**In diesem Kapitel**:

- *Die* System *Klasse*
  - *Standard I/O Streams*
  - *System Properties*
- *Prozesse kreieren*
  - *Die* Process *Klasse*
  - *Die Prozess Umgebung*
  - *Portabilität*
- *Shutdown*
  - *Shutdown Fallen*
  - *Shutdown Abfolge*
  - *Shutdown Strategien*
- *Der Rest von Runtime*
  - *Laden von Native Code*
  - *Debugging*
- Security
  - *Die* SecurityManager *Klasse*
  - *Permissions*
  - *Security Policies*
  - *Zugriffskontrolle und privilegierte Ausführung*

# *System Programmierung*

*GLENDOWER: I can call spirits from the vasty deep.*
*HOTSPUR: Why, so can I , or so can any man;*
*But will they come when you call for them?*
- William Shakespeare, *King Henry IV, Part I*

Jetzt beschreiben wir, wie Ihre Applikationen mit dem Laufzeitsystem der Java Virtual Machine zusammenarbeiten und wie das Laufzeitsystem benutzt werden kann, um mit dem darunterliegenden Betriebsystem zusammen zu arbeiten.

Dies umfasst das Lesen von Systemeigenschaften, welche beispielsweise die Kommunikation zwischen dem Betriebsystem und dem Laufzeitsystem gestatten, insbesondere das Ausführen anderer Programme und das Herunterfahren des Laufzeitsystems.

Drei Klassen aus dem `java.lang` Package gestatten diese Zugriffe:

- die System Klasse stellt statische Methoden zur Verfügung, mit deren Hilfe der Systemzustand manipuliert werden kann. Sie können damit die Systemeigenschaften lesen und verändern, in einem eingeschränkten Rahmen. Zudem stellt diese Klasse die Standard Eingabe und Ausgabe Streams und einige weitere Hilfsmethoden zur Verfügung. Aus Bequemlichkeit agieren mehrere Methoden der `System` Klasse auf dem aktuellen `Runtime` Objekt.

- die `Runtime` Klasse stellt dem Laufzeitsystem ein Interface zum Laufzeitsystem der aktiven Virtual Machine zur Verfügung. Das aktuelle `Runtime` Objekt ermöglicht den Zugriff auf Funktionalitäten, wie beispielsweise dem Garbage Collector, zur Ausführung anderer Programme und zum Herunterfahren des Laufzeitsystems zur Verfügung.

- die `Process` Klasse stellt einen laufenden Prozess dar, der mittels der `Runtime.exec()` Methode gestartet wurde (also beispielsweise ein Prozess ausserhalb der JVM: mit `Runtime.exec(cmd/k)` können Sie beispielsweise die Shell von Windows NT starten, mit `Runtime.exec(winhelp.exe)` das Windows Help System).

Diese Zugriffe sind zum Teil so direkt ins Betriebsystem, dass die Sicherheitsfragen eine zentrale Rolle spielen. Daher müssen wir uns einmal mehr mit Sicherheitsfragen befassen.

## *1.1. Die System Klasse*

Die `System` Klasse stellt statische Methoden zur Verfügung, mit deren Hilfe der Systemzustand manipuliert werden kann. Die Klasse agiert sozusagen als Repository für systemweite Resssourcen. Die Klasse `System` stellt Funktionalitäten in den vier Bereichen zur Verfügung:

- die Standard I/O Streams
- Manipulation der System Properties
- Hilfsmittel und praktische Methoden für den Zugriff auf das aktuelle Runtime Objekt und
- Security

Schauen wir uns die einzelnen Themen genauer an.

### 1.1.1. Standard I/O Streams

Die Standard Eingabe, Ausgabe und Fehler Ströme sind als statische Felder der `System` Klasse definiert.

`public static final InputStream in`
: Standard Eingabestrom zum Einlesen von Daten.

`public static final PrintStream out`
: Standard Ausgabestrom zum Drucken von Nachrichten

`public static final PrintStream err`
: Standard Fehlerstrom für die Ausgabe von Fehlermeldungen. Der Benutzer kann die Standard Ausgabe in eine Datei umleiten. Aber es wird auch ein Ausgabekanal für die Fälle benötigt, in denen die Ausgabe in eine Datei nicht mehr möglich ist. `err` ist ein spezieller Ausgabekanal, in den auch jene Meldungen geschrieben werden sollen, die nicht zu einer Standardausgabe gehören.

Aus historischen Gründen sind sowohl `out` als auch `err PrintStream` Objekte., also nicht `PrintWriter` Objekte.

Obschon alle Standardströme als final deklariert sind, können Sie mit den Methoden `setIn`, `setOut` und `setErr` die aktuellen Ströme neu definieren. Dazu werden de facto Methoden eingesetzt, welche auf einem tieferen Level angesiedelt sind, als die Java Sprachroutinen. Aus Sicherheitsgründen sind diese Methoden geschützt und werfen SecurityException falls der Benutzer nicht die passenden Zugriffsrechte hat.

```
public static void setIO() {
    try {
      // START setIO
      System.setIn(new java.io.FileInputStream("meineEingabeDatei"));
      System.setOut(new PrintStream(new
      java.io.FileOutputStream("meineAusgabeDatei")));
      System.setErr(new PrintStream(new
      java.io.FileOutputStream("meineFehlerDatei")));
```

```
      System.out.println("Hallo Ausgabedatei");      // out
      System.err.println("Hallo Fehlerdatei");       // err
      // END setIO

      } catch (java.io.IOException e) {
          e.printStackTrace();
      }
}
```

### 1.1.2. System Properties

System Properties definieren die Systemumgebung. Diese werden von der `System` Klasse in einem `Properties` Objekt gespeichert. Property Namen bestehen aus mehreren Teilen, welche durch ein Komma getrennt werden. Beispielsweise liefert eine Liste aller Properties auf meinem Notebook:

```
package einfuehrendesbeispiel;

import java.util.Date;
import java.util.Properties;
import java.util.Enumeration;
import java.io.*;

public class Main {
  public static void main(String[] args) {
    Properties props = System.getProperties(); //  list  properties
    for (Enumeration enum = props.propertyNames(); enum.hasMoreElements();)
    {
      String key = (String)enum.nextElement();
      System.out.println(key + " = " + (String)(props.get(key)));
    }
  }
}
```

mit der Ausgabe:

```
java.runtime.name = Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path = C:\JBuilder4\jdk1.3\jre\bin
java.vm.version = 1.3.0-C
java.vm.vendor = Sun Microsystems Inc.
java.vendor.url = http://java.sun.com/
path.separator = ;
java.vm.name = Java HotSpot(TM) Client VM
file.encoding.pkg = sun.io
java.vm.specification.name = Java Virtual Machine Specification
user.dir = D:\NDKJava\Programme\lang\System\EinfuehrendesBeispiel
java.runtime.version = 1.3.0-C
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment
os.arch = x86
java.io.tmpdir = C:\TEMP\
line.separator =

java.vm.specification.vendor = Sun Microsystems Inc.
java.awt.fonts =
os.name = Windows NT
java.library.path =
C:\JBuilder4\jdk1.3\bin;.;C:\WINNT\System32;C:\WINNT;C:\WINNT\system32;C:\W
INNT;c:\jc211\bin;C:\Programme\Rational\common;C:\JBuilder4\jdk1.3\bin;
```

```
java.specification.name = Java Platform API Specification
java.class.version = 47.0
os.version = 4.0
user.home = C:\WINNT\Profiles\zajoller.000
user.timezone =
java.awt.printerjob = sun.awt.windows.WPrinterJob
file.encoding = Cp1252
java.specification.version = 1.3
user.name = joller
java.class.path =
D:\NDKJava\Programme\lang\System\EinfuehrendesBeispiel;C:\JBuilder4\jdk1.3\
lib\imap.jar;C:\JBuilder4\jdk1.3\lib\mail.jar;C:\JBuilder4\jdk1.3\lib\maila
pi.jar;C:\JBuilder4\jdk1.3\lib\pop3.jar;C:\JBuilder4\jdk1.3\lib\smtp.jar;C:
\JBuilder4\jdk1.3\lib\activation.jar;C:\JBuilder4\jdk1.3\lib\concurrent.jar
;C:\commapi\comm.jar;C:\JBuilder4\lib\cx.jar;C:\JBuilder4\lib\dx.jar;C:\JBu
ilder4\lib\beandt.jar;C:\JBuilder4\lib\dbswing.jar;C:\JBuilder4\lib\interne
tbeans.jar;C:\JBuilder4\lib\jaxp.jar;C:\JBuilder4\lib\parser.jar;C:\JBuilde
r4\lib\jbcl.jar;C:\JBuilder4\lib\jdsremote.jar;C:\JBuilder4\lib\jdsserver.j
ar;C:\JBuilder4\lib\jds.jar;C:\JBuilder4\samples\OpenToolsAPI\layoutassista
nt\classes;C:\JBuilder4\lib\jbuilder.jar;C:\JBuilder4\lib\help.jar;C:\JBuil
der4\lib\gnuregexp.jar;C:\JBuilder4\lib\servlet.jar;C:\JBuilder4\jdk1.3\dem
o\jfc\Java2D\Java2Demo.jar;C:\JBuilder4\jdk1.3\jre\lib\i18n.jar;C:\JBuilder
4\jdk1.3\jre\lib\jaws.jar;C:\JBuilder4\jdk1.3\jre\lib\rt.jar;C:\JBuilder4\j
dk1.3\jre\lib\sunrsasign.jar;C:\JBuilder4\jdk1.3\lib\dt.jar;C:\JBuilder4\jd
k1.3\lib\tools.jar
java.vm.specification.version = 1.0
java.home = C:\JBuilder4\jdk1.3\jre
user.language = de
java.specification.vendor = Sun Microsystems Inc.
awt.toolkit = sun.awt.windows.WToolkit
java.vm.info = mixed mode
java.version = 1.3.0
java.ext.dirs = C:\JBuilder4\jdk1.3\jre\lib\ext
sun.boot.class.path =
C:\JBuilder4\jdk1.3\jre\lib\rt.jar;C:\JBuilder4\jdk1.3\jre\lib\i18n.jar;C:\
JBuilder4\jdk1.3\jre\lib\sunrsasign.jar;C:\JBuilder4\jdk1.3\jre\classes
java.vendor = Sun Microsystems Inc.
file.separator = \
java.vendor.url.bug = http://java.sun.com/cgi-bin/bugreport.cgi
sun.cpu.endian = little
sun.io.unicode.encoding = UnicodeLittle
user.region = CH
sun.cpu.isalist = pentium_pro+mmx pentium_pro pentium+mmx pentium i486 i386
```

Diese Properties sind auf allen Systemen definiert, allerdings variieren die Werte und die Schlüssel von Plattform zu Plattform. Einige der Standard Properties werden auch in den Standard Packages eingesetzt. Beispielsweise verwendet die `File` Klasse die `file.separator` Property, um Pfadnamen zu zerlegen.

Das folgende Beispiel benutzt die Properties, um das Home Verzeichnis des Benutzers zu bestimmen und zu verwenden:

```
public static class personalConfig(String fileName) {
      String home = System.getProperty("user.home");
      if (home == null)
            return null;
      else
            return new File(home, fileName);
}
```

Die Methoden der System Klasse, welche sich mit den Systemeigenschaften befassen, sind:

`public static Properties getProperties()`
liefert das `Properties` Objekt, welches die Systemeigenschaften beschreibt

`public static String getProperty(String key)`
liefert den Wert der Systemeigenschaft zum Schlüssel `key`.

`public static Properties getProperty(String key, String defaultValue)`
liefert den Wert der Systemeigenschaft `key`, falls vorhanden. Falls kein Wert vorhanden ist, wird der Standardwert zurückgegeben.

`public static Properties setProperty(String key, String value)`
setzt den Wert der Systemeigenschaft zum Schlüssel `key` auf den Wert `value` und liefert den alten Wert zurück. Falls vorgängig kein Wert definiert war,wird das `null` Objekt zurückgeliefert.

`public static Properties setProperties(Properties props)`
setzt das `Properties` Objekt gemäss dem Objekt props und definiert alle Systemeigenschaften.

```
package setproperty;

public class Main {
    public static void main(String[] args) {
//
      String oldVal = System.setProperty("user.dir", "D:\\Temp");
      System.out.println("Alter Wert: " + oldVal);
      System.out.println("Neuer Wert: " + System.getProperty("user.dir"));
//
    }
}
```

liefert die Ausgabe:

```
Alter Wert: D:\NDKJava\Programme\lang\System\setProperty
Neuer Wert: D:\Temp
```

Beispiel für den Einsatz von Standardwerten

```
      {
            // START getprop
            // user's home directory
            String homeDir = System.getProperty("user.home");
            // falls unbekannt, verwende 'homeDir' als Default
            String outDir = System.getProperty("testdir", homeDir);
            // END getprop
            System.out.println("homeDir: " + homeDir);
            System.out.println("outDir: " + outDir);
      }
```

Alle diese Methoden werden sicherheitsmässig überprüft und werfen eine SecurityException, falls die Sicherheitsvorschriften verletzte werden. Dabei kann es durchaus sein, dass Sie die Systemeigenschaften nicht setzen, wohl aber abfragen können. Auch könnte es sein, dass Sie lediglich Zugriff auf bestimmte Systemeigenschaften haben.

Systemeigenschaften werden als Zeichenketten gespeichert, aber die Zeichenketten können auch Zahlen oder Boole'sche Werte enthalten. Verschiedene Methoden gestatten das Lesen der Systemeigenschaften und umsetzen in Basisdatentypen. Diese Methoden sind in den Wrapperklassen der Basisdatentypen beschrieben. Diese Entschlüsselungsmethoden sind in der Regel statische Methoden der Basisdatentypklasse. Jede Methode enthält sozusagen als Schlüssel eine Zeichenkette, mit dessen Hilfe die Eigenschaft benannt werden kann. Einige Methoden besitzen einen zweiten Parameter. Dieser entspricht dem Standardwert der Eigenschaft, und wird zurückgeliefert, falls die Eigenschaft nicht gefunden wird. Falls die Methode keinen Standardwert als Parameter vorsieht, liefern den Standardwert als Rückgabewert. Alle diese Methoden verwenden folgende Methoden, um Datentypen in Basisdatentypen umzuwandeln:

```
public static boolean Boolean.getBoolean(String name)
public static Integer Integer.getInteger(String name)
public static Integer Integer.getInteger(String name, Integer def)
public static Integer Integer.getInteger(String name, int def)
public static Long Long.getLong(String name)
public static Long Long.getLong(String name, Long def)
public static Long Long.getLong(String name, long def)
```

`def` steht dabei für den Standardwert.

Die `getBoolean` Methode unterscheidet sich von den andern - sie liefert einen `boolean` Wert anstelle eines Objekts der Klasse `Boolean`. Falls eine Eigenschaft fehlt, liefert `getBoolean()` den Wert `false`; alle anderen Methoden liefert den `null` Wert.

Die Klassen `Character, Byte, Short, Float` und `Double` besitzen keine eigenen Methoden zur Bestimmung der Systemeigenschaften. Sie können die Werte als Zeichenketten bestimmen und mittel Zeichenkettenkonversionen diese in die passenden Basisdatentypen umwandeln.

### 1.1.3. Hilfsmethoden

Die System Klasse enthält eine Anzahl Hilfsmethoden:

`public static long currentTimeMillis()`

liefert die aktuelle Zeit in Millisekunden seit Epoch = 00:00:00 GMT, Januar 1, 1970. Die Zeit wird als long dargestellt. Falls Sie eine detaillliertere Kontrolle über die Zeit und Zeitformate benötigen, können Sie weitere Klassen, beispielsweise `Calendar, TimeZone` oder `DateFormat` einsetzen.

`public static void arraycopy(Object src, int srcPos, Object dst, int dstPos, int count)`

kopiert den Inhalt des `src` Arrays, ab Position `srcPos`, auf das `dst` Array, ab Position `dstPos`. Insgesamt werden `count` Elemente kopiert werden. Falls Sie dabei die Arraygrenzen überschreiten, wird eine `IndexOutOfBoundsException` geworfen. Falls die Werte im Ausgangsdatenfeld inkompatibel mit dem Zieldatenfeld sind, wird eine `ArrayStoreException` geworfen. "Kompatibel" heisst in diesem Zusammenhang, dass jedes Objekt aus dem Ausgangsdatenfeld einem Komponententyp des Zieldatenfeldes zugeordnet werden kann, typenmässig. Falls der Datentyp ein Basisdatentyp ist, müssen die Datentypen beider Datenfelder identisch

sein. `arraycopy()` können Sie also nicht einsetzen, um beispielsweise ein `short` Datenfeld auf ein `int` Datenfeld abzubilden, zu kopieren. Die Methode `arraycopy()` können Sie auch einsetzen, um Teile eines Datenfeldes auf andere Teile des selben Datenfeldes zu kopieren. Das heisst, Sie können Teile eines Datenfeldes verschieben, beispielsweise an den Anfang oder das Ende des Datenfeldes.

`public static int identityHash(Object obj)`

liefert einen Hash-Code für das Objekt obj mit Hilfe des Algorithmus in `Object.hashcode()`. Sie haben damit die Möglichkeit auf einer tieferen Ebene auf den systemeigenen Hash-Code zuzugreifen, selbst wenn die Methode `hashCode()` in der Klasse von `obj` überschrieben wird. Deswegen heisst die Methode auch `identity...()`.

Eine Anzahl anderer Methoden in der `System` Klasse wurden hinzugefügt, um Ihnen das Leben beim Umgang mit diesen Objekten - beispielsweise dem aktuellen `Runtime` Objekt - zu vereinfachen.

Das `Runtime` Objekt können Sie mit der statischen Methode `Runtime.getRuntime()` bestimmen. Damit sind Sie in der Lage Methoden der Klasse `Runtime` auszuführen: `Runtime.getRuntime().`*`methode()`*. Diese Methoden sind:

```
public static void exit(int status)
public static void gc()
public static void runFinalization()
public static void loafLibrary(String libname)
public static void load(String filename)
```

## *1.2. Kreieren von Prozessen*

Ein laufendes System kann jede Menge Threads besitzen, gleichzeitig natürlich. Die meisten Systeme, auf denen eine Java VM läuft, können auch gleichzeitig mehrere Programme ausführen. Sie können aus einem Java Programm heraus weitere Prozesse, auf VMs starten, beispielsweise mit der `Runtime.exec()` Methode. Jeder erfolgreiche Aufruf von `exec()` kreiert ein neues Process Objekt, welches ein Programm als eigenständigen Prozess beschreibt. Mit dem Process Objekt können Sie den Zustand des Prozesses abfragen oder Methoden aufrufen, welche dessen Ausführung beeinflussten. `Process` ist eine abstrakte Klasse, deren Unterklassen auf jeder Plattform mit deren plattformabhängigen Prozessen zusammenarbeiten muss.

Es existieren zwei Grundformen der `exec()` Methode:

`public Process exec(String[ ] cmdArray) throws IOException`
: führt den Befehl `cmdArray` auf dem aktuelen System aus. Die Methode liefert ein `Process` Objekt (siehe oben), welches den ausführenden Prozess beschreibt. Die Zeichenkette `cmdArray[0]` enthält den Namen des Befehls, die weiteren Elemente des Datenfeldes werden dem Befehl als Parameter mitgegeben.

`public Process exec(String command) throws IOException`
: diese Form ist äquivalent zur obigen Methode, ausser dass die einzelnen Datenfeldelemente mit einem `StringTokenizer` implizit zerlegt wird.

Der neu kreierte Prozess wird als *child* Prozess bezeichnet, als Unterprozess. Analog zu dieser Bezeichnung nennt man den kreierenden Prozess den *parent* Prozess.

Das Kreieren eines Prozesses ist eine privilegierte Operation und eine `SecurityException` wird geworfen, falls Sie keine Berechtigung haben, Unterprozesse zu generieren. Die Ausnahme ist nicht gerade vielsagend: `IOException`. Das hängt damit zusammen, dass das Kreieren des Unterprozesses mit IO zu tun hat.

### 1.2.1. Die Process Klasse

Die `exec()` Methode liefert ein `Process` Objekt zurück, je eines pro *child*-Prozess. Dieses Objekt repräsentiert den *child*-Prozess in zweierlei Hinsicht:

1. es stellt Methoden zur Verfügung, um Eingabe-, Ausgabe- und Fehler- Ströme für den *child*-Prozess zu bestimmen.
2. es stellt Methoden zur Verfügung, um den Prozess zu kontrollieren und dessen Status zu erfahren, beispielsweise den Abschluss, das Beenden des Prozesses.

`public abstract OutputStream getOutputStream()`
: liefert einen `OutputStream`, der mit dem Eingabestrom des child-Prozesses verbunden ist. Die Daten, welche in diesen Strom geschrieben werden, werden vom child-Prozess als Eingabe gelesen.

`public abstract InputStream getInputStream()`

liefert den `InputStream`, der mit dem Ausgabestrom des child-Prozesses verbunden ist. Die Daten, welche in diesem Strom vorhanden sind, wurden vom child-Prozess als Ausgabe produziert.

`public abstract InputStream getErrorStream()`

liefert einen `InputStream`, der mit dem Fehlerausgabestrom des child-Prozesses verbunden ist. Die Daten, welche in diesem Strom vorhanden sind, wurden in der Regel in Fehlern des child-Prozesses generiert.

Hier ein Beispiel:

```
package processbeispiel;

import java.io.*;

public class ProcessAnwendungsbeispiel {
    public static void main(String[] args) {
        //
        try {
            String cmd = "cmd.exe /c FTYPE";
            // Windows : liefert Dateityp - Pgm Mapping
            Process child = Runtime.getRuntime().exec(cmd);
            InputStream in = child.getInputStream();
            int c;
            while ((c = in.read()) != -1) {
                System.out.print((char)c);
            }
            in.close();
            //
            try {
                child.waitFor();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //
            System.out.println("child mit Status " + child.exitValue()+"
abgeschlossen");
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

Komplexer wäre beispielsweise die Definition von Threads, welche die Ein/Ausgabe umleiten:

```
public static Process userProg(String cmd) throws IOException {
      Process proc = Runtime.getRuntime().exec(cmd);
      plugTogether(System.in, proc.getOutputStream() ); // Thread verknüpft
      plugTogether(System.out, proc.getInputStream() );
      plugTogethet(System.err, proc.getERrorStream() );
      return proc;
}
```

wobei das Problem die Definition der Verbindungsmethode der jeweils zwei Streams ist.

Die zweite Art und Weise ein Process Objekt für den child-Prozess zu beschreiben, benutzt folgende Methoden zur Kontrolle des Prozesses und seinem Abschluss:

`public abstract int waitFor() throws InterruptedException`
wartet unbestimmte Zeit, bis der Prozess beendet ist und liefert den Abschlusswert, der beispielsweise durch `System.exit(int i)` gesetzt wurde: 0 bedeutet erfolgreicher Abschluss; jeder andere Wert bedeutet einen Fehler.

`public abstract int exitValue()`
liefert den Abschlusswert für diesen Prozess. Falls der Prozess noch nicht abgeschlossen / beendet ist, wird eine `IllegalThreadStateException` geworfen.

`public abstract void destroy()`
hält den Prozess an. Falls der Prozess schon beendet ist, geschieht nichts. Garbage Collection des Prozess Objekts besagt nicht, dass der Prozess zerstört wurde: es besagt lediglich, dass auf den Prozess nicht mehr zugegriffen werden kann.

Hier ein weiteres Beispiel, welches ein Windows Programm startet:

```
package starteinesexefiles;

public class RunTimeWindows {
    public static void main (String[] args) {
        Runtime rt = Runtime.getRuntime();
        String strDir = "cmd.exe /k cmd.exe"; // /K = Fenster offen lassen
        try {
            System.out.println("Total Memory : "+rt.totalMemory());
            System.gc();
            System.out.println("Freier Speicher :"+rt.freeMemory());
            System.out.println(" traceInstructions und traceMethods wird
                                eingeschaltet ");

            rt.traceInstructions(true);
            rt.traceMethodCalls(true);
            rt.exec(strDir);
            // einige Beispiele
            //rt.exec ("cp test.java dd.txt"); // für Unix
            //rt.exec("cmd.exe /c copy eins.txt dd.txt");// für Windows
            //rt.exec("explorer.exe");
            //rt.exec("winfile.exe");
            //rt.exec("c:\\Programme\\Plus!\\Microsoft
                     Internet\\Iexplore.exe");

            //rt.exec("c:\\Programme\\Netscape\\Communicator\\Program\\Nets
            cape.exe");
            //rt.exec("posix.exe"); //posix shell starten
            //rt.exec("winchat.exe"); // RAS chat System starten
            rt.exec("c:\\Programme\\WinAmp\\winamp.exe
                  c:\\WinNT\\Media\\Windows NT-Abmeldeklang.wav");
         } catch (Exception e) {
            System.out.println(" Fehler beim Ausführen des Windows / DOS
                        Befehls : "+e.getMessage()+"\n");
            e.printStackTrace ();
         }
    }
}
```

Das Programm setzt voraus, dass Sie die WinAmp Applikation installiert haben.

Falls Sie eine der Command Optionen oder Befehle im DOS Fenster ausführen wollen, ist es hilfreich, wenn Sie dies erst einmal von Hand üben, also ausserhalb von Java, zumindest bin ich so zu den lauffähigen Beispielen gelangt.

Hier ein Beispiel für den Einsatz der destry() Methode: das erste Beispiel, welches wir betrachtetetn, lieferte sehr viel Informationen. Wir wollen aber nur die ersten paar Zeilen:

```
package destroychild;
import java.io.*;

public class distroyBeispiel {
    public static void main(String[] args) {
        try {
            String cmd = "cmd.exe /c FTYPE";
            // fünf Zeilen lesen und dann den child Prozess zerstören
            Process child = Runtime.getRuntime().exec(cmd);
            InputStream in = child.getInputStream();
            int c, newline = 0;
            while ((c = in.read()) != -1 && newline < 5) {
                char ch = (char)c;
                System.out.print(ch);
                if (ch == '\n')
                    ++newline;
            }
            in.close();
            System.out.println("destroy() : ");
            child.destroy(); // Prozess killen
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

`Process` ist eine abstrakte Klasse. Jede Implementation einer Java Virtual Machine kann eine oder mehrere Implementationen dieser Klasse mitliefern, je nach Betriebsystem. Diese Klassen können auch gleich weitere Methoden und Datenfelder anbieten, welche für die betreffende Plattform Sinn machen. Es liegt an Ihnen herauszufinden, welche Funktionalitäten auf Ihrer Plattform angeboten werden.

Beachten Sie, dass Sie keine Kontrolle darüber haben, ob der neu generierte Prozess asynchron abläuft oder in irgend einer Art und Weise mit Ihrem Programm synchronisiert wird. Alle diese Fragen sind plattformabhängig.

**Selbsttestaufgabe 1**

Bestimmen Sie mittels des `Reflection` Interfaces die Methoden und Datenfelder eines Prozess Objekts auf Ihrer Entwicklungsplattform.

Nochmal:
falls Ihre Beispiele oder Programme nicht so laufen, wie Sie es erwarten, sollten Sie die Kommandos ausserhalb des Java Programms ausführen, um besser sehen zu können, was passiert.

### 1.2.2. Prozess Umgebungen

Zwei andere Formen von `Runtime.exec(...)` gestatten Ihnen auch noch Umgebungsvariablen zu setzen. Diese Formen sind plattformabhängig und gestatten es, Schlüssel/Werte Paare zu setzen. Umgebungsvariablen können auch vom neuen Prozess abgefragt werden. Sie werden als Zeichenketten Datenfelder an die Methode `exec(...)` übergeben. Jedes Element des Datenfeldes definiert ein *name=wert* Paar. Der Name darf keine Leerzeichen enthalten; der Wert darf aber eine beliebige Zeichenkette sein. Die Umgebungsvariablen werden als zweiter Parameter aufgeführt:

```
public Process exec(String[] cmdArray, String[] env) throws IOException
public Process exec(String command, String[] env) throws IOException
```

Falls man als `env` das `null` Element übergibt, erhält man das selbe Ergebnis wie beim Aufruf ohne `env` Variable.

Umgebungsvariablen werden, wie oben erwähnt, plattformabhängig interpretiert. Sie können beispielsweise den Benutzernamen, das aktuelle Verzeichnis, Pfade oder andere Informationen umfassen, Informationen, die von laufenden Programmen benutzt werden können. Die Werte der Umgebungsvariable können Sie auf der einen Seite mit der System.getEnv() abfragen, obschon diese Methode nicht mehr in Gnade ist. Der bevorzugte Mechanismus für das Verwalten solcher Zusatzinformationen sind neu die Properties. Aber der Umgebungsvariablenmechanismus ist eigentlich universeller, Java unabhängig.

Es gleiben noch zwei weitere Formen von `exec(...),` mit dem das Arbeitsverzeichnis des Subprozesses definiert werden kann:

```
public Process exec(String[] cmdArray, String[] env, File dir)
      throws IOException
public Process exec(String command, String[] env, File dir)
      throws IOException
```

Dem Subprozess wird ein Arbeitsverzeichnis mitgegeben in Form eines Verzeichnisses. Falls `dir null` ist, erbt der Subprozess das aktuelle Verzeichnis des parent-Prozesses - also der Wert, der dem `user.dir` entspricht.

### 1.2.3. Portabilität

Jedes Programm, welches `exec(..)` einsetzt, ist nicht portabel. Nicht alle Betriebsysteme kennen nebenläufige Programme und Umgebungsvariablen. Auch die Befehle für einzelne Betriebsystemfunktionen sind von System zu System unterschiedlich. Sie müssen daher entsprechend vorsichtig sein beim Einsatz der `exec(..)` Methode.

**Selbsttestaufgabe 2**

Schreiben Sie ein Programm, welches Ihr Lieblingsprogramm startet. Ist es möglich eine neue Windows Shell zu kreieren, also Windows On Windows (WoW)?

## *1.3. Shutdown*

Normalerweise wird die Ausführung einer virtuellen Maschine beendet, sobald der letzte Benutzerthread beendet ist. Ein `Runtime` Objekt kann auch explizit heruntergefahren werden, indem die `exit()` Methode mit einem integer Statuscode aufgerufen wird, der auch an das Umgebungssystem der virtuellen Maschine weitergegeben wird - Null, falls der Abschluss erfolgreich ist, ungleich Null, falls der ein Fehler auftrat.

Die Methode `exit()` beendet die Ausführung des Laufzeitsystems abrupt und alle Threads des Systems werden beendet. Die Threads werden nicht beendet oder gestoppt, die Threads existieren einfach nicht mehr. Es wird auch kein `finally()` ausgeführt.

Sobald die `exit()` Methode ausgeführt wird oder alle Benutzerthreads beendet werden, beginnt der Shutdown Prozess. Die virtuelle Maschine kann auch extern gestoppt werden, beispielsweise durch Eingabe von CTRL / C oder wenn der Benutzer aus dem System ausloggt.

Alle Methoden, welche mit dem Shutdown der virtuellen Maschine zusammenhängen sind geschützt und werfen eine `SecurityException`, falls Sie dafür keine Privilegien haben.

### 1.3.1. Shutdown Hooks

Eine Applikation kann einen sogenannten *shutdown hook* beim Laufzeitsystem registrieren. Es gibt Threads, welche Aktionen repräsentieren, welche auf jeden Fall ausgeführt werden müssen, bevor die virtuelle Maschine beendet wird. Hooks räumen typischerweise externe Ressourcen auf, beispielsweise Dateien schliessen oder Netzwerkverbindungen abschliessen.

`public void addShutdownHook(Thread hook)`
: registriert einen neuen VM Hook. Falls der Hook schon registriert worden war oder schon gestartet wurde, wird eine `IllegalArgumentException` geworfen.

`public boolean removeShutdownHook(Thread hook)`
: löscht die Registration eines Threads bei der VM als Shutdown Hook. Die Methode liefert `true`, falls der Hook registriert war und gelöscht werden konnte. Falls der Hook vorgängig nicht registriert war, wird `false` zurückgeliefert.

Sie können keine Hooks mehr hinzufügen oder löschen sobald der Shutdown Prozess angefangen hat. Falls Sie dies trotzdem tun, wird eine `IllegalStateException` geworfen.

Beim Herunterfahren der virtuellen Maschine wird die start() Methode aller Hooks aufgerufen. Die Reihenfolge der Ausführung der einzelnen Threads ist unbestimmt.

Das Fehlen einer Ausführungsordnung kann zu Problemen führen, beispielsweise beim Speichern eines Zustandes in einer Datenbank. Beispielsweise könnte einer der Hooks die Datenbank herunterfahren. Gleichzeitig kann aber auch ein Hook dafür definiert sein, Daten persistent abzuspeichern. Damit könnte ein Konflikt auftreten. In diesem einfachen Beispiel ist die Lösung einfach: Sie könnten beispielsweise die zwei Threads zusammenlegen : der

Thread speichert zuerst den Zustand und fährt anschliessend die Datenbank herunter, sofern die Zeit reicht.

Die Shutdown Hooks sind auch gleichzeitig mit anderen Threads aktiv, es treten also alle Probleme auf, die in einem Multithreading -System auftreten können. Selbst wenn das Herunterfahren der VM nach Abschluss aller Benutzerthreads gestartet wird, können noch Daemon Threads im System aktiv bleiben. Anders sieht die Situation aus, wenn Sie die `exit(..)` Methode aufrufen: in diesem Fall sind die Benutzerthreads und Daemonthreads aktiv. Es liegt bei Ihnen dafür zu sorgen, dass keine Deadlocks auftreten.

Alle Shutdown Hooks müssen schnell ausgeführt werden. Falls der Benutzer beispielsweise ein Programm unterbricht, erwartet er, dass das Programm möglichst schnell beendet wird. Falls ein Benutzer aus dem Rechner auslogged, erwartet er, dass das System sich möglichst schnell verabschiedet.

Genauso muss eine VM beim Herunterfahren möglichst schnell alle Hooks ausführen.

### 1.3.2. Die Shutdown Sequenzen

Das Herunterfahren wird nach Abschluss des letzten Benutzerthreads initialisiert. Dies geschieht auch, falls `Runtime.exit()` ausgeführt wird, oder durch externe Einflüsse die Ausführung der VM gestoppt wird. Sobald das Herunterfahren der VM eingeleitet wird, starten alle Hook Threads. Falls einer der Hook Threads abbricht, wird das Herunterfahren gestoppt. Falls das Herunterfahren intern ausgelöst wurde, wird die VM nicht heruntergefahren. Falls die VM extern gestoppt wurde, wird eine erzwungene Beendigung der VM die Folge sein.

Falls beim Herunterfahren eine nicht abgefangene Ausnahme eintritt, wird die `uncaughtException` Methode der Threadgruppe ausgeführt. Der Shutdown Prozess wird weitergeführt, es tritt kein Unterbruch ein.

Nachdem alle Hooks abgeschlossen sind, wird die `halt()` Methode ausgeführt. Diese Methode beendet die VM, sie existiert danach nicht mehr. Sie sollten allerdings nicht versuchen, die halt() Methode direkt auszuführen - einzelne Hooks könnten unterbrochen werden. `halt(..)` akzeptiert einen Parameter, den Returncode mit der selben Bedeutung wie bei `exit(..)`. Ein Hook kann die `halt()` Methode aufrufen, aber eben mit Vorsicht. In der Regel ist der Aufruf von `halt(..)` das Dümmste was Sie tun können.

Falls Sie `exit()` *während des Herunterfahrens* aufrufen, wird dies zu einer Blockade führen. Daher sollten Sie auch dies vermeiden.

Die VM wird selten abbrechen, ohne sauber herunterzufahren. Dies könnte beispielsweise beim Auftreten eines internen Fehlers passsieren. Aber es kann auch sein, dass das Wirtesystem, beispielsweise Unix, ... einen Absturz der VM zur Folge hat. In diesen Fällen kann man nicht garantieren, dass das System in einem definierten Zustand abgeschlossen wird.

## *1.4. Der Rest von Runtime*

Die Runtime Klasse stellt weitere Funktionen zur Verfügung, insgesamt in den folgenden fünf Bereichen:

- die Interaktion mit dem Garbage Collection
- das Ausführen externer Programme
- das beenden des Laufzeitsystems
- das Laden von Code Libraries
- Debugging

Die ersten drei Punkte haben wir schon besprochen (`...gc()`, `...exec(...)`, `...exit(...)`). Nun befassen wir uns mit den zwei restlichen Punkten.

### 1.4.1. Das Laden von Code Bibliotheken

Wir haben bereits gesehen, wie man Java Klassen dynamisch laden und entladen kann. Auch Module in einer Java externen Sprache können geladen werden (native Code).

Neben Modulen kann man auch ganze Bibliotheken laden. Dies geschieht allerdings auf eine plattformabhängige Art und Weise.

Die Methoden, welche Runtime dafür zur Verfügung stellt, sind:

`public void loadLibrary(String libname)`
: lädt die dynamische Bibliothek mit dem Namen libname. Die Bibliothek entspricht einer Datei im lokalen Dateisystem. Die aktuelle Zurodnung des Bibliotheknamens zur Bibliothek ist also plattformabhängig.

`public void load (String filename)`
: lädt die Datei mit dem Dateinamen filename als eine dynamische Bibliothek. Im Gegensatz zur obigen Methode gestattet es diese Variante die Bibliothek irgendwo im Dateisystem anzugeben

Typischerweise laden Klassen, die den direkten Zugriff auf Systembibliotheken benötigen, als Teil ihrer Initialisierung die dynamischen Bibliotheken. Das Laden selbst ist eine privilegierte Aktion. Falls die Bibliothek nicht geladen werden kann, wird ein `UnsatisfiedLinkError` geworfen.

Im der Systemklasse existiert eine verwandte Methode, `mapLibraryName()`, welche eine Bibliothek auf eine systemseitige Bibliothek abbildet.

Beispielsweise wird der Bibliotheksname "awt" auf "awt.dll" abgebildet (unter Windows bzw. "lobawt.so" unter Unix).

### 1.4.2. Debugging

Zwei Methoden des Laufzeitsystems werden für das Debugging eingesetzt:

`public void traceInstructions(boolean on)`
: schaltet das Verfolgen der Ausführung von Instruktionen auf der VM ein oder aus. Das muss allerdings nicht automatisch dazu führen, dass die VM irgendwelche Ausgaben produziert.

`public void traceMethodcalls(boolean on)`
: schaltet das Verfolgen der Methodenaufrufe ein oder aus. Auch hier bedeutet das Einschalten der Verfolgung, dass die VM Ausgaben produzieren sollte. Dies hängt aber von der VM ab.

Die Ausgabe der Debugging Information erfolgt je nach Plattform. Jede VM besitzt bestimmte Charaktersitiken, welche die obigen Methoden wirkungslos werden lassen.

## *1.5. Security*

Sicherheit ist ein sehr komplexes Thema. Deswegen wird das Thema auch anderswo detaillierter besprochen. Das Java Team hat die Sicherheitsarchitektur in einem Buch:

*Inside Java 2 Platform Security: API, Design and Implementation*

festgehalten. Dieses Buch enthält alle Details zum Thema. Hier geht es um eine Übersicht, nicht so sehr der allgemeinen Themen, da diese im Security Text stehen, sondern aus Sicht der APIs, der Programmierschnittstellen.

Um sicherheitsüberprüfte Operationen ausführen zu können, benötigen Sie die Erlaubnis (*Permission*) für diese Operationen. Alle Rechte werden in der *Security Policy* festgelegt. Ein *Protection Domain* umschliesst ein Set von Klassen, deren Instanzen bestimmte Rechte gegeben werden und die alle von der selben Codequelle (*code source*) stammen. Protection Domains, Schutzdomänen also, werden mittels Class Loadern festgelegt. Um die Sicherheitsrichtlinien einzuschalten und die Schutzdomänen zu aktivieren, müssen Sie einen *Security Manager* installieren.

Die Klassen und Interfaces, welche für das Einhalten der Sicherheitrichtlinien verwendet werden, sind über mehrere Packages verteilt.

### 1.5.1. Die SecurityManager Klasse

Die java.lang.SecurityManager Klasse gestattet es den Applikationen Sicherheitsrichtlinien mitzugeben, eine Security Policy. Der SecurityManager überprüft bei vielen Operationen als erstes, ob diese oder jene Operation gesattet ist. beim Starten einer Applikation kann der SecurityManager mehrfach aufgerufen werden, Grund ist , dass die Security Richtlinien mehr oder weniger pro Objekt definiert werden können und innerhalb der Objekte auch noch differenziert werden kann.

Die `SecurityManager` Klasse enthält viele Methoden, welche mit check...() beginnen. Diese Methoden werden durch andere Methoden aufgerufen, um sensitive Operationen zu schützen. Beispielsweise wird vor dem Dateizugriff zuerst geprüft, ob Sie oder das Programm Zugriffsrechte oder Modifikationsrechte für diese Datei besitzen.

Typischerweise sehen solche Prüfungen folgendermassen aus:

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
      security.checkXXX(argument, ...);
}
```

Der Security Manager hat also eine Chance die Gültigkeit eines Aufrufes einer Methode zu verhindern, falls Sie dazu nicht berechtigt sind. In diesem Fall würde der Security Manager eine Ausnahme werfen: `SecurityException`.

Der aktuelle Security Manager kann mittels Systemklassenmethoden bestimmt werden:

`public static void setSecurityManager(SecurityManager s)`
: setzt den Security Manager mittels des selbstdefinierten Security Managers. Falls bereits ein Security Manager existiert und gesetzt wurde, wird diese überschrieben, falls die Sicherheitsrichtlinien es Ihnen gestatten den Security Manager zu ersetzen. Falls dies nicht der Fall ist, wird eine `SecurityException` geworfen.

`public static void SecurityManager  getSecurityManager()`
: liefert den Systemsecuritymanager. Falls keiner gesetzt wurde, wird `null` als Objekt geliefert und es wird angenommen, dass Sie alle Rechte besitzen.

Der Security Manager delegiert die aktuellen Sicherjeitstests an ein Zugriffskontrollobjekt. Jede `check...()` Methode benutzt den Security Manager, speziell dessen `checkPermission()` Methode, an die das passende `java.security.Permission` Objekt für diese betreffende Methode übergeben wird. Die Standard Implementation der `checkPermission()` Methode ruft dann die

```
java.security.AccessController.checkPermission(perm);
```

Falls ein angefragter Zugriff gestattet ist, dann wird checkPermission() einfach beendet. Falls dies nicht der Fall ist, wird eine `java.security.AccessControllException` geworfen, eine Unterklasse der `SecurityException`.

Diese Art der Sicherheitsüberprüfung findet immer im Rahmen, Kontext, des aktuellen Threads statt - also der Schutzdomäne (*protection domain*), in der sich der Thread befindet. Eine Schutzdomäne ist eine Menge bestehend aus Klassen,welche alle die selben Sicherheitsprivilegien besitzen. Deswegen kann ein Thread auch gleichzeitig zu mehreren Schutzdomänen gehören:

falls er Methoden von Objekten verwendet,welche zu verschiedenen Schutzdomänen gehören.

Die Methode `getSecurityContext()` der SecurityManager Klasse liefert den Sicherheits-Kontext für einen bestimmten Thread in Form eines `java.security.AccessControlContext` Objekts. Diese Klasse definiert auch eine `checkPermission()` Methode, aber konzeptionell besteht ein Unterschied zu den checkXXX() Methoden des Security Managers:

- die `checkPermission()` Methode des `AccessControlContext` Objekts überprüft die Sicherheit im Rahmen des Kontexts

- die `checkXXX()` Methoden des Security Managers überprüfen die Sicherheit im Rahmen des Threads.

Der Kontext spielt beispielsweise dann eine Rolle, wenn Aufgaben an einen Thread delegiert werden und dieser Thread in einem Kontext ist, in dem er viele Privilegien besitzt. Der andere Thread, jener der die Aufgabe an diesen delegiert, kann aber völlig unprivilegiert sein. Daher müssen die Kontexte gegenseitig ihre Rechte und Sicherheitsvorschriften abstimmen.

Im einfachsten Fall verwendet die `checkPermission()` Methode des Kontextes zwei Parameter:
ein `Permission` Objekt und ein `AccessControlContext` Objekt. Allerdings sind auch komplexere Konstrukte vorgesehen. Auf diese werden wir noch eingehen.

### 1.5.2. Permissions

Permissions, Rechte, können in mehrere Kategorien eingeteilt werden. Beispielsweise:
- File - `java.io.FilePermission`
- Netzwerk - `java.net.NetPermission`
- Properties - `java.util.PropertyPermission`
- Reflection - `java.lang.reflect.ReflectPermission`
- Runtime - `java.lang.RuntimePermission`
- Security - `java.security.SecurityPermission`
- Serialization - `java.io.SerializablePermission`
- Sockets - `java.net.SocketPermission`

Alle ausser `FilePermission` und `SocketPermission` sind Unterklassen der `java.security.BasicPermission`, welche selbst eine abstrakte Unterklasse der top-level Klasse aller Permissionklassen, der `java.security.Permission` Klasse ist. `BasicPermission` definiert einfache Rechte basierend auf einfachen Namen. Beispielsweise repräsentiert "exitVM" die `RuntimePermission`, um die Methode `Runtime.exit()` aufzurufen.

Hier einige weitere Rechte und deren Klassen:
- "createClassLoader" - ruft den `ClassLoader` Konstruktor auf
- "setSecurityManager" - ruft `System.setSecurityManager()` auf
- "modifyThread" - ruft eine der Thread Methoden `interrupt(), setPriority(), setDaemon()` oder `setName()`

Die grundlegenden Rechte haben Sie entweder oder Sie haben Sie nicht! Die Benennung der Grundrechte folgt jener der Properties. Die Spezifikation kann auch *wildcard* Zeichen enthalten:
beispielsweise ist "java.*" oder "*" erlaubt; "*.java" oder "y*x" allerdings nicht.

`FilePermission` und `SocketPermission` sind Unterklassen der `Permission` Klasse. Diese Klassen können eine komplexere Namenskonvention haben als bei den Grundrechten. Beispielsweise können Sie bei `FilePermission` die Rechte für Verzeichnisse und Unterverzeichnisse oder generische Dateinamen angeben: "*.a_x*" ist ein korrekter Name. Die Syntax ist genau dieselbe wie bei der Auswahl von Dateien im `dir` Befehl.

Alle Rechte können auch *action lists*, Aktionslisten, zugeordnet bekommen. Diese definieren bestimmte Aktionen, welche mit oder im Zusammenhang mit diesen Objekten gestattet sind.

Beispielsweise können Sie bei den Dateirechten den Objekten spezielle Aktionen zuordnen, wie "read", "write", "execute", "delete". Diese geben einfach an, welche Aktionen erlaubt oder verboten sind (falls sie fehlen). Viele Basisrechte benutzen keine Aktionslisten.

`PropertyPermission` sieht eine solche Aktionsliste vor. Hier ein Beispiel:
Sie können beispielsweise mit dem Namen "java.*" und der Aktion "read" zulassen, dass Sie die Werte aller `System.Properties` bestimmen können, welche mit java anfangen. Sie können diese Rechte mit `System.setProperty()` und `System.getProperty()` setzen oder abfragen.

### 1.5.3. Security Policies

Die Sicherheitsrichtlinien für ein gegebenes Laufzeitsystem wird durch das `java.security.Policy` Objekt oder spezifischer durch eine mögliche Unterklasse der abstrakten `Policy` Klasse definiert. Das `Policy` Objekt unterhält die Rechte, welche den Schutzbereichen (*protection domains*) zugeordnet sind. Wie die Sicherheitsrichtlinien dem `Policy` Objekt mitgeteilt werden, hängt von der Implementation der Schutzmechanismen ab. Der Standardmechanismus ist der, eine Policy Datei zu verwenden, in der die Rechte festgehalten werden. Diese Dateien werden typischerweise mit dem Policytool festgelegt.

Beispiel:

```
grant codeBase "file:/home/sysadmin" {
      permission java.io.FilePermission "/tmp/abc", "read";
}
```

legt fest, dass Sie Zugriff auf die Datei "/tmp/abc" im Verzeichnis "file:/home/sysadmin" besitzen.

### 1.5.4. Zugriffskontrolle und privilegierte Ausführung

Die AccessController Klasse wird für drei typische Anwendungsfälle eingesetzt:

- sie stellen die grundlegenden `checkPermission()` Methoden für den Security Manager zur Verfügung.
- sie stellen Mechanismen zur Bestimmung eines Snapshots des aktuellen Kontextes zur Verfügung: `getContext()`, welche ein `AccessControlContext` Objekt zurückliefert.
- sie stellen Mechanismen zur Verfügung, um Programmcode *privilegiert* auszuführen. Damit werden die Privilegien temporär oder dauerhaft überschrieben bzw. geändert.

Einem *Protection Domain* (repräsentiert durch `java.security.ProtectionDomain`) ist eine *Code Source* (repräsentiert durch `java.security.CodeSource`) zugeordnet plus Zugriffsrechte der Programme in diesen Code Sourcen (Mengen von Programmen). Code Sourcen sind eine Verallgemeinerung der Codebase, einem Konstrukt, welches speziell bei Applets angewandt wird und eine Lokation, ein Verzeichnis festlegt, aus dem die Programme und Class Dateien stammen. Dieses Konzept wurde entwickelt, um neben der Lokation der Dateien auch andere Kriterien zuzulassen, beispielsweise der Schutz von Verzeichnissen oder Dateien mittels Zertifikaten, mit denen die Zugriffsrechte festgelegt und überprüft werden können.

Die `doPrivileged()` Methode des AccessController Objekts kann eingesetzt werden, von einem privilegierten Thread, um einem anderen Thread temporär die Ausführung einer bestimmten oder mehrerer Methoden zu gestatten.

Und so könnte ein Programmbeispiel aussehen:

```
void eineMethode() {

      // normaler Programmcode
      //...
      AccessController.doPrivileged(new PrivilegedAction() {
            public Object run() {
                  // privilegierter Programmcode
                  // zum Beispiel
                  System.loadLibrary("awt");
                  return null;
            }
      });
      // normaler Programmcode
      // ...
}
```

Sie sollten diese doPrivileged() mit äusserster Vorsicht einsetzen, falls überhaupt.
Das Thema ist also eher für jene gedacht, die eh schon fast alles über Java ausprobiert haben.

# SYSTEM PROGRAMMIERUNG