N Zusammenfassung der 3. Übung

- Exceptions
- Streams
- Threads (Teil1)

O Überblick über die 4. Übung

- Lösung der Aufgabe 3
- Sockets
- Design-Patterns
- Threads (Teil 2)
- Serialisierung
- ClassLoader
- SecurityManager
- Hinweise zur 4. Aufgabe

O.1 Lösung der 3. Aufgabe

- Shape.java
- ShapeContainer.java
- ShapeContainerImpl.java

1 Shape.java

```
import java.awt.Graphics;
public abstract class Shape {
        int x,y,width,height;
        public Shape(int x, int y, int width, int height) { /* ... */}
        public void move(int x, int y) { /* ... wie in Aufgabe 2 */}
        public int getX(){ return x;}
        public int getY(){ return y; }
        public int disX(int x) { return x-this.x; }
        public int disY(int y) { return y-this.y; }
        public abstract void draw(Graphics g);
        public abstract boolean isInside(int x, int y);

        public void resize(int w, int h) {
                this.width = w;
                this.height = h;
        }
        public String getSpec() {
                return getClass().getName()
                            + " " + x
                            + " " + y
                            + " " + width
                            + " " + height;
        }
} //Shape
```

2 ShapeContainer.java

```
import java.io.*;
import java.awt.Graphics;
import java.util.StringTokenizer;

public interface ShapeContainer {

        public void addShape(String spec) throws FormatException;

        public Shape getShape(int x, int y);

        public void deleteShapes(int x, int y);

        public void load(String filename)
                              throws IOException, FormatException;

        public void save(String filename) throws IOException;

        public void paintDrawing(Graphics g);

}// ShapeContainer
```


3 ShapeContainerImpl.java

```
import java.io.*;
import java.awt.Graphics;
import java.util.StringTokenizer;

public class ShapeContainerImpl implements ShapeContainer{

    Shape[] shapes;

    public ShapeContainerImpl (){
        this(100);
    }
    public ShapeContainerImpl (int number){
        shapes = new Shape[number];
    }

    public void addShape(Shape shape){
        for(int i=0; i < shapes.length; i++) {
            if (shapes[i] == null) {
                shapes[i] = shape;
                break;
            }
        }
    }
                                                //- Fortsetzung folgt -
```


3 ShapeContainerImpl.java (2)

```
    public void addShape(String classname, String xstr, String ystr)
                                                    throws FormatException {
            /* wie in Aufgabe 2 */
    }

    public void addShape(String classname, String xstr, String ystr,
                String wstr, String hstr) throws FormatException {
            /* ... */
    }


    public void addShape(String spec) throws FormatException {
            StringTokenizer t = new StringTokenizer(spec);
            if (t.countTokens() != 5) {
                throw new FormatException("incomplete shape spec");
            }
            addShape(t.nextToken(), t.nextToken(), t.nextToken(),
                                        t.nextToken(), t.nextToken());
    }



                                                //- Fortsetzung folgt -
```


3 ShapeContainerImpl.java (3)

```
public void save(String filename) throws IOException{
        PrintWriter out = new PrintWriter(new FileWriter(filename));
        for(int i=0; i<shapes.length; i++) {
            if (shapes[i] != null) {
                out.println(shapes[i].getSpec());
            }
        }
        out.close();
    }
public void load(String filename) throws IOException, FormatException {
        clear();
        BufferedReader in = new BufferedReader(new FileReader(filename));
        String line;
        while((line = in.readLine()) != null) {
            addShape(line);
        }
    }
public void clear() {
        for(int i=0; i<shapes.length; i++) {
            shapes[i] = null;
        }
    }
                                                //- Fortsetzung folgt -
```

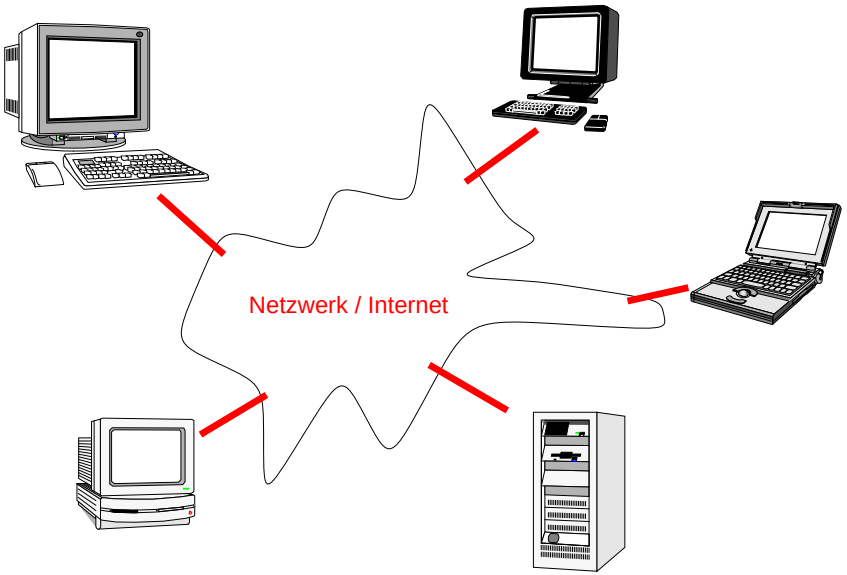
3 ShapeContainerImpl.java (4)

```
    public Shape getShape(int x, int y){
        Shape selected = null;
        for(int i=0; i < shapes.length; i++) {
            if (shapes[i] != null && shapes[i].isInside(x,y)) {
                selected = shapes[i];
                break;
            }
        }
        return selected;
    }
    public void deleteShapes(int x, int y){
        for(int i=0; i<shapes.length; i++){
            if (shapes[i] != null && shapes[i].isInside(x,y)) {
                shapes[i] = null;
            }
        }
    }
    public void paintDrawing(Graphics g) {
        for(int i=0; i<shapes.length; i++) {
            if (shapes[i] != null)
                shapes[i].draw(g);
        }
    }
}// ShapeContainerImpl.java
```

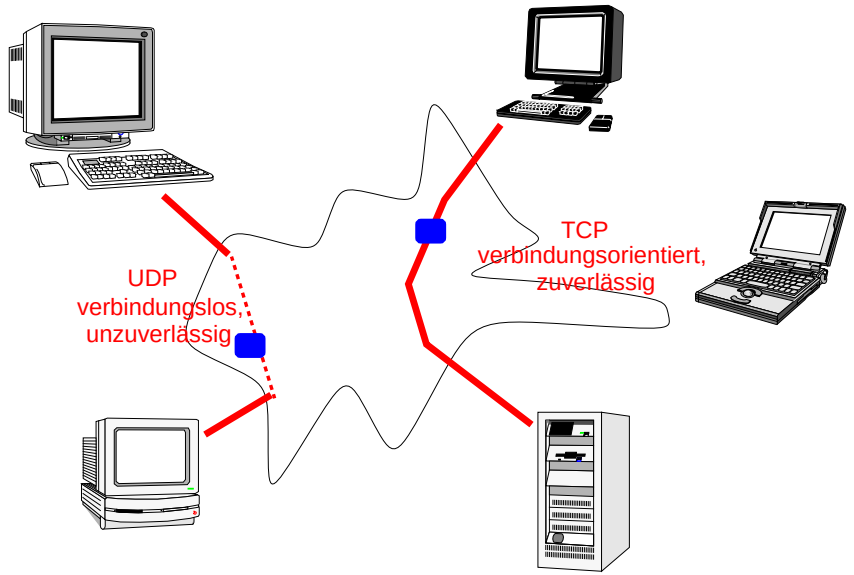

P.1 Adressierung: InetAddress

- IP-Adresse:
  - DNS-Form: www4.informatik.uni-erlangen.de
  - durch Punkte getrenntes Quadtupel: 131.188.34.42
- **`java.net.InetAddress`** enthält IP-Adressen
- **`InetAddress`** hat keinen öffentlichen Konstruktor.
  Instanzen können folgendermassen erzeugt werden:
  - **`getLocalHost()`**
  - **`getByName(String hostname)`**
  - **`getAllByName(String hostname)`**
- **`InetAddress`** stellt Konvertierungsmethoden zur Verfügung:
  - **`byte[] getAddress()`**: IP-Adresse in Bytearray
  - **`String getHostAddress()`**: Stringdarstellung
  - **`String getHostName()`**: Rechnername (DNS-Form)

P Netzwerkprogrammierung



P.2 Sockets



1 Verbundungsorientierte Sockets

- **`java.net.Socket`**
  - TCP/IP
  - zuverlässig
  - Repräsentiert einen Kommunikationsendpunkt bei einem Client oder einem Server
- Erzeugen eines neun Sockets:

```
socket= new Socket("www4.informatik.uni-erlangen.de",80);
```

- Ein Kommunikationsendpunkt ist definiert durch *Rechnername und Port* (Ports: 16 bit, < 1024 privilegiert)
- **`close`** schließt den Socket.

2 ServerSocket

- **`java.net.ServerSocket`**
  - wird serverseitig verwendet um auf Verbindungsanfragen von Clients zu warten
- **`accept`** wartet auf Verbindungsanfragen
- für eine neue Verbindung wird ein neues **`Socket`**-Objekt zurückgegeben:

```
ServerSocket ServerSocket = new ServerSocket(10412);
Socket socket = serverSocket.accept();
```

- **`close`** schließt den Port.

3 Ein- / Ausgabe über Sockets

- Lesen von einem Socket mittels **`InputStream`**:

```
InputStream inStream = socket.getInputStream();
```

- Schreiben auf einen Socket mittels **`OutputStream`**:

```
OutputStream outStream = socket.getOutputStream();
```

- aus diesen Strömen können leistungsfähigere Ströme erzeugt werden:

```
DataOutputStream out =
  new DataOutputStream(new BufferedOutputStream(outStream));
```


4 TCP Client/Server

Server

```
ServerSocket serverSocket = new ServerSocket(5124);
```

Client

4 TCP Client/Server (2)

Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept(); // accept blockiert
```

Client

```

```


4 TCP Client/Server (3)

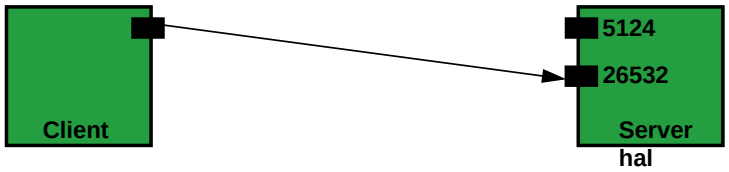
Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept(); // accept blockiert
```

Client

```
Socket socket = new Socket("hal",5124);
```


4 TCP Client/Server (4)



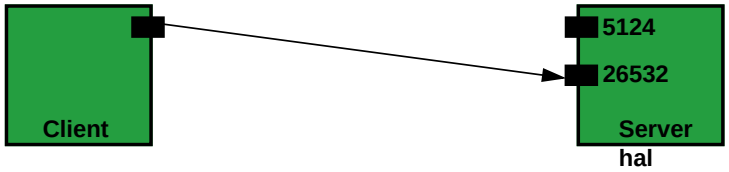
Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept(); // accept kehrt zurück
```

Client

```
Socket socket = new Socket("hal",5124);
```


4 TCP Client/Server (5)



Server

```
ServerSocket serverSocket = new ServerSocket(5124);
Socket socket = serverSocket.accept();
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```

Client

```
Socket socket = new Socket("hal",5124);
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
```

P.3 Verbindungslose Sockets

- **`java.net.DatagramSocket`**
  - UDP/IP
  - unzuverlässig: Datagramme können verloren gehen!
  - geringe Latenzzeit
  - Konstruktoren:

    `DatagramSocket(int port)`
    bindet an den lokalen Port **`port`**

    `DatagramSocket()`
    bindet an irgendeinen lokalen Port
  - Methoden:

    `send(DatagramPacket packet)`
    sendet ein Paket, die Zieladresse muss im Paket eingetragen sein

    `receive(DatagramPacket packet)`
    empfängt ein Paket, die Absenderadresse ist im Paket enthalten

1 Empfänger

- Pakete an einem bestimmten Port empfangen:

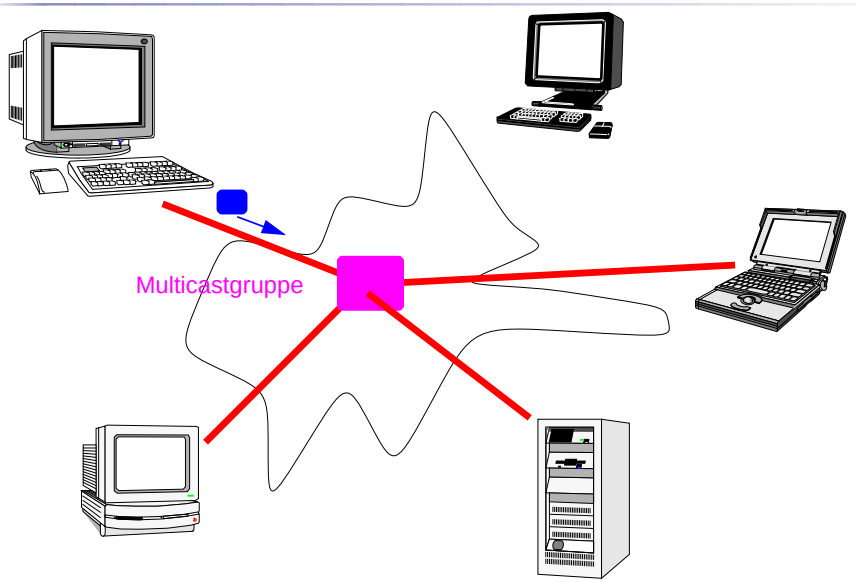
```
DatagramSocket socket = new DatagramSocket(10412);
byte[] buf = new byte[1024];
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
InetAddress from = packet.getAddress();
int bytesReceived = packet.getLength();
```


2 Sender

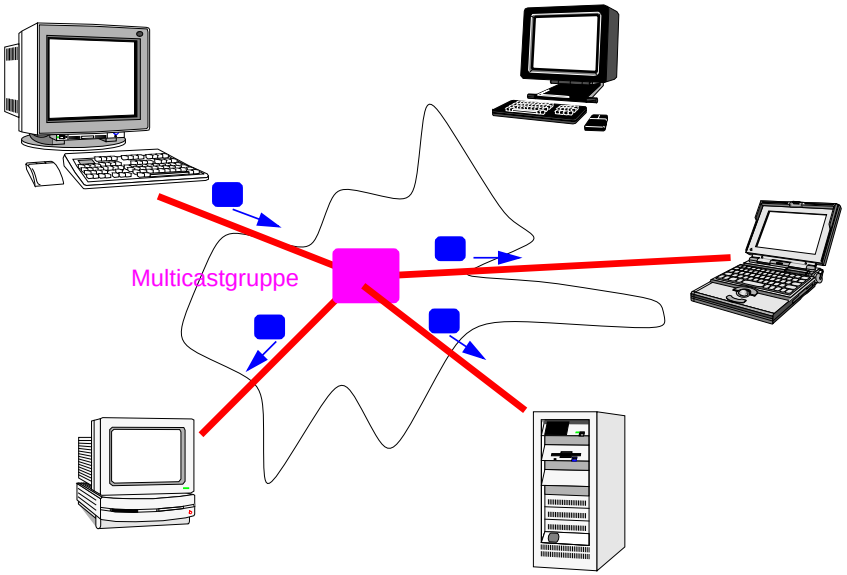
- Pakete von einem beliebigen Port verschicken:

```
DatagramSocket socket = new DatagramSocket();
byte[] buf = new byte[1024];
buf[0] = ...

InetAddress addr = InetAddress.getByName("faui40");
int port = 10412;
DatagramPacket packet;
packet = new DatagramPacket(buf, buf.length, addr, port);
socket.send(packet);
```


P.4 Multicast-Sockets



P.4 Multicast-Sockets



P.4 Multicast-Sockets

- **`java.net.MulticastSocket`**
  - verbindungslos (Unterklasse von **`DatagramSocket`**)
  - verwendet IP-Adressen der *Klasse D* (**`224.0.0.1`** bis **`239.255.255.255`**)
  - nach dem Erzeugen des Sockets kann man Pakete verschicken
  - Um Pakete zu empfangen muss man der Gruppe beitreten mittels **`joinGroup()`**
  - Die Paketverbreitung wird mit Hilfe des Parameters “time-to-live” (TTL) gesteuert.
- Beispiel:

```
InetAddress group = InetAddress.getByName("228.5.6.7");
MulticastSocket socket = new MulticastSocket(6789);
socket.setTimeToLive((byte)2);
socket.joinGroup(group);
```

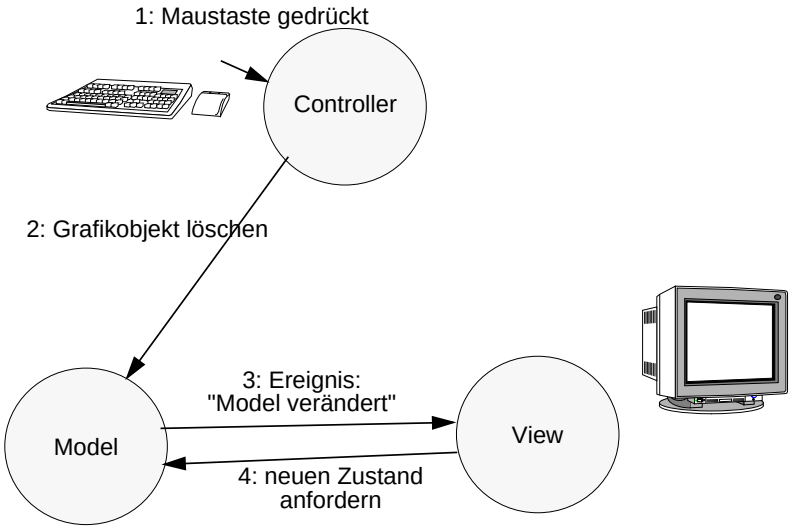

P.5 Zusammenfassung

- **`Socket`**: Endpunkt (Server oder Client) einer TCP-Kommunikation
  - enthält Zieladresse / -port und lokalen Port
  - Daten werden mittels Strömen (Streams) gelesen und geschrieben
- **`SocketServer`**: ein Server-Endpunkt, erzeugt Instanzen von **`Socket`**
- **`DatagramSocket`**: UDP-Kommunikation
  - **`send()`**/**`receive()`** um Daten zu verschicken / empfangen.
  - Zieladresse ist in einem **`DatagramPacket`**-Objekt enthalten.
- **`MulticastSocket`**: Multicast-UDP-Kommunikation
  - verwendet einen reservierten Bereich von IP-Adressen
  - bevor man Daten empfängt, muss man mittels **`joinGroup()`** einer Multicastgruppe beitreten

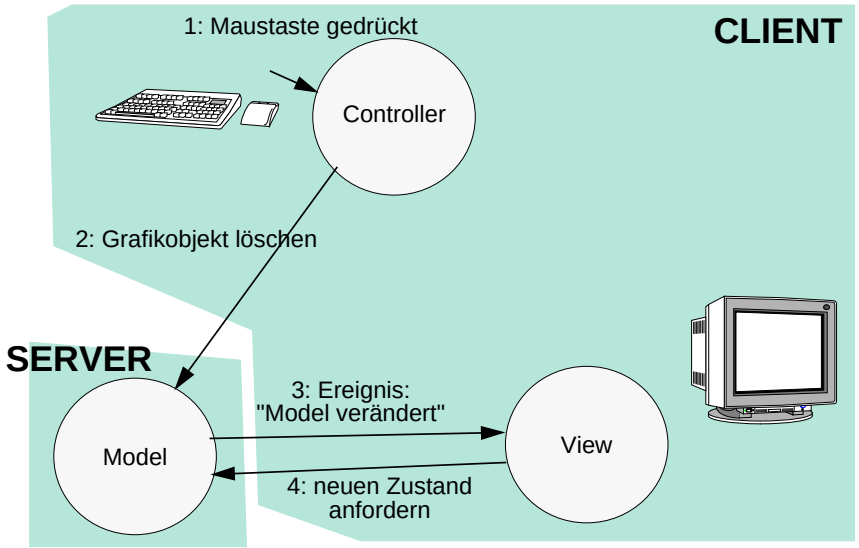
Q Design Patterns (Entwurfsmuster)

- "elements of reusable design"
- Beispiele:
  - Model-View-Controller
  - Observer
  - Iterator
  - Proxy
  - Command
  - Factory

1 Model-View-Controller

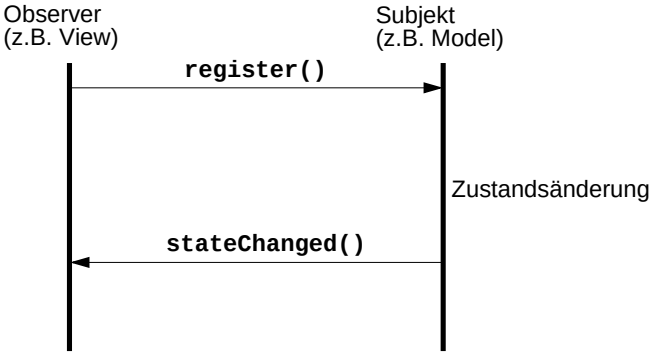


1 Model-View-Controller (2)



2 Observer

- wird z.B. im MVC-Muster verwendet um Änderungen des Modells zu beobachten



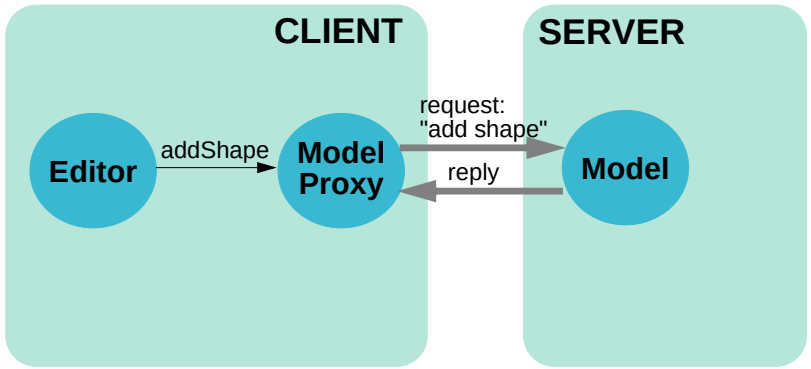
3 Iterator

- wird verwendet um durch eine Menge von Objekten zu "laufen"
- Ein Iterator ist dafür verantwortlich die aktuelle Position zu verwalten.

```
class Iter implements java.util.Enumeration {
  int cursor;
  Shape[] shapes;

  public Iter(Shape[] shapes) {
    this.shapes = new Shape[shapes.length];
    System.arraycopy(shapes, 0, this.shapes, 0, shapes.length);
  }

  public boolean hasMoreElements() {
   while (cursor<shapes.length && shapes[cursor]==null) cursor++;
    return cursor < shapes.length; }

  public Object nextElement() { return shapes[cursor++]; }
}
```

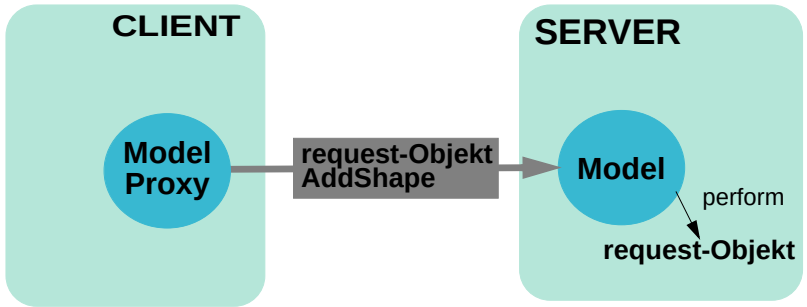
4 Proxy

- wird verwendet um einen lokalen Stellvertreter zu einem entfernten Objekt zu haben
- implementiert die gleiche Schnittstelle wie das “echte” Objekt
- WhiteBoard mit entferntem Modell:



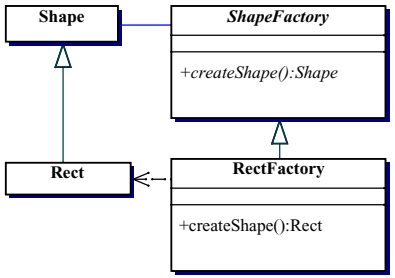
5 Command

- wird verwendet um eine Anfrage zum Server zu transferieren
- Der Zustand enthält Informationen vom Client
- Die **`perform()`**-Methode wird vom Server aufgerufen
- Die Parameter enthalten Informationen vom Server



6 Factory

- Definiert eine Klassenschnittstelle zum Erzeugen von Objekten
- Konkrete Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist



R Ausgewählte Kapitel des Java Laufzeitsystems (Teil 2)

- Threads (Teil 2)
- Serialisierung
- ClassLoader
- SecurityManager

R.1 Threads (Teil 2)

- Daemon-Threads
- Threadgruppen

1 Daemon-Threads

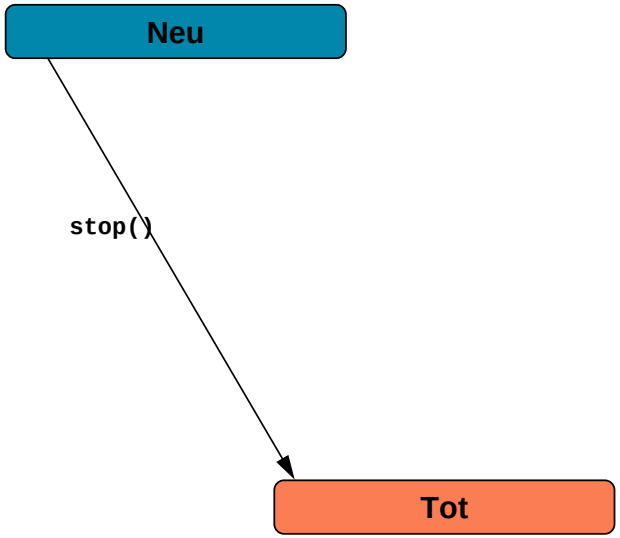
- Daemon-Threads werden für Hintergrundaktivitäten genutzt
- Sie sollen nicht für die Hauptaufgabe einer Programmes verwendet werden
- Sobald alle *nicht-daemon* Threads beendet sind, ist auch das Programm beendet.
- Woran erkennt man, ob ein Thread ein Daemon-Thread sein soll?
  - Wenn man keine Bedingung für die Beendigung des Threads angeben kann.
- Wichtige Methoden der Klasse **`Thread`**:
  - **`setDaemon(boolean switch)`**: ein- oder ausschalten der Daemon-Eigenschaft
  - **`boolean isDaemon()`**: Prüft ob ein Thread ein Daemon ist.

2 Die Klasse `ThreadGroup`

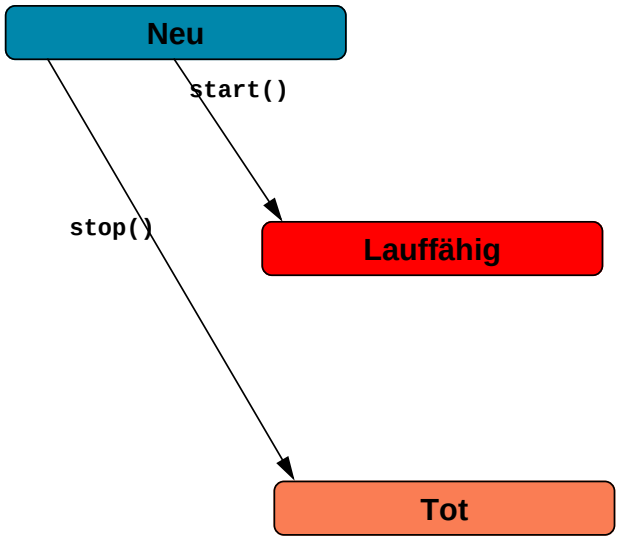
- Gruppe von verwandten Threads (**`ThreadGroup`**):
  - Eine Threadgruppe kann Threads enthalten und andere Threadgruppen.
  - Ein Thread kann nur Threads in der eigenen Gruppe beeinflussen.
- Methoden, die nur mit Threads der gleichen Gruppe angewendet werden können:
  - **`list()`**
  - **`stop()`**
  - **`suspend()`**
  - **`resume()`**

3 Zustände von Threads

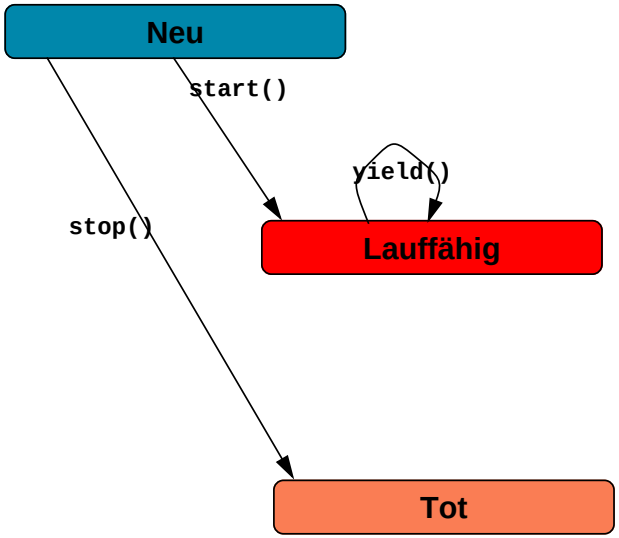
3 Zustände von Threads (2)



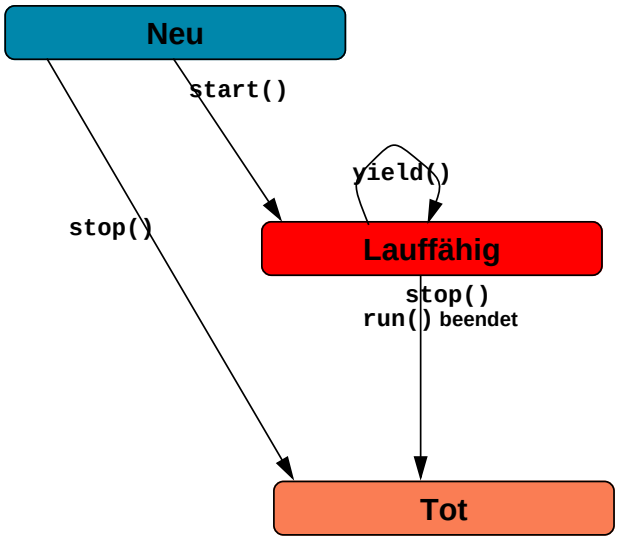
3 Zustände von Threads (3)



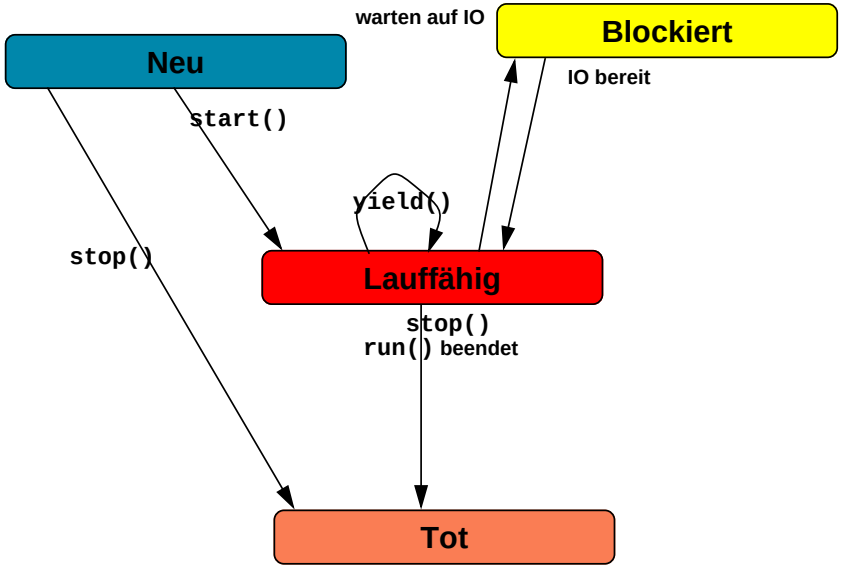
3 Zustände von Threads (4)



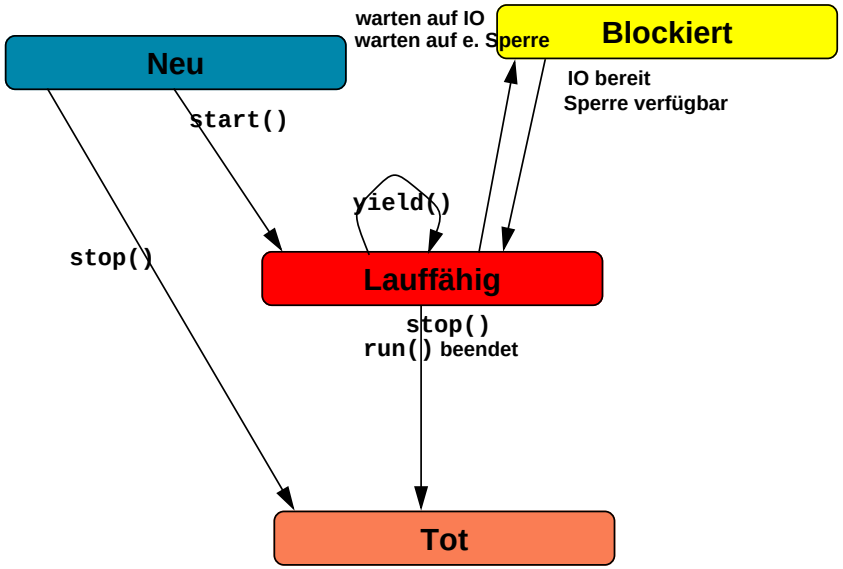
3 Zustände von Threads (5)



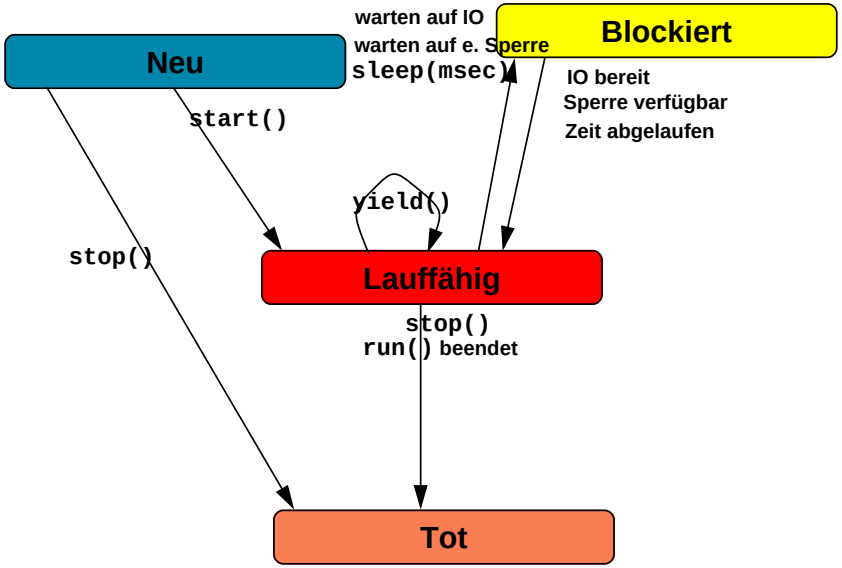
3 Zustände von Threads (6)



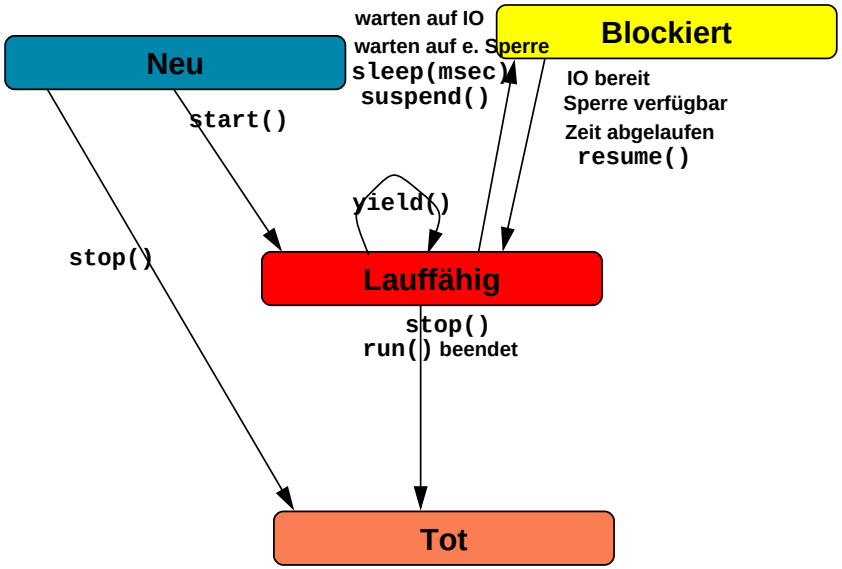
3 Zustände von Threads (7)



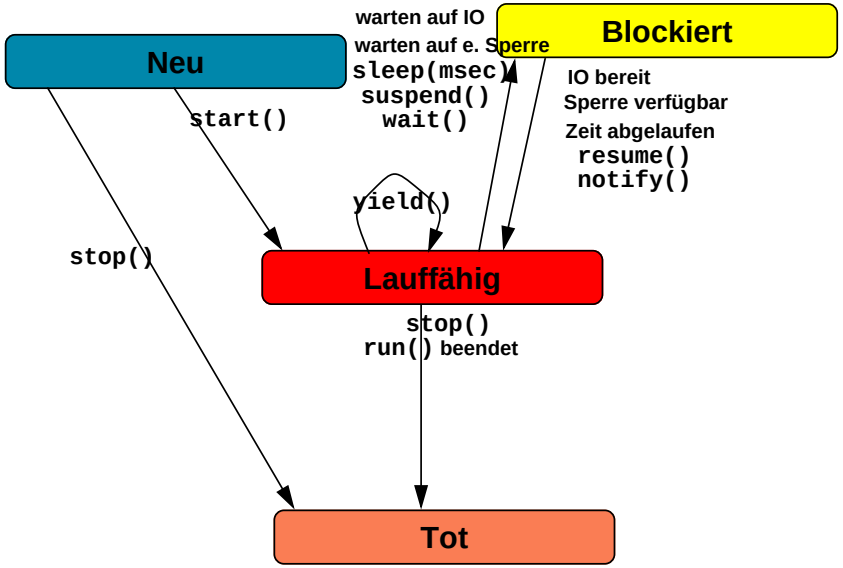
3 Zustände von Threads (8)



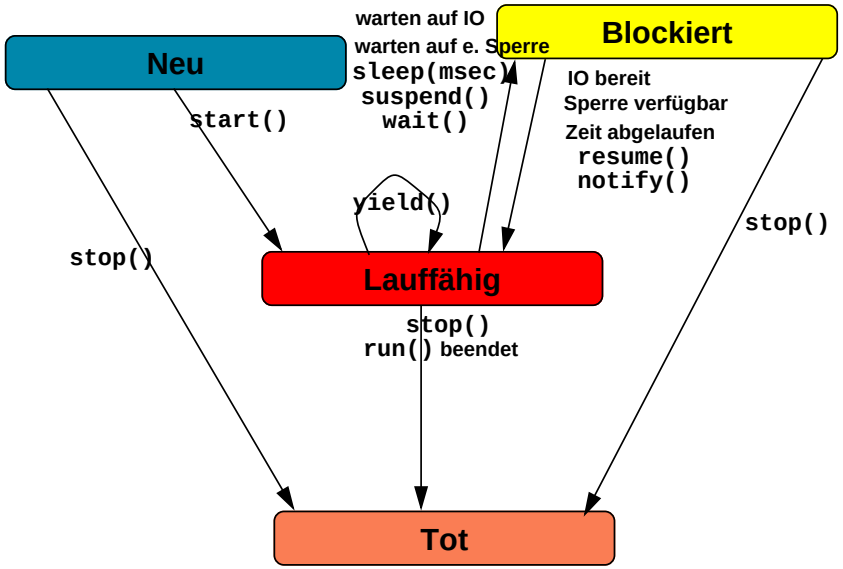
3 Zustände von Threads (9)



3 Zustände von Threads (10)



3 Zustände von Threads (11)



4 Veraltete Methoden der Klasse Thread

- **`stop()`**, **`suspend()`**, **`resume()`** sind seit Java 1.2 unerwünscht.
- **`stop()`** gibt alle Sperren des Thread frei - kann zu Inkonsistenzen führen
- **`suspend()`** und **`resume()`** können zu einem Deadlock führen:
  - suspend gibt keine Sperren frei
  - angehaltener Thread kann Sperren halten
  - Thread, der **`resume`** aufrufen will blockiert an einer Sperre

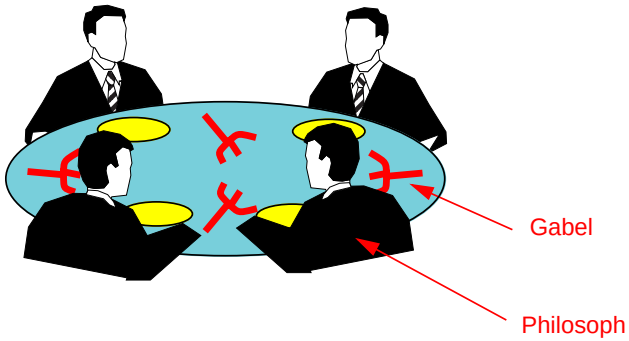
R.2 Korrektheit nebenläufiger Programme

- Safety: "Es passiert niemals etwas Schlechtes"
- Liveness: "Es passiert überhaupt etwas"

1 Safety

- gegenseitiger Ausschluss durch **`synchronized`**
- optimistische Nebenläufigkeitskontrolle

2 Deadlock: Das Philosophenproblem



denken → essen → denken → essen → ...

- ein Philosoph braucht beide Gabeln zum Essen
- alle Philosophen nehmen zuerst die rechte Gabel dann die linke → Verklemmung

3 Liveness

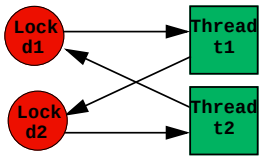
- keine Sprachunterstützung zur Deadlock-Verhinderung/Erkennung
- Deadlock-Beispiel:

```
class Deadlock implements Runnable {
  Deadlock other;
  void setOther(Deadlock other) {this.other = other; }
  synchronized void m() {
    try { Thread.sleep(1000); }
       catch(InterruptedException e) {}
    other.m();
  }
  public void run() { m(); }
}
```


3 Liveness (2)

- Verwendung, die zum Deadlock führt:

```
Deadlock d1 = new Deadlock();
Deadlock d2 = new Deadlock();
d1.setOther(d2); d2.setOther(d1);
Thread t1 = new Thread(d1);
t1.start();
Thread t2 = new Thread(d2);
t2.start();
try { t1.join(); t2.join(); } catch(InterruptedException e) {}
```



4 Deadlocks

- Counter

```
class Counter {
    private int count = 0;

    public synchronized void inc() { count++; }

    public int getCount() {return count; }

    public void setCount(int count) { this.count = count; }

    public synchronized void swap(Counter counter) {
        synchronized (counter) { // Deadlock Gefahr
            int tmp = counter.getCount();
            counter.setCount(count);
            count = tmp;
        }
    }
}
```


5 Deadlock - Vermeidung (1)

- Verhinderung zyklischer Ressourcenanforderung, Ordnung auf Locks

```
class Counter {
  ...
  public void swap(Counter counter) {
    Counter first = this;
    Counter second = counter;
    if (System.identityHashCode(this)
         < System.identityHashCode(counter)) {
      first = counter;
      second = this;
    }
    synchronized (first) {
      synchronized (second) {
        int tmp = counter.getCount();
        counter.setCount(count);
        count = tmp;
      }
    }
  }
```


5 Deadlock Vermeidung (2)

- Ressourcen (Locks) werden atomar angefordert:

```
class Counter {
  ...
  static Object lock = new Object();
  public void swap(Counter counter) {
    synchronized (lock) {
      synchronized(this) {
        synchronized (counter) {
          int tmp = counter.getCount();
          counter.setCount(count);
          count = tmp;
        }
      }
    }
  }
}
```


6 Nachteile der Java-Locks

- Methoden (lock, unlock) von Java-Locks sind unsichtbar, nur mit synchronized beeinflussbar
- kein Timeout beim Warten auf ein Lock möglich (Deadlock-Erkennung)
- Es können keine Unterklassen von Locks mit speziellem Verhalten erzeugt werden (Authentifizierung, Queueing-Strategien, ...)
- Locks können nicht referenziert werden (keine Deadlock-Erkennung oder Recovery möglich)

7 Optimistische Nebenläufigkeitskontrolle

- Vorteile:
  - keine Deadlocks
  - höhere Parallelität möglich
- Nachteile:
  - Designs werden komplexer
  - ungeeignet bei hoher Last
- Rollback/Recovery
  - Aktionen müssen umkehrbar sein, keine Seiteneffekte
  - zu jeder Methode muss es eine “Antimethode” geben
- Versioning
  - Methoden arbeiten auf shadow-Kopien des Objektzustandes
  - atomares commit überprüft, ob sich Ausgangszustand geändert hat (Konflikt) und setzt shadow-Zustand als neuen Objektzustand

7 Optimistisch - Beispiel (1)

- Counterzustand (Instanzvariablen) ist in separates Objekt ausgelagert (Memento Design-Pattern)

```
class CounterState {
  int count;
  CounterState(CounterState state) { ... }
  void inc() { ... }
  void dec() { ... }
  void swap(CounterState counter) { ... }
}
```


7 Optimistisch - Beispiel (2)

- Counter führt alle Operationen auf Kopie des Zustands aus
- am Ende wird die Kopie in einer atomaren Operation als Ist-Zustand gesetzt

```
class Counter {
  CounterState state;
  synchronized boolean commit(CounterState assumed,
                              CounterState next) {
    if (state != assumed) return false;
    state = next;
    return true;
  }
  void inc {
    do {
      assumed = state;
      next = new CounterState(assumed);
      next.inc();
    } while ( ! commit(assumed, next));
  }
}
```


R.3 Objekt Serialisierung

- Motivation:
  - Objekte sollen von der Laufzeit einer Anwendung unabhängig sein
  - Objekte sollen zwischen Anwendungen ausgetauscht werden können

1 Objektströme

- Mit Objektströmen können Objekte in Byteströme geschrieben werden und von dort gelesen werden.
- Klasse **`java.io.ObjectOutputStream`**
  - **`void writeObject(Object o)`**: Serialisierung eines Objekts (transitiv) (java.io.NotSerializableException)
- Klasse **`java.io.ObjectInputStream`**
  - **`Object readObject()`**: Lesen eines serialisierten Objekts (ClassNotFoundException)

2 Beispiel

- Speichern eines Strings und eines **`Date`**-Objekts:

```
FileOutputStream f = new FileOutputStream("/tmp/objects");
ObjectOutput s = new  ObjectOutputStream(f);
s.writeInt(42);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
f.close();
```

- Lesen der Objekte:

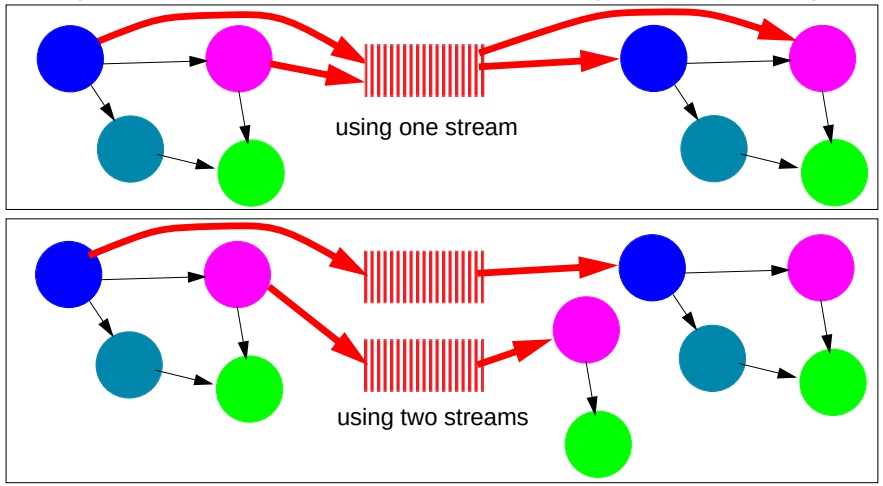
```
FileInputStream in = new FileInputStream("/tmp/objects");
ObjectInputStream s = new ObjectInputStream(in);
int i = s.readInt();
String today = (String)s.readObject();
Date date = (Date)s.readObject(); //! ClassNotFoundException
in.close();
```


3 Schnittstellen

- "Marker Interface" **`java.io.Serializable`**:
  - Instanzvariablen werden automatisch gesichert
  - Variablen, die mit **`transient`** deklariert wurden, werden nicht gesichert
- Interface **`java.io.Externalizable`**:
  - Ein Objekt kann seine Serialisierung selbst vornehmen
  - folgende Methoden müssen implementiert werden:
    - **`writeExternal(ObjectOutput out)`**
    - **`readExternal(ObjectInput in)`**

4 Probleme

- Alle Objekte eines Objekt-Graphen sollten in den gleichen Strom geschrieben werden, ansonsten werden die Objekte beim lesen dupliziert



4 Probleme (2)

- der Objekt-Graph muss atomar geschrieben werden
- Klassen werden nicht gespeichert: sie müssen verfügbar sein, wenn ein Objekt später wieder eingelesen wird
- statische Elemente werden nicht gesichert
  - Lösung: Serialisierung beeinflussen:
    - private void writeObject(java.io.ObjectOutputStream out)
    - private void readObject(java.io.ObjectInputStream in)

5 Versionskontrolle der Klassen

- serialisierte Objekte müssen mit der “richtigen” Klasse gelesen werden
- serialisierte Objekte enthalten dazu eine Klassenreferenz, welche den Namen und eine Versionsnummer der Klasse enthält
- Die Versionsnummer wird durch einen Hash-Wert repräsentiert, der über den Klassennamen, die Schnittstellen und die Namen der Instanzvariablen und Methoden gebildet wird.
- Problem: kleine Änderungen an der Klasse führen dazu, dass alte serialisierte Objekte unlesbar sind.
- Lösung:
  - eine Klasse kann ihre Versionsnummer festlegen:
    `static final long serialVersionUID = 1164397251093340429L;`
  - initiale Versionsnummer kann mittels **`serialver`** berechnet werden.
  - ⇒ Versionsnummer nur nach inkompatiblen Änderungen verändern

R.4 ClassLoader

- Wie werden Klassen in die Java Virtual Machine (JVM) geladen?
  - vom lokalen Dateisystem (**`CLASSPATH`**)
  - von einer Instanz eines **`ClassLoader`**
- ... und wann?
  - Wenn sie zum ersten Mal gebraucht werden!

```
class Test {
   public String toString() {
      Hello hello = new Hello();
   }
}
```

```
ClassLoader cl = new ...;
Class c = cl.loadClass("Test");
Object t=c.newInstance();
t.toString();
```

(1) Die Klasse **`Test`** wird vom Klassenlader **`cl`** geladen

(2) **`toString`** wird ausgeführt, dabei wird die Klasse **`Hello`** benötigt

(3) der Klassenlader von **`Test`** (= **`cl`**) wird beauftragt **`Hello`** zu laden

R.4 ClassLoader (2)

- Ein ClassLoader erzeugt einen Namensraum.
  - Es können Klassen mit demselben Namen innerhalb einer JVM existieren, wenn sichergestellt ist, dass sie von verschiedenen Classloader-Instanzen geladen werden.
- **`java.lang.ClassLoader`**
  - kann eine Klasse aus einem Array von Bytes erzeugen, welches dem Format einer Klassendatei entspricht (**`defineClass()`**)
  - wenn diese Klasse andere Klassen verwendet, wird der ClassLoader angewiesen diese Klassen zu laden (**`loadClass(String name)`**)
  - nach zuvor geladenen Klassen kann mittels
    `Class findLoadedClass(String name)`
    gesucht werden
- Mehrere **`ClassLoader`** können in einer JVM aktiv sein.

R.4 ClassLoader - Beispiel

```
import java.io.*;

public class SimpleClassLoader extends ClassLoader {
   public synchronized Class loadClass(String name, boolean
resolve) {

      Class c = findLoadedClass(name);
      if (c != null) return c;

      try {
          c = findSystemClass(name);
          if (c != null) return c;
      } catch(ClassNotFoundException e) {}

      try {
          RandomAccessFile file =
             new RandomAccessFile("test/" + name + ".class", "r");
          byte data[] = new byte[(int)file.length()];
          file.readFully(data);
          c = defineClass(name, data, 0, data.length);
      } catch(IOException e) {}

      if (resolve)
            resolveClass(c);
      return c;
} }
```


R.5 Java Sicherheit

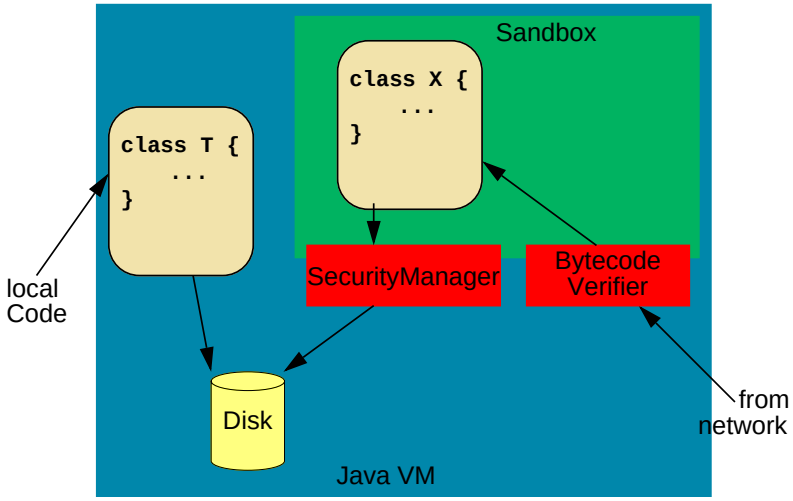
- Sandboxing
- Bytecode Verifier
- Security Manager
- Implementierung der Java-Bibliotheken

R.4 Beispiel Appletviewer

- **`appletviewer`** verfügt über einen speziellen ClassLoader
- Folgende Schritte laufen beim Start des **`appetviewer`** ab:
  1. Eine Zeichenkette wir in ein Objekt vom Typ **`URL`** umgewandelt.
  2. Der Zeichenstrom des **`URL`**-Stroms wir nach einem **`<applet>`**-Tag durchsucht.
  3. Ein **`AppletClassLoader`** wird mit Inhalt des Tags **`codebase`** erzeugt.
  4. Die mit dem Tag **`code`** festgelegte Klasse wird durch den **`AppletClassLoader`** geladen.
- Der **`AppletClassLoader`** wird mit einem Objekt vom Typ **`URL`** initialisiert:
  `AppletClassLoader(URL baseURL)`
- **`loadClass(String name)`** erzeugt eine URL aus **`baseURL`** und dem Namen der Klasse und ruft dann **`loadClass(URL url)`** auf
- **`loadClass(URL url)`** lädt die Klasse unter Verwendung von **`defineClass()`**

1 Sandboxing

- Klassen, die vom Klassenlader geladen wurden, werden innerhalb einer *Sandbox* ausgeführt



2 SecurityManager

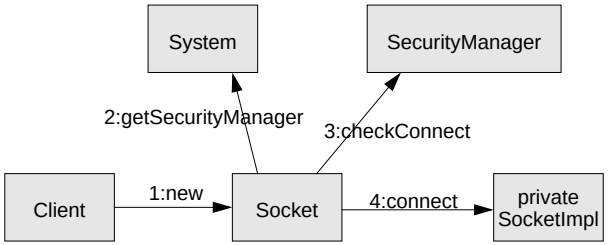
- Prüft, ob eine Klasse eine Operation durchführen darf.
- **`System.getSecurityManager()`** liefert den globalen SecurityManager zurück
- Überprüfungen müssen von der geschützten Klasse selbst durchgeführt werden

```
SecurityManager security = System.getSecurityManager();
if (security != null) {
    security.checkConnect(address.getHostAddress(), port);
}
```

- **`System.setSecurityManager()`** installiert einen globalen SecurityManager
  - ◆ diese Methode kann nur einmal aufgerufen werden!

3 SecurityManager und Sockets

- Beispiel Netzwerkkommunikation: Erzeugung eines neuen Sockets:



4 Beispiel SecurityManager

```
public class SimpleSecurityManager extends SecurityManager
{
    public void checkRead(String s) {
        if (...) {
            throw new SecurityException("checkread");
        }
    }
}
```

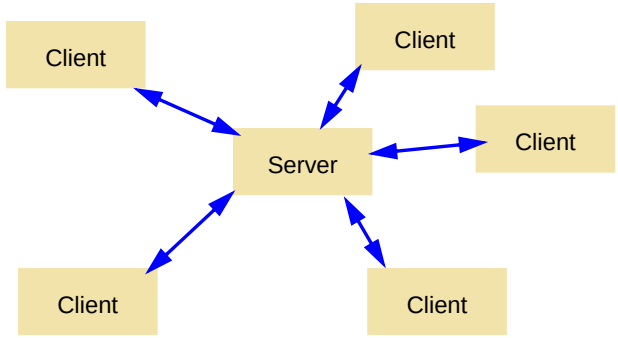

5 Was wird durch einen SecurityManager geschützt?

- Zugriffe auf das lokale Dateisystem
- Zugriffe auf das Betriebssystem
  - ◆ Ausführen von Programmen
  - ◆ Lesen von System Informationen
- Zugriffe auf das Netzwerk
- Thread Manipulationen
- Erzeugung von *factory* Objekten (Socket Implementierung)
- JVM: Linken von native code, Verlassen des Interpreters, Erzeugung eines Klassenladers
- Erzeugung von Fenstern
- ...

6 AppletSecurityManager

Security Checks	AppletSecurityManager
checkCreateClassLoader	nicht erlaubt
checkConnect	nur erlaubt, wenn die URL des Klassenladers der Klasse den Zielrechner enthält
checkExit	nicht erlaubt
checkExec	nicht erlaubt

S.2 Architektur für Aufgabe 4



S Hinweise zur 4. Aufgabe

S.1 Objektidentität in verteilten Programmen

- Wenn Objekte durch einen ObjectStream transferiert werden, verlieren sie ihre Identität
- Referenzen können nicht mehr zur Identitätsprüfung eingesetzt werden
- Lösung: eine Objekt ID
  - verändert sich nicht zwischen verschiedenen Rechnern und mehrmaligen Ausführungen des Programmes

S.2 Architektur für Aufgabe 4 (2)