In diesem Kapitel:

1. *Einführung in Java Security*

1.1.1. Übersicht

Nach dem Durcharbeiten dieser Einheit sollten Sie in der Lage sein:

- fünf praxiserprobte Sicherheitspraktiken zu identifizieren und zu beschreiben; Sie sollten in der Lage sein aufzuzeigen wie Java diese Praktiken konkret unterstützt.
- die Security Features des Byte Verifiers und der Java Virtual Machine zu beschreiben
- einen Security Manager oder einen Class Loader zu implementieren, um Sicherheitsaspekte zu adressieren
- die Security Klassen von Java einzusetzen, um neue Policy Dateien zu kreieren oder eigene Permission Types zu definieren und zu implementieren

1.1.2. Lernziele

Nach dem Durcharbeiten dieser Unterlagen sollten Sie:

- wissen, wie Klassen schrittweise geladen und überprüft werden.
- welche Funktion der Byte Code Verifier hat
- wann der Class Loader aktiv wird
- wie ein Security Manager definiert werden kann

1.2. Modul 1 - Übersicht über die Java Security

1.2.1. Einführung

In diesem Modul beschreiben wir generell akzeptierte Sicherheitspraktiken. Zudem gehen wir auf Java Lösungen ein, die diese Praktiken implementieren.

1.2.1.1. Lernziele

Nach dem Durcharbeiten sollten Sie in der Lage sein:

- Security zu definieren und fünf Rechnersicherheit zu definieren und fünf gängige Verfahren aufzuführen
- die Security Features von Java zu erläutern :
  Sandbox
  Byte Code Verifier
  Class Loader
  Security Manager
- die Java Lösungen der Sicherheitsprobleme aufzeichnen können.

1.2.2. Was ist Security?

Rechnersicherheit kann definiert werden als Schutz der Rechner Hardware, Software und Daten vor zufälligem oder böswilligem

- Zugriff,
- Nutzung,
- Zerstörung oder
- Offenlegung. Werkzeuge für die Erhaltung der Sicherheit fokusieren sich auf
- Verfügbarkeit,
- Vertraulichkeit und
- Integrität.

1.2.2.1. Referenzen

Die folgende Referenzen liefern Ihnen zusätzliche Informationen zum Thema dieses Moduls:

- ***Secure Computing With Java: Now and the Future*** erhältlich von http://www.javasoft.com/marketing/collateral/security.html
- ***Java Security Solution from JavaOne*** Unterlagen von der JavaOne http://www.javasoft.com/javaone/sessions/slides/TT01/tt01_21.htm
- ***The Cuckoo's Egg*** von Clifford Stoll (Doubleday 1989)
- ***Java Security-Hostile Applets, Holes, and Antidotes*** von McGraw und Felton

1.2.3. Gute Security Praktiken

Da die meisten Rechner mit einem Intranet oder dem Internet verbunden sind, wurde das Thema Rechnersicherheit in den letzten Jahrzehnten zunehmend wichtiger. Zudem wurde das Thema Sicherheit auch wesentlich komplexer durch neue Zugriffsarten (Modem, Kabelnetz, VPN) und neue Möglichkeiten unerlaubt auf Rechner zuzugreifen und neuartige Sicherheitsverletzungen. Die meisten Experten sind sich jedoch einig, dass eine gute Sicherheitspraktik sich um folgende Themen kümmern muss:

- Identifikation und Authentifizierung / Bestätigung / Beglaubigung
- Autorisation
- Ressourcen Kontrolle und Beherrschung
- Vertraulichkeit und Integrität
- Unleugbarkeit
- Prüfen / Auditing

Um ein zuverlässiges Sicherheitsmodell implementieren zu können muss sich der Benutzer der Streitpunkte bewusst sein und Vorsichtsmassnahmen und Sicherheitspläne sollten definiert sein. Effektive Sicherheit kann nur durch folgende Massnahmen erreicht werden:

- Schulung der Benutzer (Mitarbeiter und Angestellte) über Sicherheitsbelange und Taktiken / Policies.
- Implementation eines Einbruchabwehrplans, in dem festgehalten wird, wann Auditdateien nachgeschaut werden müssen und auf was geachtet werden muss.
- Implementation eines Rettungsplans / Wiederherstellungsplans / Recovery Plans für den Fall eines Einbruchs und Festlegung wer was zu tun hat, wer für was verantwortlich ist.

Identifikation und Authentifizierung legen die Legitimierung / Rechtmässigkeit eines Knoten oder eines Benutzers fest, bevor der Zugriff auf einen Rechner oder benötigte Informationen freigegeben wird. Im Verlaufe dieses Prozesses muss unter Umständen ein Benutzer einen Benutzernamen oder eine Kontonummer (Identifikation) und Passwort (Authentifizierung) eingeben, bevor ihm der Zugriff zum Rechner gestattet wird.

Nachdem das System den Benutzer identifiziert und authentifiziert hat, erwartet er, dass falls er eine Kommunikation zu einem andern Rechner aufbaut:

- dieser (andere) Rechner kein Betrüger ist
- die Dateien, die er herunter lädt das enthalten, was er erwartet, insbesondere keine Viren, trojanische Pferde oder Code, der die Integrität oder Sicherheit seines Systems beeinträchtigen könnte.

Das Java Security Package enthält die Werkzeuge `keytool` und `jarsigner` (und ab JDK 1.1 `javakey`), mit deren Hilfe digitale Signaturen und Zertifikate gehandhabt werden können. Digitale Signaturen identifizieren Programmquellen, nicht aktuelle Benutzer. Auf der Browser Ebene hängt die Java Security von der Benutzerauthentifizierung auf Betriebsystemebene ab. Danach schützt eine digitale Signatur den Benutzer.

Autorisierung ist der Prozess zur Bestimmung der erlaubten Aktivitäten für einen authentifizierten Benutzer. Nachdem der Benutzer authentifiziert ist, kann er für unterschiedliche Zugriffe oder Aktivitäten autorisiert werden. Administratoren können über den Level der Zugriffsrechte für autorisierte Benutzer entscheiden. Die Granularität / Körnigkeit dieser Zugriffsberechtigungen ist ein wichtiger Bestandteil der Autorisierung.

Rechnerbenutzer hängen von der Vertraulichkeit und Datenintegrität der Rechner und Rechneranwendungen ab. Vertraulichkeit kann viele unterschiedliche Bedeutungen haben. Die meisten Benutzer verstehen darunter, dass die Informationen vor unberechtigtem Lesen oder Kopieren geschützt sind, sofern der Besitzer der Informationen diese Zugriffe nicht explizit gestattet. Im UNIX, VMS, ... System werden Dateien im Rahmen des Dateisystems geschützt (Lesen, Schreiben, Ausführen, Löschen).

Datenintegrität bedeutet, dass Daten nicht auf unautorisierte Art und Weise verändert wird. Wie könnten Sie mit einem Rechner arbeiten, falls Sie nicht dieses Vertrauen hätten?

Authentifizierung ist eine Voraussetzung für Unleugbarkeit. Unter Unleugbarkeit verstehen wir in diesem Zusammenhang die Unleugbarkeit des Ursprungs, also der eindeutigen Zuweisbarkeit einer Verantwortung. Mittels Unleugbarkeit kann das Sicherheitssystem einen Benutzer eindeutig für eine bestimmte Transaktion verantwortlich machen. Damit werden auch elektronische Vereinbarungen möglich, da eindeutig die Partizipation der einzelnen Teilnehmer bewiesen werden kann.

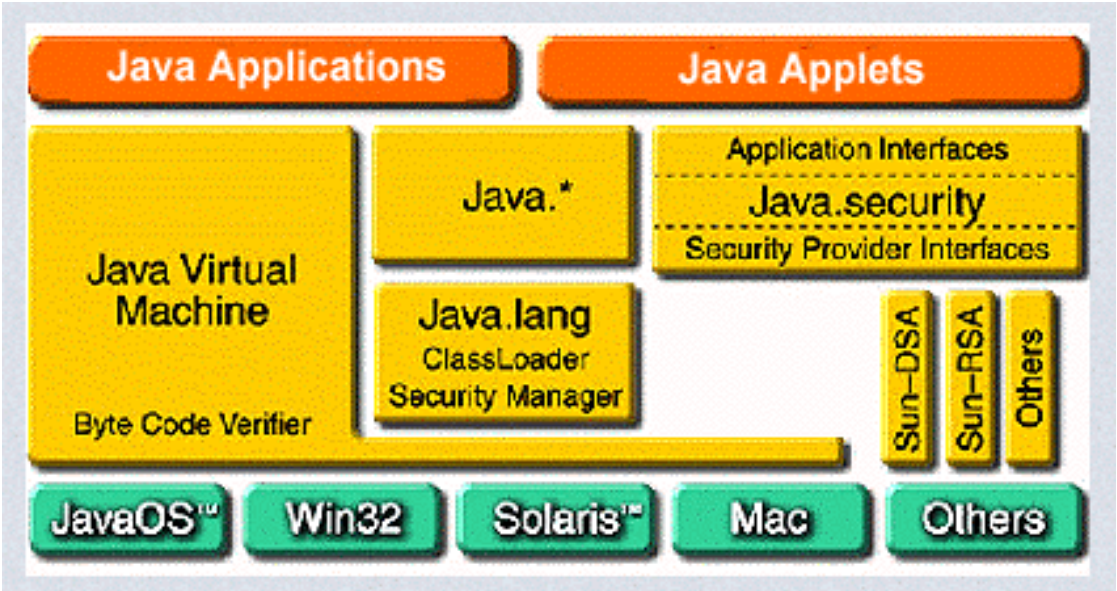
Gute Sicherheitsrichtlinien enthalten Prüfungsgrundsätze. Diese helfen den Administratoren Probleme zu isolieren und zu beheben, wann immer eine Abnormalität oder ein Verstoss eingetreten ist. Prüfungen / Audits bewahren nicht vor Attacken. Aber eine Prüfung kann Aufschluss liefern über den Umfang eines Verstosses und ist damit auch behilflich bei der Behebung des Schadens.

1.2.4. Security Praxis und Java

Die Java Plattform und das Java Entwicklungssystem (JDK) implementiert gute Security Praktiken mit folgenden Spezifika:

- Authentifizierung
  Meldungsauszüge, digitale Signaturen, Zertifikate
- Autorisierung
  Zugriffskontrolllisten, Security Manager, protected Domains (JDK1.2)
- Vertraulichkeit
  public / private Key Encryption / Verschlüsselung
- Unleugbarkeit (engl. non-repudiation)
  Meldungsauszüge und digitale Signturen
- Prüfen / Auditing
  Security Manager und zukünftige Erweiterungen
- Sicherheitsbehälter (engl. containment)
  VM, Class Loader, Security Manager und seit JDK 1.2 geschützte Domänen

Digitale Signaturen, Verschlüsselungsunterstützung, Zugriffslisten und Meldungsauszüge (message digests) sind Teil des `java.security` Pakets. Die Java Cryptography Architecture (JCA) ist Teil des Java Security Package (`java.security`) innerhalb JDK. Das API wurde definiert; aber es bestehen / bestanden bestimmte Exporteinschränkungen. Der Security Manager und Class Loader sind Teil von `java.lang`. Diese Bestandteile ergeben zusammen eine gute Basis für eine gute Sicherheitspraxis.

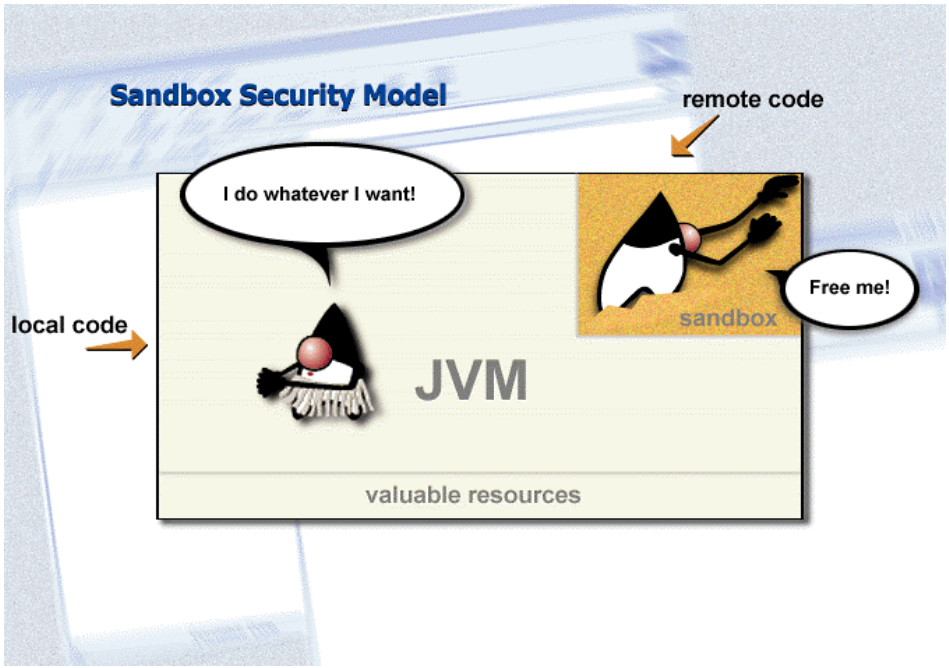


1.2.5. Das Sandbox Sicherheitsmodell

Das ursprüngliche Sicherheitsmodell von Java wird als das Sandbox (Sandkasten) Modell bezeichnet. Dieses Modell wurde ab JDK 1.2 wesentlich erweitert. Um das erweiterte Modell verstehen zu können ist es hilfreich, das Originalmodell zu studieren:

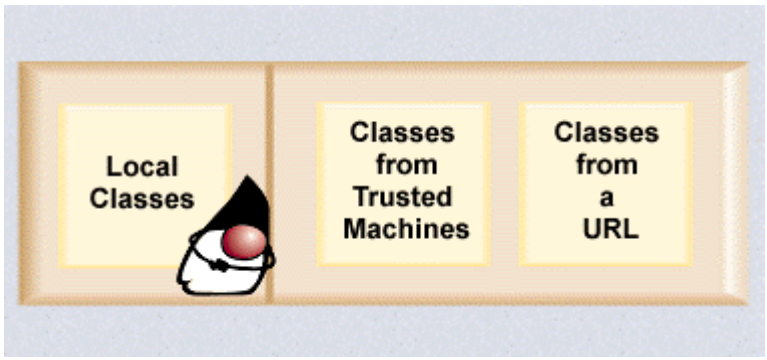
- Byte Code Verifier
- Class Loader
- Security Manager

Diese drei Komponenten werden oft als der Dreizack der Java Sicherheit bezeichnet. Diese Methode ist wie ein Dreifuss: Sie können keines der Beine wegnehmen ohne dass die Stabilität verloren geht. Alle drei Komponenten werden benötigt damit das System sicher Java Code ausführen kann.



Wann immer der Java Compiler eine Java Quelldatei übersetzt, wird eine `.class` Datei generiert. Diese `.class` Datei besteht aus Java Bytecode, einem plattformunabhängigen Zwischenformat. Wenn ein Applet vom Web geladen wird (oder vom Intranet oder von sonstwo), wird der Applet Class Loader vom Browser gestartet. Der Applet Class Loader lädt der Code (den Bytecode für das Applet) und erzwingt Namensraum- und Speichergrenzen. Diese Einschränkungen (engl. controls) bestimmen, auf welche Teile der Java VM das Applet zugreifen kann.

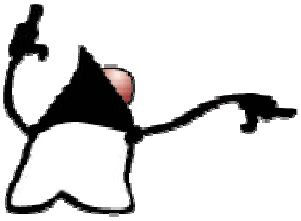
Java Name Space



ACHTUNG - Jede Applet Codebase (Verzeichnis, aus dem der Applet Code geladen wird) erhält einen eigenen Class Loader. Der System Class Loader behandelt alle Ladevorgänge aus dem CLASSPATH.

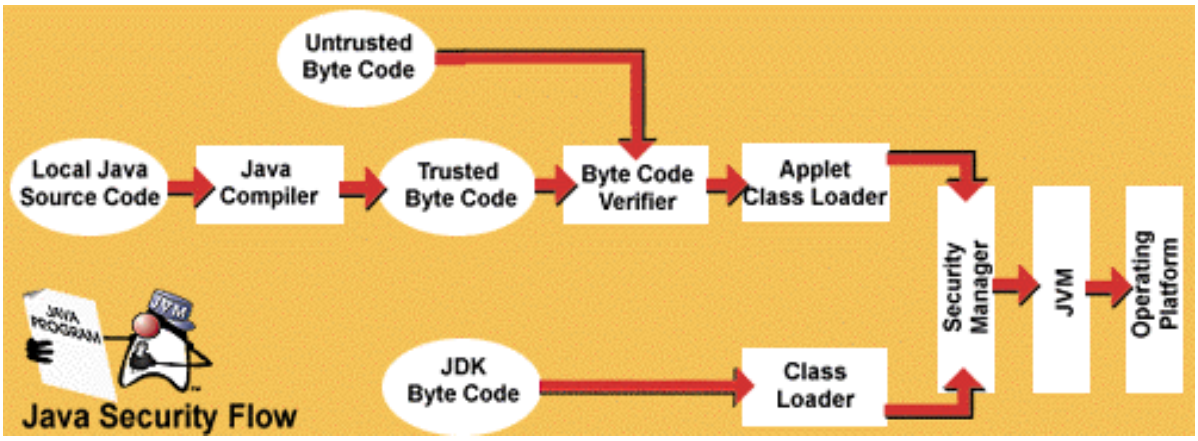
Bevor der Applet Code ausgeführt wird, wird der Byte Code Verifier vom Applet Class Loader geladen.

Die Hauptaufgabe des Byte Code Verifiers besteht in der Überprüfung des Applet Codes auf Konformität mit der Java Sprachspezifikation und darauf, dass keine der folgenden Situationen eintritt:



- Verletzung der Java Sprachregeln
- Verletzung von Namensraum- Einschränkungen
- Stapel Adressüberschreitung oder Unterschreitung (engl. Stack Overflow oder Underflow)
- unzulässige Datentypumwandlungen

Lokaler Programmcode (Code vom lokalen Rechner, der nicht im CLASSPATH steht) wird vom Java Compiler übersetzt und an den Byte Code Verifier weitergereicht. JDK Byte Code (und Programmcode im CLASSPATH) durchläuft den Byte Code Verifikationsprozess nicht.



Es liegt also an Ihnen dem Code zu vertrauen - vertrauenswürdiger Code kann in den CLASSPATH aufgenommen werden. Für nicht vertrauenswürdige (engl. untrusted) Applikationen gibt es die seit JDK 1.2 neue Java Eigenschaft `java.app.class.path`. Diese Eigenschaft wird im Class Loader Modul und auch in den JDK Security Klassen gelesen.

ACHTUNG - das Einbeziehen des aktuellen Verzeichnis (".") im CLASSPATH kann gefährlich sein.

Falls während des Bytecode Verifikationsprozess keine Probleme auftauchen startet die JVM den Applet Code. Wann immer das Applet versucht eine Aktion auszuführen, die ausserhalb des Sandkastens liegt, wird der Security Manager aufgerufen, von den Java Basisklassen (`java.net` oder `java.io`), um diese Aktion zu bestätigen. Falls die Aktion vom Security Manager bestätigt und somit genehmigt wird, kann die Aktion ausgeführt werden. Sonst wird eine Sicherheitsausnahme geworfen.

Security Manager **Java Core Klassen**

Bemerkung - Falls man mit native Methoden arbeitet, sollten Sie alle native Programmteile private machen.

1.2.6. Selbsttestaufgaben

Die Selbsttestaufgaben zu diesem Lernmodul " Einführung - Übersicht über die Java Security" enthält drei Fragen, mit denen geprüft wird, ob Sie den Stoff verstanden haben. Die Lösungshinweise und die Musterlösung helfen Ihnen, falls Sie nicht alleine zurecht kommen.

1. Welche Rolle spielen digitale Signaturen in einer guten Sicherheitspraxis?

a) digitale Signaturen stellen unterschiedliche Autorisierungen für Zugriff und Aktivitäten zur Verfügung
b) digitale Signaturen stellen Prüfungsmöglichkeiten zur Verfügung
c) digitale Signaturen identifizieren Programmquellen.

2. Welche Sicherheitspraktik stellt die Legitimation eines Knoten oder Benutzers fest, bevor diesem der Zugriff auf einen Rechner oder verlangte Information zur Verfügung gestellt wird?

a) Autorisierung
b) Ressourcenkontrolle und Sicherheitsbehälter
c) Identifikation und Authentifizierung
d) Überprüfung (Auditing)

3. Welche der folgenden Beschreibungen passt am besten zur Funktion des Byte Code Verifiers?

a) Herunterladen von Programmcode und erzwingen der Namensraumbegrenzungen
b) bewilligen von Aktionen, die vom Applet verlangt werden
c) überprüfen, ob der Applet Code den Java Sprachspezifikationen entspricht und dass keine Java Sprachregeln oder Namensraumrestriktionen, Stapel Überläufe oder Unterläufe oder illegale Datentypenkonversionen auftreten.

1.2.7. Musterlösungen

1. Die dritte der möglichen Antworten ist korrekt:

a) Autorisierung wird mittels Zugriffskontrollisten (Access Control Lists, ACLs), dem Security Manager und ab JDK 1.2 den geschützten Domänen (protected domains) implementiert.
b) Prüfungsmöglichkeiten (Audits) werden von Security Manager zur Verfügung gestellt
c) Digitale Signaturen identifizieren den Programmcode, nicht den Benutzer. Auf der Ebene der Browser hängen die Java Browser vom Betriebsystem ab. Dieses muss die Benutzer authentifiieren und dann digitale Signaturen verwenden, um diese Benutzer zu schützen.

2. Die dritte der möglichen Antworten ist korrekt:

a) Autorisierung ist der Prozess, mit dessen Hilfe festgestellt wird, welche Aktivitäten einem authentifizierten Benutzer erlaubt sind.
b) Ressourcenkontrolle und Sicherheitsbehälter werden eingesetzt, um einen autorisierten Benutzer in eine Umgebung einzubetten, in der ihm klar definierte Ressourcen zur Verfügung stehen.
c) Das Java Security Paket stellt die Werkzeuge `keytool` und `jarsigner` zur Verfügung, mit deren Hilfe digitale Signaturen und Zertifikate für den Identifikations- und Authentisierungs- Prozess verwaltet werden können.
d) Überprüfungsmöglichkeiten helfen dem Systemadministrator Probleme zu isolieren und zu beheben, falls eine Abnormalität oder ein Verstoss gegen gewilligte Rechnerbenutzung eintreten sollte.

3) Die dritte der möglichen Antworten ist korrekt:

a) Der Class Loader ist verantwortlich für das Herunterladen des Codes und erzwingt Namensraumgrenzen
b) Falls ein Applet versucht Aktionen auszuführen, die "ausserhalb" des Sandkastens sind, wird der Security Manager von den Java Core Klassen (Basisklassen: `java.io`, `java.net`) aufgerufen, um diese Aktionen bewilligen zu lassen.
c) Der Byte Code Verifier wird vom Applet Class Loader aufgerufen bevor der Applet Code ausgeführt wird.

1.2.8. Zusammenfassung - Übersicht über Java Security

In diesem Modul sollten Sie gelernt haben, was Rechnersicherheit ist und um was sich gute Rechnersicherheitspraktiken bemühen:

- Identifikation und Authentifizierung / Beglaubigung
- Autorisierung
- Ressourcenkontrolle und Sicherheitsbehälter
- Vertraulichkeit und Integrität
- Unleugbarkeit (engl. non-repudiation)
- Prüfbarkeit (engl. Auditing)

Java Sicherheitslösungen wurden diskutiert, inklusive dem Sandkastenmodell und seinem Einsatz der drei Säulen:

- Byte Code Verifier
- Class Loader
- Security Manager

1.3. Modul 2 - JVM und der Verifikationsprozess

1.3.1. Einführung

Dieses Modul bespricht Grundlagen der Java Virtual Machine (JVM) und des Byte Code Verifiers.

Nach dem Durcharbeiten dieser Studieneinheit sollten Sie in der Lage sein:

- Byte Codes zu definieren
- die Prinzipien zu beschreiben, nach denen die statische Integrität des Byte Codes überprüft wird.
- die Checks, welche zur Laufzeit ausgeführt werden können (und müssen) zu umschreiben
- zu erklären, warum Typenprüfung grundlegend für die Java Umgebung ist

---

Referenzen - Die folgenden Referenzen enthalten zusätzliche Details zu den Themen in diesem Modul:

- Details des Verifikationsprozesses und des Byte Code Verfiers, also der "Low Level Security in Java", finden Sie unter http://www.javasoft.com/sfaq/verifier.html oder http://java.sun.com/sfaq/verifier.html
- Details zum Kimera Forschungsprojekt finden Sie unter "Kimera, A Java System Architecture (A System Architecture for Networked Computers" http://kimera.cs.washington.edu/index.html
- Informationen über das class Format der Java Virtual Machine und den Verifikationsprozess finden Sie unter Tim Lindholm & Frank Yellin (1996) "The Java Virtual Machine Specification" http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html
- Eine Diskussion der Typensicherheitsaspekte in Java finden Sie im Buch von Gary McGraw & Edward Felton (1996) "Java Security - Hostile Applets, Holes and Antidotes" John Wiley & Sons, inc.

1.3.2. Java Virtual Machine

In ihrem Buch ***The Java Virtual Machine Specification,*** definieren Yellin und Lindholm die Java Virtual Machine als einen abstrakten (imaginären) Rechner mit einem Instruktionsset. Eine Implementation der JVM emuliert den Instruktionsset in Software auf einem echten Rechner. Implementierungen existieren für Win32, MacOS, Solaris und viele weitere Architekturen. Es sind auch C++/C Implementation erhältlich, die auf den gewünschten Zielrechner portiert werden können.

Die Implementierung einer Programmiersprache mittels einer virtuellen Maschine ist nicht neu: eines der bekanntesten Beispiele war die P-Code Maschine für UCSD PASCAL.

Die JVM ist die Java Technologiekomponente, welche es erlaubt, Java Lösungen auf mehreren Plattformen anzubieten. JVM zeichnet sich durch folgende Charakteristiken aus:

- sie versteht nur das `class` Dateiformat
  Die JVM weiss nichts über die Java Programmiersprache. Das Dateiformat wird in Kapitel 4 der ***The Java Virtual Machine Specification*** spezifiziert. Der Code in der `class` Datei umfasst JVM Instruktionen, Byte Code genannt. Byte Codes sind ein Zwischen-, plattformunabhängiges Format. Byte Codes werden vom Java Compiler `javac` oder andern Compilern erzeugt.
- es werden strikte Format- und strukturelle Beschränkungen auferlegt
  Einschränkungen betreffend der `class` Datei umfassen statische Einschränkungen betreffend der Instruktionen im Code Array, einem Array von Byte Codes für die Klassendatei. Zusätzlich müssen strukturelle Beschränkungen (Beschränkungen in Relation zu JVM Instruktionen) beachtet werden.
- arbeitet mit primitiven Werten und Referenzwerten
  Primitive Werte und Referenzwerte sind zwei Arten von Werten, welche in Variablen gespeichert werden können. Sie werden in Java als Argumente, Rückgabewerte oder Arbeitsvariablen verwendet.
- unterstützt die Ausführung mehrerer Threads
  Die JVM kann nebenläufig mehrere Threads, jeden mit eigenem Programmzähler, Register und Stapel (Stack).

1.3.3. Java Interpreter

In ***The Java Virtual Machine Specification*** stellt eine Art Hardware Spezifikation dar auf der alle Java Programme lauffähig sein müssen. Der Java Interpreter für ein bestimmtes Hardwaresystem garantiert, dass das Programm auf dieser Hardware ausführbar ist.

Das Class Dateiformat für den Programmcode der Virtuellen Maschine besteht aus kompaktem, effizientem Bytecode. Jeder Java Interpreter muss in der Lage sein, Programme in Class Dateien, welche der ***The Java Virtual Machine Specification*** entsprechen, auszuführen.

Um übersetzten Programmcode auszuführen, werden drei Schritte durchlaufen:

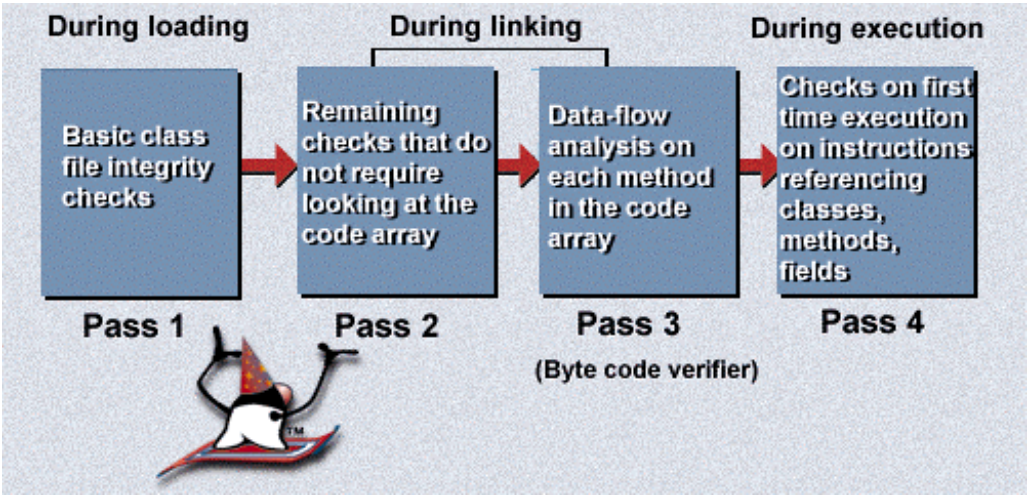
1. Laden des Bytecodes -
   Diese Aufgabe wird durch Class Loader ausgeführt.
2. Verifizieren des Bytecodes -
   Diese Aufgabe wird durch den Bytecode Verifier ausgeführt.
3. Ausführen des Bytecodes -
   Diese Aufgabe wird durch den Runtime Interpreter ausgeführt.

In diesem Modul befassen wir uns mit dem Bytecode Verifier und dem Verifikationsprozess.

1.3.4. Class Datei Verfier

Der Java Compiler (`javac` aus dem JDK) sollte eigentlich nur Class Dateien generieren, welche allen Anforderungen genügen, welche in der Spezifikation der JVM angegeben sind. Allerdings hat die JVM keine Garantie, dass jede Datei, die geladen wird, auch von JDK Compiler generiert wurde oder korrekt formatiert ist. Dies hat aber grosse Auswirkungen auf die Java Sicherheit. Nachdem der Bytecode geladen wurde, aber bevor der Code ausgeführt wird, muss eine Verifikation durchgeführt werden, mit der sichergestellt wird, dass der Class Datei vertraut werden kann. Sonst könnte es passieren, dass fehlerhafter und unfreundlicher Programmcode vom Java Interpreter ausgeführt werden sollte.

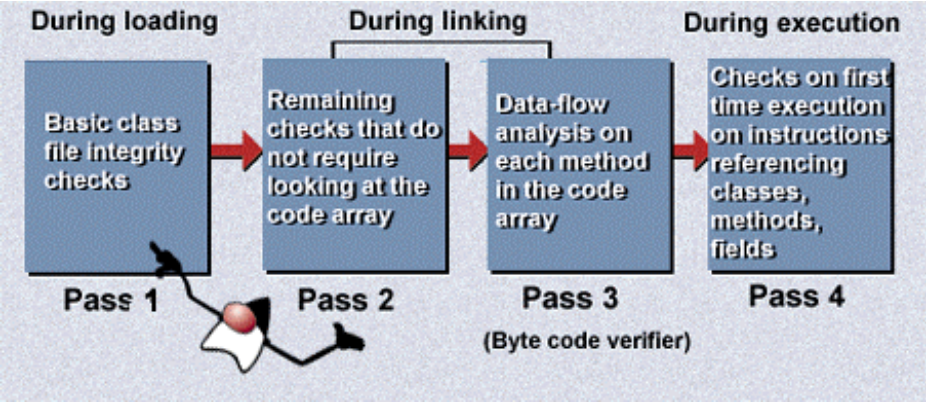
Für diese Verifikation ist der Class File Verifier zuständig. Die Verifikation besteht aus vier Durchgängen. Hier die offizielle Abbildung von Sun:



Den dritten Durchlauf bezeichnet man offiziell als den Byte Code Verifier.
Alle Schritte tragen dazu bei, sicherzustellen, dass der Programmcode der JVM Spezifikation entspricht und die Systemintegrität nicht verletzt wird.

Hinweis
Alle Class Dateien, welche von irgend einem Class Loader geladen werden, ausser dem Sysrem Class Loader, durchlaufen den Verifikationsprozess.

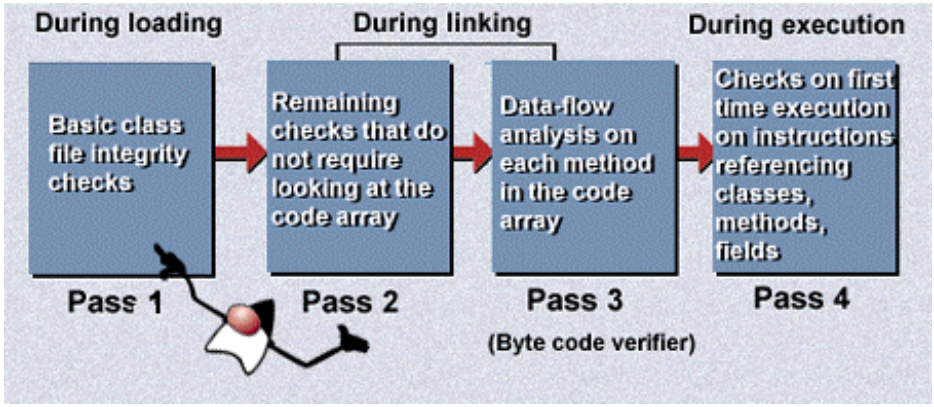
In Durchlauf 1 wird das grundlegende Format der Class Datei überprüft. Dies geschieht beim Lesen in den Java Interpreter. Mit diesen grundlegenden Tests kann folgendes verifiziert werden:



- 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[1] enthält keine unbekannten Informationen

[1] Konstanten Pool, Constant Pool
In Section 3.5.5 der The Java Virtual Machine Specification steht: "A constant pool is a per-class or per-interface runtime representation of the `constant_pool` table in a Java class file. It contains several kinds of constants, ranging from numeric literals known at compile time to method and field references that must be resolved at run time."
Der Konstanten Pool dient einer ähnlichen Funktion wie die Symboltabelle einer konventionellen Programmiersprache, obschon der Constant Pool zusätzliche Informationen enthält.

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:

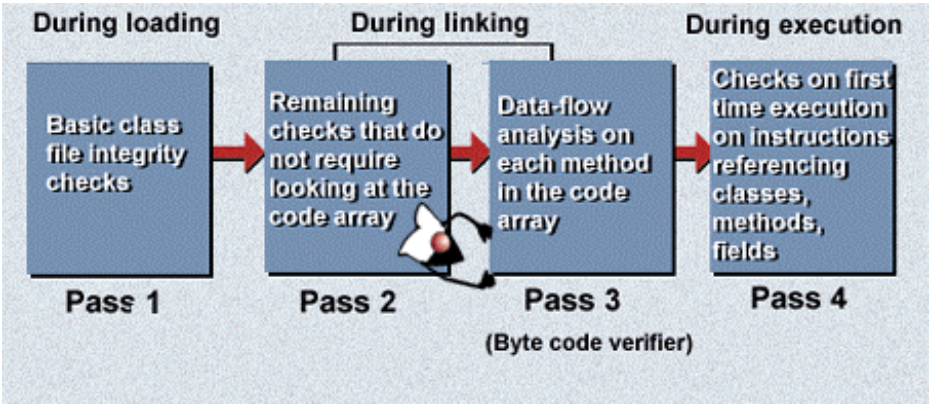
- finale Klassen keine Unterklassen haben
- alle Klassen eine Oberklasse haben, ausser der Klasse Object
- der Konstanten Pool korrekt formatiert ist
- alle Datenfelder und Methodenaufrufe im Konstanten Pool gültige Namen, Typen und Beschreibungen haben.

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

Wie bereits erwähnt, wird Pass 3 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 eine Datenflussanalyse durchgeführt, für jedes Methode (dieser Pass 3 wird im Detail in der JVM Spezifikation beschrieben, inklusive allen Details).

Diese Analyse garantiert insbesondere, dass unabhängig davon, wie man bei der Programmausführung zu dieser Anweisung gelangt, folgende Aussagen zutreffen:

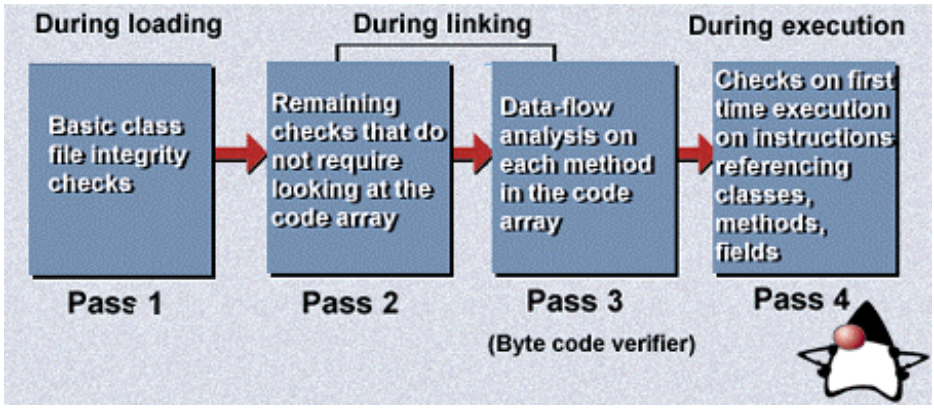
- dass der Operanden Stack immer die korrekte Grösse hat und gleiche Objekttypen betrifft.
- auf keine lokalen Variablen zugegriffen wird, ausser es ist bekannt, dass sie einen Wert des korrekten Datentyps enthalten.
- Methoden werden mit korrekten Argumenten aufgerufen.
- Datenfelder Zuweisungen sind typengerecht
- alle OpCodes verwenden korrekte Argumenttypen



Die Spezifikation der Virtuellen Maschine lässt es dem Implementierer frei, einzelne Prüfungen in Phase 3 oder 4 durchzuführen. Je nach Implementation treten also Unterschiede auf.

Bei der Sun JVM Implementation wird beim ersten Mal, bei dem eine Anweisung eine 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.

Hinweis
Falls dieser Durchgang 4 fehl schlägt, wird eine Instanz der Subklasse `LinkageError` geworfen.

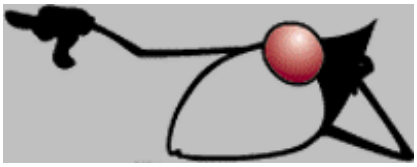
In der Sun Implementation werden Anweisungen nach der Verifikation durch alternative Darstellungen der Anweisungen ersetzt. Diese alternative Darstellung zeigt, dass die Verifikation erfolgreich durchlaufen wurde.

Hier ein einfaches Beipiel: `new` wird ersetzt durch `new quick`. Damit ist auch klar, dass bei zukünftigen Aufrufen dieser geprüften Methoden der Aufruf schneller ist, da verschiedene Tests nicht mehr nötig sind. Diese alternativen Darstellungen dürfen natürlich in der Class Datei nicht vorkommen.

Falls der Verifier alle vier Schritte erfolgreich abschliessen kann, dann sind Sie sicher, dass:

- die Class Dateien gemäss der Spezifikation der JVM aufgebaut sind.
- keine Zugriffsrechtsverletzungen geschehen.
- kein Overflow oder Underflow geschieht.
- alle Parameter der Operatoren vom korrekten Datentyp sind.
- keine illegalen Datenkonversionen geschehen
- Zugriffe auf Objektattribute korrekt sind
- lokale Variablen beim ersten Zugriff darauf initialisiert sind.

1.3.5. Typensicherheit

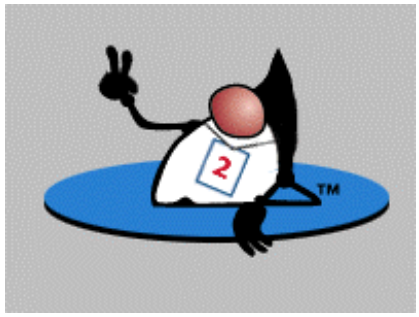


Der Byte Code Verifier sorgt für Typensicherheit. Die Java Programmiersprache ist eine strikt typengeprüfte Programmiersprache. Das heisst, dass in Java nicht auf unerlaubte Speicherbereiche zugegriffen werden kann. Sie können einen Zugriff auf oder Operationen mit Daten nicht ausführen, falls diese Operationen für diesen Datentyp nicht vorgesehen ist.

Im Buch von McGraw und Felten ***"Java Security - Hostile Applets, Holes, and Antidotes"*** wird beschrieben, wie wichtig Typenprüfung in Java ist und wie dieses Konzept mit der Java Security verknüpft ist.

Objekte werden als im Java Code als Speicherblöcke betrachtet, Referenzen auf Objekte sind dabei Zeiger, Pointer auf Speicheradressen. Um die Typenprüfung zu erleichtern, werden Referenzen mit einem Typenkennzeichen versehen, aus dem hervor geht, von welchen Typus ein Objekt ist, auf das die Referenz zeigt.

Jedes Java Objekt wird im Rechnerspeicher mit einem Klassenlabel versehen. Die Typenprüfung kann mit Hilfe des Labels, des Tags, auf zwei Wegen geprüft werden (die in Java verwendete Technik wird als *static type checking* bezeichnet):



- **dynamisch** - während der Laufzeit (*dynamisch*) wird der Klassentag jeweils vor der Ausführung irgend einer Operation auf oder mit dem Objekt überprüft, um sicherzustellen, dass dies gestattet ist. Diese Überprüfung dauert Zeit und verlangsamt die Programmausführung
- **statisch** - vor der Programmausführung analysiert der Verifier die obigen Überprüfungen. Falls der Verifier feststellt, dass eine Überprüfung immer zu einem positiven Ergebnis führt, also keine Probleme auftauchen, dann eliminiert der Verifier den Test aus dem Programm.

Der Byte Code Verifier führt diese statische Typenprüfung durch. Damit werden die meisten Prüfoperationen zur Laufzeit überflüssig und das Programm wird typensicher und schneller.

Der Java Byte Code Verifier ist ein wichtiges Element der Java Security, weil er die Typenprüfung durchführt. Alle Klassendateien, ausser den Systemdateien, welche auf der JVM ausgeführt werden sollen, müssen durch den Verifier durch und seinen Anforderungen genügen. Sicherheitsvorschriften, die mit dem Compiler nicht überprüft werden können, müssen vom Verifier geprüft werden, damit Java seinen hohen Sicherheitsanforderungen gerecht werden kann. Sonst besteht die Gefahr, dass der Code zu tief ins System eindringen kann und Probleme verursachen könnte.

1.3.6. Fehler im Verifier früherer JDKs

Da der Verifier eine kritische Komponente ist, wurde er auch sehr gründlich getestet und untersucht. Und wahrscheinlich auch deswegen entdeckte man in früheren Releases vom JDK mehrere Fehler. Für diese Tests wurde ein spezielles Team von Sicherheitsexperten zusammengestellt und Laborversuche durchgeführt. Der Erfolg ist, dass bisher zwar einige Personen behaupteten Sicherheitslöcher nutzen zu können; diese konnten aber nicht nachgewiesen oder reproduziert werden.

Wir wollen zwei mögliche Attacken herausnehmen und etwas genauer ansehen:

- Type Confusion
- Verifier und Class Loader umgehen

Bemerkung
Sie können sich bei Sicherheitsfragen nicht nur auf den Compiler und den Programmcode stützen. Ein Hacker oder jemand, der Ihr System attackieren will, wird unter Umständen einen eigenen Compiler entwickeln. Daher ist die Laufzeitüberprüfung wesentlich.

1.3.7. Typenvortäuschung

Bei der Typentäuschung versucht der Angreifer den Byte Code Verifier betreffend des Objekttyps zu täuschen. Die Täuschung besteht darin, dass der Angreifer zwei Referenzen auf das selbe Objekt erstellt aber jeweils mit einem anderen Label versieht. Das kann, clever ausgenutzt, bereits zu einer vollen Systemübernahme durch den Angreifer führen.

Da die statische Typenprüfung sehr komplex ist, besteht die Gefahr, dass etwas übersehen wird. Dies würde oder könnte zu einem Sicherheitsloch führen. Ein solches Sicherheitsloch wurde durch Drew Dean, Dirk Balfanz und Edward Felten im Juni 1996 entdeckt, speziell im Netscape Explorer 3.05. Dort wurden Arrays falsch implementiert und konnten zu Systemübernahmen durch Angreifer führen. In Version 3.06 wurde das Problem behoben.

1.3.8. Verifier und Class Loader Attacken

Im März 1996 entdeckte das Safe Internet Programming (SIP) Team an der Princeton University einen Fehler im Byte Code Verifier und eine Schwachstelle im Class Loader. Zusammen konnten diese zwei Fehler zu einer vollen Systemübernahme durch Angreifer führen. Auch diese Fehler zeigten sich zuerst im Netscape Browser und wurden umgehend behoben (Netscape 2.01 auf Netscape 2.02). Zudem wurde ein Sicherheitsalarm ausgelöst, ein sogenannter CERT Alert.

Was war das Problem?
Um das Sicherheitsloch zu verstehen, benötigt man vertiefte Kenntnisse über die Art und Weise, wie Klassen gebildet, geladen und in der Virtuellen Maschine gespeichert werden.

Die Java Programme bestehen aus Klassen, welche in unterschiedlichen Dateien abgespeichert werden, Klassen können durch andere Klassen mit Hilfe ihres Namens referenziert werden. Sie werden nach Bedarf geladen.

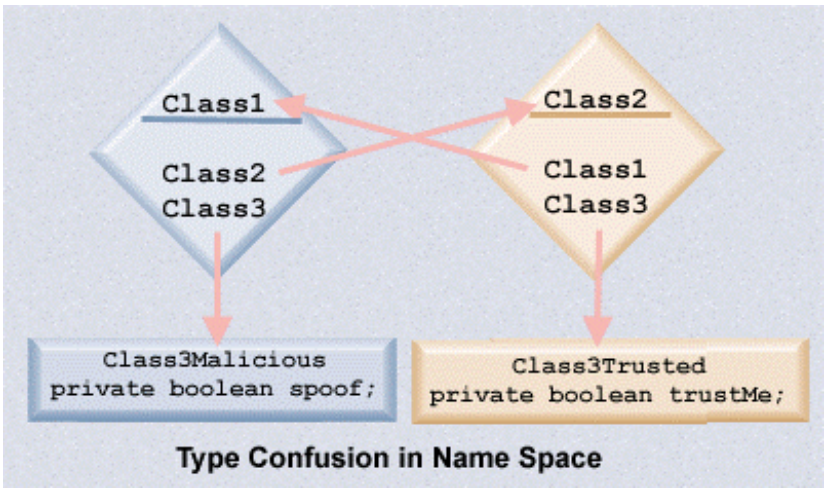
Referenzen müssen zum Laden aufgelöst werden, speziell falls sich die Class Datei irgendwo im Internet befindet.

In Java spricht man von *Name Spaces*, Namensräumen. Diese sind Teil des Adressraumes der Programme. In Java wird pro geladene Klasse ein Namensraum definiert. Lokale Klassen und Applet Klassen verwenden ebenfalls getrennte Namensräume. Jeder Namensraum wird von einem Objekt, dem Class Loader verwaltet. Die Virtual Machine verwendet einen Class Loader, um Klassennamen zu übersetzen und die Klassen zu laden.

Nach der Übersetzung des Namens wird der Byte Code für die referenzierte Klasse geladen (lokal oder über das Netzwerk). Die Methode `defineClass()` des Class Loaders konvertiert den Byte Code in eine verwendbare Klasse und liefert diese Klasse an die JVM.

Applets dürfen keine Class Loader kreieren. Der Konstruktor für den Class Loader (die man erweitern müsste, falls man einen eigenen Class Loader definieren möchte) prüft mit dem Security Manager, ob es sich um ein Applet handelt und würde in diesem Fall abbrechen, mit einer SecurityException.

Das SIP Team stellte fest, dass die Schwachstelle des Verifiers das Kreieren eines eigenen Class Loaders zulies. Dieser eigene Class Loader benutzte natürlich nicht den Class Loader Konstruktor und konnte damit den Security Manager umgehen. Anschliessend benutzte der fremde Class Loader eine Typenkonfusion , um das System zu übernehmen.



Falls es jemandem gelingt, einen unfreundlichen Class Loader einzusetzen und die `defineClass` Methode auszuführen mit inkompatiblen Namensräumen, dann könnte es einem gelingen das System zu übernehmen:

eine Klasse `Class1` wird instanziert und verweist auf `Class2` (und `Class3`), welche ihrerseits auf `Class1` und `Class3` verweist. Durch Typeconfusion könnten wir erreichen, dass `Class2` auf eine korrekte, `Class1` aber auf eine boshafte Klasse `Class3` zeigt.

Der Verifier prüft die Klasse `Class2` und stellt fest, dass alles okay ist. Nun übergeben wir aber zur Laufzeit die falsche Klasse "`Class3boshaft`" oder `Class3Malicious` (gemäss obigem Diagramm aus der Java Dokumentation) an die Klasse Class2. Natürlich schleusen wir jede Menge gefährlichen Code ein und übernehmen die Maschine.

Das problem bestand in Netscape, einmal mehr in Netscape! Natürlich wurde es sofort behoben. Der erste Fix war bedenklich einfach: falls nicht der echte ClassLoader geladen wurde, also ein anderer Konstruktor ausgeführt wurde, musste abgebrochen werden, weil ein Flag nicht gesetzt war. In neueren Versionen wurde das Problem gründlicher behoben.

1.3.9. Kimera Projekt

Das Kimera Projekt wurde von mehreren Dokoranden an der Washington University gestartet. Leider sind die meisten Teile des projekts gesperrt. Aber Sie finden einige Hinweise zum Projekt unter :

http://www.cs.washington.edu/homes/egs/papers/egs-seajug/ppframe.htm

Der eigentliche Web Site wäre (ist aber oft nicht online):

http://kimera.cs.washington.edu/

Kimera ist ein Java Sicherheitsprojekt. Im projekt wurden verschiedene JVMs untersucht, mit dem Ziel eine sichere zu entwickeln, für den Einsatz in verteilten, hochzuverlässigen Systemen.

Teil des projekts war die Entwicklung eines Byte Code Verifiers. Das Kimera Team nimmt natürlich für sich in Anspruch, einen sichereren Byte Code Verifier entwickelt zu haben, als andere Stellen. Die Implementation des Verifiers geschah entlang den Vorgaben in der JVM Spezifikation.

Zum Testen konnte man eine URL angeben, eine Class Datei, die man gerne testen wollte, und bekam in einem Applet die Antwort.

Neben Kimera gibt es andere Byte Code Verifier Projekte. Sun selber unterhält ein Labor, in dem laufend geprüft wird. Sun und andere haben spezielle Web Sites, ab denen man beliebig viele Informationen zum Thema erhält.

Das Kimera Team informierte Sun 1996, dass im damaligen Verifier ein Fehler enthalten war. Dieser ermöglichte es Applets, im schlimmsten Fall fremde Systeme zu übernehmen. Auch dieser Fehler wurde umgehend behoben.

Eine der wichtigsten Referenzen zum Thema ist:

http://java.sun.com/security

dort findet man auch Details zum Thema Kimera Bug (1997) und neueren Bugs, beispielsweise:

"November 29, 2000 - Potential Security Issue in Class Loading
Through its own research and rigorous testing, Sun has discovered a potential security issue in the Java™ Runtime Environment that affects both Java Development Kit (JDK™) 1.1.x and Java 2 Standard Edition SDK™ v 1.2.x releases. The issue poses a possible security risk by allowing an untrusted class to call into a disallowed class under certain circumstances."

Aber auch hier handelt es sich um ein recht altes JDK!

1.3.10. Was wurde unternommen

Forscher und Experten aus unterschiedlichen Fachgebieten haben versucht, die Spezifikation von JVM zu formlisieren. Damit würden weniger Freiräume entstehen, die unterschiedlichen Implementationen damit auch vergleichbarer und kontrollierbarer.

Zudem wurden Kompatibilitätstests entwickelt, Sie haben sicher vom Streit Microsoft versus Sun betreffend Java Kompatibilität gehört.

Eine grössere Anstrengung unternahm ein Gremium um Schlumberger, Sun und anderen (unter anderem wegen der JavaCard). In deren Auftrag entwickelte die Firma Computational Logic Inc (CLI) ein formales Modell, wenigstens für einen Teil der Spezifikation.

Dieses wurde aber seit 1997 nicht weiter nachgeführt. Es ist von folgender Adresse herunter ladbar:

http://www.cli.com/software/djvm/index.html

Das Modell diente aber bereits als Basis für Verbesserungen des Byte Code Verifiers.

Wie gut ist die JVM, die Sie einsetzen?
Auf Grund der Bemerkungen oben, sollte klar sein, dass diese Frage sehr wesentlich ist. Sun hat aus diesem Grunde ein Test Kit entwickelt, mit dem die Java Konformität geprüft werden kann. Leider hat sie dafür auch noch viel Geld verlangt. Aber das Kit kann gratis heruntergeladen werden, wenigstens konnte ich dies, obschon etwas anderes auf den Web Seiten steht.

1.3.10.1. Formale Modelle

Das formale Modell einer JVM wurde in ACL2 beschrieben, einer Sprache, die sich an Common Lisp anlehnt.

1.3.10.2. Java Kompatibilitätstest

Damit Benutzer und Entwickler sicher sind, dass ihre Java Systeme der Spezifikation entsprechen, wurde das Java Compatibility Kit (JCK) von Sun entwickelt.

Man kann damit weder die Robustheit des Codes, noch die Qualität der Programme testen. Aber die Konformität des Systems mit der Spezifikation wird überprüft.

1.3.10.3. Java Kompatibilität versus 100% Pure Java

Sun vergibt zwei Labels:
Java Compatibility
und
100% Pure Java

Die Java Kompatibilität hängt mit der Implementation von Java zusammen, also Compiler, Laufzeitsysteme, APIs usw.

100% Pure Java wird an Software Entwickler vergeben und besagt lediglich, dass Applikationen zu 100% in Java geschrieben wurden.

1.3.10.4. Java Compatibility Kit

Das JCK besteht aus Tests und Werkzeugen, mit dem lizenzierte Tester die Kompatibilität testen können. Das Kit besteht aus:

- Java Test Harness - ein Window Programm, mit dem die Tests administriert werden können.
- Tests - Programmcode der Tests; übersetzte Tests; erwartete Resultate

1.3.10.4.1. Java Test Harness

Mit dieser Oberfläche können Sie:

- den Programmcode der Tests starten
- positive oder negative Tests laufen lassen
- Tests nach Stichworten auswählen
- Test nachts laufen lassen

1.3.10.4.2. Kompatibilitätseinschränkungen

Damit die Tests auch fair ablaufen, mussten sie nach bestimmten Regeln entwickelt werden:

- Tests mussten unabhängig vom Tester implementiert werden.
- Tests müssen Applets und Applikationen testen können.
- Tests müssen folgende Aspekte berücksichtigen:
  - Dateisystem
  - Netzwerk
  - Sicherheitseinschränkungen
  - Anzeige / Display / Grafik
  - andere Bibliotheken.

Die Tests selbst verwenden das Dateisystem oder andere Systemressourcen nicht. Damit sind die Tests systemunabhängig.

1.3.10.4.3. Test Komplexität

Die Tests müssen nach bestimmten Kriterien gebaut werden:

- was will man testen
- wie will man testen
  - selbsttestend : der Test bestimmt selber ob er erfolgreich war oder nicht

Die Spezifikation der Tests ist oft nicht so ganz klar. Damit sind auch die Ergebnisse unklar.

1.3.10.4.4. Testwerkzeuge

JCK verwendet verschiedene Testtools, mit deren Hilfe die Tests durchgeführt werden:

- zum übersetzen der Tests
- um die Ergebnisse anzuschauen
- um Reports zu generieren
- um zu analysieren, ob der gesamte Code getestet wurde (Code Coverage)

1.3.10.4.5. Testprozesskomplexität

Weil Java auf vielen Plattformen lauffähig ist, müssen die Tests auch verteilte Aspekte testen:

- Client Server
- vernetzt
- verteilt
- interaktive Anwendungen

Einige Tests beispielsweise der grafischen Oberflächen (AWT) bedingen die Generierung von Events, welche sicher plattformabhängig sind.

1.3.10.4.6. Zu testende Komponenten

Die Tests werden in drei Gruppen eingeteilt:

- Compiler Tests - prüfen, ob die *The Java Language Specification* eingehalten wird.
- JVM Tests - entspricht die VM der *The Java Virtual Machine Specification*
- API Tests - entsprechen die APIs den vorhandenen APIs

Sicherheitstests werden in allen drei Fällen durchgeführt.

1.3.10.4.7. Compilertests

Compiler sind relativ einfach zu testen. Es werden etwa 3000 Tests durchgeführt, fast alle automatisiert. Tests umfassen auch Konzepte wie RMI, für Remote Methode Invocation, also die Programmierung verteilter Systeme.

1.3.10.4.8. Virtual Machines

Die VM Komponente ist viel schwieriger zu testen als die Sprache! Für die Tests werden auch Dateien verwendet, die nicht von einem Compiler stammen, um unfreundliche Übernahmen zu simulieren. Dabei werden programme assembliert und deassembliert. Insgesamt gibt es über 1700 Tests, über 90% aller Aussagen in der Spezifikation werden getestet.

1.3.10.4.9. API Libraries

Neue APIs zu testen ist etwas schwierig. Standardtests sind kaum definierbar. Was man prüfen kann ist, ob die Signatur der Aufrufe gemäss Spezifikation erfolgt.

- es werden etwa 200 Tests automatisch durchgeführt
- einige Tests werden von Hand, grafisch geführt durchgeführt
- die Automatisierung weiterer Tests wird untersucht, erscheint aber schwierig

1.3.10.4.10. Erweiterungen des JCK

Sun arbeitet an weiteren Verbesserungen der Tests.

- bessere Tests der Applet Umgebung
- bessere Tests der grafischen Elemente
- insgesamt breitere Abdeckung der Tests

1.3.11. Indirekte Ausführung

Welchen Einfluss hat die Einführung des Just in Time (JIT) Compilers auf die Sicherheit in Java gehabt? Welchen Einfluss haben die neuen Java Chips?

Im Buch ***Advanced Techniques for Java Developers*** der beiden Autoren Berg und Fritzinger wird erläutert, dass die indirekte Ausführung des Java Codes wichtiger sei als die Typensicherheit. Indirekte Ausführung bedeutet, dass Java Code innerhalb der JVM aufbereitet wird und erst dann auf der CPU ausgeführt wird (also indirekt, nicht direkt auf der CPU). Diese indirekte Ausführung erhöht die Sicherheit des Java Codes beträchtlich. Applets werden nie direkt ausgeführt, da sie keinen Zugriff auf Betriebsystem und tiefere Schichten haben (ausser Sie geben dem Applet die Rechte dazu). Die JVM ist also der Türwärter für Applets.

JIT Compiler übersetzen den Byte Code in Maschinencode, allerdings können Sie selbst mit Applets keinen Maschinencode einschmuggeln. Der Class Verifikationsprozess und der Byte Code Verifier bleiben aktiv!

Bemerkung
Achten Sie darauf, dass Sie nur JIT Compiler von renomierten Firmen einsetzen, genau wie bei den JVMs. Damit stellen Sie sicher, dass Sie keine Attacken einschlesen oder Löcher offen lassen.

Im Falle der Java Chips ist der Byte Code der Maschinencode und Java wird direkt ausgeführt. Die JVM läuft auf dem Chip und dem Betriebsystem. Da die JVM weiter bestehen bleibt, haben Java Chips keinen Einfluss auf die Sicherheit von Java Applikationen. Sun arbeitet an einem modernen Java Chip (MaJiC), der im Momement (Feb 2001) getestet wird.

1.3.12. Übungen - mit Bytecode arbeiten

Lernziel dieser Übung ist es, dass Sie sich mit Bytecode vertraut machen und sich über die Bedeutung des Byte Verifiers klar werden. Dazu analysieren Sie Programme, die bereits fertig geschrieben sind, oder Ausgaben von Programmen. Damit Sie etwas über die Hintergründe und die Bedeutung des Kimera Projekts erfahren, sind auch einige kurze Auszüge aus Arbeiten zu diesem Projekt wiedergegeben.

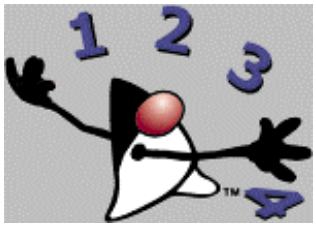
Die Übung besteht aus drei Teilen:

- Kimera Verifier und Disassembler
- Eine Klasse, deren Verifikation nicht gelingt
- Das Jasmin - Java Interface

1.3.12.1. Der Kimera Verifier und Disassembler

Diese Übung besteht aus den folgenden 7 Teilen:

1. schreiben eines `HelloWorld` Programms
2. lesen einer Kurzübersicht über den Kimera Verifier
3. eine typische Ausgabe des Kimera Verifiers anschauen
4. Vergleich der Kimera mit der Standardverifikation
5. lesen einer Kurzübersicht über den Kimera Disassembler
6. studieren der Kimera Disassembler Ausgabe
7. vergleich der Kimera Ausgabe mit `javap`



1.3.12.1.1. Schritt 1 - HelloWorld.java

Schauen Sie sich die folgende Version des Hello World Programms an. Wir werden den Rest mit diesem Programm arbeiten.

```
public class HelloWorld {
        public static void main(String arg[]){
                System.out.println("Hello World!");
                System.out.println("Hello World!");
                System.out.println("Hello World!");
                System.out.println("Code");
        }
}
```

Bemerkung
Sie können, falls der Server wieder einmal läuft, Ihre eigenen Class Dateien an Kimera senden, interaktiv am Web :
http://kimera.cs.washington.edu/disassembler.html

1.3.12.1.2. Schritt 2 - Der Kimera Verifier

Der folgende Artikel stammt von den Projektleitern des Kimera Projekts. Lesen Sie ihn und versuchen Sie zu verstehen, welches die wichtigsten Ziele des Kimera Verifiers sind.

1.3.12.1.3. Kimera Bytecode Verification - Literaturauszug

At the University of Washington, we have built a clean-room implementation of a robust and secure verifier as part of our security architecture. We have empirically demonstrated that our implementation is safer than the current state of the art commercial verifiers.

There are numerous advantages to our approach to separate verification in the Kimera firewall. Namely,

- A bug in the runtime cannot be exploited to weaken the verifier.
- There are no implied semantics as a result of hidden interfaces.
- The amount of code is small and can be tested for coverage and correctness.

The small size and clean structure of our verifier enables us to have a clear mapping between the Java virtual machine specification and the Java safety axioms implemented in our verifier. Whereas typical commercial implementations of integrated Java virtual machines may distribute security checks thoughout the verification, compilation, interpretation, and runtime stages, our verifier performs complete bytecode verification in a single component. Further, the implementation is annotated with safety axioms that we have distilled from the Java virtual machine specification. Hence, it is possible to easily check the verifier for correctness. We believe that our verifier embodies the most complete set of safety axioms about Java byte code.

The implementation of our verifier facilitates code coverage testing. It is trivial to figure out the set of axioms that have not been tested by a given test suite, which can be used to guide the development and construction of test cases. The fine grain nature of the axioms further simplifies this task.
The actual implementation of our verifier closely follows the Java virtual machine specification. Four passes are performed during verification. The first pass verifies that the class file is consistent and that its structures are sound. The second pass determines the instruction boundaries of the bytecode and verifies that all branches are safe. The third and most complicated pass involves data flow analysis and verifies that the operands in the Java virtual machine are used in accordance with their types. The first three passes together ensure that the submitted class file does not perform any unsafe operations in isolation. The final and fourth pass ensures that the combination of the submitted class with the rest of the Java virtual machine does not break any global type safety or interface restrictions.
Kimera verification service is accessible from the web. You can run the verifier on a class you would like to examine by providing a URL to the class (the default is a simple hello world applet that passes verification). The verifier that you are about to run does not perform phase four link checks. The submitted class files are logged.

Sean McDirmid & Emin Gün Sirer & Brian Bershad
Project Kimera
Department of Computer Science and Engineering

1.3.12.1.4. Schritt 3 - Die Kimera Ausgabe

Das obige Programm wurde an den Kimera Verifier übergeben und das Ergebnis in einer Datei gespeichert. Hier die Ausgabe des Kimera Verifiers:

Verifying: ...HelloWorld.class

Kimera verifier says:

Class is safe HelloWorld

You can get a dissassembly of the submitted class file here

- User time (seconds): 0.00
- System time (seconds): 0.00
- Percent of CPU this job got: 78%
- Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.01
- Average shared text size (kbytes): 0
- Average unshared data size (kbytes): 0
- Average stack size (kbytes): 0
- Average total size (kbytes): 0
- Maximum resident set size (kbytes): 1600
- Average resident set size (kbytes): 0
- Major (requiring I/O) page faults: 0
- Minor (reclaiming a frame) page faults: 42
- Voluntary context switches: 0
- Involuntary context switches: 1
- Swaps: 0
- File system inputs: 0
- File system outputs: 0
- Socket messages sent: 0
- Socket messages received: 0
- Signals delivered: 0
- Page size (bytes): 8192
- Exit status: 1

Emin Gün Sirer & Sean McDirmid & Brian Bershad
Project Kimera
© 1997, University of Washington

1.3.12.1.5. Schritt 4 - Aufruf des Java Verifiers

In Version 1.1x konnte man mit dem Aufruf

```
java -verify -cp . HelloWorld
```

den Verifier explizit aufrufen. In neueren Versionen wurde das Flag durch -Xverify ersetzt. Allerdings habe ich dazu keine Sun Dokumentation mehr gefunden. Ein Test des Switches zeigte keinerlei Wirkungen bei JDK1.3x. Allerdings generiert der Schalter auch keine Fehlermeldung. Etwas scheint also doch zu passieren. Wir werden beim Testen eines fehlerhaften Applets darauf zurück kommen.

1.3.12.1.6. Schritt 5 - Der Kimera Disassembler Literaturauszug

Disassembly is the process of examining the contents of a Java class. Whereas a reverse compiler will reconstruct the source code of the applet, a disassembler will print the low-level Java byte codes as they are shipped on the network. A disassembly service is useful for debugging, auditing and post-mortem crash and security violation analysis. In the course of testing our verifier implementation with automatically generated class files, it was determined that a tool was needed to examine the class files that trigger security flaws in Java implementations.
While Sun provides an option to `javap` to disassemble class files, we found that its output was found to be unsuitable for parsing and assembly. This need for an assembler-compatible disassembler prompted us to write our own. A Java disassembler has been developed that generates jasmin compatible output from Java class files.
The disassembler has been used for debugging as well as post-mortem analysis of JVM security attacks. The disassembler output has hypertext links in order to aid easy code browsing. The output can be directly fed into the jasmin assembler.

Kimera disassembly service is accessible from the Web. You can run the disassembler on a class you would like to examine by providing a URL to the class you are interested in (the default is a sample applet from JavaSoft). The submitted class files are logged.

Emin Gün Sirer & Brian Bershad
Project Kimera
Department of Computer Science and Engineering

1.3.12.1.7. Schritt 6 - Ausgabe des Kimera Disassemblers

Schauen wir uns die vom Kimera Disassembler generierte Ausgabe für das HelloWorld Programm genauer an:

```
; Produced by the Kimera disassembler
from the University of Washington
; Wed Nov 12 07:19:00 PST 1997
.class  public synchronized HelloWorld
.super java/lang/Object

.method public ()V
  .limit locals 1
  .limit stack 1

  aload_0
  invokenonvirtual java/lang/Object.()V
  return
.end method

.method public static main([Ljava/lang/String;)V
  .limit locals 1
  .limit stack 2

  getstatic       java/lang/System/out Ljava/io/PrintStream;
  ldc             "Hello World!"
  invokevirtual   java/io/PrintStream.println(Ljava/lang/String;)V
  getstatic       java/lang/System/out Ljava/io/PrintStream;
  ldc             "Hello World!"
  invokevirtual   java/io/PrintStream.println(Ljava/lang/String;)V
  getstatic       java/lang/System/out Ljava/io/PrintStream;
  ldc             "Hello World!"
  invokevirtual   java/io/PrintStream.println(Ljava/lang/String;)V
  getstatic       java/lang/System/out Ljava/io/PrintStream;
  ldc             "Code"
  invokevirtual   java/io/PrintStream.println(Ljava/lang/String;)V
  return
.end method
```

1.3.12.1.8. Schritt 7 - Vergleich der Ausgabe von javap und des Kimera Disassemblers

Mit Hilfe des `javap` Programms (im JDK bin Verzeichnis) können Sie ebenfalls disassemblieren.

Mit dem Befehl

```
javap -c -l -classpath . -verbose HelloWorld > DisassembledHelloWorld.txt
```

wird die Ausgabe in eine Datei umgeleitet. Der Inhalt dieser Datei ist:

```
Compiled from HelloWorld.java
public class HelloWorld extends java.lang.Object {
    public HelloWorld();
      /* Stack=1, Locals=1, Args_size=1 */
    public static void main(java.lang.String[]);
      /* Stack=2, Locals=1, Args_size=1 */
}

Method HelloWorld()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Line numbers for method HelloWorld()
   line 1: 0

Method void main(java.lang.String[])
   0 getstatic #2 <Field java.io.PrintStream out>
   3 ldc #3 <String "Hello World!">
   5 invokevirtual #4 <Method void println(java.lang.String)>
   8 getstatic #2 <Field java.io.PrintStream out>
  11 ldc #3 <String "Hello World!">
  13 invokevirtual #4 <Method void println(java.lang.String)>
  16 getstatic #2 <Field java.io.PrintStream out>
  19 ldc #3 <String "Hello World!">
  21 invokevirtual #4 <Method void println(java.lang.String)>
  24 getstatic #2 <Field java.io.PrintStream out>
  27 ldc #5 <String "Code">
  29 invokevirtual #4 <Method void println(java.lang.String)>
  32 return

Line numbers for method void main(java.lang.String[])
   line 3: 0
   line 4: 8
   line 5: 16
   line 6: 24
   line 7: 32
```

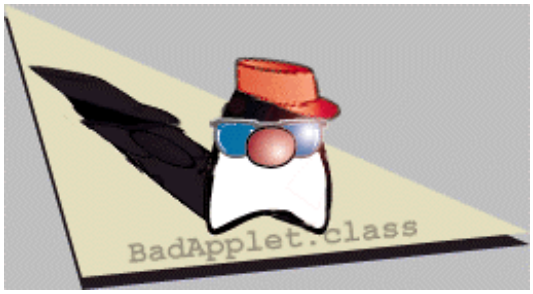
Vergleichen Sie diese Ausgabe mit jener des Kimera Projekts.

1.3.12.2. Konstruktion einer Klasse, die im Verifier verworfen wird

Eine Klasse `badapplet.class` wurde an das Kimera Projekt zur Prüfung geschickt. Die URL des Applets war :

http://kimera.cs.washington.edu/applets/badapplet.class

Schauen wir uns einmal die Ausgaben des Kimera Disassemblers und des Kimera Byte Code Verifiers genauer an.



1.3.12.2.1. Kimera Byte Code Verifiers Ausgabe

Als Eingabe diente:

```
http://kimera.cs.washington.edu/applets/badapplet.class.
```

Als Ausgabe erhielt man:

```
Verifying: http://kimera.cs.washington.edu/applets/badapplet.class

Kimera verifier says:

Security flaw: DFA_STACK_WRONG_TYPE 608 in class file badappletSecurity
flaw: DFA_STACK_WRONG_TYPE 608 in class file badapplet (init ()V @ pc =
2)Security flaw: DFA_STACK_WRONG_TYPE 608 in class file badapplet (init ()V
@ pc = 2)
Security flaw: DFA_STACK_WRONG_TYPE 608 in class file badapplet (init ()V @
pc = 2)

     Must pop proper types from method stack.
     Must pop proper types from method stack.
     Must pop proper types from method stack.
     Must pop proper types from method stack.

     You can get a dissassembly of the submitted class file here
        o User time (seconds): 0.00
        o System time (seconds): 0.00
        o Percent of CPU this job got: 72%
        o Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.01
        o Average shared text size (kbytes): 0
        o Average unshared data size (kbytes): 0
        o Average stack size (kbytes): 0
        o Average total size (kbytes): 0
        o Maximum resident set size (kbytes): 1472
        o Average resident set size (kbytes): 0
        o Major (requiring I/O) page faults: 0
        o Minor (reclaiming a frame) page faults: 40
        o Voluntary context switches: 0
        o Involuntary context switches: 0
        o Swaps: 0
        o File system inputs: 0
        o File system outputs: 0
        o Socket messages sent: 0
        o Socket messages received: 0
        o Signals delivered: 0
        o Page size (bytes): 8192
        o Exit status: 0

            ------------------------------------------------
               Emin Gün Sirer & Sean McDirmid & Brian Bershad
                              Project Kimera
                     © 1997, University of Washington
```

1.3.12.2.2. Kimera Disassembler Ausgabe

Jetzt schauen wir uns die Ausgabe des Kimera Disassemblers an, für das gleiche Applet wie oben:

```
; Produced by the Kimera disassembler from the University of Washington
; Mon Dec  1 00:29:50 PST 1997
.class  public synchronized badapplet
.super java/applet/Applet

.method public init()V
        .limit locals 1
        .limit stack 20

        iconst_1
        aconst_null
        iadd
    getstatic       java/lang/System/out Ljava/io/PrintStream;
    ldc             "Hello world"
    invokevirtual   java/io/PrintStream.println(Ljava/lang/String;)V
    return
.end method

.method public ()V
    .limit locals 1
    .limit stack 1

    aload_0
    invokenonvirtual java/applet/Applet.()V
    return
.end method

        ----------------------------------------------------
          Emin Gün Sirer & Sean McDirmid & Brian Bershad
                  Project Kimera
```

Unser manipuliertes Applet versucht sich in Pointer Arithmetik: zu einer Objektreferenz wird ein Integer addiert (**rot** markiert oben).

Ein solches Applet darf nicht ausgeführt werden, weil es im schlimmsten Fall versuchen würde, die Sandbox zu verlassen. Die JVM darf solche Applets auf keinen Fall ausführen. Der Byte Code Verifier zeigt klar an, dass er das Sicherheitsloch entdeckt hat.

Ziel dieser Übung war es, Sie auf mögliche Sicherheitsprobleme und deren (automatische) Prüfung durch Byte Code Verifier hinzuweisen.

Ab JDK 1.2 haben Sie mit `SecureClassLoader` und der `AccessController` Klasse, sowie dem neuen Protection Domains Modell noch mehr Möglichkeiten, Ihr Sicherheitsmodell den Gegebenheiten optimal anzupassen.

1.3.12.3. Jasmin - ein Java Assembler Interface

Weiter vorne haben wir bereits gesagt, dass der Kimera Disassembler Jasmin kompatiblen Code generieren kann.

Jasmin finden Sie entweder auf dem Kurs Server oder unter

http://www.mrl.nyu.edu/meyer/jasmin/

Die eigentlich Homepage für Jasmin (gemäss dem Buch ***Java Virtual Machine*** von Jon Meyer, Troy Downing; O'Reilly Associates March 1997) ist

http://cat.nyu.edu/meyer/jasmin

Das `jasmin` Programm akzeptiert ASCII Beschreibungen der Java Klassen in Assembler ähnlicher Syntax und konvertiert diese in binären Bytecode, der dann in die JVM geladen werden kann.

Das Hello World Beispiel von Jon Meyer in `jasmin` :

```
;     Jon Meyer (meyer@cs.nyu.edu), Troy Downing (downing@nyu.edu)
;     http://mrl.nyu.edu/
;     February 1997

;     Feel free to use and modify this code. Please leave
;     this header in any derived works.

;     Copyright (c) 1997 Jon Meyer and Troy Downing All rights reserved

.class public HelloWorld
.super java/lang/Object

; specify the constructor method for the Example class

.method public <init>()V
     ; just call Object's constructor
     aload_0
     invokespecial java/lang/Object/<init>()V
     return
.end method

; specify the "main" method - this prints "Hello World"

.method public static main([Ljava/lang/String;)V
     ; set limits used by this method
     .limit stack 2

     ; Push the output stream and the string "Hello World" onto the stack,
     ; then invoke the println method:
     getstatic java/lang/System/out Ljava/io/PrintStream;
     ldc "Hello World!"
     invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
     return
.end method
```

und das dazugehörende Java Programm:

```
*     Jon Meyer (meyer@cs.nyu.edu), Troy Downing (downing@nyu.edu)
      http://mrl.nyu.edu/
      February 1997

      Feel free to use and modify this code. Please leave
      this header in any derived works.

      Copyright (c) 1997 Jon Meyer and Troy Downing All rights reserved
*/

public class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

Bemerkung
Neben Kimera und Jasmin gibt es weitere Assembler und Disassembler:

- Alternative zu Jasmin : http://siesta.cs.wustl.edu/~djh4/
- Alternative zum Kimera Disassembler: http://home.cc.umanitoba.ca/~umsilve1/djava

1.3.13. Quiz

Frage 1 :
Was ist Bytecode?[2]
a) JVM Befehle, in einem plattformunabhängigen Zwischenformat
b) Werte, in denen Man Variablen speichern kann
c) mehrere ausgeführte Threads

Frage 2 :
Der Bytecode wird nach dem Laden und vor dem Ausführen überprüft, damit [3]
a) sichergestellt ist, dass der Code von einem Java Compiler in korrektem Format generiert wurde.
b) um zu vermeiden, dass Sicherheitslöcher beim Ausführen im Java Interpreter entstehen können.
c) beide a) und b)

Frage 3 :
Der Bytecode Verifier führt eine statische Typenprüfung durch:[4]
a) beim Ausführen, durch die Überprüfung aller Klassenbeschreibungen
b) durch Analyse vor der Ausführung des Programms
c) durch Überprüfen der Klassen vo allen Zugriffen

Frage 4 :
Welche der folgenden Aufgaben ist nicht Aufgabe des Java Interpreters?[5]
a) Code laden
b) Code verifizieren
c) die Class Datei generieren
d) den Code ausführen

Frage 5 :
Welche der folgenden Aussagen ist korrekt bzgl dem Verifier?[6]
a) zur Laufzeit wird jeder Class Tag analysiert und geprüft, ob der Zugriff erlaubt ist
b) bevor das Programm ausgeführt wird, prüft der Verifier die Classes und eliminiert Tests, die immer korrekt sind.

[2] a) ist korrekt : Bytecode wird mt javac generiert (aus Java Dateien)
[3] c) ist korrekt : ausser dem System Class Loader werden alle Class Dateien überprüft
[4] b) ist korrekt : falls der Verifier feststellen kann, dass eine Prüfung immer korrekt abläuft, wird der Test eliminiert (im Cache der Class Datei, vor der Ausführung).
[5] c) ist korrekt : die Class Datei wird vom Compiler generiert.
[6] a) ist korrekt : die Tests werden bereits vor der Ausführung gemacht

1.3.14. Zusammenfassung des Moduls

In diesem Modul haben Sie gelernt:

- wie der Byte Code verifiziert wird
  (mit dem Byte Code Verifier)
- welche Sicherheitstests erst zur Laufzeit ausgeführt werden (können)
- warum die Typensicherheit fundamental für die Java Sicherheit ist
- wie in Forschungsprojekten Löcher im Verifier von früheren Versionen gefunden werden konnten.

1.4. Module 3 - Class Loaders

Nach der Verifikation müssen wir uns mit dem Laden der Klassen befassen.

1.4.1. Einleitung

Der Class Loader ist ein weiteres wichtiges Hilfsmittel zur Sicherstellung der Java Sicherheit.

1.4.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein,

1. wissen, welche Funktion der Class Loader hat und welche Rolle er zur Sicherstellung der Java Sicherheit hat.
2. den Begriff Namespace / Namensraum zu erläutern und erklären zu können, welche Rolle Namensräume bei der Festlegung von Java Sicherheiten spielen.
3. zu unterscheiden zwischen einem Applet Class Loader und dem System Class Loader
4. die Bedeutung der folgenden Konzepte zu erklären:
   - java.security.SecureClassLoader und
   - java.net.URLClassLoader,
   welche in JDK 1.2 eingeführt wurden.
5. einfache Class Loader zu schreiben, bei denen spezielle Sicherheitsaspekte berücksichtigt werden.

1.4.1.2. Referenzen

Die folgenden Dokumentationen enthalten weitergehende Informationen zum Thema:

- `docs/api/java.lang.ClassLoader.html` und `docs/api/java.security`
- `SecureClassLoader.html` der JDK 1.2 Dokumentation
- im Java Tutorial von Sun Microsystems, welches Sie auch herunter laden können, gibt es einen speziellen Java Security "Trail".

1.4.2. Wann werden Klassen geladen?

Die Frage ist: wann wird eine Klasse in die JVM geladen oder was verursacht das Laden einer Klasse in die JVM?

Es gibt drei Gründe, die zum Laden einer Klasse in die JVM führen:

1. Sie schreiben Programmcode wie
   `MeineKlasse meinObjekt = new MeineKlasse();`
   um eine Instanz zu kreieren. Natürlich muss eine Klasse erst geladen werden bevor sie in der JVM ausgeführt werden kann. Einige JVMs laden die Klassen sofort, einige erst zum Zeitpunkt, zu dem sie benötigt werden.

2. Sie schreiben Code, der explizit das Laden einer Klasse zur Folge hat, selbst wenn diese Klasse eventuell noch nicht benötigt wird:

   ```
   Class c = Class.forName("meineKlassen.MeineKlasse");
   ```

3. als dritte Möglichkeit könnten Sie einen Class Loader starten. Sie würden damit einfach von Hand das auslösen, was normalerweise im Hintergrund geschieht:

   ```
   ClassLoader cl = this.getClass().getClassLoader();

   // oder Sie kreieren eine Class Loader explizit:
   // ClassLoader cl = new MyClassLoader;

   Class c = cl.loadClass("meineKlassen.MeineKlasse");
   ```

1.4.3. Wie wird eine Klasse geladen?

Eine Klasse wird mit Hilfe der Methode `loadClass()` nach Bedarf geladen. Somit stelt sich die Frage:
- wie funktioniert diese Methode?
- was muss gemacht werden, um die Methode in einem eigenen Class Loader zu implementieren?

Die Methode verwendet eine Zeichenkette als Parameter. Diese beschreibt die Klasse plus Package, also den voll qualifizierten Namen der Klasse.

Die Methode ist überladen, besitzt also unterschiedliche Formen:
```
protected loadClass(String name, boolean resolve)
```

Falls der Wert von resolve `true` ist, ruft die Methode `loadClass()` die Methode `resolveClass()` auf.

Unter *resolving* versteht man den Prozess, bei dem man die Oberklassen und allfällige Interfaces der Klasse sucht und 'auflöst'. Die VM kann eine Klasse auch laden ohne aufzulösen.

Folgende Tätigkeiten werden zum Laden einer Klasse in die JVM durchgeführt:
- validieren des Klassennamens
- überprüfen des Cache
- überprüfen der Oberklasse des Class Loaders
- kreieren einer Referenz auf eine Datei oder ein Verzeichnis
- laden des Bytecodes
- installieren der Klasse
- Klasse in den Cache einfügen
- auflösen der Klasse

Die Reihenfolge, in der die Operationen ausgeführt werden, ist dabei eher unwichtig. Falls beispielsweise der Cache erst später abgefragt wird, stört das kaum.

1.4.3.1. Validieren des Klassennamens

Um Klassen zu laden, muss als erstes der Name der Klasse und der Package Name validiert werden. In alten Versionen des JDKs fand man Fehler, speziell Sicherheitslöcher, in diesem Prozessschritt.

Als Beispiel müssen viele Class Loader zum Laden einer Klasse die Punkte aus der voll qualifizierten Schreibweise einer Klasse (<package>.<klasse>) umwandeln in das Trennzeichen des Dateisystems (beispielsweise '\' oder '/'). Falls führende Punkte erlaubt sind, führt dies bei der Auflösung oft zu Problemen. Auch mehrfache Punkte können zu Problemen führen.

Falls ein illegaler Klassen- oder Package Namen angegeben wird, sollte dies zu einer Ausnahme führen (`ClassNotFoundException`).

1.4.3.2. Überprüfen des Cache

Nach dem Validieren des Klassen- oder Package- Namens sollte der Class Loader prüfen, ob die Klasse bereits geladen wurde. Dies geschieht mit Hilfe des Class Cache. Falls die Klasse dort gefunden werden kann, sollte einfach die existierende Klasse zurück gegeben werden (bzw. eine Referenz darauf). Das bedingt, dass eine Liste der geladenen Klassen verwaltet werden muss, der Cache der geladenen Klassen. Falls eine Klasse nicht durch den Cache geladen wird, könnte es vorkommen, dass ein und dieselbe Klasse mehrfach geladen wird. Dies würde dazu führen, dass auch statische Members der Klasse mehrfach geladen werden könnten und Speicherplatz verschwendet würde. Gravierender sind die Zugriffsprobleme, die entstehen könnten. Beispielsweise könnte es passieren, dass eine Instanz einer Klasse entdecken könnte, dass sie nicht mehr auf ihre privaten Members zugreifen könnte, weil diese durch eine neue Kopie der Klase überschrieben wurden.

In JDK 1.2 wird dieser Cache mit einem Konstrukt gemäss `java.util.Map`, ein Interface, spezifiziert. Auch die Hashtabellen : `java.util.Hashtable` implementiert das Interface. Hashtabellen sind auch in älteren JDK Versionen bereits vorhanden.

Vor JDK1.1 musste der Class Loader eine Cache Funktion zur Verfügung stellen. Dies konnte unter Umständen zu Problemen führen, speziell falls man vergass diese Funktion zu implementieren. Deswegen wurde ab JDK 1.1 ein Cache im System Class Loader eingeführt. Damit werden alle Klassen, die in die JVM geladen werden sollen, automatisch auch in den Cache des System Class Loaders eingetragen. Dieser Cache ist allen Class Loadern gemeinsam.

Damit das Konzept der Namensräume (Name Spaces) verhebt, sollte auf Stufe Class Loader ein Cache definiert und implementiert werden. Dies ist ab JDK 1.2 der Fall: dort wird ein Cache pro Class Loader definiert.

Zum Prüfen, ob eine Klasse bereits geladen wurde, steht die Methode `findLoadedClass()` zur Verfügung. Diese Methode sollte jeweils aufgerufen werden, bevor überhaupt auch nur versucht wird, eine neue Klasse zu laden.

Bemerkung
Unterschiedliche Namenräume (Name Spaces, Class Loaders) verbergen keine Klassen vor anderen Class Loaders. Lediglich die Zugriffsrechte der einzelnen Klassen werden damit geregelt.

1.4.3.3. Überprüfen der Oberklasse des Class Loaders

Es kann auch vorkommen, dass die Oberklasse des Class Loaders eine bestimmte Klasse bereits geladen hat. In JDK 1.1 ist die Oberklasse des Class Loaders der System Class Loader. Ab JDK 1.2 bilden die Class Loader eine Baumstruktur mit dem System Class Loader als Wurzel. Dabei besteht die Beziehung zum System Class Loader nicht im Sinne einer Package Hierarchie. Vielmehr handelt es sich um eine Verschachtelung, analog zu AWT. Der Konstruktor des Class Loaders enthält einen Parameter, einen Class Loader. Dieser ist in der Regel der System Class Loader.

Damit wird garantiert, dass als erstes immer der System Class Loader zum Zuge kommt.

Der Class Loader muss auch die Methoden `checkPackageAccess()` und `checkPackageDefinition()` aufrufen. In der Regel benötigt der Class Loader auch Zugriff auf das lokale Dateisystem und die Netzwerk Ressourcen.

1.4.3.4. Kreieren einer Referenz auf eine Datei oder ein Verzeichnis

Falls der Class Loader geprüft hat, dass der Klassennamen legal und korrekt ist und auch geladen werden sollte, muss mit Hilfe des Package Namens eine verzeichnisreferenz aufgebaut werden.

Falls beispielsweise die Klasse `meineUtilities.math.Complex` geladen werden soll, muss zuerst die Klasse `Complex.class` gefunden werden. Diese Datei sollte sich im Pfad `meineUtilities/math` irgendwo unter dem Stammverzeichnis befinden, in dem der Class Loader die Klassen sucht.

Die Verzeichnisstruktur kann sich auf lokales Laufwerk oder auf eine URL, einen Web Server beziehen, auf den mit Hilfe von HTTP zugegriffen wird. Aber es kann sich auch um eine Verzeichnisstruktur innerhalb eines ZIP oder JAR Archives handeln.

1.4.3.5. Laden des Bytecodes

Nachdem der Pfad zur Klassendatei eindeutig festgelegt wurde, muss die Datei als Byte Array geladen werden. Idealerweise definiert man ein Array, in dem die gesamte Datei Platz findet. Falls die Datei aber von einem entfernten Server stammt, ist dies oft nicht möglich, weil die Übertragung unzuverlässig ist. In diesem Fall muss die Datei aus mehreren Teilen zusammengesetzt werden.

1.4.3.6. Installieren der Klasse

Bei gegebenem Byte Array wird die Klasse mit dem Befehl (der Methode des Class Loaders) `defineClass` in der VM installiert.

Die Definition von `public abstract class` **`ClassLoader`** definiert mehrere Methoden zum Installieren einer Klasse.

`protected Class defineClass(byte[] b, int off, int len)`
Deprecated. *Replaced by defineClass(java.lang.String, byte[], int, int)*

`protected Class defineClass (String name, byte[] b, int off, int len)`
konvertiert ein Byte Array in eine Instanz der Klasse `Class`.

`protected Class defineClass(String name, byte[] b, int off, int len,ProtectionDomain protectionDomain)`
konvertiert ein Byte Array in eine Instanz der Klasse `Class` mit einer optionalen Protection Domäne.

Die zweite Methode ist die üblicherweise verwendete Methode. Die erste Methode konnte zu einem Sicherheitsproblem führen: man konnte vorgeben eine andere Klasse zu sein und damit mehr Zugriffsrechte erlangen.

1.4.3.7. Installieren der Klassen

In der Klasse java.security.SecureClassLoader, welche als Basis für eigene Class Loader verwendet werden sollte, wird eine andere Version der Methode defineClass verwendet.

```
protected Class defineClass(String name, byte[] b, int off, int len,
CodeSource cs)
```

konvertiert ein Byte Array in eine Instanz der Klasse `Class`, mit der optionalen Angabe der Quelle des Codes (`CodeSource`).

Diese Klasse wird als Basis für den URLClassLoader verwendet, mit dem typischerweise Klassen über das Internet geladen werden.

1.4.3.8. Klasse im Cache einfügen

Die Methode `defineClass()` liefert als Ergebnis ein `Class` Objekt. Dieses sollte für den weiteren Gebrauch im Cache registriert werden. Der Cache ist dabei in der Regel eine Hashtabelle und als Schlüssel wird der voll qualifizierte Klassennamen verwendet.

Bemerkung
Vor JDK 1.1 musste jeder Class Loader seinen eigenen Cache verwalten. Ab JDK 1.1 verwaltet `defineClass()` die Klasse im System Cache. Ab JDK 1.2 wird der Class Loader Cache automatisch definiert und mittels finalen Methoden unausweichlich fixiert.

1.4.3.9. Die Klasse auflösen

Das zweite Argument der geschützten Methode zum Laden der Klasse

```
protected loadClass(String name, boolean resolve)
```

gibt an, ob die Klasse aufgelöst werden soll. Falls das Argument `true` ist oder die Methode ohne Flag aufgerufen wird (diese ist äquivalent zur obigen mit `true` als zweitem Parameter), dann wird die Klasse aufgelöst. Dies geschieht mit einem Aufruf der Methode

```
resolveClass()
```

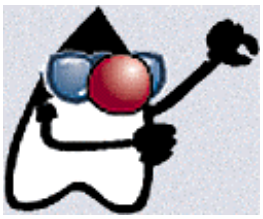
mit der neu geladenen Klasse als Argument.

Bemerkung
Die Auflösung der Klasse geschieht auch, falls die Klasse eine Systemklasse ist.

1.4.4. Class Loaders

In der Regel werden ein oder mehrere Class Loaders bereits vorhanden sein, um Ihre Klassen zu laden. Standardmässig ist der System Class Loader bereits vorhanden. Falls Sie ein Applet ausführen, wird mindestens ein weiterer Class Loader vorhanden sein.

Bemerkung



In Java 1.2 und neuer stehen weitere Class Loader zur Verfügung:

```
java.security.SecureClassLoader
java.net.URLClassLoader
```

werden standardmässig mit dem JDK ausgeliefert. Aber diese werden nur instanziert, falls Sie dies explizit verlangen. In JDK1.1 ist lediglich der `RMIClassLoader` Teil des Core Java APIs.

Der `RMIClassLoader` ist selbst keine Unterklasse des Class Loaders. Intern verwendet er Objekte, welche das Interface `java.rmi.server.LoaderHandler` implementieren.

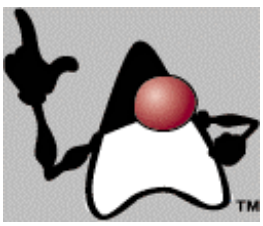
1.4.4.1. Standard Class Loader

Falls Sie ein Applet in einen Browser laden, ist das Laden, also auch die Wahl des Class Loaders, eine interne Angelegenheit des Browsers. Allerdings kann das Applet selbst dann auch diesen Class Loader des Browsers einsetzen. Das hängt unter anderem damit zusammen, dass der Security Manager dem Applet gar nicht die Möglichkeit gibt einen eigenen Class Loader zu definieren. Einzig der System Class Loader kann als Alternative verwendet werden.

Beide, der System Class Loader und der Browser Class Loader erlauben normalerweise das Laden von Klassen aus Archiven (JAR und ZIP). Ab JDK 1.2 ist dies durch den `URLClassLoader` garantiert, selbst wenn die Verbindung über eine HTTP Verbindung geschieht.

Bemerkung

Class Loader sollten auch signierte Archive unterstützen. Diese Fähigkeit wurde allerdings durch viele JDK 1.1 komaptible Browser nicht implementiert. Alle Browser, welche JDK 1.2 kompatibel sind, sollten jedoch diese Fähigkeit besitzen.



Der `URLClassLoader` kann Klassen aus JAR Dateien lesen, welche auf einem Web Server gespeichert sind. Das Protokoll für diese Operation ist in der Klassenbeschreibung für `java.net.JarURLConnection` beschrieben.

Hier ein Beispiel:

falls Klassen aus einem Archiv von der URL http://www.meineArchive.org/classes/utils.jar geladen werden soll, dann muss der URL Class Loader (das Objekt) mit einer URL auf dieses Archiv kreiert werden:

das Argument ist in diesem Fall jar: http://www.meineArchive.org/classes/utils.jar.

1.4.5. Kreieren eigener Class Loader

Falls bereits System und Browser Class Loader existieren, warum sollte man dann überhaupt noch eigene Class Loader kreieren?

Vergessen Sie nicht: einen Class Loader können Sie nur für Applikationen definieren; Applets dürfen dies nicht!

Bemerkung
Ab JDK 1.2 ist es kaum mehr nötig, eigene Class Loader zu kreieren. Einfacher und sicherer ist es, einen Protokoll Handler zu definieren (das werden wir im Teil Java Netzwerk Programmierung sehen) und den URLClassLoader einzusetzen. In älteren JDKs kann es aber durchaus sein, dass Sie mit eigenen Class Loader arbeiten müssen.

In einer Applikation könnte es verschiedene Gründe für einen eigenen Class Loader geben:

1. die Anwendung muss die Klassen ab einem speziellen Server laden und die Definition eines eigenen Protokoll Handlers ist nicht angebracht.
   Dieser Fall könnte auch in JDK 1.2 auftreten. Allerdings werden Sie sehen, dass Protokoll Handler sehr flexibel definiert werden können und somit dieser Fall eher unwahrscheinlich ist.
2. die Anwendung benötigt spezielle Sicherheitsmechanismen. Auch in diesem Fall könnte das Problem mit einem Protokoll Handler gelöst werden.

Achtung
Das Schreiben eines Class Loaders ist eine sicherheitsrelevante Tätigkeit. Sie sollten entsprechend vorsichtig vorgehen, um keine Sicherheitslöcher zu kreieren.

3. Die Applikation benötigt spezielle Caches für remote Klassen.
4. Die Applikation benötigt ein eigenes Class Datei Format. Diese Klassendateien müssen aufbereitet werden bevor sie an die JVM abgegeben werden.

Bemerkung
In der Regel ist die Definition eines eigenen Class Loaders zwar eine nette Übung. Aber sehr empfehlenswert ist es kaum. Die Einführung des `URLClassLoaders` in JDK 1.2 führte zu weiteren Verbesserungen. In der Regel reicht also ein Protokoll Handler.

1.4.6. Laden von Ressourcen

Bereits ab JDK 1.1 wurde der Class Loader auch zum Laden von Ressourcen verwendet. Eine Ressource kann dabei sein:

- ein Applet
- ein Bild (eine .gif Datei)
- ...

Im Prinzip ist eine Ressource eine Sequenz von Bytes, die einem Package zugeordnet ist. Schauen wir uns ein Beispiel an:

```
ClassLoader cl = this.getClass().getClassLoader();
URL url = null;
if (cl != null) {
      url = cl.getResource("myImage.gif");
}
else {
      url = ClassLoader.getSystemResource("meinImage.gif");
}
```

dabei setzen wir voraus, dass das Handling des Inhalts durch entsprechende Klassen sichergestellt ist.

Alternativ könnten Sie eine Ressource auch als Stream behandeln:

```
ClassLoader cl = this.getClass().getClassLoader();
InputStream in = null;
if (cl != null) {
      in = cl.getResourceAsStream("meinImage.gif");
}
else {
      in = ClassLoader.getSystemResourceAsStream("meinImage.gif");
}
```

Bemerkung
Die Methoden `getSystemResource` und `getSystemResourceAsStream` kann man ersetzen durch die `getResource` Methoden, welche in der Klasse `java.lang.Class` definiert sind.

Falls Sie einen eigenen Class Loader definieren, sollten Sie auch analog zu oben, eigene Ressourcen Methoden implementieren, um den Byte Code zu laden.

Falls Sie zum Laden ein eigenes Protokoll benötigen, sollten Sie einen Protokoll Handler und einen dazu passenden Content Handler definieren.

1.4.7. Class Loaders in JDK 1.2

Ab JDK 1.2 steht Ihnen der `java.security.SecureClassLoader` zur Verfügung. Er kann auch leicht erweitert werden und als Basis für eigene Class Loader dienen.

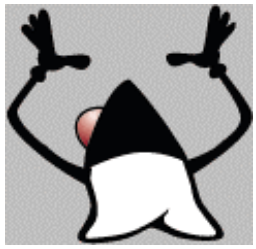
Alles was Sie zu tun haben ist, passende Unterklassen zu definieren und entsprechende Konstruktoren zu implementieren.

Ab JDK 1.2 bilden die Class Loader eine Hierarchie. In älteren Versionen wurde zuerst immer mit dem System Class Loader versucht die Klassen zu laden. In JDK 1.2 und neueren Versionen kann pro Class Loader also ein "Parent" definiert werden. Zum Laden wird immer zuerst die Oberklasse eingesetzt. Falls sie in der Lage ist, die Klasse zu laden, ist alles okay. Sonst wird versucht die Klasse anders zu laden.

Falls keine Oberklasse angegeben werden kann, dann wird der System Class Loader verwendet.

Die Methode `findLocalClass` wird immer dann eingesetzt, falls die Klasse weder im Cache gefunden werden kann, noch der Parent Class Loader die Klasse kennt.

Damit wird auch eine bestimmte Sicherheit eingebaut.



Auch ab JDK1.2 wird in `java.net` eine Unterklasse des Secure Class Loaders definiert, die Klasse `URLClassLoader`. Dieser Class Loader kann Klassen ab URLs laden. In der Regel wird man ihn also für die eigene Definition lokaler Class Loader nicht benötigen.

1.4.8. Class Loader Sicherheitsfragen

Die Sicherheitsaspekte beim Implementieren von Class Loadern kann man in zwei Kategorien einteilen:
- wesentliche Merkmale aller Class Loaders
- zusätzliche Merkmale spezieller Class Loaders

Die folgenden vier Punkte sollten Sie in jeder der beiden Situationen beachten:

- überprüfen der Package und Klassen-Namen.
  Besonders aufpassen muss man, dass nicht eine Klasse aus *irgend* einem Verzeichnis geladen werden kann. Illegale Zeichenketten, die an den Class Loader als Klassennamen übergeben werden, sollten sofort zurück gewiesen werden und eine `ClassNotFoundException` werfen.
- vor dem Laden einer Klasse sollte der Cache überprüft werden, um sicherzustellen, dass die Klasse nur einmal geladen wird, speziell bei entfernten Klassen.
- der Verifier muss ausgeführt werden. Dies ist nicht so schwierig, da er Teil der Methode `defineClass` ist und kaum umgangen werden kann.

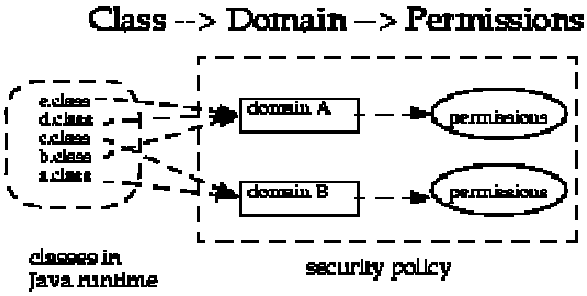
Zusätzlich zu den grundsätzlichen Sicherheitsüberlegungen können Sie beliebig viele eigene Abfragen einbauen. Beispiele hierfür könnten sein:
- Definition bevorzugter Server
- Einschränkung möglicher Server

Dies sind die zwei am meisten implementierten Features:
- überprüfen der Zugriffsrechte in der Package Hierarchie
- Behandlung von digitalen Signaturen

Falls in einer Hierarchie der Zugriff geschützt ist, muss jeder Versuch, eine Klasse aus dieser Hierarchie zu laden, verworfen werden. Dies ist beispielsweise im Appletviewer und im HotJava Browser von Sun implementiert.

Sie können die Sicherheit vergrösseren, wenn Sie Ihre Klassen mit einer Signatur versehen. In diesem Fall müssen Sie vor dem Laden der Klasse auch noch die Signatur analysieren und prüfen. Die Class Loader Klasse verfügt über eine Methode `protected final setSigners()` mit der eine Klasse markiert werden kann. Die Methode enthält zwei Argumente, ein Objekt vom Typ Class und ein Array von Objekt Instanzen.:
`protected void` **setSigners**`(`**Class** `c,` **Object**`[] signers)`

Diese Objekt Instanzen sind in der Regel Unterklassen der `java.security.Signer`, welche ihrerseits eine Unterklasse von `java.security.Identity` ist. Die `Identity` Klasse liefert Informationen über den Namen der signierenden Identität und auf das Zertifikat dieser Identität.



Protection Domain

Die Klasse
`public class` **`ProtectionDomain`** `extends` **`Object`**

fasst das Sicherheitsverhalten bestimmter Klassen zusammen. Alle Klassen in einem Protection Domain besitzen die selben Rechte.

Neben den Rechten besteht eine Domäne aus einer `CodeSource`:
`public class` **`CodeSource`** `extends` **`Object`** `implements` **`Serializable`**

Diese Klasse erweitert das Konzept der Codebase: neben der Lokation (URL) werden auch noch die Zertifikate, mit denen der signierte Code aus dieser URL signiert wird, verwaltet.

Jede Klasse gehört nur zu einer Protection Domäne.

Der Class Loader, mit dem eine Klasse geladen wird, wird Teil des Name Space dieser Klasse. Das besagt insbesondere, dass falls zwei unterschiedliche Class Loader die selbe Klasse laden, diese zwei (identischen) Klassen nicht miteinander kommunizieren können, da sie zu unterschiedlichen Namensräumen gehören.

Dieses Verghalten ist insbesondere bei Applets wichtig. In diesem Fall kann ein Applet von unterschiedlichen Web Sites stammen. Der Browser verwendet pro Codebase einen eigenen Class Loader, kreiert also einen eigenen Namensraum.

1.4.9. Übungen

Lernziele für diese Übungen sind:

- sich mit dem inneren Aufbau eines Class Loaders vertraut zu machen.
- einen Class Loader zu basteln, mit dem zusätzliche Security Checks möglich sind.
- Zusammenhänge zwischen Class Loading und Ressource Loading kennen zu lernen.

1.4.9.1. Mit dem Cache arbeiten

Zuerst schauen wir uns die grundlegende Struktur eines Class Loaders an, also ein Raster, kein fertiges Programm.

```
public class LoadClass extends ClassLoader {
protected Class loadClass(String name, boolean resolve) {
      Class c;
      SecurityManager sm = System.getSecurityManager();

      // Step 1 -- ist die Klasse bereits geladen
      c = findLoadedClass(name);
      if (c != null)
            return c;

      // Step 2 -- ist der Zugriff auf diese Klasse möglich
      if (sm != null) {
            int i = name.lastIndexOf('.');
            if (i >= 0) sm.checkPackageAccess(name.substring(0, i));
      }

      // Step 3 -- prüfe die Systemklasse
      try {
          c = findSystemClass(name);
          return c;
      } catch (ClassNotFoundException cnfe) {
          // keine Systemklasse
      }

      // Step 4 -- können wir die Klasse definieren
      if (sm != null) {
            int i = name.lastIndexOf('.');
            if (i >= 0) sm.checkPackageDefinition(name.substring(0, i));
      }

      // Step 5 -- lesen der Class Datei
      byte data[] = lookupData(name);

      // Step 4 and 5 -- definieren der Klasse
      //           übergeben der Daten an den Byte Code Verifier
      c = defineClass(name, data, 0, data.length);

      // Step 6 -- auflösen der internen Referenzen
      if (resolve)
            resolveClass(c);

      return c;
}
      byte [] lookupData(String n) {
          return new byte[1];
      }
```

```
}
```

Und nun schauen wir uns einen Java Class Loader an, mit dem Klassen geladen werden können:

```
package securityclassloader;

import java.util.*;
import java.net.*;
import java.io.*;

public class MeinClassLoader extends ClassLoader {
    // JDK 1.0.x : jeder ClassLoader besitzt einen eigenen Cache
    // private Hashtable cache = new Hashtable();
    private final File myBase;
    private static final boolean verbose = Boolean.getBoolean("Debug");

    public MeinClassLoader(File myBase) {
      if (verbose) { log("MeinClassLoader: constructor, mybase = " +
          myBase); }
      this.myBase = myBase;
    }

    private byte[] loadClassData(String name)
      throws ClassNotFoundException {
      byte [] buffer;

      if (verbose) { log("MeinClassLoader: loadClassData, name=" +
          name); }
      File source = null;
      InputStream s = null;
      try {
          source = new File(myBase, name);
          s = new FileInputStream(source);
          buffer = new byte [(int)(source.length())];
          s.read(buffer);
      }
      catch (Exception e) {
          System.out.println("Probleme mit " + source);
          throw new ClassNotFoundException(name);
      }
      return buffer;
    }

    public synchronized Class loadClass(String name, boolean resolve)
      throws ClassNotFoundException {
      if (verbose) { log("MeinClassLoader: loadClass, name=" + name +
          " resolve=" + resolve); }
      if (name.charAt(0) == '.') {
          throw new SecurityException("'Absolute' package name");
      }
      if (name.indexOf("..") != -1) {
          throw new SecurityException("Illegal package name");
      }

      // jdk1.0.x
      // Class c = (Class)(cache.get(name));
      Class c = (findLoadedClass(name));

      if (c == null) {
          try {
```

```
            c = findSystemClass(name);
          }
          catch (Exception e) {
            try {
                String path = name.replace('.', File.separatorChar) +
                    ".class";
                byte data[] = loadClassData(path);
                c = defineClass(name, data, 0, data.length);
                // JDK 1.1+ defineClass cached die Klassen
                // JDK 1.0.x - benutzen des eigenen Caches
                // cache.put(name, c);
            }
            catch (ClassFormatError ex) {
                throw new ClassNotFoundException(" Class ist: " + name +
                  " Problem ist: " + ex.getMessage());
            }
          }
      }
      if (resolve) {
          resolveClass(c);
      }
      return c;
    }

    private void log(String s) {
      System.err.println(s);
    }

    public static void main(String args[]) throws Throwable {
      if (args.length < 2) {
          System.out.println("MeinClassLoader : fehlendes Argument ");
          System.exit(1);
      }
      File base = new File(args[0]);

      if (!(base.exists() && base.isDirectory())) {
          System.out.println("Verzeichnisfehler " + args[0]);
          System.exit(1);
      }

      ClassLoader cl = new MeinClassLoader(base);
      Class c = cl.loadClass(args[1]);
      Runnable r = (Runnable)(c.newInstance());
      r.run();
    }
}
```

Als Parameter beim Start muss das Verzeichnis angegeben werden in dem sich die Datei `Runner.class` befindet, plus dieser Klassennamen (also ohne `.class`).

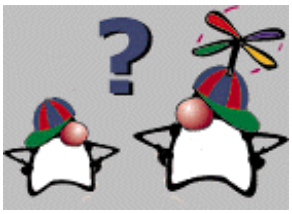
Jetzt versuchen wir eine Klasse zweimal zu laden:
aus diesem Grunde modifizieren wir das Programm in der main Methode leicht und prüfen, ob beide Klassen identisch sind. Im diesem Fall muss die zweite Klasse aus dem Cache stammen.

Dieser Test zeigt einfach, wie dies überprüft werden kann. Die Ausführung der zwei Runner lassen wir dann gleich weg. Das ist zu uninteressant.

```
package securityclassloaderv2;

import java.util.*;
import java.net.*;
import java.io.*;

public class MeinClassLoader extends ClassLoader {
    // JDK 1.0.x : jeder ClassLoader besitzt einen eigenen Cache
    // private Hashtable cache = new Hashtable();
    private final File myBase;
    private static final boolean verbose = Boolean.getBoolean("Debug");

    public MeinClassLoader(File myBase) {
      if (verbose) { log("MeinClassLoader: constructor, mybase = " +
          myBase); }
      this.myBase = myBase;
    }

    private byte[] loadClassData(String name)
      throws ClassNotFoundException {
      byte [] buffer;

      if (verbose) { log("MeinClassLoader: loadClassData, name=" +
          name); }
      File source = null;
      InputStream s = null;
      try {
          source = new File(myBase, name);
          s = new FileInputStream(source);
          buffer = new byte [(int)(source.length())];
          s.read(buffer);
      }
      catch (Exception e) {
          System.out.println("Probleme mit " + source);
          throw new ClassNotFoundException(name);
      }
      return buffer;
    }

    public synchronized Class loadClass(String name, boolean resolve)
      throws ClassNotFoundException {
      if (verbose) { log("MeinClassLoader: loadClass, name=" + name +
          " resolve=" + resolve); }
      if (name.charAt(0) == '.') {
          throw new SecurityException("'Absolute' package name");
      }
      if (name.indexOf("..") != -1) {
          throw new SecurityException("Illegal package name");
      }

      // jdk1.0.x
      // Class c = (Class)(cache.get(name));
      Class c = (findLoadedClass(name));

      if (c == null) {
          try {
            c = findSystemClass(name);
          }
          catch (Exception e) {
            try {
                String path = name.replace('.', File.separatorChar) +
```

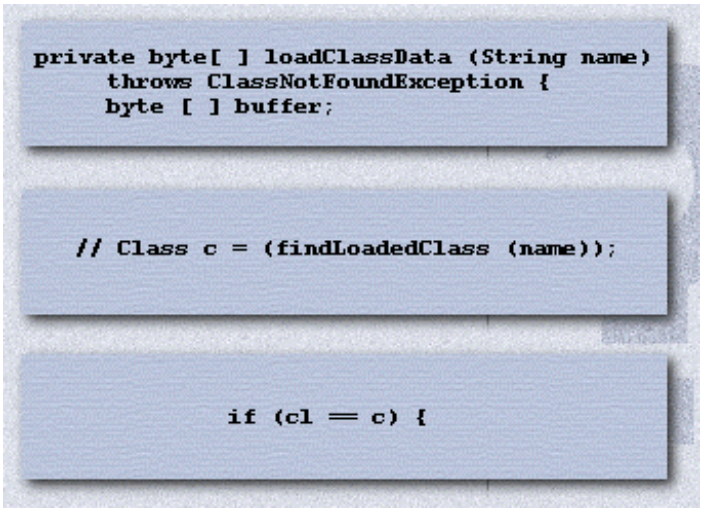
```
                    ".class";
                byte data[] = loadClassData(path);
                c = defineClass(name, data, 0, data.length);
                // JDK 1.1+ defineClass cached die Klassen
                // JDK 1.0.x - benutzen des eigenen Caches
                // cache.put(name, c);
              }
              catch (ClassFormatError ex) {
                throw new ClassNotFoundException(" Class ist: " + name +
                  " Problem ist: " + ex.getMessage());
              }
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }

    private void log(String s) {
      System.err.println(s);
    }

    public static void main(String args[]) throws Throwable {
      if (args.length < 2) {
          System.out.println("MeinClassLoader : fehlendes Argument ");
          System.exit(1);
      }
      File base = new File(args[0]);

      if (!(base.exists() && base.isDirectory())) {
          System.out.println("Verzeichnisfehler " + args[0]);
          System.exit(1);
      }
      ClassLoader cl = new MeinClassLoader(base);
      Class c = cl.loadClass(args[1]);
      Class c1 = cl.loadClass(args[2]);
      if (c1 == c) {
          System.out.println("Die zwei Klassen sind identisch "+
            "(referenzieren auf das gleiche Objekt); die zweite Klasse  "+
            "wurde somit aus dem Cache geladen.");
      } else {
          System.out.println("Die zwei Klassen sind unterschiedlich;"+
            " die zweite Klasse wurde somit nicht im Cache gefunden.");
      }
      //Runnable r = (Runnable)(c.newInstance());
      //r.run();

    }
}
```

Fett ausgezeichnet ist der geänderte Bereich im Programm.
Sie können das Programm mit den Parametern <Verzeichnis> <Class1> <Class2> aufrufen.
Dabei sollten Sie das Programm mehrfach ausführen: einmal mit zwei gleichen Klassen; einmal mit unterschiedlichen Klassen.



Und hier nun einige Fragen:

Wie müssen Sie das Programm ändern, damit der Cache nicht mehr abgefragt wird? In diesem Fall muss das System feststellen, dass eine Klasse zweimal geladen werden soll und damit ein Sicherheitsproblem entstehen könnte.[7]



a)

b)

c)

Sie finden im Programm eine Variante, die sich vom obigen Vorschlag unterscheidet.

1.4.9.2. Checksum Validation

Mit dem Programm Sum wird eine Class Datei um ihre Checksumme ergänzt. Diese Class Datei kann anschliessend nicht mehr direkt ausgeführt werden. Sie müssten einen eigenen Class Loader schreiben, der diese Zusatzinformation wieder überprüft und löscht, bevor die Class Datei an die JVM übergeben wird.

```
package securityclassloader;

import java.io.*;

public class Sum {
    public static int calculate(byte[] data) {
      int sum = 0;
      for (int i = 0; i < data.length; i++) {
          sum += data[i];
      }
      return sum;
    }

    public static void main(String args[]) throws Throwable {
      File f = new File(args[0]);
      if (!((args.length > 0) && f.exists() && f.isFile()
          && f.canRead())) {
          System.out.println("Datei " + args[0]+" kann nicht gelesen
werden");
          System.exit(1);
      }
      FileInputStream in = new FileInputStream(f);
```

[7] b) : falls der Test fehlt, wird der Cache nicht abgefragt und das Sicherheitssystem von Java muss aktiv werden

```
      byte [] buffer = new byte[(int)(f.length())];
      in.read(buffer);
      //in.close();
      int sum = calculate(buffer);
      if (args.length > 1) {
            System.out.println("Anlegen der Ausgabedatei¨"+args[1]+"\nDiese
besitzt eine Bad Magic Number");
          f = new File(args[1]);
          DataOutputStream out = new DataOutputStream( new
              FileOutputStream(f));
          out.writeInt(0xadc0ffee);
          out.writeInt(sum);
          out.write(buffer);
          out.close();
      }
      System.out.println("Summe ist " + sum);
    }
}
```

Damit endet unsere kurze Tour durch den Class Loader. Sie finden in der Literatur (zum Beispiel im Buch Scott Oaks ***Java Security***, O'Reilly Verlag) viele weitere Beispiele, unter anderem zum Laden von Klassen aus unterschiedlichen Servern. Einige der verwendeten Beispiele stammen aus diesem Buch.

1.5. Modul 4 - Security Manager

1.5.1. Einleitung

In diesem Modul zeigen wir Ihnen, wie Sie Ihr System schützen können, mit Hilfe eines Security Managers. Applets verwenden immer einen Security Manager; Applikationen können, müssen aber nicht, mit einem Security Manager arbeiten.

Der Security Manager war in JDK 1.1 recht komplex implementierbar. Ab JDK1.2 wurde das Ganze wesentlich erleichtert. Man könnte sagen, dass die Protection Domains den Security Manager (fast) überflüssig machen.

Alle Browser benutzen (für die Applets) einen Security Manager. Daher ist es wichtig zu wissen, wie er funktioniert.

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

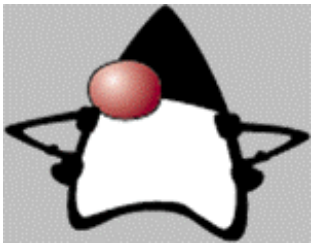
- zu beschreiben, welche Rolle ein Security Manager für die Java Sicherheit spielt.
- die Rollen eines Security Managers mit jener eines Class Loaders zu vergleichen und gegeneinander abzugrenzen.
- die Security Manager Klasse und deren Methoden zu erklären und zu wissen, welche Tests der Security Manager durchführt.
- einen einfachen Security Manager für ein Java Programm zu implementieren

Referenz

```
JDK1.3\java\guide\security\index.html
JDK1.3/java/api/java/lang/SecurityManager.html
```

1.5.2. Security Manager - Übersicht

Das meiste, was Sie hier kennen lernen, ist für JDK 1.1.x entwickelt worden. Die Änderungen für JDK 1.2+ werden im nächsten Modul genauer beschrieben. In der Regel müssen Sie aber rückwärtskompatible Anwendungen entwickeln. Daher müssen Sie wissen, wie der Security Manager in JDK 1.1.x funktionierte.



Der Security Manager ist eine abstrakte Klasse, welche bestimmte Rechte und Einschränkungen für einen bestimmten Name Space definiert. Die Einschränkungen können das lokale Dateisystem, Netzwerkzugriffe, Abstract Windowing Toolkit (AWT) oder andere Bereiche betreffen.

Der Class Loader ist für das Laden der Klassen und Festlegen der Namensräume zuständig; der Security Manager definiert die Grenzen der Sandbox in JDK 1.1.x. Der Security Manager wird immer dann von der JVM aufgerufen, wenn ein Applet eine Aktion ausführen möchte und dabei den Zugriff auf System Ressourcen benötigt. Jedes Programm kann höchstens einen Security Manager besitzen.

Schauen wir uns ein einfaches Programm an und den Aufruf des Security Managers:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World ");
  }
}
```

Jetzt starten wir dieses Programm mit einem Security Manager (dem Standard Security Manager):

java -cp . -**Djava.security.debug=all** HelloWorld

Dann erhalten wir folgende Ausgabe:

```
scl:  getPermissions (file:/D:/NDKJava/Programme/SecurityClassLoader/ <no certificates>)
policy: reading file:C:/JBuilder4/jdk1.3/jre/lib/security/java.policy
policy: Adding policy entry:
policy:   signedBy null
policy:   codeBase file:C:/JBuilder4/jdk1.3/jre/lib/ext/*
policy:
policy:   (java.security.AllPermission <all permissions> <all actions>)
policy:
policy: Adding policy entry:
policy:   signedBy null
policy:   codeBase null
policy:
policy:   (java.lang.RuntimePermission stopThread)
policy:   (java.net.SocketPermission localhost:1024- listen,resolve)
policy:   (java.util.PropertyPermission java.version read)
policy:   (java.util.PropertyPermission java.vendor read)
policy:   (java.util.PropertyPermission java.vendor.url read)
policy:   (java.util.PropertyPermission java.class.version read)
policy:   (java.util.PropertyPermission os.name read)
policy:   (java.util.PropertyPermission os.version read)
policy:   (java.util.PropertyPermission os.arch read)
policy:   (java.util.PropertyPermission file.separator read)
policy:   (java.util.PropertyPermission path.separator read)
policy:   (java.util.PropertyPermission line.separator read)
policy:   (java.util.PropertyPermission java.specification.version read)
policy:   (java.util.PropertyPermission java.specification.vendor read)
```

```
policy:   (java.util.PropertyPermission java.specification.name read)
policy:   (java.util.PropertyPermission java.vm.specification.version read)
policy:   (java.util.PropertyPermission java.vm.specification.vendor read)
policy:   (java.util.PropertyPermission java.vm.specification.name read)
policy:   (java.util.PropertyPermission java.vm.version read)
policy:   (java.util.PropertyPermission java.vm.vendor read)
policy:   (java.util.PropertyPermission java.vm.name read)
policy:
policy: reading file:C:/WINNT/Profiles/zajoller.000/.java.policy
policy: Adding policy entry:
policy:   signedBy null
policy:   codeBase file://d:/SecurityEinschub/WriteFileApplet/
policy:
policy:   (java.io.FilePermission writetest.txt write)
policy:   (java.io.FilePermission <<ALL FILES>> write)
policy:
policy: evaluate((file:/D:/NDKJava/Programme/SecurityClassLoader/ <no certificates>))
policy:   granting (java.lang.RuntimePermission stopThread)
policy:   granting (java.net.SocketPermission localhost:1024- listen,resolve)
policy:   granting (java.util.PropertyPermission java.version read)
policy:   granting (java.util.PropertyPermission java.vendor read)
policy:   granting (java.util.PropertyPermission java.vendor.url read)
policy:   granting (java.util.PropertyPermission java.class.version read)
policy:   granting (java.util.PropertyPermission os.name read)
policy:   granting (java.util.PropertyPermission os.version read)
policy:   granting (java.util.PropertyPermission os.arch read)
policy:   granting (java.util.PropertyPermission file.separator read)
policy:   granting (java.util.PropertyPermission path.separator read)
policy:   granting (java.util.PropertyPermission line.separator read)
policy:   granting (java.util.PropertyPermission java.specification.version read)
policy:   granting (java.util.PropertyPermission java.specification.vendor read)
policy:   granting (java.util.PropertyPermission java.specification.name read)
policy:   granting (java.util.PropertyPermission java.vm.specification.version read)
policy:   granting (java.util.PropertyPermission java.vm.specification.vendor read)
policy:   granting (java.util.PropertyPermission java.vm.specification.name read)
policy:   granting (java.util.PropertyPermission java.vm.version read)
policy:   granting (java.util.PropertyPermission java.vm.vendor read)
policy:   granting (java.util.PropertyPermission java.vm.name read)
scl:   java.security.Permissions@3e86d0 (
 (java.lang.RuntimePermission exitVM)
 (java.lang.RuntimePermission stopThread)
 (java.net.SocketPermission localhost:1024- listen,resolve)
 (java.io.FilePermission \D:\NDKJava\Programme\SecurityClassLoader\- read)
 (java.util.PropertyPermission java.vendor read)
 (java.util.PropertyPermission java.specification.version read)
 (java.util.PropertyPermission line.separator read)
 (java.util.PropertyPermission java.class.version read)
 (java.util.PropertyPermission java.specification.name read)
 (java.util.PropertyPermission java.vendor.url read)
 (java.util.PropertyPermission java.vm.version read)
 (java.util.PropertyPermission os.name read)
 (java.util.PropertyPermission os.arch read)
 (java.util.PropertyPermission os.version read)
 (java.util.PropertyPermission java.version read)
 (java.util.PropertyPermission java.vm.specification.version read)
 (java.util.PropertyPermission java.vm.specification.name read)
 (java.util.PropertyPermission java.specification.vendor read)
 (java.util.PropertyPermission java.vm.vendor read)
 (java.util.PropertyPermission file.separator read)
 (java.util.PropertyPermission path.separator read)
 (java.util.PropertyPermission java.vm.name read)
 (java.util.PropertyPermission java.vm.specification.vendor read)
)
scl:
```

Der klassische Security Manager existiert also auch noch in JDK 1.2. Allerdings wurden die Methoden der Klasse umdefiniert, so dass die Protection Domain Infrastruktur die Zugriffssicherheit garantiert (der `AccessController`, den wir noch diskutieren werden, ist für diese Entscheidungen verantwortlich). Im Falle der Applets ist der Security Manager eine wesentliche Komponente für die Sicherstellung der Sicherheit, insbesondere gegen Zugriffe auf lokale Daten und Dateisysteme. Der Security Manager garantiert, dass Daten nur ab der Codebase, also dem Heimverzeichnis des Applets geladen werden dürfen.

1.5.3. Security Managers und Applikationen

Java Applikationen haben in der Regel keinen Security Manager. Falls Sie die Rechte einer Java Applikation einschränken möchten, müssen Sie einen Security Manager laden. Allerdings haben wir bereits gesehen, dass die Zugriffsrechte auf Objekte und Klassen systematisch geprüft werden. Sie haben auch die Möglichkeit auf der Kommandozeile einen Security Manager anzugeben. Wir haben dies beim Beispiel weiter vorne bereits gezeigt (-Djava....).

Einen Security Manager benötigen Sie auch, falls Sie Klassen über ein Netzwerk herunter laden möchten. Einen eigenen Security Manager können Sie als Erweiterung der `SecurityManager` Klasse definieren. Ihre Aufgabe besteht dann im Wesentlichen im Überschreiben der dort definierten Methoden.

Ob eine Applikation einen Security Manager besitzt und wie dieser heisst, können Sie mit Hilfe der Methode `getSecurityManager` feststellen:

```
public class AbfrageDesSecurityManagers {

  public static void main(String[] args) {
    SecurityManager sec = System.getSecurityManager();
    System.out.println("Security Manager : "+sec);
 }
}
```

Ein Security Manager Objekt ist ein fixer Schutz, also immer aktiv. Die Security Manager Klasse ist als abstrakt definiert, legt also nur das generelle Verhalten fest.

Im obigen Beispiel wird `null` als Ergebnis geliefert. Dies besagt, dass kein (gültiger) Security Manager installiert wurde. Im Falle eines Applets können Sie aus Sicherheitsgründen den Security Manager nicht überschreiben. Sie können generell einen bereits installierten Security Manager nicht überschreiben. Der Security Manager ist per default sehr restriktiv: er verhält sich analog zu einem Hardware Router, der per default kein Protokoll akzeptiert. Es liegt an Ihnen festzulegen, was konkret gestattet wird.

1.5.4. Die `checkXXX` Methoden

Die Klasse `SecurityManager` besitzt jede Menge `check`-Methoden. Alle werfen eine `SecurityException`, falls Probleme auftauchen.

Diese sind in die JVM eingebaut worden, also auch in JDK 1.2+ vorhanden.
Schauen wir uns kurz einige dieser Methoden an:

`void` **`checkAccept`**`(String host, int port)`
wirft eine Exception, falls die Verbindung zum Host / Port nicht gestattet ist.

`void` **`checkAccess`**`(Thread t)`
wirft eine Exception, falls der aktive Thread den Thread t nicht modifizieren darf.

`void` **`checkCreateClassLoader`**`()`
wirft eine Exception, falls versucht wird, einen eigenen Class Loader zu kreieren.

1.5.5. Methoden und Operationen

Die folgende Tabelle zeigt die Objekte und die entsprechenden checkXXX Methoden.

Operation an	Prüfung

Sockets	`checkAccept(String` *host*`, int` *port*`)`
	`checkConnect(String` *host*`, int` *port*`)`
	`checkConnect(String` *host*`, int` *port*`, Object` *executionContext*`)`
	`checkListen(int` *port*`)`

Threads	`checkAccess(Thread` *thread*`)` `checkAccess(ThreadGroup` *threadgroup*`)`
Class loader	`checkCreateClassLoader()`

File system	`checkDelete(String` *filename*`)`
	`checkRead(FileDescriptor` *filedescriptor*`)`
	`checkRead(String` *filename*`)`
	`checkRead(String` *filename*`, Object` *executionContext*`)`
	`checkWrite(FileDescriptor` *filedescriptor*`)`
	`checkWrite(String` *filename*`)`

System commands	`checkExec(String` *command*`)` `checkLink(String` *library*`)`
Interpreter	`checkExit(int` *status*`)`
Package	`checkPackageAccess(String` *packageName*`)` `checkPackageDefinition(String` *packageName*`)`
Properties	`checkPropertiesAccess()` `checkPropertyAccess(String` *key*`)`
Networking	`checkSetFactory()`
Windows	`checkTopLevelWindow(Object` *window*`)`

Um eine Security Policy festzulegen, müssen Sie sich Gedanken darüber machen, welche dieser Methoden Sie konkret überschreiben müssen. In einem Beispiel werden wir den Zugriff auf Dateien modifizieren. Dazu benötigen wir aus der obigen Gruppe den Lese- und den Schreib-Zugriff.

1.5.6. FileInputStream Beispiel

Falls wir aus einer Datei des lokalen Dateisystems Daten heraus lesen möchten, geschieht der Reihe nach folgendes:

- zuerst wird geprüft, ob ein Security Manager vorhanden ist
- falls ja, dann wird die `checkRead()` Methode des Security Managers aufgerufen
- falls diese Methode fehlerfrei abgeschlossen wird, dann darf man auf die Datei zugreifen
- falls diese Methode eine Exception wirft, ist der Zugriff auf die Datei nicht erlaubt.

Schauen wir uns einmal den Programmcode für das Gerüst der check Methode an:

```
public void checkRead(String file) {
                                throw new SecurityException();
}
```

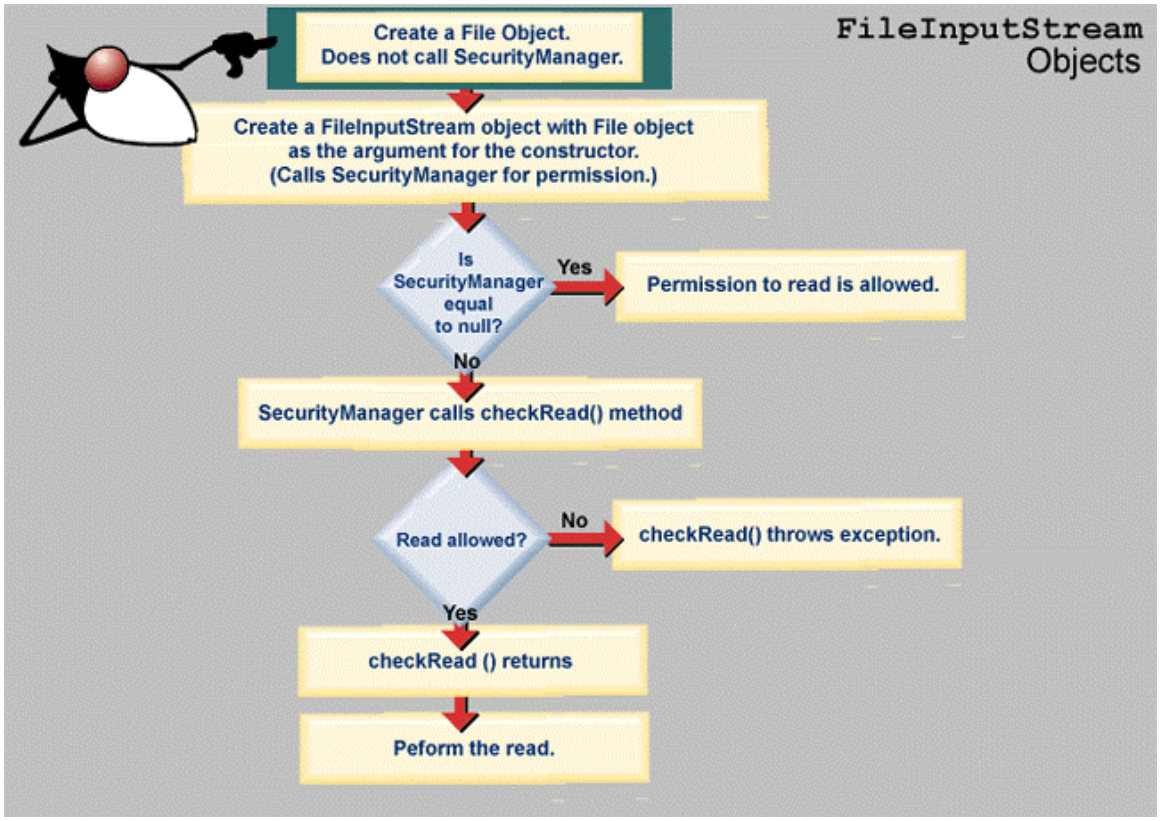
Wie diese in einen Security Manager eingebaut wird, sehen wir weiter unten. Unser Zugriffsprogramm könnte beispielsweise folgendermassen aussehen:

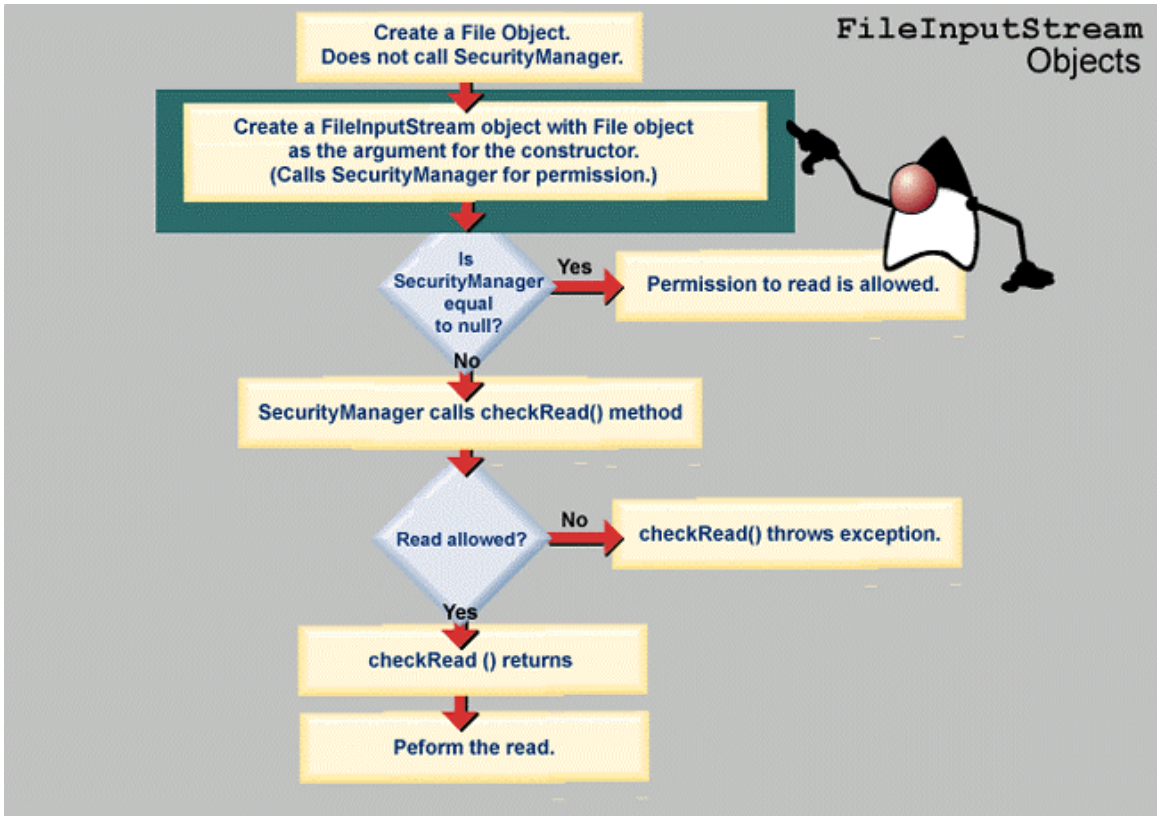
```
import java.io.*;
public class LesenEinerDatei {
  public static void main(String[] args) {
    try {
      FileInputStream fis = new FileInputStream("Textdatei.txt");
      int i=0;
      Integer iObj = new Integer(i);
      while ((i=fis.read())!=-1) {// EOF = -1
        System.out.print( (char)i); // Ausgabe als Zeichen : casten
      }
    } catch(IOException ioe) {
      System.err.println("Datei existiert nicht");
      ioe.printStackTrace();
    }
  }
}
```

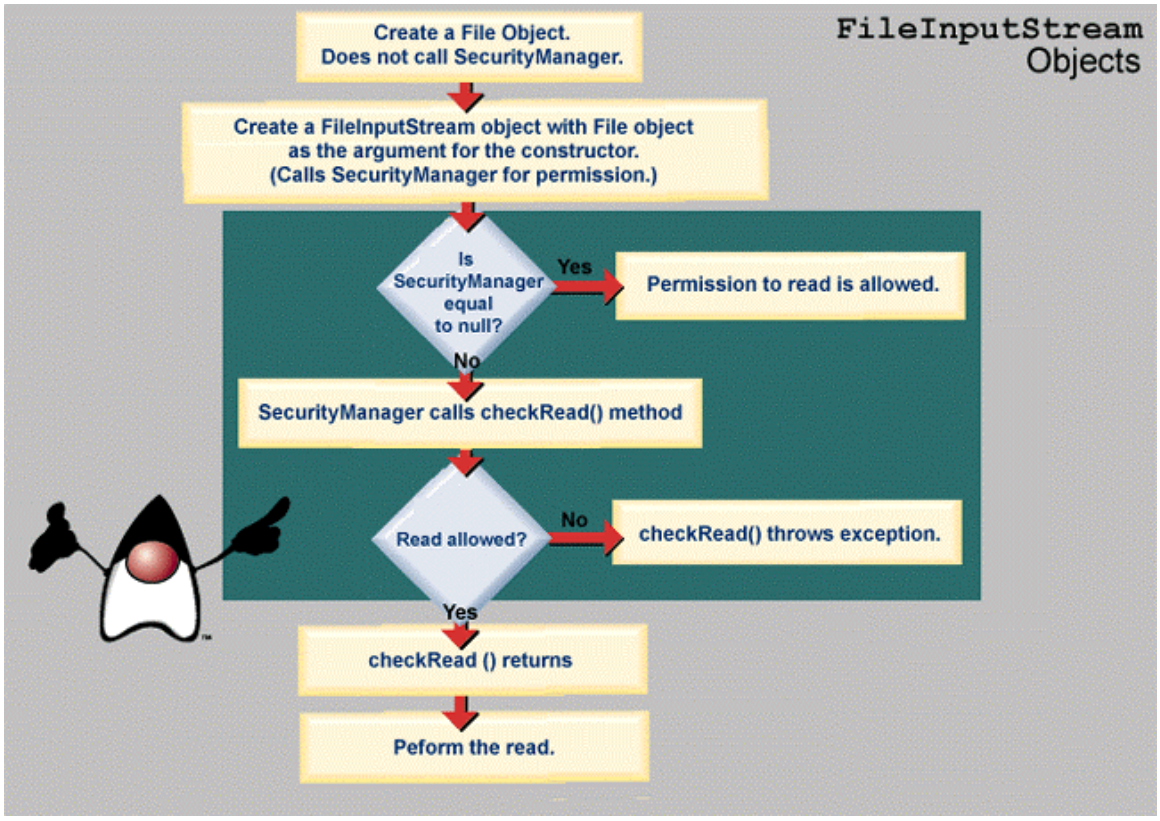
Wenn Sie dieses Programm starten, werden Sie keinerlei Probleme bei der Ausführung haben. Der Grund ist einfach der, dass kein Security Manager geladen wurde.

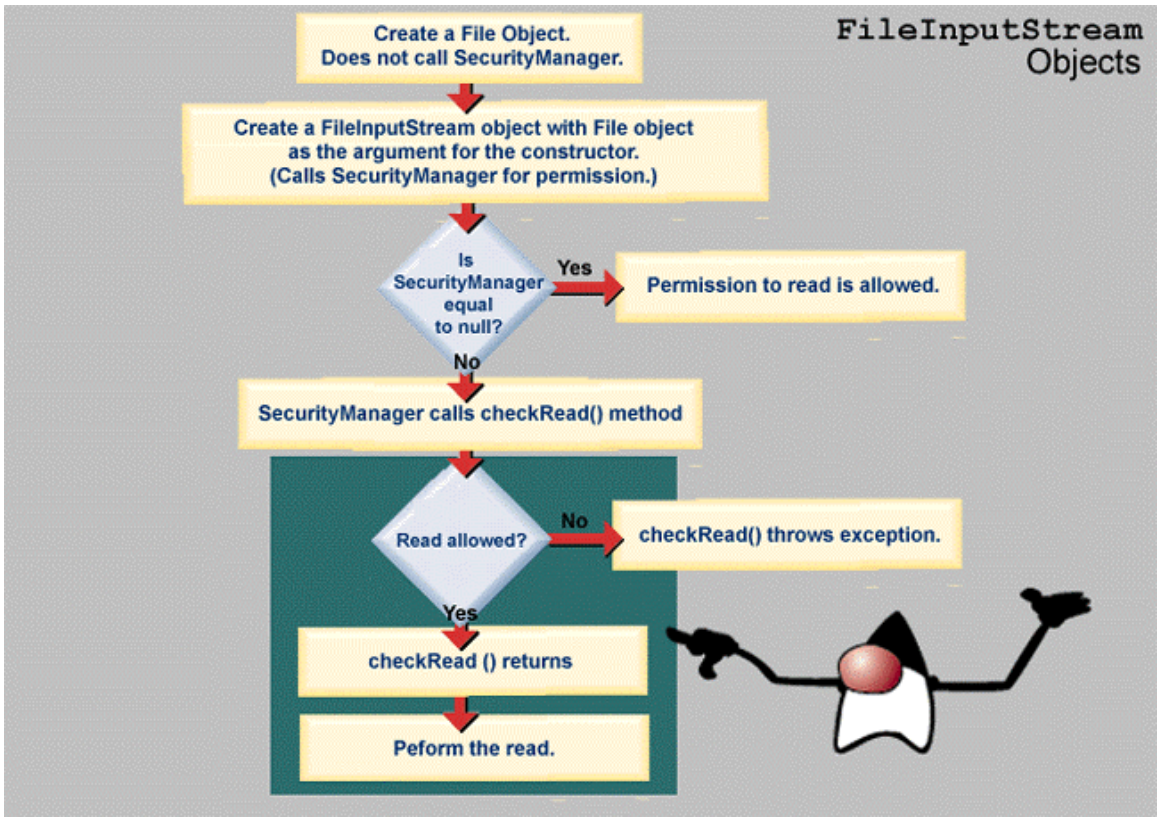
In den folgenden Seiten und Abschnitten werden wir schrittweise diesen Security Manager konstruieren.

Zuerst wollen wir uns jedoch den Ablauf der Sicherheitsprüfung verdeutlichen.


FileInputStream Objects
Create a File Object.
Does not call SecurityManager.
Create a FileInputStream object with File object as the argument for the constructor.
(Calls SecurityManager for permission.)
Is SecurityManager equal to null?
Yes
Permission to read is allowed.
No
SecurityManager calls checkRead() method
Read allowed?
No
checkRead() throws exception.
Yes
checkRead () returns
Peform the read.


FileInputStream Objects
Create a File Object.
Does not call SecurityManager.
Create a FileInputStream object with File object as the argument for the constructor.
(Calls SecurityManager for permission.)
Is SecurityManager equal to null?
Yes
Permission to read is allowed.
No
SecurityManager calls checkRead() method
Read allowed?
No
checkRead() throws exception.
Yes
checkRead () returns
Peform the read.


FileInputStream Objects
Create a File Object.
Does not call SecurityManager.
Create a FileInputStream object with File object as the argument for the constructor.
(Calls SecurityManager for permission.)
Is SecurityManager equal to null?
Yes
Permission to read is allowed.
No
SecurityManager calls checkRead() method
Read allowed?
No
checkRead() throws exception.
Yes
checkRead () returns
Peform the read.


FileInputStream Objects
Create a File Object.
Does not call SecurityManager.
Create a FileInputStream object with File object as the argument for the constructor.
(Calls SecurityManager for permission.)
Is SecurityManager equal to null?
Yes
Permission to read is allowed.
No
SecurityManager calls checkRead() method
Read allowed?
No
checkRead() throws exception.
Yes
checkRead () returns
Peform the read.

1.5.7. Kreieren eines eigenen Security Managers

Um einen eigenen Security Manager zu kreieren, müssen Sie die `SecurityManager` Klasse erweitern. Diese enthält eine Menge Methoden, mit deren Hilfe die einzelnen Java Klassen die Sicherheitsprüfungen vornehmen. Es liegt an Ihnen, diese zu überschreiben und damit eigene Sicherheitsrichtlinien zu implementieren.

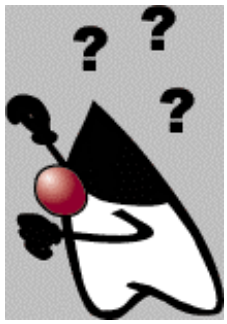
Beispielsweise könnten Sie eine Passwortabfrage implementieren (dazu gibt es in Java bereits viele vorgefertigte Klassen, das Beispiel illustriert nur das Konzept):

```
public class PasswortSecurityManager extends SecurityManager {
      ...
}
```

In JDK 1.x wurden Sicherheitsrichtlinien für alle native Zugriffe (beispielsweise das Aufrufen von C Routinen) mit solchen Mecahnismen abgesichert.

Die harte Codierung der Sicherheitsrichtlinien in die Applikation ist eher ungeschickt. Besser ist die Lösung in JDK1.2+: mit Hilfe einer Policy Datei werden die Zugriffsrechte festgelegt. Diese Datei lässtsich leicht mit Hilfe des PolicyTools (im bin Verzeichnis des JDK) definieren. Aber es ist Ihnen überlassen, die Properties zu bestimmen und eigene Sicherheitsrichtlinien zu implementieren.

Die konkrete Definition des Security Managers geschieht, indem Sie die Methoden überschreiben. Ihre Aufgabe ist es zu entscheiden, welche Methoden überschrieben werden müssen und wie:



- welche Methoden sollen überschrieben werden?
- müssen neue Methoden hinzugefügt werden?
- wie muss ich die zu überschreibenden Methoden definieren?
- wie strikt sollen meine Sicherheitsregeln sein?

1.5.8. Security Manager Methoden

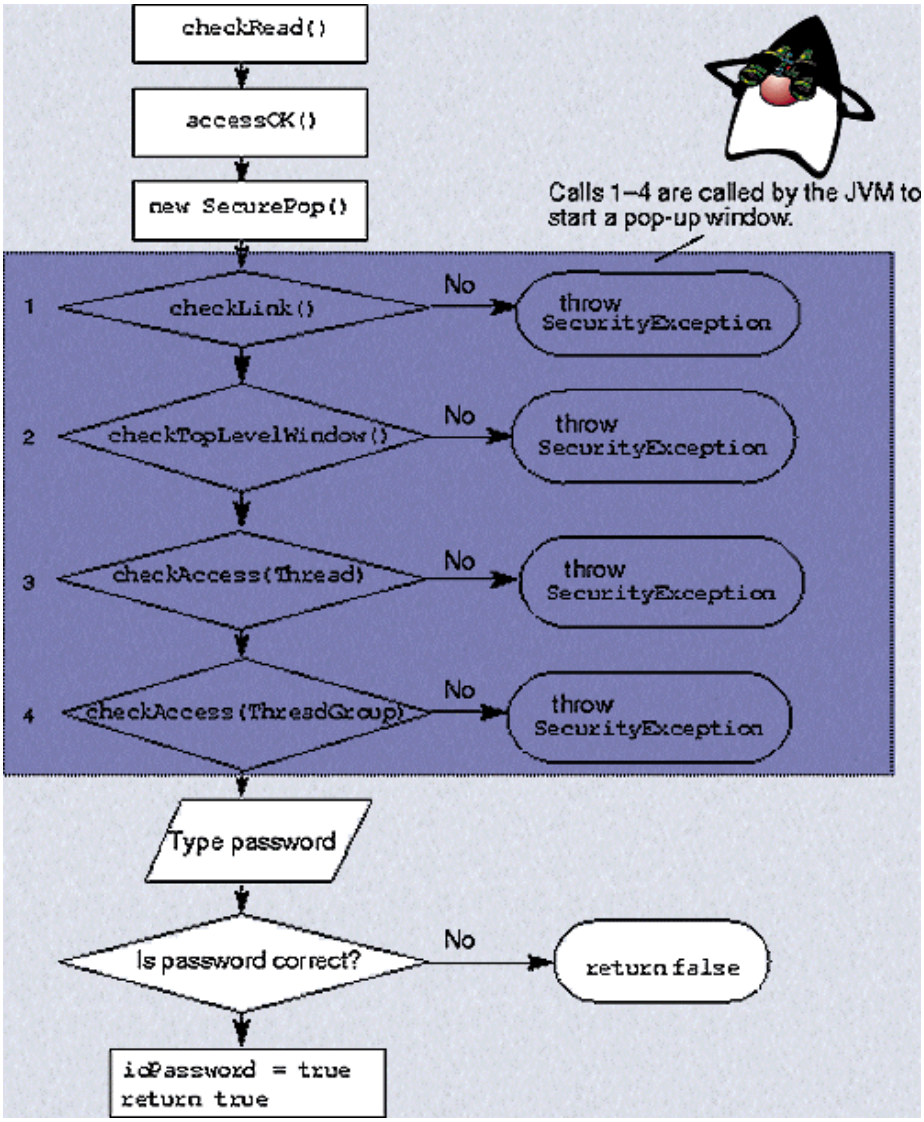
Die folgenden drei Standard Methoden müssen sogut wie immer angepasst werden:

```
public void checkRead(FileDescriptor fd) throws SecurityException{
      throw new SecurityException();
}

public void checkRead(String file) throws SecurityException{
      throw new SecurityException();
}

public void checkRead(String file, Object context)throws SecurityException{
      throw new SecurityException();
}
```

Alle drei Methoden werfen eine `SecurityException`. Falls wir beispielsweise ein Passwort überprüfen wollen, könnte dies folgendermassen aussehen (als Skizze):

```
 public void checkRead(String filename) throws SecurityException {
      System.out.println("checkRead(" + filename + ")");
            if (!accessOK()) { // Passwort ist falsch
                  throw new SecurityException("No Way!");
            }
 }

private boolean accessOK() {
      boolean ok = true;
      if (inClassLoader()) {
            if (!ioPassword) {
                  if (sp.getPassword().equals(password)) {
                        ioPassword = true;
                  } else {ok = false;}
            }
      }
      return ok;
}
```

Der Flow Chart zeigt den geplanten Ablauf, der dem obigen Code zu Grunde liegt.



Die Methode `checkLink` überprüft das Vorhandensein bestimmter Bibliotheken. Wenn Sie beispielsweise grafische Oberflächen definieren, müssen die entsprechenden Bibliotheken vorhanden sein und der Zugriff darauf gestattet sein.

```
public void checkLink(String library){
      //Code für Checking
}
```

Die Methode `checkTopLevelWindow` prüft, ob ein Top Level Window kreiert werden darf. Diese Abfrage kann sinnvoll sein, um zu verhindern, dass der Benutzer ein eigenes Login Fenster definiert.

```
public synchronized boolean checkTopLevelWindow(Object obj){
      return !inClassLoader();
}
```

Die Methode `checkAccess(Thread)`wird immer dann aufgerufen, wenn der Thread modifiziert werden soll.

```
public synchronized void checkAccess(Thread t){
      ...
}
```

Die Methode `checkAccess(ThreadGroup)`wird aufgerufen, wenn die Thread Gruppe modifiziert werden soll.

```
public synchronized void checkAccess(ThreadGroup tg){
      ...
}
```

Die Dateizugriffsmethoden `checkWrite()` mit unterschiedlichen Parametern überprüfen den Zugriff auf Dateien. Analog verhält es sich mit den `checkRead()` Methoden.

```
public void checkWrite(String filename) {
      System.out.println("checkWrite("+filename+")");
      if (!accessOK()) {
            throw new SecurityException("Nichts da!");
      }
}
```

1.5.9. Installieren eines Security Managers

Nachdem Sie einen Security Manager geschrieben haben, die Sicherheitsrichtlinien definiert haben, ist es kaum ein Problem, den Security Manager in die Applikation einzubinden. Dies geschieht mit der statischen `System.setSecurityManager` Methode.

Das folgende Fragment zeigt, wie dies geschehen kann:

```
import java.io.*;

class SecurityManagerTest {
  public static void main(String[] args) {
   try {
    System.setSecurityManager(new PasswordSecurityManager("PASSWORD"));
   } catch (SecurityException se) {
    System.err.println("SecurityManager wurde schon gesetzt!");
   }

    try {
     ClassLoader cl = new SampleClassLoader(new File("try"));
      Class c = cl.loadClass("Copier");
      Runnable r = (Runnable)(c.newInstance());
      r.run(); // lesen und schreiben
    } catch (Exception e) {
      e.printStackTrace();
    }
    System.exit(0); // das war's
  }
}
```

1.5.10. Datei Zugriff

Im obigen Beispiel stecken alle Dateizugriffe im Thread `Copier`. Diese Klasse wird vom Class Loader geladen.

```
import java.io.*;
public class Copier implements Runnable {
  public void() {
   try {
    Buffered Reader fis = new BufferedReader (new
                                    FileReader<br>("inputtext.txt));
    BufferedWriter fos = new BufferedWriter (new
                                    FileWriter<br>("outputtext.txt"));
    char [] buffer = new char [4096];
    int count}
    while ((count = fis.read(buffer)) > -1) {
     fos.write(buffer, 0, count);
    }
    fis.close();
    fos.close();
   } catch (Exception e) {
     e.printStackTrace();
   }
  }
}
```

1.5.11. Nach der Installation

Es kann lediglich ein Security Manager installiert werden. Falls Sie versuchen, mehr als einen Security Manager zu installieren, wird eine Exception geworfen. Es ist sogar so, dass der Security Manager niemals explizit aufgerufen werden muss, seine Anwesenheit reicht bereits.

Bei Applets wird der Browser einen Security Manager installieren, mit dem das Sandbox Modell implementiert wird.

1.5.12. Übung - Security Manager

Die folgenden Dateien fassen die Programmfragmente aus diesem Modul zusammen.

1.5.12.1. SampleClassLoader

```
package securitymanagerzusammenfassung;

import java.util.*;
import java.net.*;
import java.io.*;
import java.security.*;
import java.math.*;

public class SampleClassLoader extends ClassLoader {
    private File myBase;
    private static final boolean verbose = Boolean.getBoolean ("Debug");

    public SampleClassLoader (File mb) {
        myBase = mb;
        log ("SampleClassLoader: constructor, mb = " + mb);
    }

    private byte [] loadClassData (String name)
        throws ClassNotFoundException {
        byte [] buffer;

        log ("SampleClassLoader: loadClassData, name=" + name);
        File source = null;
        InputStream s = null;
        try {
            source = new File (myBase, name);
            s = new FileInputStream (source);
            buffer = new byte [(int)(source.length ())];
            s.read (buffer, 0, 8); // read 8 bytes
            DataInputStream in =
                new DataInputStream (new ByteArrayInputStream (buffer));
            int magic = in.readInt ();
            int sum = in.readInt ();
            if (magic == 0xadc0ffee) {
                // signed, lies alles in den Buffer,
                if (verbose) {
                    log ("Classfile " + source.getName () +
                        " ist mit Quersumme versehen");
                }
                s.read (buffer);
                int sumRead = Sum.calculate (buffer);
                if (sum != sumRead) {
                    if (verbose) { log ("Quersumme stimmt nicht!"); }
                    throw new ClassNotFoundException ("Classfile " +
                        source.getName () +
```

```
                                " besitzt eine fehlerhafte Quersumme");
                    }
                } else {
                    // Classfile ohne Checksumme
                    // lies den Rest
                    if (verbose) {
                        log ("Classfile " + source.getName () +
                            " ist ohne Quersumme");
                    }
                    s.read (buffer, 8, buffer.length - 8);
                }
            }
            catch (Exception e) {
                System.out.println ("Probleme mit " + source);
                throw new ClassNotFoundException (name);
            }
            return buffer;
        }
        public synchronized Class loadClass (String name, boolean resolve)
            throws ClassNotFoundException {
            log ("SampleClassLoader: loadClass, name=" + name +
                    " resolve=" + resolve);
            if (name.charAt (0) == '.') {
                throw new SecurityException ("'Absolute' package name");
            }
            if (name.indexOf ("..") != -1) {
                throw new SecurityException ("Illegal package name");
            }

            Class c = findLoadedClass (name);

            if (c == null) {
                try {
                    c = findSystemClass (name);
                }
                catch (Exception e) {
                    log ("SampleClassLoader: class ist non-system");
                    try {
                        String path = name.replace
                            ('.', File.separatorChar) + ".class";
                        byte [] data = loadClassData (path);
                        c = defineClass (name, data, 0, data.length);
                    }
                    catch (ClassFormatError ex) {
                        throw new ClassNotFoundException (name);
                    }
                }
            }
            if (resolve) {
                resolveClass (c);
            }
            return c;
        }
        private void log (String s) {
            if (verbose) {
                System.err.println (s);
            }
    }
        public static void main (String args[]) throws Throwable {
            if (args.length < 2) {
                System.out.println (
                    "SampleClassLoader  ");
```

```
            System.exit (1);
        }
        File base = new File (args[0]);

        if (!(base.exists () && base.isDirectory ())) {
            System.out.println ("Verzeichnisfehler " + args[0]);
            System.exit (1);
        }

        System.setSecurityManager (new SampleSecurityManager ());
        ClassLoader cl = new SampleClassLoader (base);

        Class c = cl.loadClass (args[1]);
        Runnable r = (Runnable)(c.newInstance ());
        r.run ();
    }
}
```

1.5.12.2. SampleSecurityManager

```
package securitymanagerzusammenfassung;

import java.io.*;
import java.net.*;

public class SampleSecurityManager extends SecurityManager {

    static {
      System.out.println (
          "SampleSecurityManager loading, verbose mode ist " +
          (Debug.VERBOSE ? "ON" : "OFF"));
    }

    public void checkCreateClassLoader () {
      Debug.println ("checkCreateClassLoader ()");
    }

    public void checkAccess (Thread g) {
      Debug.println ("checkAccess (Thread g)");
    }

    public void checkAccess (ThreadGroup g) {
      Debug.println ("checkAccess (ThreadGroup g)");
    }

    public void checkExit (int status) {
      Debug.println ("checkExit (int status)");
    }

    public void checkExec (String cmd) {
      Debug.println ("checkExec (String cmd)");
    }

    public void checkLink (String lib) {
      Debug.println ("checkLink (String lib)");
    }

    public void checkRead (FileDescriptor fd) {
      Debug.println ("checkRead (FileDescriptor fd)");
    }
```

```
public void checkRead (String file) {
  Debug.println ("checkRead (String file)");
}

public void checkRead (String file, Object context) {
  Debug.println ("checkRead (String file, Object context)");
}

public void checkWrite (FileDescriptor fd) {
  Debug.println ("checkWrite (FileDescriptor fd)");
}

public void checkWrite (String file) {
  Debug.println ("checkWrite (String file)");
}

public void checkDelete (String file) {
  Debug.println ("checkDelete (String file)");
}

public void checkConnect (String host, int port) {
  Debug.println ("checkConnect (String host, int port)");
}

public void checkConnect (String host, int port, Object context) {
  Debug.println (
      "checkConnect (String host, int port, Object context)");
}

public void checkListen (int port) {
  Debug.println ("checkListen (int port)");
}

public void checkAccept (String host, int port) {
  Debug.println ("checkAccept (String host, int port)");
}

public void checkMulticast (InetAddress maddr) {
  Debug.println ("checkMulticast (InetAddress maddr)");
}

public void checkMulticast (InetAddress maddr, byte ttl) {
  Debug.println ("checkMulticast (InetAddress maddr, byte ttl)");
}

public void checkPropertiesAccess () {
  Debug.println ("checkPropertiesAccess ()");
}

public void checkPropertyAccess (String key) {
  Debug.println ("checkPropertyAccess (String key)");
}

public void checkPropertyAccess (String key, String def) {
  Debug.println ("checkPropertyAccess (String key, String def)");
}

public boolean checkTopLevelWindow (Object window) {
  return true;
}
```

```
    public void checkPrintJobAccess () {
      Debug.println ("checkPrintJobAccess ()");
    }

    public void checkSystemClipboardAccess () {
      Debug.println ("checkSystemClipboardAccess ()");
    }

    public void checkAwtEventQueueAccess () {
      Debug.println ("checkAwtEventQueueAccess ()");
    }

    public void checkPackageAccess (String pkg) {
      Debug.println ("checkPackageAccess (String pkg)");
    }

    public void checkPackageDefinition (String pkg) {
      Debug.println ("checkPackageDefinition (String pkg)");
    }

    public void checkSetFactory () {
      Debug.println ("checkSetFactory ()");
    }

    public void checkMemberAccess (Class clazz, int which) {
      Debug.println ("checkMemberAccess (Class clazz, int which)");
    }

    public void checkSecurityAccess (String provider) {
      Debug.println ("checkSecurityAccess (String provider)");
    }
}

class Debug {
    public static final boolean VERBOSE =
      Boolean.getBoolean ("security.verbose");

    public static void println (String s) {
      if (VERBOSE) {
          System.out.println (s);
      }
    }

    public static void print (String s) {
      if (VERBOSE) {
          System.out.print (s);
      }
    }
}
```

1.5.12.3. Sum - Checksumme

```
package securitymanagerzusammenfassung;

import java.io.*;

public class Sum {
    public static int calculate(byte[] data) {
      int sum = 0;
      for (int i = 0; i < data.length; i++) {
          sum += data[i];
      }
      return sum;
    }

    public static void main(String args[]) throws Throwable {
      File f = new File(args[0]);
      if (!((args.length > 0) && f.exists() && f.isFile()
          && f.canRead())) {
          System.out.println("Datei" + args[0]+
                  " kann nicht gelesen werden");
          System.exit(1);
      }
      FileInputStream in = new FileInputStream(f);
      byte [] buffer = new byte[(int)(f.length())];
      in.read(buffer);
      //in.close();
      int sum = calculate(buffer);
      if (args.length > 1) {
          f = new File(args[1]);
          DataOutputStream out = new DataOutputStream( new
              FileOutputStream(f));
          out.writeInt(0xadc0ffee);
          out.writeInt(sum);
          out.write(buffer);
          out.close();
      }
      System.out.println("Checksumme ist " + sum);
    }
}
```

1.5.12.4. Runner

```
package securitymanagerzusammenfassung;

import java.io.*;

public class Runner implements Runnable {
    public void run () {
      System.out.println ("Runner ist up and running!");
      try {
          BufferedReader in = new BufferedReader (
            new FileReader ("test.txt"));
          String s;

          while ((s = in.readLine ()) != null) {
            System.out.println (s);
          }
      }
      catch (Exception e) {
          e.printStackTrace ();
      }
    }
}
```

1.6. Erweiterung des Sandbox Security Modells

Das Sandbox Modell ist das ursprüngliche Sicherheitsmodell der JVM: Applets werden so restriktive wie nur immer möglich behandelt.

1.6.1. Einleitung

Wir gehen auf die Weiterentwicklung des klassischen Sandbox Modells ein, das Java Protection Domains Security Modell und auch warum diese Weiterentwicklung nötig wurde.

1.6.1.1. Lernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein

- das klassische Sandbox Modell mit dem neueren Java Protection Domains Sicherheits Modell zu vergleichen.
- in einem Diagramm aufzuzeigen, wie das klassische und das neue Sicherheitsmodell funktionieren.
- Schwächen des Sandbox Modells aufzuzeigen und zu erläutern warum die Erweiterung des Modells nötig war.
- zu unterscheiden zwischen einer System Domäne und einer Applikationsdomäne.
- zu beschreiben, wie Threads die Rechte bestimmen, falls mehrere Applikationsdomänen betroffen sind.
- zu erläutern, welche Rolle die Policy Datei spielt, im Rahmen des Protection Domain Sicherheitsmodells.

1.6.1.2. Referenzen

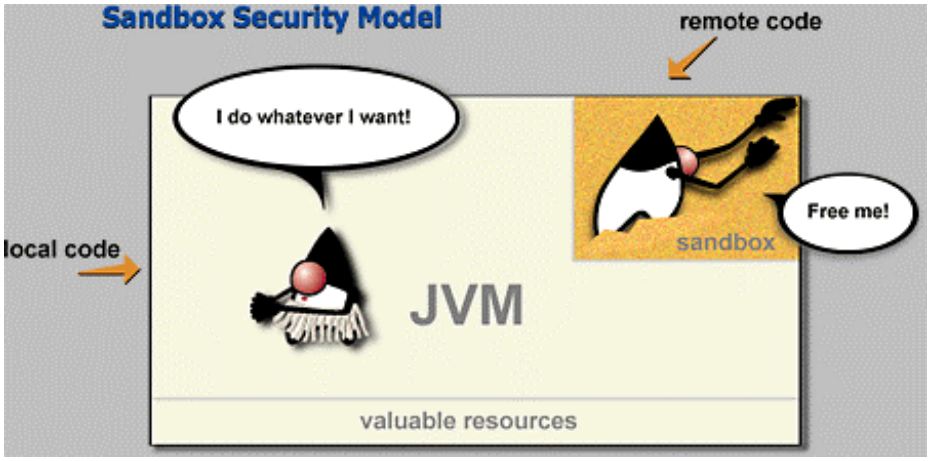
Folgende Links führen Sie zu zusätzlicher Information für diesen Modul:

- ***FAQs - Java Protected Domains Security Model***. [Online]:
  `http://java.sun.com/marketing/collateral/prot_domain_faq.html`

- ***Secure Computing With Java: Now and the Future***. Whitepaper. [Online]:
  `http://java.sun.com/marketing/collateral/security.html`

1.6.2. Kurzwiederholung - Das Sandbox Security Modell

Das ursprüngliche Sandbox Modell basierte auf dem Prinzip:



- lokalem Programmcode kann vertraut werden
- nichtlokalem Programmcode muss misstraut werden. Dies implizitiert, dass der Programmcode in der Sandbax bleiben muss und diese also nicht verlassen darf. Es sind auch keine Zugriffe auf Ressourcen des lokalen Systems möglich. Die Sandbox zeigt also die Grenze des nicht vertrauenswürdigen Programmcodes. Innerhalb der Schranken kann das Programm tun und lassen was es will. Aber lesen und schreiben ausserhalb der Grenzen ist nicht gestattet.

Die Sandbox besteht aus drei Teilen:

- dem Class Loader
- dem Byte Code Verifier
- dem Security Manager

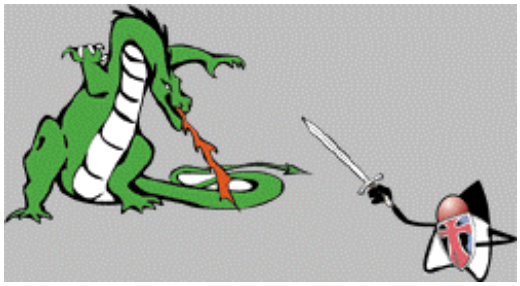
Diese drei Komponenten arbeiten zusammen und garantieren die Intergrität des Systems. Sie unterscheiden vertrauenswürdigen von nicht vertrauenswürdigem Programmcode. Jede Komponente trägt ihren Teil zur Sicherheit bei:

- der Class Loader garantiert, dass lediglich die korrekten Klassen geladen werden. Was die Klassen tun oder worauf sie zugreifen wird nicht bestimmt.
- der Verifikationsprozess (mit dem Byte Code Verifier) garantiert das korrekte Format der Class Dateien.
- der Security Manager arbeitet mit der Java Virtual Machine und garantiert, dass lediglich vertrauenswürdige Klassen Zugriff auf die System Ressourcen haben und somit gefährliche Klassen keinen gefährlichen Code ausführen können.

1.6.3. Die Sandbox schützt nicht

Die Sandbox schützt Sie nicht vor Applets, welche

- viel CPU verbrauchen
- Anzeigen:
  - falsche Login Fenster
  - störende Effekte haben
  - Meldungen, mit denen sensitive Benutzerdaten abgefragt werden
- denial of Service starten (beispielsweise viele Fenster öffnen)
- Text in riesigen Fonts starten



Diese Eigenschaften werden schlechten Applets zugeordnet. Diese Themen werden an anderer Stelle besprochen.

1.6.4. Applet Fähigkeiten

In der folgenden Zusammenstellung finden Sie, was ein Applet tun darf und was nicht, gemäss dem Java Sandbox Security Modell. Jeglicher Programmcode, der über das Netzwerk geladen wird, darf folgendes auf dem Gastsystem (dem Client) nicht tun:

- lesem, schreiben, umbenennen oder löschen von Dateien.
- Verzeichnisse anlegen oder deren Inhalt abfragen
- Informationen über Dateien abfragen (existiert eine Datei? wie gross ist die Datei? von welchem Typus ist die Datei? Zeitstempel?)
- Informationen über den Benutzer abfragen.
- eine Netzwerkverbindung zu irgend einem Rechner aufbauen, der mit dem Rechner, von dem der Code stammt, nicht identisch ist.
- auf Netzwerkverbindungen warten (aktiv, als Server) oder Netzwerkverbindungsanfragen aktiv beantworten (als Server)
- Netzwerkkontrollfunktionen ausführen
- System Eigenschaften definieren
- Programme mittels der `Runtime.exec()` Methoden ausführen oder dynamisch Bibliotheken laden.
- Class Loaders oder Security Managers kreieren
- Threads kreieren oder manipulieren, die nicht zur `ThreadGroup` des Applets gehören.
- Klassen definieren, die nicht zu einem Package auf dem Client gehören.

Auf Unix Systemen könnte beispielsweise versucht werden mit `getProperty()` die Eigenschaften `user.name` und `user.dir` abzugfragen. Unter Windows ist dies nicht möglich.

In JDK 1.1 war es möglich, Applets zu signieren (mit dem javakey Utility). Vertrauenswürdige und nicht so vertrauenswürdige Applets konnte man auf Grund der Signatur unterscheiden.

Im neuen Modell hängen die Möglichkeiten eines Applets vom Protection Domain des Applets ab und den Rechten, die dem Protection Domain zugeordnet sind.

1.6.5. Das Java Protection Domain Sicherheitsmodell

Im alten Sicherheitsmodell war man sehr unflexibel. Daher wurde ab JDK 1.2 das Sicherheitsmodell überarbeitet. An Stelle der hard-coded Sicherheits-Richtlinien kommen flexiblere Spezifikation mittels des PolicyTools und der java.policy Datei.

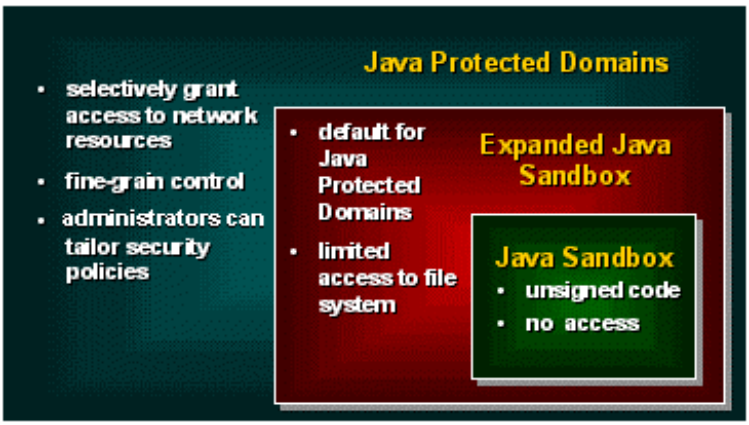
Diese neue Modell hat folgende klare Vorteile im Vergleich zum alten Sandbox Modell:

- Sicherheits-Policies sind leicht konfigurierbar:

  wie bereits beim Sandbox Modell erklärt, wurden Sicherheitsvorschriften in der Regel über checkXXX Methoden überprüft und auch festgelegt.
  Im neuen Modell werden die Sicherheitsvorschriften in einer ASCII Policy Datei festgelegt. Dieses kann mit Hilfe des `PolicyTool` generiert und unterhalten werden. Sie können auch weiterhin die Sicherheitseinstellungen aus einer Datenbank lesen und festlegen, das ist Ihnen überlassen. Sie können auch die Klasse `Policy` verwenden und das `Policy` Objekt serialisieren, also als Datei abspeichern.

- vordefinierte Sicherheitsstufen, welche Sie im `PolicyTool` auswählen können.

  Dies war auch im Sandbox Modell möglich. Aber dort wurde alles programmiert. Im neuen Modell werden einfach bestimmte Rechte ausgewählt und in die Policy Datei geschrieben.
  Der mühsame Prozess über den Security Manager entfällt damit (inklusive Unterklassenbildung und Anpassungen des Security Managers und Class Loaders).
  Die Spezifikation mittels einer Policy Datei ist wesentlich bequemer und kann leicht, ohne Programmänderungen angepasst werden.
  Auch die Spezifikation von Netzwerk-Zugriffen ist leicht steuerbar.



Weitere Vorteile sind:

- im alten Modell musste man jeweils bei zusätzlichen Anforderungen den Security Manager anpassen (eine weitere checkXXX Methode spezifizieren). Im neuen Modell ist dies leicht durch mutieren der Policy Datei möglich.

- Security Checks können auch auf Anwendungen angewandt werden
  Im Sandbox Modell waren die Sicherheitsrichtlinien auf Applets beschränkt.

Und hier ein Beispiel für eine Policy Datei:

```
/* AUTOMATICALLY GENERATED ON Tue Feb 27 18:12:47 GMT+01:00 2001*/
/* DO NOT EDIT */

grant {
  permission java.io.FilePermission "Textdatei7.txt", "write";
  permission java.net.NetPermission "requestPasswordAuthentication";
  permission java.lang.reflect.ReflectPermission "suppressAccessChecks";
  permission java.security.SecurityPermission "getProperty.<property name>";
};
```

In der ersten Zeile wird festgelegt, dass in die Textdatei 7 geschrieben werden darf, anschliessend werden weitere Zugriffsrechte spezifiziert.

In der Regel kann sehr detailliert spezifiziert werden was wo möglich ist. Dies geschieht mit Hilfe der CODEBASE Spezifikation.

1.6.6. Protection Domains

Das grundlegende Konzept im Java Protection Domains Sicherheitsmodell ist, wie der Name bereits sagt, das Protection Domain (die Sicherheitsdomäne). Mittels Domänen werden Objekte, die geschützt werden sollen, gruppiert und isoliert.

Sobald der Programmcode in die JVM geladen wird, wird auch eine Sicherheitsdomäne angelegt. Jeder Sicherheitsdomäne sind Rechte zugewiesen. Diese Rechte basieren auf zwei Attributen: der Lokation und dem Signierer. Der Signierer ist die Entität, die den Code signiert hat. Falls niemand den Code signiert, ist der `signer` **`null`**. Die Lokation oder die CODEBASE legt fest (als URL) woher die Java Klassen stammen.

Schauen wir uns ein Beispiel an, bei dem alle Klassen akzeptiert werden, welche von einer der beiden URLs stammen:

- Sun Microsystems inklusive Signatur;
  diese Klassen gehören zur Sicherheitsdomäne Sun;
- IBM inklusive Signatur;
  diese Klassen gehören zur Sicherheitsdomäne IBM;
- Microsoft inklusive Signatur;
  diese Klassen gehören zur Sicherheitsdomäne Microsoft.

Die Sicherheitspolicy würde jeder der Domänen bestimmte Sicherheitseinschränkungen zuweisen und bestimmte Zugriffe gestatten.

Protection Domains gehören zu einer der folgenden Kategorien:

- System Domain
  Klassen, die zu dieser Domäne gehören, haben vollen Zugriff auf die Maschine des Benutzers. Klassen, die typischerweise zu dieser Domäne gehören, sind
  `java.io.File`
  `java.awt`
  `java.net`
  usw. (Klassen, die mit externen Ressourcen verknüpft sind). Alle Klassen im `CLASSPATH` gehören zu dieser Domäne.
  Die anderen Klassen nutzen die System Domain Klassen, um auf die entsprechenden Ressourcen zuzugreifen.

- Application Domains
  Dazu gehören alle anderen Schutzdomänen. Alle Class Dateien, welche von einer und derselben Quelle stammen und die selbe Signatur aufweisen (oder keine) gehören zur selben Sicherheitsdomäne.
  Die Sicherheitsdomäne wird beim Laden der Klasse kreiert.

1.6.7. Security Policy Datei

Die Ssystem Security wird festgelegt durch eine System- und eine Benutzer- Policydatei.

Die System Policydatei steht normalerweise imVerzeichnis `/lib/security` des JDK. Sie haben dies bereits weiter vorne in der Ausgabe der Security Abfrage gesehen. Diese Datei heisst `java.policy`.

Dieser Name kann angepasst werden. Es ist in der Datei java.security im obigen Verzeichnis eingetragen:

```
#
# This is the "master security properties file".
#
# In this file, various security properties are set for use by
# java.security classes. This is where users can statically register
# Cryptography Package Providers ("providers" for short). The term
# "provider" refers to a package or set of packages that supply a
# concrete implementation of a subset of the cryptography aspects of
# the Java Security API. A provider may, for example, implement one or
# more digital signature algorithms or message digest algorithms.
#
# Each provider must implement a subclass of the Provider class.
# To register a provider in this master security properties file,
# specify the Provider subclass name and priority in the format
#
#    security.provider.<n>=<className>
#
# This declares a provider, and specifies its preference
# order n. The preference order is the order in which providers are
# searched for requested algorithms (when no specific provider is
# requested). The order is 1-based; 1 is the most preferred, followed
# by 2, and so on.
#
# <className> must specify the subclass of the Provider class whose
# constructor sets the values of various properties that are required
# for the Java Security API to look up the algorithms or other
# facilities implemented by the provider.
#
# There must be at least one provider specification in java.security.
# There is a default provider that comes standard with the JDK. It
# is called the "SUN" provider, and its Provider subclass
# named Sun appears in the sun.security.provider package. Thus, the
# "SUN" provider is registered via the following:
#
#    security.provider.1=sun.security.provider.Sun
#
# (The number 1 is used for the default provider.)
#
# Note: Statically registered Provider subclasses are instantiated
# when the system is initialized. Providers can be dynamically
# registered instead by calls to either the addProvider or
# insertProviderAt method in the Security class.

#
# List of providers and their preference orders (see above):
#
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider

# Class to instantiate as the system Policy. This is the name of the class
# that will be used as the Policy object.
policy.provider=sun.security.provider.PolicyFile

# The default is to have a single system-wide policy file,
# and a policy file in the user's home directory.
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

```
# whether or not we expand properties in the policy file
# if this is set to false, properties (${...}) will not be expanded in policy
# files.
policy.expandProperties=true

# whether or not we allow an extra policy to be passed on the command line
# with -Djava.security.policy=somefile. Comment out this line to disable
# this feature.
policy.allowSystemProperty=true

# whether or not we look into the IdentityScope for trusted Identities
# when encountering a 1.1 signed JAR file. If the identity is found
# and is trusted, we grant it AllPermission.
policy.ignoreIdentityScope=false

#
# Default keystore type.
#
keystore.type=jks

#
# Class to instantiate as the system scope:
#
system.scope=sun.security.provider.IdentityDatabase

#
# List of comma-separated packages that start with or equal this string
# will cause a security exception to be thrown when
# passed to checkPackageAccess unless the
# corresponding RuntimePermission ("accessClassInPackage."+package) has
# been granted.
package.access=sun.

#
# List of comma-separated packages that start with or equal this string
# will cause a security exception to be thrown when
# passed to checkPackageDefinition unless the
# corresponding RuntimePermission ("defineClassInPackage."+package) has
# been granted.
#
# by default, no packages are restricted for definition, and none of
# the class loaders supplied with the JDK call checkPackageDefinition.
#
#package.definition=
```

`policy.url.1=file:${java.home}/lib/security/java.policy`
`policy.url.2=file:${user.home}/.java.policy`

dabei ist `${java.home}` in meinem Fall `JBuilder4/jdk1.3/jre` und `${user.home}` verweist auf das Profil in WinNT.

Die Standard `java.policy` Datei, welche mit JDK mitgeliefert wird, sieht folgendermassen aus:

```
// Standard extensions get all permissions by default

grant codeBase "file:${java.home}/lib/ext/*" {
        permission java.security.AllPermission;
};

// default permissions granted to all domains

grant {
        // Allows any thread to stop itself using the java.lang.Thread.stop()
        // method that takes no argument.
        // Note that this permission is granted by default only to remain
        // backwards compatible.
        // It is strongly recommended that you either remove this permission
        // from this policy file or further restrict it to code sources
        // that you specify, because Thread.stop() is potentially unsafe.
        // See "http://java.sun.com/notes" for more information.
        permission java.lang.RuntimePermission "stopThread";
```

```
        // allows anyone to listen on un-privileged ports
        permission java.net.SocketPermission "localhost:1024-", "listen";

        // "standard" properies that can be read by anyone

        permission java.util.PropertyPermission "java.version", "read";
        permission java.util.PropertyPermission "java.vendor", "read";
        permission java.util.PropertyPermission "java.vendor.url", "read";
        permission java.util.PropertyPermission "java.class.version", "read";
        permission java.util.PropertyPermission "os.name", "read";
        permission java.util.PropertyPermission "os.version", "read";
        permission java.util.PropertyPermission "os.arch", "read";
        permission java.util.PropertyPermission "file.separator", "read";
        permission java.util.PropertyPermission "path.separator", "read";
        permission java.util.PropertyPermission "line.separator", "read";

        permission java.util.PropertyPermission "java.specification.version", "read";
        permission java.util.PropertyPermission "java.specification.vendor", "read";
        permission java.util.PropertyPermission "java.specification.name", "read";

        permission java.util.PropertyPermission "java.vm.specification.version", "read";
        permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
        permission java.util.PropertyPermission "java.vm.specification.name", "read";
        permission java.util.PropertyPermission "java.vm.version", "read";
        permission java.util.PropertyPermission "java.vm.vendor", "read";
        permission java.util.PropertyPermission "java.vm.name", "read";
};
```

Beim Starten der JVM wird zuerst die System Policy Datei gelesen, dann werden auch die Benutzerpolicies hinzugefügt. Falls diese Dateien fehlen, wird automatisch das Sandbox Modell angewandt.

Man kann auch den Java Applikation Sicherheitsrichtlinien zuordnen. Die Namenskonvention für solche Policydateien sieht folgendermassen aus:

`policy.`*`application-name`*`=policy_name`

Zum Beispiel:

```
policy.navigator=navigator.policy
policy.hotjava=hotjava.policy
policy.explorer=explorer.policy
```

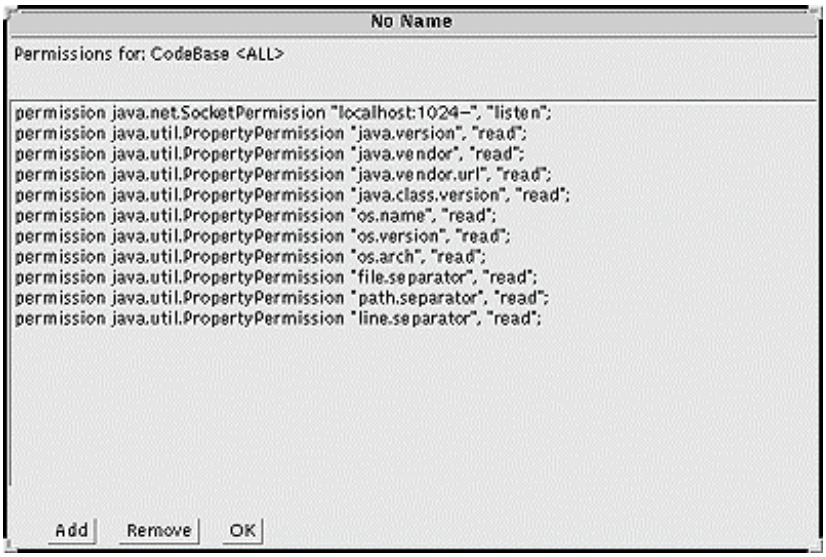
Sie können aber auch einen beliebigen Namen wählen und die Datei mit dem Laufzeitparameter `-Djava.security.policy=<dateiname>` angeben.

Falls Sie eigene Policy Provider definieren wollen, müssen Sie dazu ein `Policy` Objekt kreieren, also eine Instanz der Klasse `Policy`.

Die Policy Dateien können Sie mit jedem Texteditor kreieren. Mitgeliefert wird ein einfaches Hilfsprogramm, welches nicht sehr überzeugend programmiert wurde, aber seine Funktion erfüllt.

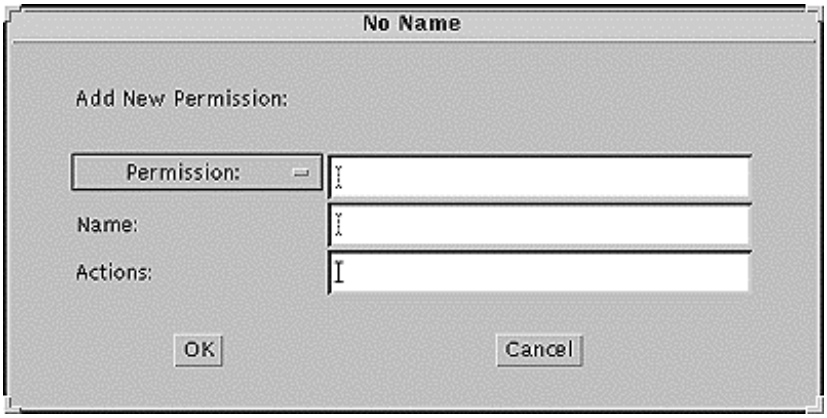
Das Tool, PolicyTool, finden Sie im JDK bin Verzeichnis.

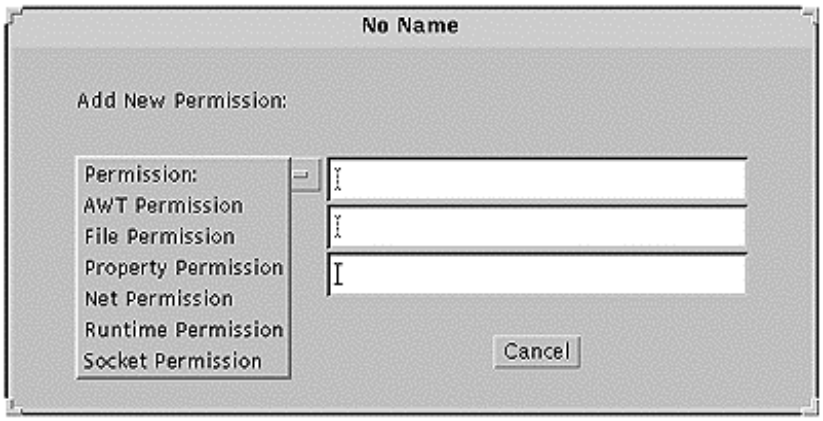
Und so sieht die System Policy Datei aus (beachten Sie das Verzeichnis: java_home):



Die möglichen Permissions sehen Sie, wenn Sie das Menü Permission anklicken:

Schauen Sie auf Ihrem Rechner die Policy Dateien an. Zudem finden Sie auf dem Server / Web / CD Beispiele für Policy Dateien, beispielsweise um einem Applet das Recht zu geben, lokale Dateien anzulegen oder zu ergänzen.





Sie brauchen die Permissions nicht auwendig zu kennen. Immer wenn Sie Probleme haben, können Sie immer noch zusätzliche Freigaben machen.

Das Format jedes Eintrags in der Policy Datei ist:

```
grant [signedBy "signer_names"] [, codeBase "URL"] {
      permission permission_class_name ["target_name"]
            [, "action"] [, signedBy "signer_names"];
      permission ...
};
```

wobei:

- Leerzeichen vor und nach dem Komma erlaubt sind
- alle Permmissions voll qualifiziert angegeben werden müssen
  zum Beispiel `java.io.FilePermission`
- das `action` Feld ist optional, muss aber, falls vorhanden, bündig folgen.
- falls die `CODEBASE` fehlt, gilt die Regel überall.

Sie können auch alle Dateien in einem Verzeichnis oder alle Dateien in einem Verzeichnisbaum in eine Policy einbeziehen:

- <directory>/*
  alle Dateien in im Verzeichnis <directory>

- <directory>-
  alle Dateien im Verzeichnisbaum unter <directory>

- -
  alle Dateien des Dateisystems

Es sind mehrere Permission Klassen definiert:

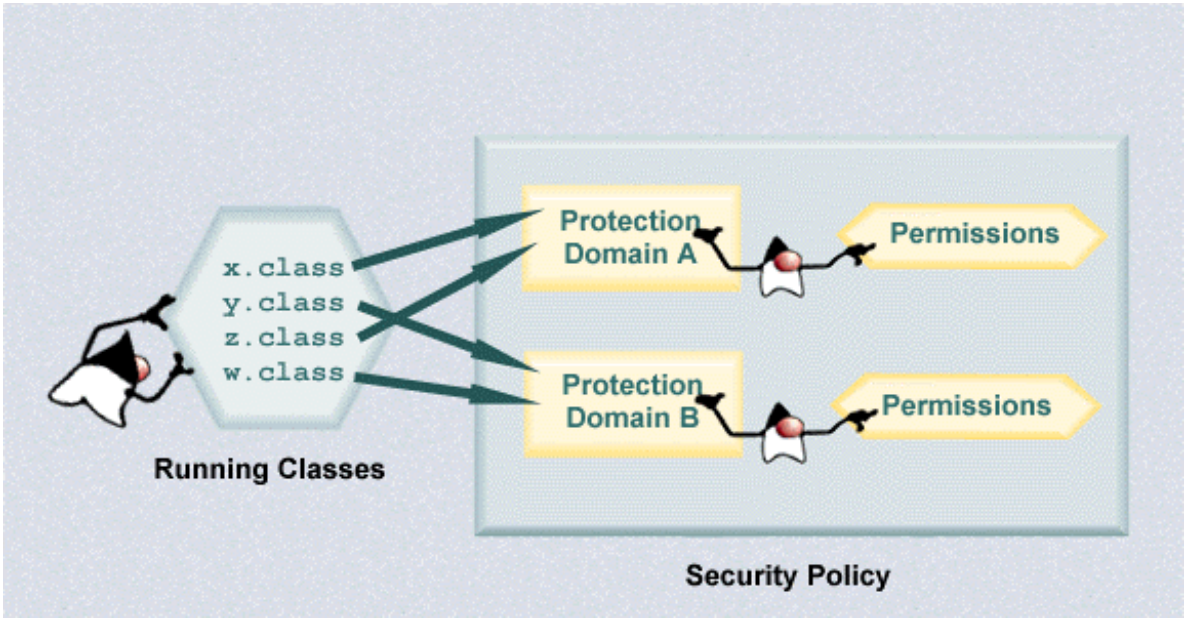
- `java.io.FilePermission`
- `java.net.NetPermission`
- `java.lang.RuntimePermission`
- `java.net.SocketPermission`
- `java.security.BasicPermission`
- `java.util.PropertyPermission`
- `java.awt.AWTPermission`

Für jede dieser Permissions sind bestimmte Aktionen und Aktionstypen definiert. Beispielsweise kann man im Falle von `FilePermission` die Rechte für `read, write, execute` und `delete` als Actionstypen definieren.

Beispiel:

```
grant signedBy "Sun,IBM", codeBase "http://java.sun.com/" {
  permission java.io.FilePermission "/tmp", "read";
  permission java.io.SocketPermission "*", "connect";
```

Falls mehrere Einträge für einen bestimmten Fall gültig sind, werden alle möglichen Rechte gegeben.



1.7. Java JDK Security Klassen

Ohne dass wir auf Details eingehen sei zum Abschluss noch erwähnt, dass in JDK verschiedene Klassen definiert wurden, mit denen die gesamte Sicherheitsarchitektur von Java von Hand programmiert werden kann.

Die relevanten Klassen sind:

- `java.security.Policy`
- `java.security.Permission`
- `java.security.AccessController`
- `java.security.SecureClassLoader`
- `java.security.GeneralSecurityException`

Sie finden eine Besprechung dieser Security Klassen in Li Gong's Paper, ***Java Security Architecture (JDK 1.2)***. [Online]:
`http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-specTOC.doc.html.`

Diese Themen werden noch detaillierter in einer weiteren Einheit
Java Security Features
sowie in
Java Security Protocols & Policies
detaillierter besprochen.

1.8. Zusammenfassung

Was haben Sie in dieser Einheit gelernt?
Wir haben einige grundlegende Mechanismen der Java Virtual Machine besprochen (Class Loader, Security Manager und vorallem Byte Code verifier) besprochen bzw. kennen gelernt.

Sicher fehlt noch viel, um zu behaupten, dass man alles verstehen kann, auf Grund des Textes. Aber in der Regel werden diese Themen nicht mehr so stark betont und fehlen in vielen Kursen sogar ganz. Sie sind aber wichtig beim Programmieren verteilter Systeme. Dort muss man die Policies setzen, da sonst das Sandbox Modell aktiv wird, mit all seinen Einschränkungen.

In diesem Sinne sollte dies eine Einführung in die Thematik sein!

Weitergehende Literatur wurde bereits erwähnt.