung nicht möglich so schlägt die Verifikation fehl).
     Das Mergen zweier Operanden Stacks/Lacal Variable Arrays wird in 5.1.1/5.1.2 genauer beschrieben.
     Setze das *changed* Bit falls sich Typen am Operanden Stack oder am Local Variable Array geändert haben. Andernfalls lösche das *changed* Bit.

5. Weiter bei Schritt 1.

### 5.1.1. Verschmelzen (Mergen) zweier Operand Stacks:

- je zwei korrespondierende Stack-Elemente müssen typmässig kompatibel sein
- Bei unterschiedlichen Referenzen nimm erste gemeinsame Superklasse
- Bei unvereinbaren Typen -> Verifikation schlägt fehl




### 5.1.2. Verschmelzen (Mergen) zweier Local Variable Arrays:

- Typmässig unvereinbare lokale Variablen erhalten speziellen Typ *unusable* -> Verifikation schlägt nicht fehl.
- Bei unterschiedlichen Referenzen nimm erste gemeinsame Superklasse

## 6. Der Security Manager

Mit dem Security Manager wird die Zugriffskontrolle realisiert. Er ist seit der Version 1.0 im JDK enthalten. Die Basis-Klassenbibliothek ist so programmiert, dass der Sicherheits-Manager immer angefragt wird, bevor potentiell gefährliche Operationen ausgeführt werden. Die Operationen, die als gefährlich eingeschätzt werden und deshalb mit dem Security Manager kontrolliert werden können, sind:

- Netzwerkzugriffe
- alle Operationen zum Manipulieren von und der Zugriff auf Threads
- der Zugriff auf Systemressourcen
- Zugriffe auf das Dateisystem
- das Aufrufen von lokalen Programmen und Betriebssystem-Kommandos

Welcher Zugriff dabei vom Security Manager auf Grundlage welcher Faktoren (z.B. Herkunft des Programms) gewährt wird, ist nicht im Java-Sicherheitsmodell festgelegt, sondern hängt von der jeweiligen Implementierung der Methoden des Security Managers ab.

Konkret ist der Sicherheits-Manager ein Objekt, das für jede kritische Operation eine Methode zur Verfügung stellt, welche für die entsprechende Operation überprüft, ob sie ausgeführt werden darf oder nicht.

Da es in Java Applets und nicht lokalen Klassen verboten ist native Methoden zu verwenden, ist es diesen Klassen nur über die Java API möglich auf Systemressourcen zuzugreifen. Die Java API ist so programmiert, dass vor der Ausführung einer kritischen Aktion, also vor dem Zugriff auf wichtige Systemressourcen, der Security Manager angefragt wird. Zu diesem Zweck stellt der Security Manager sogenannte check-Methoden zur Verfügung.

Beispielsweise wird die Methode *public void checkDelete(String file)* des Security Managers von der Java API immer aufgerufen, bevor eine Datei gelöscht wird. Diese Methode muss überprüfen, ob die als Parameter angegebene Datei gelöscht werden darf und wenn dies nicht der Fall ist, eine Exception erzeugen, so dass die Operation abgebrochen wird.

Vor JDK Version 1.2. war die Klasse *java.lang.SecurityManager* eine abstrakte Klasse. Um benutzerdefinierte Sicherheitsrichtlinien zu installieren musste man seinen eigenen Security Manager schreiben, und von der Klasse *java.lang.SecurityManager* ableiten.
Sobald eine Applikation dann den Security Manager instantiiert und installiert kümmert sich der Security Manager um die Einhaltung der Sicherheitsrichtlinien, die durch die check-Methoden definiert wurden.

Um eine differenziertere Sicherheitspolitik basierend auf signierten Code zu erreichen, wurde in Java 1.2 die Klasse *java.lang.SecurityManage*r eine konkrete Klasse, die eine default Implementierung des Security Managers darstellt. Der default Security Manager kann durch Aufruf folgender Option über die Kommandozeile installiert werden:

```
-Djava.security.manager
```

Seit JDK 1.2 hat man nun die Möglichkeit eine benutzerdefinierte Sicherheitsrichtlinie, anstatt in Java-Code, in einem ASCII-File, genannt Policy File, zu definieren.

Beim Zugriff auf Systemressourcen wird in Java 2 zwar weiterhin der Security Manager konsultiert, die Zugriffsregeln allerdings sind nicht mehr in ihm implementiert, sondern werden an den sogenannten Access Controller weitergeleitet. Wenn also eine check-Methode des default Security Managers aufgerufen wird, so wird der Request an die Klasse AccessControler weitergeleitet. Der Access Controller verwendet die Information, die in den Protection Domain Objekten der Klassen, die auf dem Call Stack liegen, enthalten ist, und führt Stack Inspection durch, um zu entscheiden, ob die Aktion erlaubt werden soll oder nicht. (siehe 8. Rechteverwaltung)

# *7. Signing*

Wie bereits erwähnt erwies sich das Sandbox Modell der JDK Version 1.0 als zu unflexibel. Im JDK 1.0 waren nicht einmal Standardanwendungen wie z.B. ein einfacher Texteditor als Applet möglich, da man hierzu die entsprechenden Schreib- bzw. Leserechte benötigt.

Um die Einschränkungen der Sandbox für vertrauenswürdige Applets aufzuheben, wurde mit dem JDK 1.1 die Möglichkeit zur Signierung von Applets eingeführt. Eine Signatur stellt eine digitale Unterschrift dar. Sie stellt sowohl sicher, dass das Applet von einen bestimmten Autor stammt (Datenidentität) als auch, dass das Applet auf Weg vom Autor zum Nutzer nicht verändert wurde (Datenintegrität).
Ein signiertes Applet, dessen Autor vom Benutzer als vertrauenswürdig akzeptiert wird, wird als vertrauenswürdig eingestuft und erhält damit erweiterte Rechte. Im JDK 1.1 erhält es genauso wie lokale Applets und Applikationen alle Rechte. Im JDK 1.2 kann der Umfang der zusätzlichen Rechte eingegrenzt werden (siehe 8.Rechteverwaltung).

## 7.1. Grundlagen

Bei der digitalen Unterzeichnung wird vom JDK das Public-Key-Konzept angewandt. Dabei wird ein Schlüsselpaar erzeugt. Ein solches Schlüsselpaar besteht aus einem privaten Schlüssel (private key) und einem zugehörige öffentliche Schlüssel (public key). Der private Schlüssel darf nur dem Schlüsselinhaber bekannt sein, während der öffentliche Schlüssel öffentlich bekannt gemacht wird. Mit dem privatem Schlüssel kodierte Nachrichten sind nur mit dem öffentlichen Schlüssel entschlüsselbar.

Bei der digitalen Unterzeichnung eines Applets wird zunächst mit Hilfe einer Einweg-Hashfunktion ein Hashwert gebildet. Dieser Hashwert wird dann mittels des privaten Schlüssels verschlüsselt. Der verschlüsselte Hashwert wird an die Ursprungsnachricht angefügt. (Abbildung 7.1)

Der Empfänger der Nachricht kann mittels des öffentlichen Schlüssel, den er vorher vom Urheber erhalten hat, diesen Hash durch Entschlüsselung ermitteln, und mit dem Hash, den er selbst aus der übermittelten Nachricht errechnet hat, vergleichen. Sind beide Werte identisch, kann der Empfänger davon ausgehen, dass die Nachricht vom Inhaber des öffentlichen Schlüssel stammt und auf dem Weg zu ihm nicht verändert wurde. Wäre die Nachricht nachträglich verändert worden, wurde der errechnete Hashwert nicht mehr mit dem verschlüsselten Hash übereinstimmen.

Es stellt sich jedoch hierbei die Frage, wer dem Empfänger garantiert, dass ein öffentlicher Schlüssel wirklich echt ist. Es könnte ja schließlich auch Hacker geben, die falsche öffentliche Schlüssel weitergeben und somit korrekte Signaturen für gefälschte Nachrichten berechnen. Es müssen also entsprechende Schlüsselzertifikate bereitgestellt werden, damit sichergestellt werden kann, dass eine Signatur auch zu dem Signierer passt. Ein Zertifikat ist eine Zusicherung eines Zertifikatsausstellers, dass der öffentliche Schlüssel des Zertifikatsinhaber korrekt ist. Gültige Zertifikate erhält man z.B. bei VeriSign (http://verisign.com) oder bei Thawte Certification (http://www.thawte.com).

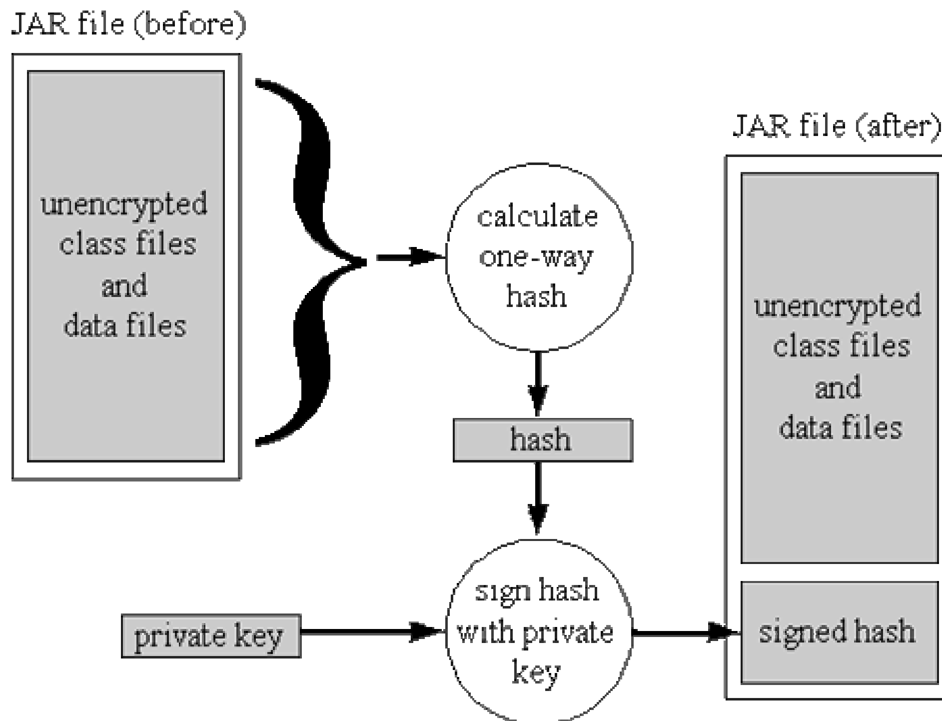



Abbildung 7.1.: Digitales Signieren eines Jar Archivs [Ven99]

## 7.2. Erstellung eines signierten Applets

Im Folgenden möchte ich die Erstellung und Verwaltung von Schlüsselpaaren und Zertifikaten sowie die Generierung von signierten Applets mit Hilfe der Tools jarsigner und keytool erläutern.

1. Erstellung eines Schlüsselpaars und entsprechenden Zertifikats

Um ein Schlüsselpaar zu erzeugen, wird keytool mit dem der Option -genkey aufgerufen. Ein Aliasname für den Schlüsselinhaber wird angegeben. Weiters können zusätzliche Optionen angegeben werden, z.B. Signieralgorithmus, Schlüsselalgorithmus, Gültigkeitsdauer, Ort des Schlüsselspeichers etc.

```
keytool -genkey -alias me
```

Das Programm fragt jetzt nach dem Passwort des Schlüsselspeichers und nach Angaben zum Schlüsselinhaber:

```
Geben Sie das Keystore-Passwort ein:  meinpasswort
Wie lautet Ihr Vor- und Nachname?
 [Unknown]:  Johanna Fitzinger
Wie lautet der Name Ihrer organisatorischen Einheit?
 [Unknown]:  Fachbereich Informatik
Wie lautet der Name Ihrer Organisation?
 [Unknown]:  Universität Linz
Wie lautet der Name Ihrer Stadt oder Gemeinde?
 [Unknown]:  Linz
```

```
Wie lautet der Name Ihres Bundeslandes oder Ihrer Provinz?
 [Unknown]:  Austria
Wie lautet der Landescode (zwei Buchstaben) f³r diese Einheit?
 [Unknown]:  AT
Ist CN=Johanna Fitzinger, OU=Fachbereich Informatik, O=Universität Linz,
L=Linz,
 ST=Austria, C=AT richtig?
 [Nein]:  j
Geben Sie das Passwort f³r <me> ein.
        (EINGABETASTE, wenn Passwort dasselbe wie f³r Keystore):
```

Dann erzeugt das Programm ein entsprechendes Schlüsselpaar und ein zugehöriges selbstzertifizierendes Zertifikat; d. h. Aussteller und Inhaber des Zertifikats sind in diesem Fall identisch. Das Werkzeug keytool bietet noch weitere Optionen, z.B. zum Export und Import von Zertifikaten, zum  Erstellen von Zertifikatsanforderungen an Zertifikatsaussteller.

2. Erstellung eines Jar-Archives

Um ein Jar-Archiv zu erstellen, wird das tool `jar` benutzt:

```
jar cvf MeinArchiv.jar *.class
```

3. Signierung eines Jar-Archives

Das Archiv kann mit dem Tool jarsigner signiert werden:

```
jarsigner MeinArchiv.jar me
Enter Passphrase for keystore: meinpasswort
```

Das unsignierte Archiv MeinArchiv.jar wird mit dem Schlüssel von *me* signiert, indem dem Archiv zwei weitere Dateien hinzugefügt werden. Die Signatur-Datei mit der Endung .sf enthält pro  Datei des Archivs einen Eintrag mit Namen der Datei, dem Namen der genutzten Hash-Funktion und den Hashwert selbst. Diese Signatur-Datei wird mit dem privaten Schlüssel des Signierers signiert. Die so gewonnene Signatur wird zusammen mit dem verschlüsselten Zertifikat des Signierers in die Signaturblock-Datei (Endung `.dsa`)  eingefügt.

## *8. Rechteverwaltung:*

Obwohl die Einführung signierter Applets eine Verbesserung  der Einsatzmöglichkeiten von Applets brachte, war das Sicherheitsmodell noch immer zu unflexibel. Entweder erhielt ein Applet nur die Sandbox Rechte, oder es bekam alle Rechte, falls es sich um ein signiertes und als vertrauenswürdig erachtetes, oder ein lokales Applet handelte.

Deshalb wurde für das JDK 1.2 ein flexibleres und feinkörnigeres Sicherheitsmodell (fine-grained security) entwickelt. Dabei können jeder geladenen Klasse entsprechend ihrem Ursprungsort (CodeBase) und ihrem Autor (Signierer) gezielt ganz bestimmte Rechte zuerkannt werden. Dazu wurden Schutzdomänen eingeführt (Protection Domains). Jede geladene Klasse befindet sich in genau einer solchen Protection Domain, eventuell zusammen mit anderen Klassen desselben Ursprungs, welche dieselben Rechte haben. Diese Rechte werden durch sogenannte Permissions dargestellt.

Für jedes Recht wurde im JDK 1.2 eine eigene Klasse definiert (abgeleitet von der Klasse *java.security.Permission*). Es können auch eigene Rechteklassen definiert werden. Diese müssen von der Klasse *java.security.Permission* abgeleitet werden.

Seit JDK 1.2 hat man außerdem die Möglichkeit eine benutzerdefinierte Sicherheitsrichtlinie, anstatt in Java-Code, in einem ASCII-File, genannt Policy File, zu definieren.

Die Überprüfung auf ausreichende Rechte wird seit JDK1.2 von der Klasse AccessController bewerkstelligt. Dabei werden Maßnahmen ergriffen, um Missbrauch von Code mit mehr Rechten durch ein Programm mit wenigen Rechten zu verhindern. Dies geschieht durch Überprüfung aller Rufer am Call Stack. Dieser Mechanismus wird als Stack Inspection bezeichnet.

## 8.1 Policy File

Die zu gewährenden Rechte werden in `.policy`-Dateien gespeichert. In ihnen sind entsprechende Policy-Einträge enthalten. Diese Einträge haben folgende Form:

```
grant [ SignedBy "Signer_name1" ] [ Codebase "url"] {
    permission permission_class_name1 "target name"[,"action"][,Signedby
"Signer_name1"]; };
```

// Angaben in eckigen Klammern können auch fehlen.

Der Eintrag besteht aus dem Schlüsselwort grant, gefolgt von dem Signierer bzw. einer Liste von Signierern (SignedBy + Alias-Namen der Signierer. Die entsprechenden Zertifikate müssen im Schlüsselspeicher enthalten sein) und dem Ort, von dem das Applet geladen wird (CodeBase + URL-Adresse). Diese Angaben sind optional. Fehlen sie, so gelten die Rechte für sowohl für signierte als auch für unsignierte Applets, bzw. für Applets, die von einem beliebigen Ort geladen wurden.
Die eigentlichen Rechte stehen in den geschweiften Klammern. Sie bestehen aus dem Namen der Rechteklasse, dem Zugriffsziel und den zulässigen Aktionen. Außerdem kann für Rechteklassen eine Signierung gefordert werden.
Sind Einträge syntaktisch fehlerhaft, so werden sie ignoriert.

Eine Policy-Datei könnte z.B. folgendermaßen aussehen:

*der Schlüsselspeicher, welcher die Zertifikate (incl. öffentlicher Schlüssel) der Signierer enthält:*
keystore “.keystore“;

*Applets, die von trustme unterzeichnet sind und von der Webadresse www.trustme.com geladen wurden, werden folgende Rechte zuerkannt:*
*- Voller Zugriff auf alle Dateien*
*- Recht zur Ermittlung der systemweiten Sicherheitseinstellungen*

```
grant SignedBy "trustme" CodeBase "http://www.trustme.com/-" {
        permission java.io.FilePermission "<<ALL FILES>>", "read, write,
        delete, execute";
        permission java.security.SecurityPermission "getPolicy";
};
```

Diese Policy-Dateien können mit Hilfe eines normalen Texteditor bearbeitet werden. Für Windows und Solaris existiert das GUI-Tool policytool. (Abbildung 8.1)

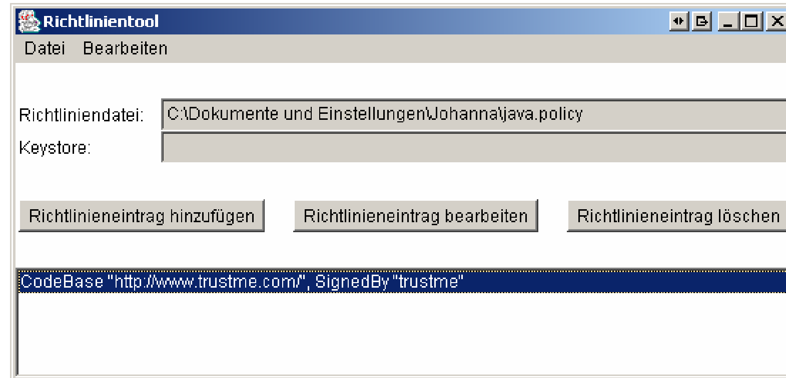



Abb. 8.1. Das policytool

## 8.2 Protection Domains

Wenn ein Typ von einem Class Loader geladen wird, wird er einer Protection Domain zugewiesen. Eine Protection Domain verwaltet alle Permissions, die einer bestimmten Code Source gewährt wderden. Eine Protection Domain entspricht also einem oder mehreren Grant-Klauseln eines Policy Files.

## 8.3 AccessController

Seit JDK 1.2 ist der AccessController ist die Instanz in der Java-API, die den Zugriff auf wichtige Systemressourcen überwacht.
Der AccessController wendet dabei Stack Inspection an, um zu entscheiden ob eine potentiell unsichere Aktion erlaubt werden soll. Dabei wird jeder Rufer am Call Stack überprüft. Nur wenn alle Rufer entsprechenden Permissions besitzen, darf auf eine Ressource zugegriffen werden.

Sicherheitskritische Operationen führen aus Gründen der Abwärtskompatibilität zu Anwendungen und Applets älterer Java-Versionen nach wie vor zum Aufruf der entsprechenden check-Methode des Security Managers. Der Security Manager entscheidet jedoch nicht mehr selbst, sondern ruft mit checkPermission() eine Methode des Access Controllers auf. Dieser überprüft ob eine Aktion erlaubt ist.

Der AccessController besteht eigentlich nur aus einer Sammlung von Methoden. Die Klasse besitzt keinen aufrufbaren Konstruktor und ist somit zumindest von außerhalb nicht instantierbar. Sämtliche Methoden sind als static deklariert, und die Klasse ist als final gekennzeichnet, somit ist es auch unmöglich, weitere Subklassen zu erzeugen.

## Stack Inspection

Wie oben erwähnt, verwendet der AccessController die Information, die in den Protection Domain Objekten der Klassen, die auf dem Call Stack liegen, enthalten ist, und führt Stack Inspection durch, um zu entscheiden, ob die Aktion erlaubt werden soll oder nicht. Bei einer Stack Inspection wird geprüft, ob alle Methoden in der Ruferkette die entsprechenden Permissions besitzen Nur wenn alle Rufer am Call Stack die entsprechende Permission besitzen, darf eine Aktion ausgeführt werden.

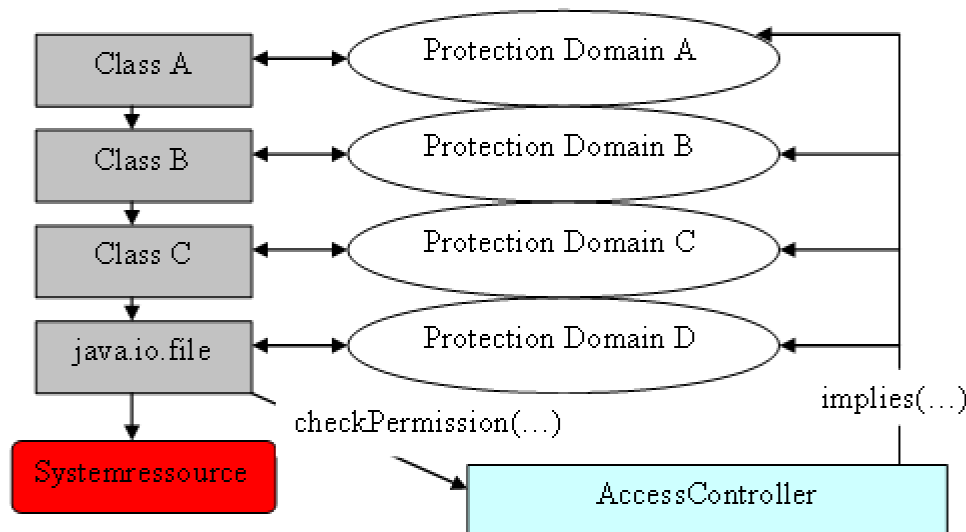



Abb. 8.2. Stack Inspection

Die Abbildung 8.2 soll den Verlauf der Stack Inspection veranschaulichen. In diesem Beispiel ruft die Klasse A eine Methode der Klasse B auf, die wiederum eine Methode der Klasse C aufruft, welche schließlich eine Methode der Klasse java.io.file aufruft. Wie bereits oben erwähnt, ist die Basisklassenbibliothek so programmiert, dass eine kritischen Aktion (im Beispiel also ein Zugriff auf das Dateisystem) vor der Ausführung zuerst mit dem Security Manager, bzw. seit JDK 1.2 mit dem Access Controller überprüft wird. Die Klasse java.io.file veranlasst also durch Aufruf der Methode checkPermission( Permission perm) den Access Controller zur Überprüfung der Rechte. (Aus Gründen Abwärtskompabilität wird zuerst der Security Manager angefragt, der wiederum den Aufruf an den AccessController weiterleitet. Dieser Umweg soll hier vernachlässigt werden.)

Der AccessController überprüft der Reihe nach alle Rufer am Stack auf die geforderte Permission, indem er die Methode implies(Permission p) aufruft. Die Methode implies() ist in der Klasse ProtectionDomain deklariert. Als einzigen Parameter erwartet diese Methode ein Objekt vom Typ Permission. Diese Methode überprüft ob eine Permission in der ProtectionDomain enthalten ist. Ist das der Fall wird true zurückgegeben, andernfalls false.

Stellt der AccessController während der Stack Inspection fest, dass einer der Rufer nicht über die geforderte Permission verfügt, wird die weitere Überprüfung mit der Erzeugung einer AccessControlException abgebrochen und der Zugriff auf die Ressource kann nicht ausgeführt werden. Die Operation bricht ihrerseits mit einer SecurityException ab.

Kann jedoch die Überprüfung bis zum Ende des Call Stacks fortgesetzt werden, wird die Überprüfung durch den AccessController erfolgreich beendet und die Methode checkPermission() kehrt ohne Exception zum Rufer zurück. Danach kann der Zugriff auf das Dateisystem ausgeführt werden.

.

## Priviligierter Code

Die oben beschriebene restriktive Rechteüberprüfung hat vor allem den Sinn, Missbrauch von Code mit hohen Privilegien zu verhindern. Allerdings wird durch diesen Mechanismus auch der normale Gebrauch erschwert. Beispielsweise müsste eine Anwendung, die ein Fenster mit Text darstellt, ebenfalls das Recht haben, die Fontdateien lesen zu dürfen. In solchen Fällen ist es sinnvoll, dass eine Klasse, insbesondere eine Systemklasse, Operationen durchführen kann, obwohl der sie aufrufende Code dazu nicht berechtigt ist.

Der AccessController bietet diese Möglichkeit über die Methode doPrivileged(PrivilegedAction action). Hierbei ist PrivilegedAction ein Interface, das nur die Methode run() besitzt, mit einem beliebigem Objekt als Rückgabewert. Der Aufruf von doPrivileged() sieht folgendermaßen aus:

```
AccessController.doPrivileged(
        new Privileged Action(){
                public Object run() {

                        // Aufruf der kritischen Operation

                        return null;
                }
        }
);
```

Der Aufruf von doPrivileged(...) erzwingt, dass die Überprüfung der Rechte nach der Überprüfung der Klasse, die doPrivileged aufgerufen hat, beendet wird. Es wird also vom AccessControler nur überprüft, ob die Klasse, die den doPrivileged() Aufruf ausführt, über entsprechende Rechte zur Ausführung der Operation besitzt.

Somit läuft der Code in der run()-Methode mit den Rechten der Klassen, die doPrivileged(...) aufgerufen hat, ohne dabei durch vorher beteiligte Klassen eingeschränkt zu werden.

## *9. Referenzen:*

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

[Yel97] Frank Yellin, Tim Lindholm: The Java Virtual Machine Specification, Addison-Wesley 1997

[Yel96] Frank Yellin: Low Level Security in Java
http://java.sun.com/sfaq/verifier.html

[Rae01] Martin Raepple: Sicherheitskonzepte für das Internet
http://www.dpunkt.de/buch/3-89864-116-3.html
http://www.dpunkt.de/leseproben/3-89864-116-3/Kap._2.17.pdf

[Pilz97] Markus Pilz: Das Sicherheitskonzept von Java
http://www.ifi.unizh.ch/richter/people/pilz/pubb/Pilz97a.pdf

[Joll98] J.M. Joller: Java Security
http://www.joller-voss.ch/ndkjava/notes/security/JavaSecurity.pdf