In diesem Kursteil

- Kursübersicht
- Modul 1 : Übersicht über Verteilte Systeme
  - Modul Einleitung
  - Kurzer geschichtlicher Rückblick
  - Verteilte Systeme - Distributed Computing
  - Techniken für das Distributed Computing mit Java Technologien
  - Vergleich der Java Verteilten Technologien
  - Praktische Übung
  - Quiz
  - Zusammenfassung
- Modul 2 : Security Managers
  - Modul Einleitung
  - Security Manager - Übersicht
  - Wer benötigt einen Security Manager?
  - Security Manager checkXXX() Methoden
  - Schreiben eines erweiterten Security Managers
  - Methoden und deren Aktionen
  - Methoden eines Security Managers
  - Installation eines Security Managers
  - Nach der Installation
  - Praktische Übung
  - Quiz
  - Zusammenfassung
- Kurs Zusammenfassung

Einführung in Java in Verteilten Systemen

.

1.1. Kursübersicht

In dieser Übersicht geht es darum, Ihnen eine erste Idee zu geben, wo und wie Sie mit Java Netzwerkprogramme erstellen und einsetzen können. Ihre Voraussetzungen sollten Java und konventionelle Verteilte Systeme sein. Sie benötigen also explizit Wissen aus dem Kommunikationsbereich. Aber auch ohne dass Sie bereits versucht haben, Netzwerkprogramme zu schreiben, werden Sie von dieser Einführung profitieren. Es geht wirklich lediglich um eine Übersicht. Die Vertiefung geschieht in den Modulen über Verteilte Systeme.

1.1.1.1. Kursvoraussetzungen

Dieser Kurs setzt voraus, dass Sie sich bereits mit der Java Programmierung und der Objekt Orientierten Analyse und Design befasst haben oder folgende Fähigkeiten besitzen:

- Sie besitzen Java Kenntnisse in einem Umfang, der es Ihnen erlaubt, Java Applikationen zu entwickeln.
- Sie haben Grundkenntnisse über Datenbanken
- Sie kennen SQL, die Strukturierte Abfragesprache
- Sie besitzen Kenntnisse über OO Programmiertechniken

1.1.1.2. Lernziele

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

- die Charakteristiken der unterschiedlichen Verteilten Systeme und Verteilten Objektsysteme zu vergleichen und gegeneinander abzugrenzen.
- drei der Schlüsseltechnologien von Sun für Java aufzuzählen und miteinander zu vergleichen.
- Sicherheitsrichtlinien anzugeben, dank denen Zugriffe mittels SecurityManager geschützt werden können.

1.2. Modul 1 : Übersicht über Verteilte Systeme

In diesem Modul

- Modul 1 : Übersicht über Verteilte Systeme
  - Modul Einleitung
  - Kurzer geschichtlicher Rückblick
  - Verteilte Systeme - Distributed Computing
  - Techniken für das Distributed Computing mit Java Technologien
  - Vergleich der Java Verteilten Technologien
  - Praktische Übung
  - Quiz
  - Zusammenfassung

1.2.1. Einleitung

Mit dem explodierenden Wachstum des Internets und der Intranets wuchs auch der Bedarf an Applikationen. In diesem Modul geht es darum, eine Übersicht zu gewinnen über die verfügbaren Technologien zur Realisierung verteilter Systeme.

1.2.1.1. Lernziele

Nachdem Durcharbeiten dieser Unterlagen sollten Sie in der Lage sein:

- die Charakteristiken der unterschiedlichen Systeme zur Realisierung verteilter Systeme und verteilter Objektanwendungen zu erkennen und gegeneinander abzugrenzen.
- die Charaktersitiken der unterschiedlichen Programmiertechniken zu erkennen und gegeneinander abzugrenzen.
- in groben Zügen beschreiben, welche Funktionalitäten die APIs für folgende Konzepte in Java vorhanden sind:
  Java Database Connectivity (JDBC)
  Remote Method Invocation (RMI) und
  JavaIDL oder ORB (CORBA)

1.2.1.2. Ressourcen und Referenzen

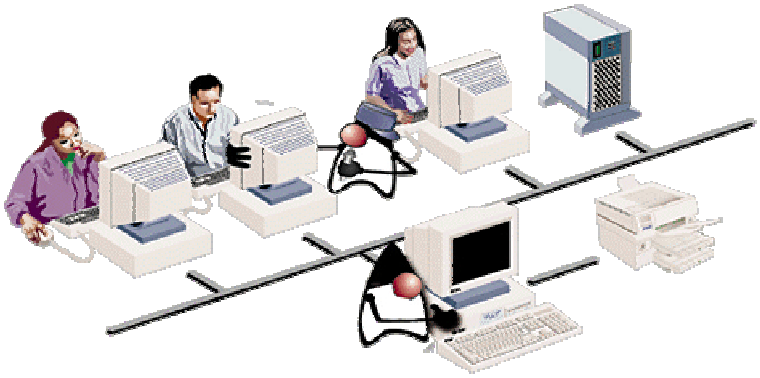
- ***The Essential Distributed Objects Survival Guide*** von Orfali, Harkey und Edwards (Wiley Press, 1996ff) .

- Auch der Bericht "A Note on Distributed Computing" von Jim Waldo, Geoff Wyant, Ann Wolrath, and Sam Kendall (Sun Microsystems, 1994); erthält wertvolle Hinweise unter http://www.sunlabs.com/technical-reports/1994/abstract-29.html oder als ganzer Bericht auf dem Web / der CD.

1.2.2. Kurzer geschichtlicher Rückblick

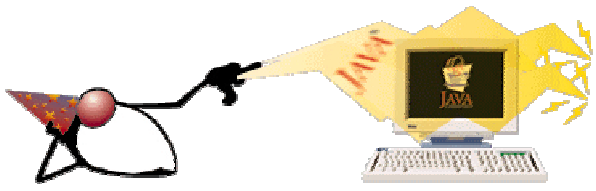
Die Geschichte der Informatik begann mit der Entwicklung von Programmen, welche auf einem einzelnen Rechner, mit einer CPU installiert und gestartet wurden. Auch die Grossrechner mit ihren TimeSharing Betriebsystemen nutzten eine CPU möglichst effizient und verwendeten einen einzigen Adressraum.

Als die Rechner leistungsfähiger und kleiner wurden, verschoben sich die Rechner mit der Zeit auf den Schreibtisch der Mitarbeiter. Sehr schnell wurde den Benutzern klar, dass es sinnvoll war, diese Rechner auf den Schreibtischen miteinander zu verknüpfen. Damit konnten Ressourcen gemeinsam genutzt werden, beispielsweise Programme, Drucker und andere Peripheriegeräte. Dazu wurden auch dedizierte Server eingesetzt. Damit wurde die Leistungsfähigkeit der Systeme deutlich verbessert. Der Client konnte Applikationen vom Server herunterladen oder direkt auf dem Server ausführen lassen.



Die Client- Server Systeme hatten aber auch klare Nachteile. In einem völlig homogenen Umfeld, bei dem jeder Client die neuste Version des Betriebsystems enthält und jede Maschine die selbe Architektur besitzt. ist das gemeinsame Nutzen von Serverprogrammen einfach. Die Realität sieht allerdings anders aus. Die einzelnen Rechner besitzen in der Regel unterschiedliche Betriebsystemversionen, die Hardwarearchitektur ist unterschiedlich, selbst wenn die Maschinen vom selben Hersteller stammen und die Wahrscheinlichkeit, dass eine Applikation gemeinsam genutzt werden kann, ist sogut wie null.

Grössere Applikationen wurden immer häufiger auf die leistungsfähigen Clients verschoben. Dies hatte zur Folge, dass auch die Client Programme grösser und grösser wurden und das Starten der Maschinen immer länger dauert.



Java versuchte einen Ausweg: die Applikationen sollten auf möglichst vielen Maschinen lauffähig sein, sofern eine relativ kleine Java Virtual Machine (JVM) darauf lauffähig ist. Dabei kann es durchaus sein, dass die Applikation sobold sie benötigt wird, vom Server herunter geladen wird. Die Schlankheit der Programme wird dadurch aber nicht Realität!

1.2.3. Verteilte Systeme - Distributed Computing

Falls man Applikationen verteilen will, muss man sie modular gestalten können und eventuell Programmcode bei Bedarf dazuladen können.

Die Lösung des Problems besteht in einer Lösung, irgendwo zwischen einem Mainframe und dem Client-Server Modell. Man muss dem Client gestatten, eine Applikation direkt dort zu nutzen, wo sie vorliegt. Damit gelangt man zu 'Distributed Computing', verteilten Systemen.

Verteilte Systeme sind aus verschiedenen Gründen wichtig:
- die Softwareemtwicklungskosten können reduziert werden. Viele Clients können auf das selbe Programm zugreifen; damit werden die Entwicklungskosten reduziert, weil keine Duplikation der Programme mehr geschieht.
- das Laden der Ressourcen ist ausbalanciert. Das Client-Server Paradigma basiert auf der Möglichkeit Dienste von einer zentralen Ressource, dem Server, zu verlangen und in Anspruch zu nehmen. Damit können CPU, Speicher und Peripheriegeräte des zentralen Servers optimal ausgenutzt werden.
- Client und Server kommunizieren über ein vereinbartes Protokoll, in der Regel einem standardisierten.

In Verteilten Systemen rufen die Programme, welche auf einem Client laufen, Programme in einem anderen Adressraum auf. Dieser Adressraum kann lokal oder entfernt vorhanden sein. Dieses Modell hat den Vorteil, dass das Client Programm optimal auf den Client angepasst werden kann, die Kommunikation mit dem Server über ein allgemeines Protokoll stattfindet und somit auch der Server mit den für ihn optimal verwendbaren Werkzeugen gemanaged werden kann und seine Applikationen auf diese Umgebung optimiert werden können. Client Architekturen und Server Architektur haben eventuell nichts gemeinsam!

Was nötig ist, ist eine saubere Analyse darüber, was lokal oder was remote installiert und betrieben werden soll. Es ist klar, dass eine Applikation in einem lokalen Umfeld schneller ladbar ist, als wenn diese erst auf einem entfernten Rechner aktiviert werden muss.

Falls eine lokale Applikation ausfällt, beispielsweise die CPU oder Memory oder eine Disk, dann läuft nichts mehr. Sie können auch nicht viel gegen den Ausfall machen: das System fällt einfach aus. Eine Recovery ist nicht möglich.

Im Verteilten Systemmodell sieht das anders aus: ein teilweiser Ausfall eines Teilsystems hat in der Regel nicht den Totalausfall zur Folge. Die Recovery wird also wichtig. Falls beispielsweise eine Client Applikation den Server nicht mehr findet haben Sie viele Möglichkeiten: sie könnten versuchen, den Server wieder zu finden, einige Male wenigstens; Sie könnten auch versuchen, auf einen alternativen Server auszuweichen. Sie können auch eine Timeout Zeitperiode angeben, nach der Sie alle weiteren Versuche abbrechen.

Schauen wir uns ein echtes Beispiel, das NFS (Network File System) von Sun Microsystems an.

Das NFS ist eine der erfolgreichsten Applikationen in Verteilten Systemen. NFS ist von vielen Software Herstellern erhältlich, für die unterschiedlichsten Plattformen. Mit NFS kann der Systemadministrator das remote Filesystem auf zwei Arten laden: Soft Mount und Hard Mount. Der Hauptunterschied besteht in der Art und Weise, in der Fehler bestimmt und behandelt werden. In einem NFS System können drei Komponenten ausfallen: der NFS Server, der NFS Client oder die Netzwerkverbindung dazwischen.

Falls ein Hard Mount verwendet wird, muss eine Schreibaktion auf die Server Disk bestätigt werden. Der Client wird bis zu 10'000 Mal erneut versuchen, die Daten korrekt zu übermitteln, bis schliesslich das Schreiben als korrekt bestätigt wird. Die Datenübertragung ist also zuverlässig.

Beim Soft Mount werden keinerlei Bestätigungen versandt. Der Fehler wird also nicht erkannt. Dafür ist die Übermittlung bedeutend schneller.

Was geschieht, falls der Client nicht länger mit dem Server kommunizieren kann?
Falls der Client Hard Mounted ist, liegt es am Client die Verbindung erneut aufzubauen. Im schlimmsten Fall hängt sich der Client auf, weil er dauernd versucht, die Verbindung erneut aufzubauen.
Falls der Client Soft Mounted ist, wird der Prozess unter Umständen eine Timeout Meldung erhalten und der Client wird einen Fehler abfagen müssen.

Und nun zurück zum allgemeinen Fall:
man hat versucht, die verteilten Systeme zu kategorisieren, also die Systeme in vordefinierte Kategorien einzuteilen. Jede dieser Einteilungen ist irgendwie künstlich!

Eines dieser Einteilungsschematas, Taxonomien, definiert sechs Technologien.
Die ersten drei Technologien bedingen synchrones, blockbasiertes Kommunizieren:
- pro Anfrage muss eine Antwort erzeugt werden.

Die nächsten drei Technologien verwenden ein asynchrones Schema:
- der Client startet eine Anfrage und wird Daten im Hintergrund empfangen. Es muss nicht auf die Antwort warten und kann in der Zwischenzeit andere Aktivitäten erledigen.

Und hier die sechs unterschiedlichen Kommunikationsschemata:
1) Datenbank-zentriert
   Der Client verlangt Daten von der Datenbank, vom Datenbankserver, beispielsweise mit Hilfe einer SQL Abfrage.

2) Remote Procedure Call (RPC)
   Der Client ruft eine lokale Funktion auf, im Adressraum des Clients. Die Funktion wird allerdings durch einen Proxy dargestellt und der Aufruf wird an einen Server weitergeleitet.

3) Common Object Request Broker Architektur (CORBA)
   Der Client ruft Methoden eines entfernten Objekts auf. Das Objekt wird durch Proxies repräsentiert, die den Methodenaufruf an das remote Objekt in einem anderen Adressraum weiterleiten.

4) Messaging
   Der Client schreibt eine Meldung in eine Warteschlange, die von einer anderen Applikation gelesen wird. Diese befindet sich in der Regel in einem anderen Adressraum.

5) Publish-And-Subscribe
   Der Client verlangt ein Objekt oder ein Programm. Er wird ein Mitglied eines Subskriptionsdienstes. Der Service liefert den Clients, welche eingetragen sind, Updates, wann immer eine Änderung stattfindet.
   OLE (Object Linking and Embedding) verwendet dieses Modell.

6) World Wide Web
   Der Client verlangt eine Seite; der Server sendet diese Seite. In der Zwischenzeit kann der Client (Browser) irgend etwas anderes machen.

Die objektorientierte Programmierung ist eine natürliche Erweiterung des 'Distributed Computing'. Mit Objekten kann man die in solchen Systemen auftretenden Probleme besser beschreiben und lösen, als mit traditionellen Methoden und Techniken. Die Objekte können in lokalen oder entfernten Adressräumen installiert und aktiviert werden.

In einem verteilten Objektmodell beschäftigt sich der Programmierer mit den unterschiedlichen Objekten und deren Wechselwirkung. Methodenaufrufe werden von Objekten auf Objekten ausgeführt. Idealerweise kann der Programmierer mit entfernten Objekten genau so umgehen, wie mit lokalen. Der aktuelle Adressraum der Objekte ist somit unwichtig. Der Programmierer bearbeitet seine Objekte mittels klar definierter Schnittstellen. Jeder Methodenaufruf sieht genauso aus wie ein lokaler Methodenaufruf.

1.2.4. Techniken für das Distributed Computing mit Java Technologien

Als erstes besprechen wir die drei synchronen Technologien. Diese sind historisch gesehen als erstes in Java übernommen worden, zusammen mit dem WWW.

Jede der drei Technologien hat ihre Vorteile und Nachteile, aber alle nutzen die Vorteile von Java als objektorientierte Sprache. Jedes dieser Systeme wurde ursprünglich im C/C++ Umfeld entwickelt. Die Komplexität der C Programme ist aber unverhältnismässig höher als in Java.

Beim Java Database Connectivity (JDBC) API wird dem Programmierer ein Set von Interfaces zur Verfügung gestellt, mit deren Hilfe Java Client (Applikationen oder Applets) auf Datenbanken zugreifen können. Das API stellt generische Interfaces zur Verfügung, welche von den Datenbank Treibern implementiert werden müssen. Die Applikationen können sowohl als zweistufige oder als dreistufige (two-tiers, three-tiers) Applikationen realisiert werden. Der Treiber kann irgend ein Zugriffsprotokoll, also nicht zwingend SQL verwenden.

Bei der Java Remote Methode Invocation (RMI) werden dem Entwickler APIs zur Verfügung gestellt, welche ähnlich aufgebaut sind, wie RPC: die Client Applikation ruft Methoden eines Objekts auf als ob dieses Objekt lokal wäre; in Wirklichkeit wird das Objekt aber durch ein Proxy Objekt repräsentiert. Das Objekt wird durch ein anderes Java Objekt implementiert, welches die selben Interfaces wie der Client implementiert. Das RMI API stellt ein Set von Klassen und Methoden zur Verfügung, mit deren Hilfe Objekte serialisiert und als Argumente bei Methodenaufrufen mitgegeben werden können. RMI muss auch Parameter passend umformattieren, um mit dem JRMI Protokoll übertragen werden zu können. RMI kommuniziert entweder über ein eigenes Wire Protokoll oder über IIOP, das von der OMG standardisierte Internet Inter Orb Protocol.

Die Java Interface Definition Language (JavaIDL) stellt dem Programmierer Hilfswerkzeuge zur Verfügung, mit deren Hilfe die Anbindung an CORBA möglich wird. JavaIDL wurde durch IDL ersetzt und ein offizielles Language Mapping von IDL auf Java in den CORBA Standard aufgenommen. RMI ist ähnlich wie CORBA. Aber CORBA ist abstrakter und allgemeiner. Client und Server werden mit Hilfe einer Interface Definition spezifiziert. Damit werden Objektdienste implementiert.

1.2.5. Vergleich der Java Verteilten Technologien

Schauen wir uns nun die drei unterschiedlichen Techniken an und versuchen einen Vergleich der Technologien anzustellen.

JDBC ist ein API, wie ein C Bibliothek, mit dessen Hilfe die Entwicklung von Applikationen recht einfach ist. Der Entwickler konzentriert sich auf den Einsatz einer handvoll Klassen und Methoden, um mit deren Hilfe auf die Datenbank zuzugreifen.

Bei RMI geht es darum Interfaces zu kreieren, und damit das Verhalten des Systems festzulegen, die vom Client und vom Server implementiert werden. Die Konstruktion eines solchen Systems bedingt mehr Aufwand, ist schwieriger als JDBC, weil man ein abstrakteres Problem vor sich hat. Die Abstraktion besteht in der Definition der Schnittstellen, also des Verhaltens, losgelöst von der Implementierung.

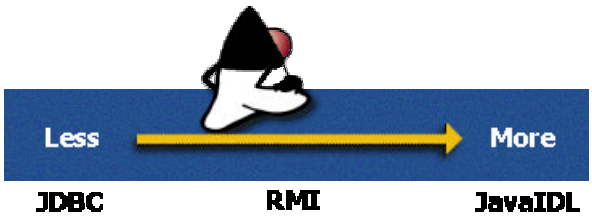
Java IDL ist noch eine Abstraktionsebene höher als RMI. Der Java IDL Compiler generiert Stubs und Skeletons, mit deren Hilfe Client und Server Implementationen gebildet werden. CORBA Applikationen setzen voraus, dass Sie in der Lage sind, Ihr Problem in IDL also losgelöst von der Implementierung und der Implementierungssprache zu spezifizieren.

Die Einfachheit eines JDBC hat auch Nachteile! Beispielsweise können Sie mit CORBA Applikationen Ihre Applikationsplattform laufend vergrössern und die alten Servant Objekte weiterlaufen lassen. In JDBC bedingt eine Änderung der Datenbankzugriffe in der Regel eine grössere Änderung der Applikation. Grössere Änderungen haben in der Regel neuen Schulungsaufwand zur Folge und verursachen somit weitere Kosten.

JDBC Applikationen sind in dieser Hinsicht am unflexibelsten. Jede Datenbankänderung führt zu einer erneuten Übersetzung der Programme, mindestens auf der Client Seite. Wenn sich das Geschäftsmodell ändert, müssen Sie von vorne anfangen.

RMI Applikationen sind flexibler als JDBC Applikationen. Der Grund ist die Trennung von Spezifikation und Implementation. Die Benutzerschnittstelle ändert sich unter Umständen weniger schnell bei Änderungen als bei einer JDBC Änderung.Falls die Methoden oder das Datenmodell sich ändern werden Sie auch in RMI grössere Änderungen vornehmen müssen. RMI ist aber im Wesentlichen eine Punkt-zu-Punkt Verbindung. Falls sich die Charakteristiken eines Punktes ändern, werden Sie Änderungen am andern Punkt haben und somit auch auf Stufe IDL Anpassungen machen müssen. Diese Änderungen auf Stufe IDL können sehr gravierend sein. Dieses Problem führte zum neusten Boom in der unternehmensweiten Kommunikation, dem Messaging oder allgemeiner, der unternehmensweiten asynchronen Kommunikation. In JavaSpaces werden Ihnen für die synchrone Kommunikation Mechanismen zur Verfügung gestellt, mit deren Hilfe Objekte lokalisiert werden können: http://java.sun.com/products/javaspaces/index.html

CORBA oder Java IDL ist gegenüber RMI eine Stufe abstrakter und somit auch flexibler. JavaIDL verwendet einen Daemon Prozess, um Meldungen zwischen Clients und Servern auszutauschen. Damit erreicht man eine hohe Flexibilität, hat aber die selben Probleme wie bei RMI (Verwaltung der IDLs, hoher Anpassungsaufwand bei Änderungen).



Bemerkung

Je mehr Sie von konkreten Daten abstrahieren, desto einfacher können Sie Änderungen an den Objekten oder Applikationen im System durchführen, ohne grössere Änderungen am Frontend, am GUI machen zu müssen.

1.2.6. Praktische Übung

1.2.6.1. Das Szenario

Wir haben ein Flugreservationssystem vor uns, von dem folgende Funktionalität erwartet wird:

- Erfassen der Passagiere
- Eingabe und Abfrage der Kundenprofile (Passagiere)
- Anzeige aller Flüge von ... nach (Abflughafen ... Destination)
- Verwaltung aller Informationen betreffend Sitzbelegung

In den folgenden Fragen geht es um die Anforderungen an dieses System. Versuchen Sie die Anforderungen zu abstrahieren und in abstrakte Methodenaufrufen festzuhalten.

Selbsttestaufgabe 1

Aufgrund der obigen Beschreibung, welche drei Java Interfaces würden Sie vermutlich definieren?[1]

a) Kunden Interface
   Gepäck Interface
   Piloteneinsatzplanung (Scheduling) Interface
b) Reservation Interface
   Flug Interface
   Snack Interface
c) Kunden Interface
   Reservation Interface
   Flug Interface

Selbsttestaufgabe 2

Werden Sie Datenbankzugriffe programmieren müssen?[2]

a) ja
b) nein

[1] c)
[2] a) viele der Informationen (Kunden, Platzreservationen, Flüge) können in Datenbanken abgespeichert werden.

Selbsttestaufgabe 3

Gegeben sei folgende Applikation. In welchem Bereich werden die Interfaces implementiert?

```
package flugreservation;

/**
 * Title:
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * @author J.M.Joller
 * @version 1.0
 */

import sunsoft.jws.visual.rt.base.*;
import sunsoft.jws.visual.rt.shadow.java.awt.*;
import java.awt.*;
import java.util.*;

// Import JDBC für miniSQL (imaginary Treiber)
import imaginary.sql.*;
import java.sql.*;

public class AirlineProg extends Group {
  private AirlineRoot gui;
  CustomerImpl cust;
  FlightImpl flt;
  CustomerReservationImpl custrv;
  String server;

  public AirlineProg () {
   addForwardedAttributes();
  }

  public AirlineProg (String db_host) {
    this();
    server = db_host;
  }


  protected Root initRoot() {
    gui = new AirlineRoot(this);

    addAttributeForward(gui.getMainChild());

    return gui;
  }

  protected void initGroup() { }


  protected void showGroup() {

    int i;

      // JDBC - Driver instanzieren
      try {
      new iMsqlDriver();
      } catch (java.sql.SQLException SQLEx) {
```

```
    System.out.println("Driver wurde nicht geladen");
    }

    cust = new CustomerImpl(server);
    flt = new FlightImpl(server);
    custrv = new CustomerReservationImpl(server);

    // GUI: zuerst Abflug- und Ankunfts-Flughafen festlegen

    /* Alle Städte mit Ankunft / Abflug */
    String origins[] = flt.getAllOrigins();
    String destinations[] = flt.getAllDest();

    /* Abflug Flughafen festlegen (originList)  */
    for (i = 0; i < origins.length; i++)
         ((List)gui.originList.getBody()).addItem(origins[i]);
    for (i = 0; i < destinations.length; i++)
         ((List)gui.destinationList.getBody()).addItem(destinations[i]);
}

protected void hideGroup() { }
protected void createGroup() { }
protected void destroyGroup() { }
protected void startGroup() { }
protected void stopGroup() { }

protected Object getOnGroup(String key) {
  return super.getOnGroup(key);
}

protected void setOnGroup(String key, Object value) {
  super.setOnGroup(key, value);
}

public boolean handleMessage(Message msg) {
  return super.handleMessage(msg);
}

/**
 *
 * public boolean mouseDown(Message msg, Event evt, int x, int y);
 * public boolean mouseDrag(Message msg, Event evt, int x, int y);
 * public boolean mouseUp(Message msg, Event evt, int x, int y);
 * public boolean mouseMove(Message msg, Event evt, int x, int y);
 * public boolean mouseEnter(Message msg, Event evt, int x, int y);
 * public boolean mouseExit(Message msg, Event evt, int x, int y);
 * public boolean keyDown(Message msg, Event evt, int key);
 * public boolean keyUp(Message msg, Event evt, int key);
 * public boolean action(Message msg, Event evt, Object what);
 * public boolean gotFocus(Message msg, Event evt, Object what);
 * public boolean lostFocus(Message msg, Event evt, Object what);
 */
public boolean handleEvent(Message msg, Event evt) {
  return super.handleEvent(msg, evt);
}

public boolean action(Message msg, Event ev, Object obj){
    int i, j;

    if (((String)ev.arg).equals ("Ausgabe der moeglichen Fluege")){
                getFlights();
```

```
      } else if (((String)ev.arg).equals ("Suchen")){
            doSearch();

      } else if (((String)ev.arg).equals ("Name gefunden")){
            String selectedName = ((List)gui.NamesFoundList.getBody
            ()).getSelectedItem();
            if (selectedName != null) {
            StringTokenizer nameT = new StringTokenizer(selectedName,"-");
            String id = nameT.nextToken();
            String lname = nameT.nextToken();
            String fname = nameT.nextToken();
            ((TextField)gui.CustIDText.getBody()).setText(id);
            ((TextField)gui.foundLastNameText.getBody()).setText(lname);
            ((TextField)gui.foundFirstNameText.getBody()).setText(fname);
            }


      } else if (((String)ev.arg).equals ("VerbergenDesSuchWindow")){
            ((TextField)gui.lastNameText.getBody()).setText("");
            ((TextField)gui.firstNameText.getBody()).setText("");
            ((List)gui.NamesFoundList.getBody()).clear();

      } else if (((String)ev.arg).equals ("SuchWindow")){
            ((TextField)gui.CustIDText.getBody()).setText("");
            ((TextField)gui.foundLastNameText.getBody()).setText("");
            ((TextField)gui.foundFirstNameText.getBody()).setText("");
            ((TextField)gui.ConfirmNumText.getBody()).setText("");

      } else if (((String)ev.arg).equals ("Neuer Kunde")){
            String addr = ((TextField)gui.AddressText.getBody()).getText();
            String last =
((TextField)gui.lastNameText.getBody()).getText();
            String first =
((TextField)gui.firstNameText.getBody()).getText();

            String ID = cust.produceID(last, first);
            cust.setNewCustomerInfo(ID, last, first, addr);


      } else if (((String)ev.arg).equals ("Reservation buchen")){
            String confirmNum = produceConfirmNumber();
            String selectedFlt = ((List)gui.AvailFlightList.getBody
            ()).getSelectedItem();
            StringTokenizer strT = new StringTokenizer(selectedFlt, "-");
            String flightNum =  strT.nextToken();
            String servClass = ((Choice)gui.ClassChoice.getBody
            ()).getSelectedItem();
            String custID =
((TextField)gui.CustIDText.getBody()).getText();

            ((TextField)gui.ConfirmNumText.getBody()).setText(confirmNum);
            custrv.setReservation(custID, flightNum, confirmNum,
servClass);

      } else if (ev.target == (TextField)gui.CustIDText.getBody()) {

            String id = ((TextField)gui.CustIDText.getBody()).getText();
            CustomerInfo cuInfo = cust.getCustomerInfo(id);
            String custInfo = cust.produceCustomerString(cuInfo);

            StringTokenizer nameT = new StringTokenizer(custInfo, "-");
```

```
            nameT.nextToken();
           String lname = nameT.nextToken();
           String fname = nameT.nextToken();

           ((TextField)gui.foundLastNameText.getBody()).setText(lname);
           ((TextField)gui.foundFirstNameText.getBody()).setText(fname);


     }


     return true;
    }

   private void getFlights() {
           int i, j;

           String month = ((Choice)gui.monthChoice.getBody
           ()).getSelectedItem();
           String day = ((Choice)gui.dayChoices.getBody
           ()).getSelectedItem();
           String year = ((Choice)gui.yearChoice.getBody
           ()).getSelectedItem();
           String ori = ((List)gui.originList.getBody
           ()).getSelectedItem();
           String dest = ((List)gui.destinationList.getBody
           ()).getSelectedItem();

           month = convertMonth(month);
           day = convertDay(day);
           year = convertYear(year);

           FlightInfo availFlts[]= flt.getAvailFlights
           (ori, dest, month + "/" + day +"/" + year);
           System.out.println
           (ori+"  "+ dest+ "  "+ month + "/" + day +"/" + year);

           i = ((List)gui.AvailFlightList.getBody()).countItems();
           if (i != 0){
                 ((List)gui.AvailFlightList.getBody()).clear();
           }
            for (i = 0; i < availFlts.length; i++){
                 String item = flt.produceFlightString(availFlts[i]);
                 System.out.println(item);

                 ((List)gui.AvailFlightList.getBody()).addItem(item);

           }
   }


   private String  convertMonth(String mth){
     if (mth.equals ("Januar"))
           return("01");
     else if (mth.equals ("Februar"))
           return("02");
     else if (mth.equals ("Maerz"))
           return("03");
     else if (mth.equals ("April"))
           return("04");
     else if (mth.equals ("Mai"))
           return("05");
```

```
      else if (mth.equals ("Juni"))
            return("06");
      else if (mth.equals ("Juli"))
            return("07");
      else if (mth.equals ("August"))
            return("08");
      else if (mth.equals ("September"))
            return("09");
      else if (mth.equals ("Oktober"))
            return("10");
      else if (mth.equals ("November"))
            return("11");
      else
            return("12");

    }


    private String  convertYear(String yr){
      if (yr.equals ("1997"))
            return("97");
      else if (yr.equals ("1998"))
            return("98");
      else if (yr.equals ("1999"))
            return("99");
      else if (yr.equals ("2000"))
            return("00");
      else if (yr.equals ("2001"))
            return("01");

      return null;
     }

    private String  convertDay(String day){
      if (day.equals (" 1"))
            return("01");
      else if (day.equals (" 2"))
            return("02");
      else if (day.equals (" 3"))
            return("03");
      else if (day.equals (" 4"))
            return("04");
      else if (day.equals (" 5"))
            return("05");
      else if (day.equals (" 6"))
            return("06");
      else if (day.equals (" 7"))
            return("07");
      else if (day.equals (" 8"))
            return("08");
      else if (day.equals (" 9"))
            return("09");

      return day;
     }

   private void doSearch(){
            int i;

            String first =
            ((TextField)gui.firstNameText.getBody()).getText();
            String last =
```

```
            ((TextField)gui.lastNameText.getBody()).getText();
            CustomerInfo nameMatches[] = cust.getCustomerInfo(last, first);

            if (nameMatches == null) {
                  return;
            }

            i = ((List)gui.NamesFoundList.getBody()).countItems();
            if (i != 0){
                  ((List)gui.NamesFoundList.getBody()).clear();
            }
            for (i = 0; i < nameMatches.length; i++){
                  String item = cust.produceCustomerString(nameMatches[i]);

                  ((List)gui.NamesFoundList.getBody()).addItem(item);

            }
      }

       private String produceConfirmNumber(){
            return Integer.toString((int)(Math.random()*1000000));
       }

}
```

Selbsttestaufgabe 4

Wie geschehen die Datenbankzugriffe mittels JDBC?[3]

a) Die Datenbankmethoden werden direkt aufgerufen.
b) Interface Methoden rufen Datenbankmethoden auf.

[3] a)

1.2.7. Quiz - Übersicht über Distributed Computing

Dieses Quiz besteht aus fünf Fragen, mit denen Sie Ihr Wissen zum Thema überprüfen können.

Selbsttestaufgabe 5 Welche der folgenden ist *nicht* charakteristisch für eine verteilte Rechnerumgebung?[4]

a) Applikationen laufen auf einem Client und können auf Applikationen, die auf einer anderen Maschine laufen, zugreifen.
b) Applikationen sind plattformabhängig.
c) Ressourcen können optimal zugeteilt werden.
d) Programme. welche auf dem Client laufen, können Programme in anderen Adressräumen aufrufen.

Selbsttestaufgabe 6 Welche der folgenden Charakteristiken trifft auf verteilte Objekt-Systeme zu?[5]

a) Verteilte Objekt-Systeme gestatten es, dass Objekte von vielen Applikationen eingesetzt werden.
b) Objektaufrufe (Methodenaufrufe) werden mit Hilfe wohl definierter Interfaces erledigt.
c) Jedes Objekt wird wie ein lokales Objekt behandelt.
d) alle obigen Antworten

Selbsttestaufgabe 7 Welche der folgenden Aussagen trifft *nicht* zu betreffend JDBC Applikationen?[6]

a) JDBC Applikationen sind sehr flexibel.
b) Bei Änderungen in der Datenbank muss die Client-seite neu geschrieben werden.
c) Die Entwicklung solcher Applikationen ist recht einfach.

[4] b)
[5] d)
[6] a)

Selbsttestaufgabe 8 Welche der folgenden Aussagen trifft *nicht* auf RMI Anwendungen zu?[7]

a) Methoden und Objekte können nicht ohne Einfluss auf die aktuellen Benutzer hinzugefügt werden.
b) Man muss das Problem abstrahieren und Java Interfaces definieren, um anschliessend Client und Server entwickeln zu können.
c) RMI gestattet es nicht, den Implementationsserver (Servant Objekt) von einer Maschine auf eine andere Maschine zu verschieben, ohne dass die Client-Seite beeinflusst würde.

Selbsttestaufgabe 9 JavaIDL ist die flexibelste Java Technologie für verteilte Systeme.[8]

a) trifft zu
b) ist nicht korrekt

[7] a)
[8] a)

1.2.8. Zusammenfassung

Nach dem Durcharbeiten dieses Moduls sollten Sie gelernt haben:

- die Charakteristiken der verschiedenen Verteilten System-Ansätze zu vergleichen und gegen abzugrenzen.
- die unterschiedlichen Programmiertechniken in Bezug auf Flexibilität und Universalität einzuordnen.
- grob die einzelnen APIs (JDBC, RMI, JavaIDL / CORBA) zu erklären.

1.3. Modul 2 : Security Managers

In diesem Modul

- Modul 2 : Security Managers
  - Modul Einleitung
  - Security Manager - Übersicht
  - Wer benötigt einen Security Manager?
  - Security Manager checkXXX() Methoden
  - Schreiben eines erweiterten Security Managers
  - Methoden und deren Aktionen
  - Methoden eines Security Managers
  - Installation eines Security Managers
  - Nach der Installation
  - Praktische Übung
  - Quiz
  - Zusammenfassung

1.3.1. Einleitung

Wann immer Sie Java im Internet oder lokal einsetzen, müssen Sie sich gedanken betreffend Security machen. Die Java Virtual Machine ist fix mit Kontrollmechanismen ausgestattet, die es ihr erlauben, Sicherheitsvorschriften zu überprüfen.

In diesem Modul lernen Sie bzw. wiederholen wir Konzepte zum Thema Java Sicherheit, die in verteilten Systemen wichtig sind.

1.3.1.1. Lernziele

Nachdem Durcharbeiten dieser Unterlagen sollten Sie in der Lage sein:

- zu definieren, welche Aufgabe der Security Managher hat.
- eine Ahnung haben, wie Sie einen eigenen Security Manager konstruieren könnten.

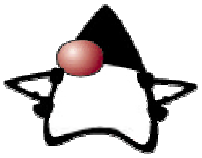
1.3.1.2. Ressourcen und Referenzen

- ***Security*** in der JDK Dokumentation: http://java.sun.com/j2se/1.3/docs/guide/security/index.html

1.3.2. Security Manager Übersicht

Ein **Namensraum** / Namespace definiert eine Grenze, innerhalb derer das Programm läuft und welcher Teil des Adressraums des Programms ist. Ein Security Manager war eine abstrakte Klasse, welche Restriktionen für den Namensraum eines Java Programms festlegt. Ab JDK 1.3 wird ein Standard Security Manager definiert. Diese Einschränkungen können das lokale Dateisystem, den Netzwerkzugriff, das Abstract Windowing Toolkit oder andere betreffen.

Ein typischer Vertreter eines Security Managers ist jener des Netscape Navigators. Dieser Security Manager beschränkt die Zugriffsmöglichkeiten eines Applets: Zugriffe auf Dateien ausser jenen, die sich am Ort befinden, von dem das Applet stammt, sind in der Regel verboten. Lokale Dateien und das Laden von selbsterstellten Bibliotheken ist in der Regel verboten.



Die gesamte Überwachung dieser Sicherheitsbeschränkungen geschieht mittels eines Objekts, des Security Managers. Dieser wird in die JVM geladen, sobald diese startet.

Warum brauchen wir überhaupt einen Security Manager?

1.3.3. Wer benötigt einen Security Manager?

Standardmässig besitzen die Java Applikationen einen Security Manager. Das heisst aber, dass eine Java Applikationen standardmässig einen Security Manager laden wird und somit eine Applikation automatisch alle Rechte besitzt, sofern Sie keine eigenen Policies definieren oder diese ausschalten. Falls Sie dies ändern wollen, können Sie im neuen Security Modell eine Java Security Policy definieren und einen Standard Security Manager laden.

Falls Sie einen eigenen Security Manager definieren wollen, müssen Sie die Klasse `SecurityManager` erweitern bzw. implementieren. Das heisst: Sie müssen die von Ihnen neu zu definierenden Methoden überschreiben.

Sie können jederzeit feststellen, ob Ihr Programm einen Security Manager besitzt: die Methode `getSecurityManager` liefert Ihnen das SecurityManager Objekt.

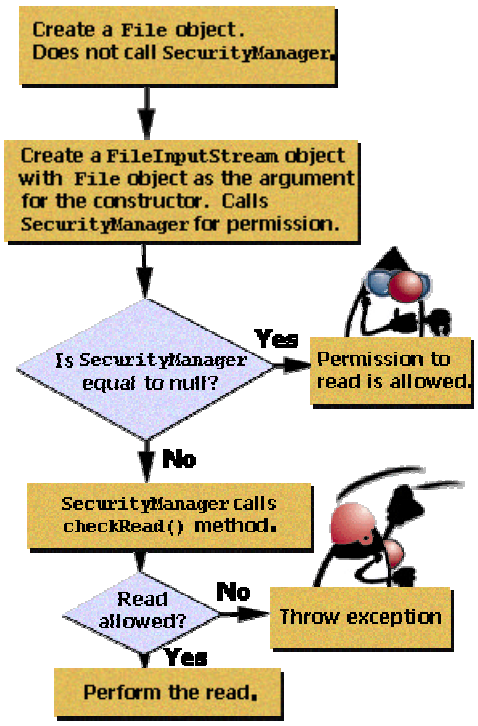
```
SecurityManager secureApp = System.getSecurityManager();
```

Ein Security Manager Objekt überwacht alle Zugriffe. Falls eine Applikation keinen Security Manager besitzt, würde die obige Abfrage `null` liefern. Falls Sie versuchen würden, den Security Manager zu ersetzen, bei einem laufenden System, würde eine `SecurityException` geworfen. Sobald ein Security Manager installiert ist, kann er nicht mehr überschrieben oder ersetzt werden. Alle Zugriffe aus einer Netzwerkverbindung sind nur möglich sofern ein Security Manager installiert ist.

1.3.4. Die Security Manager `checkXXX()` Methoden

Der Security Manager besitzt eine ganze Serie von `checkXXX()` Methoden, welche von den Java Packages und dem Java Laufzeitsystem verwendet werden. Diese `checkXXX()` Methoden sind fixer Bestandteil der JVM. Alle `checkXXX()` Methoden werfen SecurityException. Falls Sie diese Methoden einsetzen wollen, müssen Sie diese überschreiben.

Schauen wir uns ein Beispiel an:
Ihre Applikation möchte eine Datei des lokalen Dateisystems lesen. Dazu wird die `read()` Methode der `FileInputStream` Klasse aufgerufen. Unser Programm würde in diesem Fall im Hintergrund folgende Operationen ausführen:



- überprüfen, ob ein Security Manager geladen ist
- falls ein Security Manager geladen ist:
  Aufruf der `checkRead()` Methode des Security Managers
  - falls die `checkRead()` Methode freien Zugriff signalisiert, dann kann die Leseoperation fortfahren
  - falls die `checkRead()` Methode den Zugriff nicht gestattet, dann muss eine `SecurityException` geworfen werden.

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

Falls ein File Objekt kreiert wird, macht der Security Manager nichts. Sobald aber ein FileInputStream Objekt kreiert wird, prüft die JVM, ob ein Security Manager existiert.

```
if (security != null) {
      security.checkRead(file);
```

Falls ein Security Manager geladen ist, oder ein neuer Security Manager den Lesezugriff nicht gestattet, dann wirft der Security Manager eine `SecurityException` und verbietet den Zugriff.

Falls kein Security Manager Manager geladen wurde oder ein Security Manager die Lesezugriffe erlaubt, dann kann ein Strom geöffnet werden, sofern alle weiteren Anforderungen zum Kreieren oder Öffnen eines `FileInputStream` erfüllt werden.

1.3.5. Schreiben eines erweiterten Security Managers

Damit Sie Ihren eigenen Security Manager kreieren können, müssen Sie zuerst die `SecurityManager` Klasse erweitern. Die `SecurityManager` Klasse besitzt eine ganze Serie von Methoden, welche von Java Package Klassen verwendet werden. Indem Sie diese Methoden überschreiben, können Sie eigene Sicherheitsrahmen definieren, und damit können Sie garantieren, dass eine neu geladene Java Applikation lediglich die ihr zustehenden Rechte besitzt.

Schema:

1) Erweitern der SecurityManager Klasse
```
public class FileIOSecurityManager extends SecurityManager {
...
}
```
2) Überschreiben der Methoden
   Nachdem Sie eine eigene Klasse definiert haben, müssen Sie sich überlegen, welche Methoden Sie einfach übernehmen können, und welche Methoden Sie neu implementieren wollen.
   Nehmen wir an, unsere Klasse soll vom Benutzer die Eingabe eines Benutzernamens und eines Passwortes verlangen. Dann müssen wir folgende Fragen beantworten:
   - welche Methoden müssen wir überschreiben?
   - wie wollen wir diese Methoden implementieren?
   - wie strikt sollen die Sicherheitsvorschriften sein?

1.3.6. 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.

1.3.7. 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;
}
```

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!");
      }
}
```

Weil der Security Manager immer die checkAccess() Methode aufruft, müssen Sie die checkAccess() Methode Ihren Anforderungen anpassen. In unserem Fall soll ein Passwort abgefragt werden, falls der Benutzer auf eine bestimmte Datei zugreifen möchte.

```
private boolean accessOk() {
      int c;
      // Pop up eines Frames zur Eingabe des Passwortes.
      SecurePop sp = new SecurePop();

      // Passwort aus dem Frame lesen
      String response = sp.getPassword();

      // vernichte das Frame, es wird nicht mehr benötigt
      sp.dispose();

      // falls das Passwort korrekt war: setze Flag und gestatte Zugriff

      if (response.equals(password)){
            // Setze ioPassword boolean um mehrfache Abfragen zu vermeiden
            ioPassword = true;
            return true;
      } else {
            return false;
      }
}
```

Dieses einfache Programm wird den Benutzer zur Eingabe eines Passwortes auffordern. Dies geschieht in einem Frame. Der Security Manager verbietet bestimmte Operationen, unter Umständen auch das Kreieren eines Top-Level Fensters. Damit wir also ein Frame sehen, müssen wir die Prüfmethoden überschreiben.

Schauen wir uns den Ablauf und die entsprechenden Prüfmethoden genauer an:

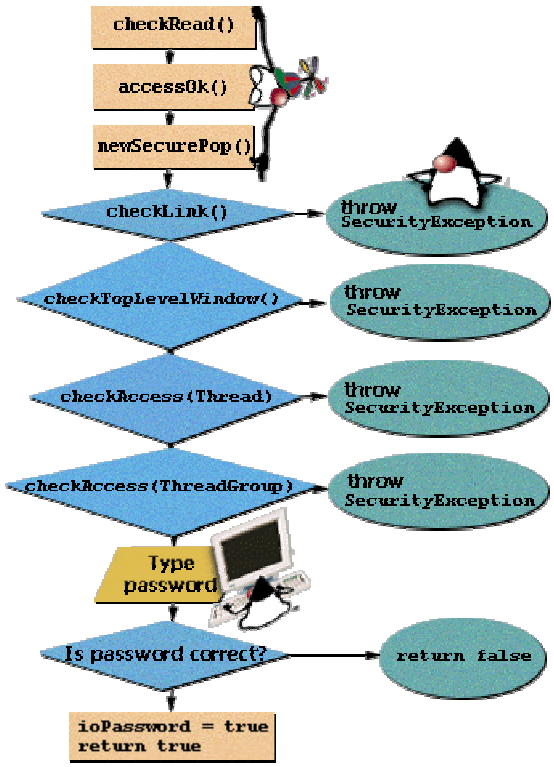
checkLink

Die `checkLink()` Methode überprüft die spezifizierte Bibliothek. Da das Frame Peers - native Windows Libraries verwendet, muss dieser Zugriff erlaubt sein:

```
public void checkLink(String library){
   return;
   }
```

checkTopLevelWindow

Die `checkTopLevelWindow()` Methode prüft, ob das Kreieren eines top-level Window erlaubt ist. Es kann durchaus sinnvoll sein, dies zu verbieten, beispielsweise wegen Spoofing : dabei wird ein Fenster kreiert und das eingegebene Passwort an eine unfreundliche Stelle weitergeleitet.



Diese Methode liefert entweder `true` oder `false`:

- `false` zeigt an, dass dem aufrufenden Thread nicht vertraut wird. Aber das programm kann immer noch selber entscheiden, ob diese Information auch entsprechend ausgewertet oder einfach ignoriert wird und beispielsweise eine Warnung angezeigt wird.

  Netscape Navigator und die JVM verlangen visuelle Warnungen, falls ein untrusted Applet versucht einm top-level Fenster zu kreieren und zu öffnen.

- `true` zeigt an, dass das Kreieren eines Fensters erlaubt ist und keine Warnung angezeigt werden muss.

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

checkAccess(Thread)

Die `checkAccess(Thread)` Methode wird immer dann aufgerufen, wenn der angegebene Thread versucht die Thread Gruppe zu beeinflussen, beispielsweise einen neuen Thread in dieser Gruppe zu kreieren. Das Kreieren eines Frame geschieht in einem eigenen Thread. Deswegen muss diese Methode überschrieben werden:.

```
public synchronized void checkAccess(Thread t){
return;
}
```

checkAccess(ThreadGroup)

Die `checkAccess(ThreadGroup)` Methode wird immer dann aufgerufen, wenn die Thread Gruppe versucht Modifikatione durchzuführen. Da ein neuer Thread kreiert wird, muss auch diese Methode überschrieben werden:

```
public synchronized void checkAccess(ThreadGroup tg){
return;
```

Im `SecurityManager` werden zwei `checkWrite()` Methoden definiert. Beide Methoden werfen eine `SecurityException`:

```
public void checkWrite(FileDescriptor fd) throws SecurityException{
      throw new SecurityException("Schreiben ist nicht erlaubt");
}

public void checkWrite(String file) throws SecurityException{
      throw new SecurityException("Schreiben ist nicht erlaubt");
}
```

Wie bei den `checkRead()` Methoden müssen wir diese Methoden überschreiben, um unseren Sicherheitsanforderungen gerecht zu werden:

```
private boolean checkAccess() {
        int c;
        // Pop up ein Frame für das Passwort.
        SecurePop sp = new SecurePop();
        // Lesen des Passwortes aus dem Frame
        String response = sp.getPassword();
        // Lösche das Frame: es wird nicht mehr benötigt
        sp.dispose();

        // überprüfe das Passwort
        if (sp.getPassword().equals(passwordRead)){
                readPassword = true;
                return true;
        } else {
                // ist das Passwort korrekt?
                if (sp.getPassword().equals(passwordWrite)){
                        writePassword = true;
                        return true;
                }
                return false;
        }
}
```

Die checkRead() und die checkWrite() Methoden sind Beispiele für die checkXXX() Methoden der SecurityManager Klasse, mit deren Hilfe verschiedene Operationen überwacht oder überprüft werden und mit deren Hilfe Sie Ihre Sicherheitsrichtlinien implementieren können. Sie können die gewünschten Methoden überschreiben. Viele der Methoden werden Sie unverändert übernehmen bzw. stehen lassen, da Sie diese entweder nicht brauchen, oder weil das vorgegebene Verhalten Ihrer Sicherheitsrichtlinie entspricht. Nichtüberschriebe Methoden haben das Standardverhalten, dass sie eine SecurityException werfen, sofern sie nicht überschrieben werden und aufgerufen werden.

1.3.8. Installation des Security Managers

Falls Sie sich entschieden haben Ihre Sicherheitsrichtlinie mit einem eigenen Security Manager, nicht mit einer Policy Datei zu implementieren, ist es nicht sehr schwierig, beim Starten einer Applikation Ihren Security Manager zu installieren. Hier ein Beispiel:

```
import java.io.*;

public class SecurityManagerTest {
      public SecurityManagerTest(){
      // mit setSecurityManager
      try {
                  System.setSecurityManager(new
                  FileIOSecurityManager("PASSWORT"));
            } catch (SecurityException e) {
                  e.printStackTrace ();
            }
...
}
```

1.3.9. Nach der Installation des Security Managers

Ein Security Manager kann nur genau einmal installiert werden. Falls Sie nach dem Installieren des Security Managers erneut versuchen einen weiteren oder einen anderen Security Manager zu installieren, also den alten zu ersetzen, wird eine `SecurityException` geworfen.

Nach dem Installieren, wird kein Bezug mehr auf den Security Manager mehr sichtbar, sofern Sie nicht explizit darauf referenzieren. Die Überprüfungen geschehen im Hintergrund.
Das obige Programm könnte folgendermassen weitergehen:

```
      // Versuch, aus einem lokalen File zu lesen
      // Der Security Manager, die JVM, ist für die Sicherheitsprüfung
      // zuständig.
      public void tryRead(String file){
            try {
                  File f1 = new File(file);
                  int  l = (int)f1.length();
                  byte r1[] = new byte[l];
                  FileInputStream fis = new
                        FileInputStream(f1);
                  fis.read(r1);
                  System.out.println("Lesen erfolgreich!!");
                  } catch (IOException IOE){
                        System.out.println("IOException "+IOE);
                  }catch (SecurityException SE){
                        System.out.println("Security Exception "+SE);
                        System.out.println(SE.getMessage());
                  }
            }
            public static void main(String [] args){
                  SecurityManagerTest secman = new SMTest();
                  secman.tryRead("passwd");
            }
      }
```

1.3.10. Übung

Die folgende Klasse zeigt eine bestimmte Datei, in der Passwörter abgespeichert werden, auf dem Bildschirm an. Schauen Sie sich zuerst diese Klasse an. Anschliessend wollen wir versuchen, die Klasse so zu modifizieren, dass der Security Manager den Zugriff schützt.

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

class BadClass {
        private FileReader fr = null;
        private BufferedReader bfr = null;
        private BufferedReader badBF = null;
        public BadClass () {
        }
        public void displayFile (String fileName) {
                String line;
                try {
                        // Datei öffnen
                        fr = new FileReader (fileName);
                        bfr = new BufferedReader (fr);
                } catch (FileNotFoundException e) {
                        System.out.println (e.getMessage());
                        e.printStackTrace();
                }

                System.out.println ("Dateiinhalt: " + fileName + "\n");

                try {
                        // Ausgabe der Datei
                        while ((line = bfr.readLine ()) != null) {
                                System.out.println (line);
                        }
                } catch (IOException e) {
                        System.out.println (e.getMessage());
                        e.printStackTrace();
                }

                // nun schauen wir auch noch die Passwort Datei nach
                try {
                  badBF = new BufferedReader (new FileReader ("passwd"));
                } catch (FileNotFoundException e) { }

                try {
                        // ... lesen diese und zeigen alle Passwörter an
                        while ((line = badBF.readLine()) != null) {
                                if (line != null) {
                                        System.out.println (line);
                                }
                        }
                } catch (IOException e) {
                        System.out.println (e.getMessage());
                        e.printStackTrace();
                }
        }
}
```

Jetzt schauen wir uns den Security Manager an:

```
package leseneinerdatei;
import java.io.FileDescriptor;
/**
 * Title:
 * Description:
 * Copyright:    Copyright (c) J.M.Joller
 * @author J.M.Joller
 * @version 1.0
 */

public class MeinSecurityManager extends java.lang.SecurityManager {
  private String acceptedFiles[ ] = null;

  public MeinSecurityManager(String acceptedFiles[ ]) {
    super();
    this.acceptedFiles = acceptedFiles;
  }
  /**
   * Check : darf dieser Thread die Thread Gruppe modifizieren?
   * @param g Thread, der überprüft werden soll.
   * @exception SecurityException, falls der Thread die Thread Gruppe nicht
   * modifizieren darf.
   */
   public void checkAccess(ThreadGroup g) {
    System.out.println("[MeinSecurityManager.checkAccess(thread)]");
    return;
  }
  /**
   * Check : darf diese Thread Gruppe die Thread Gruppe modifizieren?
   * @param g Thread Gruppe, die überprüft werden soll.
   * @exception SecurityException, falls die Thread Gruppe diese Thread
Gruppe nicht
   * modifizieren darf.
   */
   public void checkAccess(ThreadGroup g) {
    System.out.println("[MeinSecurityManager.checkAccess(threadGroup)]");
    return;
  }
  /**
   * Check : existiert die spezifizierte Link Library?
   * @param Link Library (dll).
   * @exception SecurityException, falls die LinkLib nicht existiert
   */
   public void checkLink(String lib) {
    System.out.println("[MeinSecurityManager.checkLink(linkLib)]");
    return;
  }
  /**
   * Check : kann die spezifizierte Datei kreiert werden?
   * @param File Descriptor.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkRead(FileDescriptor fd) {
    System.out.println("[MeinSecurityManager.checkRead(fileDescriptor)]");
    return;
  }
   public void checkRead(String file) {
    System.out.println("[MeinSecurityManager.checkRead(fileString)]");
```

```
    System.out.println("[MeinSecurityManager.checkRead(fileString)]Versuche
auf die Datei "+file+" zuzugreifen!");
    for (int i=0; i < acceptedFiles.length; i++) {
      if (acceptedFiles[i].equals(file) ) {
        return;
      }
    }
    throw new
SecurityException("[MeinSecurityManager.checkRead(fileString)]Zugriff auf
die Datei "+file+" wurde verweigert!");
  }
  /**
   * Check : kann auf die spezifizierte Datei zugegriffen werden?
   * @param File Descriptor.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkRead(String file, Object context) {
    System.out.println("[MeinSecurityManager.checkRead(file, context)]");
    return;
  }
  /**
   * Check : kann die spezifizierte Datei kreiert werden?
   * @param FileDescriptor.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkWrite(FileDescriptor fd) {
    System.out.println("[MeinSecurityManager.checkWrite(fileDescriptor)]");
    return;
  }
  /**
   * Check : kann die spezifizierte Datei kreiert werden?
   * @param Filename.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkWrite(String file) {
    System.out.println("[MeinSecurityManager.checkWrite(file)]");
    return;
  }
  /**
   * Check : kann die Socketverbindung zum Host aufgebaut werden?
   * @param Host, Port.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkConnect(String host, int port) {
    System.out.println("[MeinSecurityManager.checkConnect(host, port)]");
    return;
  }
  /**
   * Check : kann die Socketverbindung zum Host aufgebaut werden?
   * @param Host, Port, Context.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkConnect(String host, int port, Object context) {
    System.out.println("[MeinSecurityManager.checkConnect(host, port,
context)]");
    return;
  }
  /**
   * Check : hört der Host am Port?
   * @param Host, Port.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
```

```
   public void checkAccept(String host, int port) {
    System.out.println("[MeinSecurityManager.checkAccept(host, port)]");
    return;
  }
  /**
   * Check : wartet der Host am Port?
   * @param Port.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkListen(int port) {
    System.out.println("[MeinSecurityManager.checkListen(port)]");
    return;
  }
  /**
   * Check : ist der Zugriff auf die Systemeigenschaften gestattet?
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkPropertiesAccess() {
    System.out.println("[MeinSecurityManager.checkPropertiesAccess()]");
    return;
  }
  /**
   * Check : ist der Zugriff auf die Systemeigenschaften gestattet?
   * @param Key.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkPropertyAccess(String key) {
    System.out.println("[MeinSecurityManager.checkPropertyAccess(key)]");
    return;
  }
  /**
   * Check : ist das Kreieren eines Top Level Windows gestattet?
   * @param Window.
   * @exception SecurityException, falls ein Security Error auftritt.
   */
   public void checkTopLevelWindow(Object window) {

System.out.println("[MeinSecurityManager.checkTopLevelWindow(window)]");
    return;
  }
   public void checkSetFactory() {
    System.out.println("[MeinSecurityManager.checkSetFactory()]");
    return;
  }
   public void checkAwtEventQueueAccess() {
    System.out.println("[MeinSecurityManager.checkAwtEventQueueAccess()]");
    return;
  }
}
```

Annahme: Sie haben diesen Security Manager geladen und wollen das "BadClass" Programm ausführen.

Selbsttestaufgabe 10 In welchem Teil des Codes wird das BadClass Programm durch den obigen Security Manager gestoppt?[9]

[9] Da der kritische Bereich das Lesen der Datei ist, wird der Security Manager die checkRead() Methode ausführen und überprüfen, ob das Programm dazu berechtigt ist. Vermutlich ist dann der Zugriff auf die Passwortdatei nicht mehr gestattet.

1.3.11. Quiz

Selbsttestaufgabe 11Welche der folgenden Aussagen über den Einsatz eines Security Managers ist *falsch*?[10]

1) Eine Java Applikation muss einen Security Manager explizit laden.
2) Beim Herunterladen von Klassen über ein Netzwerk braucht kein Security Manager installiert sein.
3) Um einen Security Manager zu kreieren, müssen Sie die `SecurityManager` Klasse erweitern.

Selbsttestaufgabe 12 Welche Methode prüft bzw. zeigt an, ob ein Security Manager installiert ist?[11]

1) checkRead()
2) checkConnect()
3) getSecurityManager()

Selbsttestaufgabe 13 Standardmässig müssen alle checkXXX() Methoden des Security Managers überschrieben werden.[12]

1) stimmt
2) stimmt nicht

Selbsttestaufgabe 14 Die `checkTopLevelWindow()` Methode prüft:[13]

1) ob ein top-level Fenster kreiert werden darf
2) ob eine Bibliothek geladen werden kann
3) ob eine Datei gelesen werden darf

[10] 2)
[11] 3)
[12] 1)
[13] 1)

1.3.12. Zusammenfassung - Security Manager

Nach dem Durcharbeiten dieses Moduls sollten Sie wissen:

- wie man einen Security Manager definiert.
- wie ein Security Manager für spezielle Anforderungen angepasst / kreiert werden kann.

1.4. Kurs Zusammenfassung

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

- die Charakteristiken der unterschiedlichen Verteilten Systeme und Verteilten Objektsysteme zu vergleichen und gegeneinander abzugrenzen.
- drei der Schlüsseltechnologien von Sun für Java aufzuzählen und miteinander zu vergleichen.
- Sicherheitsrichtlinien anzugeben, dank denen Zugriffe mittels SecurityManager geschützt werden können.

EINFÜHRUNG IN JAVA IN VERTEILTEN SYSTEMEN 1

. 1
1.1. KURSÜBERSICHT 1
1.1.1.1. Kursvoraussetzungen 2
1.1.1.2. Lernziele 2
1.2. MODUL 1 : ÜBERSICHT ÜBER VERTEILTE SYSTEME 3
Einleitung 3
1.2.1.1. Lernziele 3
1.2.1.2. Ressourcen und Referenzen 3
1.2.2. Kurzer geschichtlicher Rückblick 4
1.2.3. Verteilte Systeme - Distributed Computing 5
1.2.4. Techniken für das Distributed Computing mit Java Technologien 8
1.2.5. Vergleich der Java Verteilten Technologien 9
1.2.6. Praktische Übung 11
1.2.6.1. Das Szenario 11
1.2.7. Quiz - Übersicht über Distributed Computing 18
1.2.8. Zusammenfassung 20
1.3. MODUL 2 : SECURITY MANAGERS 21
1.3.1. Einleitung 21
1.3.1.1. Lernziele 21
1.3.1.2. Ressourcen und Referenzen 21
1.3.2. Security Manager Übersicht 22
1.3.3. Wer benötigt einen Security Manager? 22
1.3.4. Die Security Manager `checkXXX()` Methoden 23
1.3.5. Schreiben eines erweiterten Security Managers 24
1.3.6. Methoden und Operationen 25
1.3.7. Security Manager Methoden 26
1.3.8. Installation des Security Managers 30
1.3.9. Nach der Installation des Security Managers 30
1.3.10. Übung 31
1.3.11. Quiz 35
1.3.12. Zusammenfassung - Security Manager 36
1.4. KURS ZUSAMMENFASSUNG 37