



hochschule mannheim

# Fakultät für Informatik **Programmierung 2 (PR2)**

**05 - Grundlagen der Nebenläufigkeit**

Prof. Dr. Frank Dopatka



Hochschule Mannheim University of Applied Sciences



hochschule mannheim



## Threads



hochschule mannheim

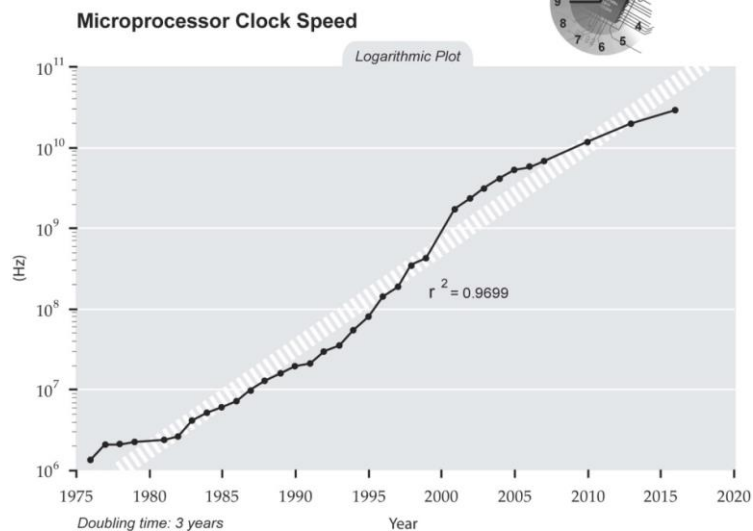
## Wozu braucht man Nebenläufigkeit?

- Ein Webserver soll mehrere Verbindungen gleichzeitig bearbeiten können.
- Ein User-Interface soll nicht blockieren, obwohl eine Berechnung im Hintergrund läuft.
- Eine Rechtschreibprüfung soll neben dem normalen Editieren des Dokuments ablaufen.
- Eine App soll eine Animation anzeigen und gleichzeitig Daten aus dem Netzwerk nachladen.
- Ein mathematisches Problem soll auf mehrere CPUs verteilt werden.



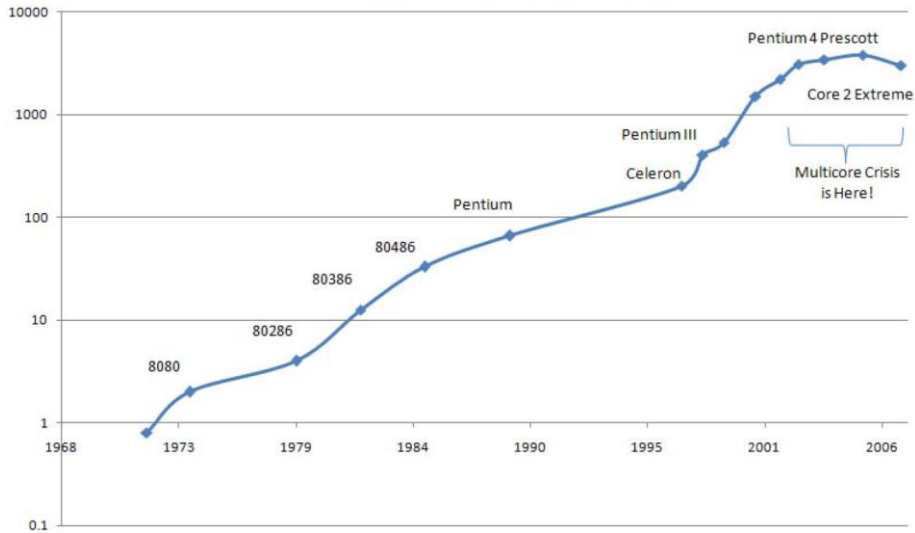
hochschule mannheim

## Parallelisierung statt Takt





Intel Processor Clock Speed (MHz)



hochschule mannheim  
Zur Motivation:  
Eine blockierende GUI



```
public class GUI implements ActionListener{
    private JFrame jf=null;
    private JButton b1=new JButton("Start");
    private JButton b2=new JButton("Abbruch");
    private JTextArea t=new JTextArea(50,50);
    private Berechnung b=null;

    public GUI(String titel){
        jf=new JFrame(titel);
        b1.addActionListener(this);
        b2.addActionListener(this);
        JPanel p1=new JPanel();
        p1.setLayout(new GridLayout(1,3));
        p1.add(b1);
        p1.add(b2);
        p1.add(t);
        jf.getContentPane().add(p1);
        jf.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        jf.setSize(400,200);
        jf.setVisible(true);
    }
}
```



## hochschule mannheim Zur Motivation: Eine blockierende GUI

```
public void addText(String text){
    t.setText(t.getText()+text+"\n");
}

@Override
public void actionPerformed(ActionEvent e) {
    Object o=e.getSource();
    if (o==b1){
        b=new Berechnung();
        b.start(this);
    }
    if (o==b2){
        if (b!=null) b.abbrechen();
    }
}

public static void main(String[] args){
    new GUI("Blockierende GUI");
}
}
```



## hochschule mannheim Zur Motivation: Eine blockierende GUI

```
public class Berechnung {
    private boolean setAbbruch=false;
    private boolean istFertig=false;
    private GUI g=null;

    public void start(GUI g){
        this.g=g; this.run();
    }

    public void run(){
        g.addText("Berechnung startet...");
        for (int i=0;i<50000000;i++){
            for (int j=0;j<100;j++){ // viel rechnen
                if (setAbbruch){
                    g.addText("ABBRUCH!");
                    return;
                }
            }
        }
        g.addText("Berechnung fertig!");
        istFertig=true;
    }

    public void abbrechen(){
        this.setAbbruch=true;
    }

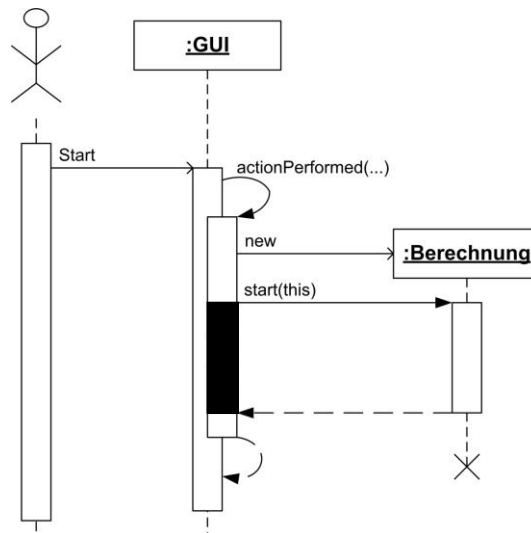
    public boolean istFertig(){
        return istFertig;
    }
}
```



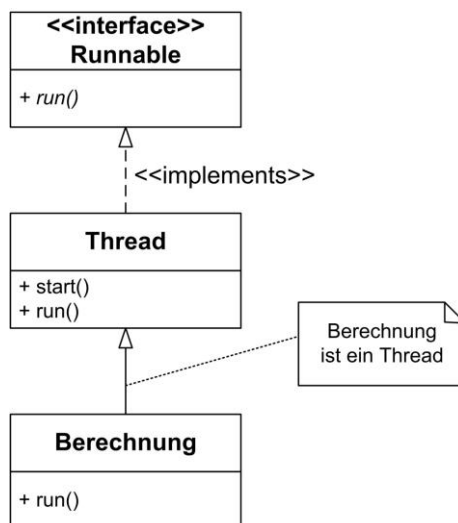
## hochschule mannheim Zur Motivation: Eine blockierende GUI

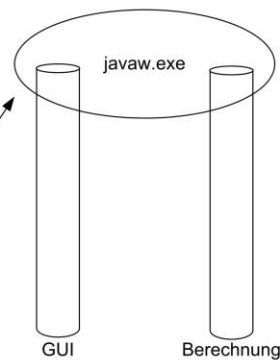
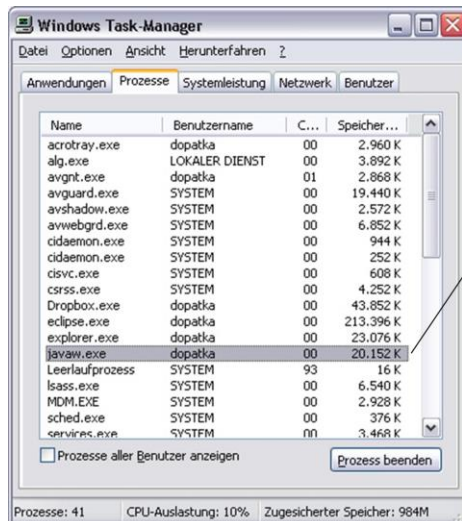
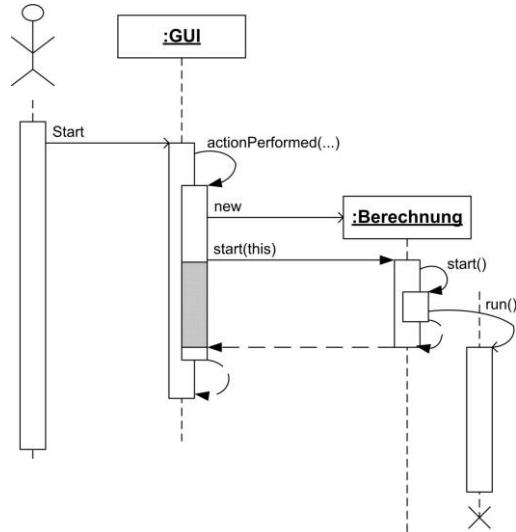


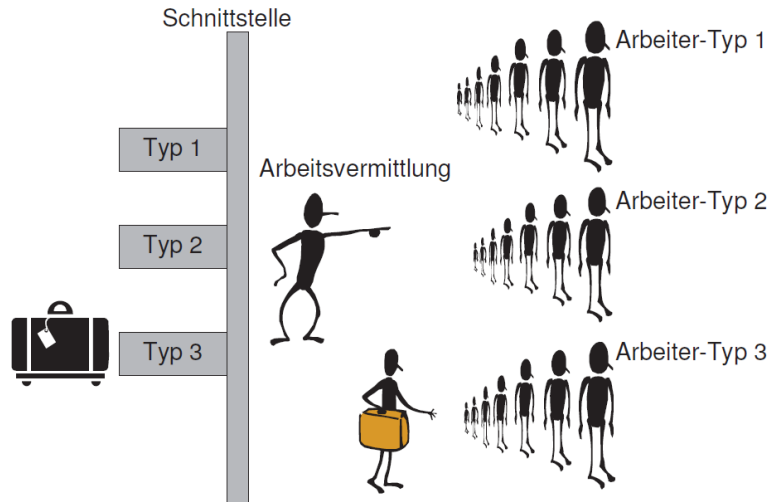
## hochschule mannheim Die Berechnung als Sequenzdiagramm der UML



- Eine Fehlermeldung (xException) ist ein Objekt von einer Klasse von Fehlermeldungen...
  - Ein Ereignis (xEvent) ist ein Objekt von einer Klasse von Ereignissen...
  - Eine GUI ist ein Objekt der Klasse JFrame...
  - Und ein Thread ist ein Objekt der Klasse Thread!
- 
- Um das Problem zu lösen, muss die Berechnung vom main-Thread mit der GUI unabhängig werden!
- 
- Die Berechnung muss ein eigener Thread sein!

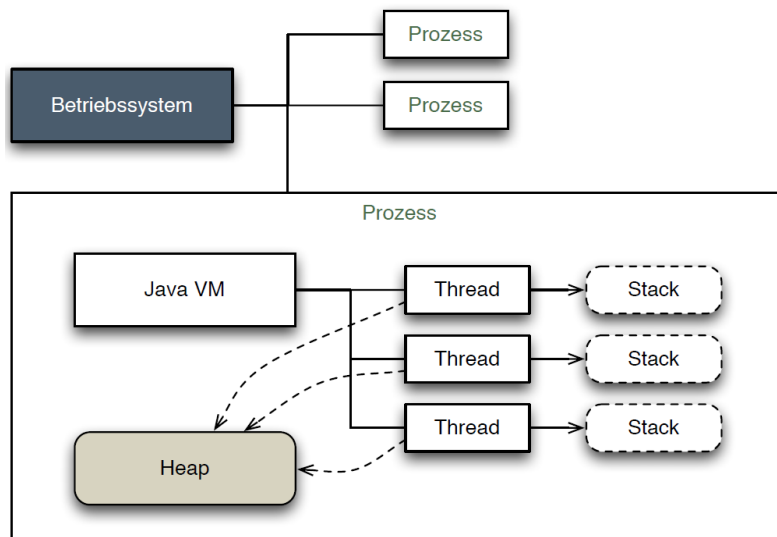
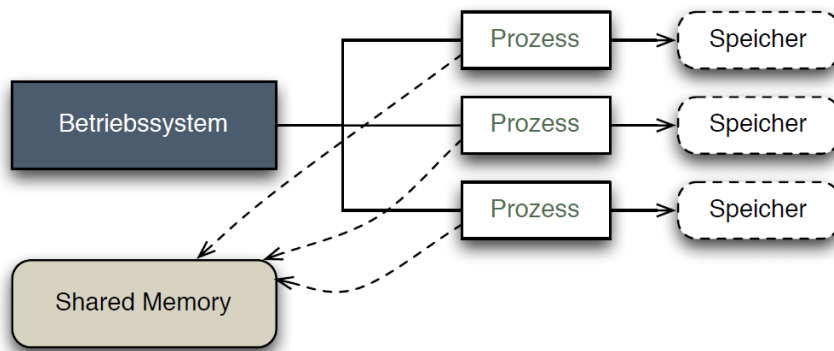






- Prozesse...
  - haben voneinander getrennte Adressräume, also einen getrennten Heap.
  - kommunizieren über Inter Process Communication (IPC).
  - können einen oder mehrere Threads enthalten.
  - sind schwergewichtig.
  - haben eigene Ressourcen wie geöffnete Dateien, Sockets, Speicher etc.
- Threads...
  - haben einen gemeinsamer Adressraum, also einen gemeinsamen Heap.
  - haben getrennte Stacks.
  - kommunizieren über den gemeinsamen Speicher.
  - sind leichtgewichtiger als Prozesse.

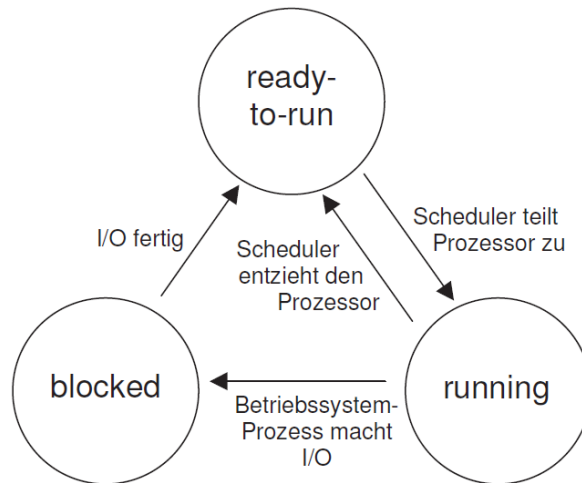






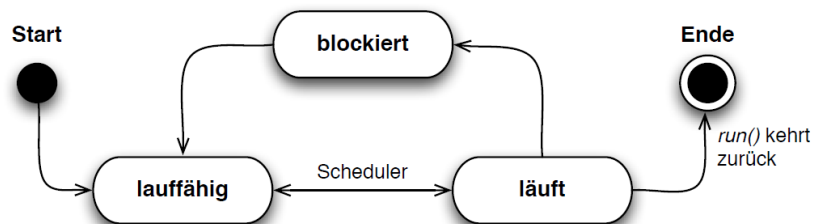
hochschule mannheim

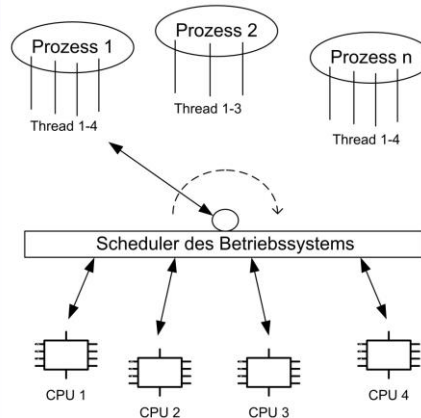
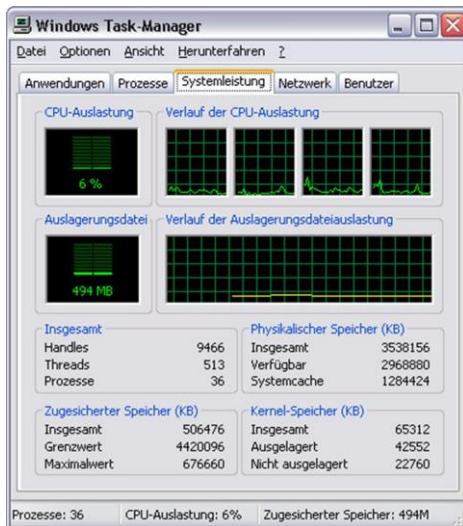
## Multi-Threading & Multi-Processing: Zustände eines Prozesses



hochschule mannheim

## Multi-Threading & Multi-Processing: Zustände eines Threads





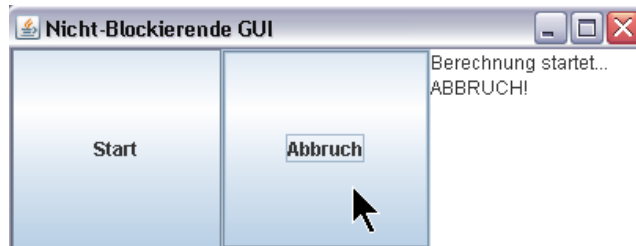
```
public class Berechnung extends Thread{
    private boolean setAbbruch=false;
    private boolean istFertig=false;
    private GUI g=null;

    public void start(GUI g){
        this.g=g; super.start();
    }

    public void run(){
        g.addText("Berechnung startet...");
        for (int i=0;i<500000000;i++){
            for (int j=0;j<100;j++){ // viel rechnen
                if (setAbbruch){
                    g.addText("ABBRUCH!");
                    return;
                }
            }
        }
        g.addText("Berechnung fertig!");
        istFertig=true;
    }

    public void abbrechen(){
        this.setAbbruch=true;
    }

    public boolean istFertig(){
        return istFertig;
    }
}
```



- Ein Thread endet normalerweise, wenn die `run()`-Methode zurückkehrt.
- Man kann den Wunsch, dass der Thread sich beenden soll durch das Setzen einer Variable von außen signalisieren.
- Eine weitere Möglichkeit besteht darin, dem Thread mit `interrupt()` zu beenden.
- Dies hat den Vorteil, dass hier der Thread auch beendet werden kann, wenn er in bestimmten Methoden blockiert ist.



- Die Vererbung von der Klasse **Thread** kann durch die direkte Implementierung des Interfaces **Runnable** umgangen werden.
- Alle Threads können gemeinsam auf denselben Speicher ihrer Prozessumgebung zugreifen.
- Weil mehrere Threads gleichzeitig aus einem Speicher lesen und/oder schreiben können, besteht die Gefahr von Race-Conditions!
- Die Thread-Umschaltung erfolgt automatisch über das Betriebssystem.
  - Sie können aber über die Methode **setPriority(int)** der Klasse **Thread** eine Priorität für einen Thread setzen.
  - Wie der Wert vom Betriebssystem nun interpretiert wird, bleibt dem Betriebssystem überlassen.



- Häufig möchte man, dass ein Thread erst weiterläuft, wenn andere Threads zu Ende gelaufen sind.
- Als Anwendungsfall möchte man gern die Ergebnisse des anderen Threads abwarten, um dann mit ihnen weiterarbeiten zu können.
- Mit der Methode **join()** kann man darauf warten, dass ein Thread fertig ist:
  - **void join(long millis)**  
Warte maximal millis Millisekunden (0 = ewig).
  - **void join(long millis, int nanos)**  
Wartet maximal millis Milli- und nanos Nanosekunden.
  - **void join()**  
Wartet ewig.
- Alle **join()**-Methoden sind unterbrechbar; sie kehren im Unterbrechungsfall mit einer **InterruptedException** zurück.



## Threads vereinigen: Ein Beispiel

```
Thread t1 = new Thread(new RunnerPrinter("Runner 1"));  
Thread t2 = new Thread(new RunnerPrinter("Runner 2"));  
Thread t3 = new Thread(new RunnerPrinter("Runner 3"));
```

```
t1.start(); t2.start(); t3.start();  
t1.join(); t2.join(); t3.join();
```

```
System.out.println("Alle fertig");
```

```
Runner 1  
Runner 3  
Runner 1  
Runner 3  
Runner 2  
Runner 1 ist fertig.  
Runner 3 ist fertig.  
Runner 2  
Runner 2  
Runner 2 ist fertig.  
Alle fertig
```



## Noch ein paar Dämonen...

- Man kann festlegen, was beim Beenden des main-Threads mit den davon abgespaltenen Threads passieren soll
  - `void setDaemon(boolean on)`
  - `boolean isDaemon()`
- Die VM beendet sich dann, wenn alle Nicht-Daemon-Threads beendet wurden.
- Welchen Sinn kann es haben, Daemon-Threads zu erstellen?





- 5 parallel ablaufende Threads sollen eine einzige Variable in Einer-Schritten jeweils 100x erhöhen.
- Dazu lesen Sie die Variable aus, müssen dann etwas berechnen und schreiben dann die erhöhte Variable wieder zurück.
- Das Objekt mit der Variablen ist das gemeinsame Betriebsmittel der 5 Threads...

```
public class Betriebsmittel {  
    private int x=0;  
  
    public void setX(int x) {  
        this.x = x;  
    }  
    public int getX() {  
        return x;  
    }  
}
```



```
public class Nutzer extends Thread {  
    private Betriebsmittel b;  
  
    public Nutzer(Betriebsmittel b){  
        this.b=b;  
    }  
  
    private void rechnen(){  
        for (int i=0;i<100000;i++){  
            for (int j=0;j<50;j++){  
            }  
        }  
    }  
  
    @Override  
    public void run(){  
        for (int i=1;i<=100;i++){  
            int wert=b.getX();  
            rechnen();  
            b.setX(wert+1);  
        }  
        System.out.println("Stand bei Ende:"+b.getX());  
    }  
}
```



```
public class TestNutzer {  
    public static void main(String[] args) {  
        Betriebsmittel b=new Betriebsmittel();  
        for(int i=0;i<5;i++){  
            Nutzer n=new Nutzer(b);  
            n.start();  
        }  
    }  
}
```

<terminated> TestNutzer [Java Application]

```
Stand bei Ende:101  
Stand bei Ende:112  
Stand bei Ende:120  
Stand bei Ende:112  
Stand bei Ende:123
```



```
@Override  
public void run(){  
    for (int i=1;i<=100;i++){  
        int wert=b.getX();  
        rechnen();  
        b.setX(wert+1);  
    }  
    System.out.println("Stand bei Ende:"+b.getX());  
}
```





```
@Override
public void run(){
    synchronized(b){
        for (int i=1;i<=100;i++){
            int wert=b.getX();
            rechnen();
            b.setX(wert+1);
        }
        System.out.println("Stand bei Ende:"+b.getX());
    }
}
```

<terminated> TestNutzer (1) [Java Application]

```
Stand bei Ende:100
Stand bei Ende:200
Stand bei Ende:300
Stand bei Ende:400
Stand bei Ende:500
```

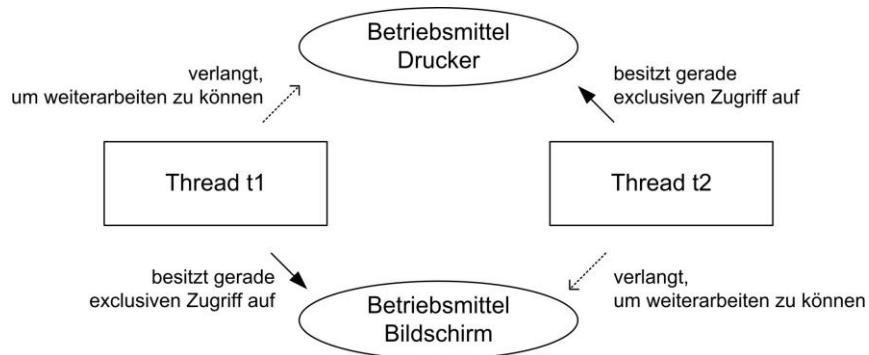


- Der **synchronized**-Block verhindert den Zugriff eines anderen Threads auf das Betriebsmittel b, wenn ein Thread bereits auf b zugreift und sich damit also im **synchronized**-Block befindet.
- Der andere Thread muss solange warten, bis der momentan zugreifende Thread das Betriebsmittel wieder freigibt, also den **synchronized**-Block verlässt.
- Zeitabhängige Fehler (Race Conditions) sind äußerst schwer zu ermitteln!
  - Sie können unter anderem plötzlich auftreten, wenn die Anwendung auf einem anderen Rechner oder einem anderen Betriebssystem ausgeführt wird.



```
@Override
public void run(){
    synchronized(b){
        for (int i=1;i<=100;i++){
            int wert=b.getX();
            rechnen();
            b.setX(wert+1);
        }
        System.out.println("Stand bei Ende:"+b.getX());
    }
}
```

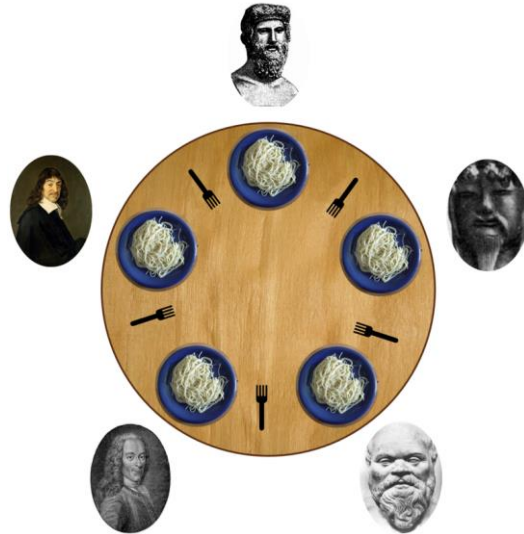
```
<terminated> TestNutzer (1) [Java Application]
Stand bei Ende:100
Stand bei Ende:200
Stand bei Ende:300
Stand bei Ende:400
Stand bei Ende:500
```





hochschule mannheim

## Das Problem der Dead-Locks: Philosophenproblem



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

37



hochschule mannheim

## Das Problem der Dead-Locks



Hochschule Mannheim University of Applied Sciences | Prof. Dr. Frank Dopatka

38



- Ein Deadlock (deutsch: Verklemmung) ist ein Zustand, bei dem mehrere Threads auf Betriebsmittel warten, die einem anderen beteiligten Thread zugeteilt sind.
- In einem solchen Fall blockiert die gesamte Anwendung und ist nicht mehr bedienbar; es hilft nur ein Neustart.
- Meist ist die Wahrscheinlichkeit, dass es zu einer solchen Situation kommt, relativ gering:
  - Dies verschlimmert jedoch die Situation, da die Anwendung „normalerweise“ ordentlich funktioniert und sich nur „plötzlich hin und wieder“ aufhängt.
- Wie soll man sowas debuggen?



1. Es existiert nur ein Thread in einem geschlossenen System.
2. Jeder Thread verwendet zu jedem Zeitpunkt maximal ein einziges Betriebsmittel.



1. Das Erwerben von Betriebsmittel und das anschließende Blockieren (englisch: hold and wait) eines Threads kann verhindert werden!
  - Dazu muss man im Vorfeld wissen, welche Betriebsmittel der Thread im schlimmsten Fall gleichzeitig beanspruchen kann.
  - Erst wenn die Betriebsmittel alle frei sind, werden Sie diesem Thread auf einen Schlag zugeteilt und er kann seine Aufgabe unterbrechungsfrei ausführen.
2. Alle Betriebsmittel können durchnummeriert und nur in aufsteigender Reihenfolge vergeben werden!
  - Dadurch kann der Effekt des gegenseitigen, kreisförmigen Wartens verhindert werden.



# Sockets



- Ein Socket ist ein vom Betriebssystem bereitgestelltes Objekt, das als Kommunikationsendpunkt dient.
- Eine Anwendung verwendet Sockets, um Daten mit anderen Anwendungen oder anderen Teilen einer verteilten Anwendung auszutauschen.
- Der Kommunikationspartner kann sich dabei
  - auf demselben Computer via localhost, IP 127.0.0.1 als Interprozesskommunikation oder
  - auf einem anderen, via Netzwerk erreichbaren Computer befinden.
- Mit Sockets kann daher eine modulare Implementierung einer Anwendung erstellt werden, die zunächst lokal ausgeführt wird und die bei höherer Nutzung später problemlos verteilt und damit skaliert werden kann.



- Die Kommunikation über Sockets erfolgt in der Regel bidirektional; über das Socket können Daten sowohl empfangen als auch gesendet werden.
- Ein Socket besteht immer aus einer IP-Adresse zur Identifikation des Computers sowie einem Port zur Identifikation der Anwendung auf dem Computer.
- Eine Kommunikation erfolgt stets zwischen einem Paar von Sockets:
  - „212.23.52.64:4246 kommuniziert mit 153.52.56.76:80“
- Da die kommunizierenden Anwendungen in verschiedenen Prozessen ablaufen, sind diese nebenläufig.

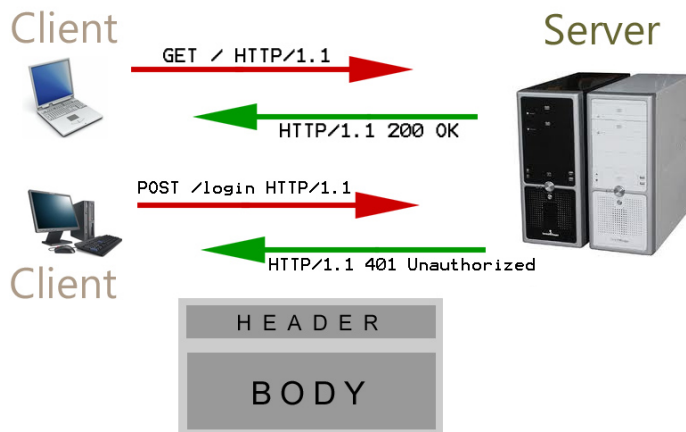
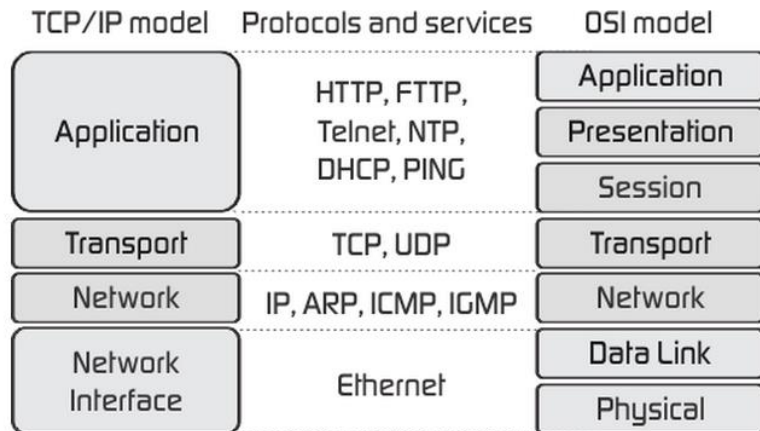


- Generell kann man unterscheiden zwischen
  - Stream Sockets als Zeichen-Datenstrom per TCP sowie
  - Datagram Sockets als separate Nachrichten per UDP.
- TCP ist verlässlicher, da die Reihenfolge und Zustellung von Paketen garantiert wird.
- TCP baut zu Beginn über einen sogenannten 3-Wege-Handshake eine virtuelle Verbindung zwischen dem Client und dem Server auf, die dann mehrfach genutzt werden kann. Bei UDP geschieht dies nicht.
- UDP ist für bestimmte Aufgaben effizienter und flexibler, oft auch zeitlich schneller aufgrund des geringeren Overheads; die Reihenfolge und die Zustellung der Pakete werden jedoch nicht garantiert. Außerdem können Pakete doppelt empfangen werden.



- Das Client- / Server-Modell beschreibt eine Möglichkeit, Aufgaben und Dienstleistungen innerhalb eines Netzwerkes zu verteilen.
- Die Aufgaben werden von Anwendungen erledigt, die in Clients und Server unterteilt werden.
- Der Client fordert einen Dienst vom Server an.
- Der Server beantwortet die Anforderung und erfüllt damit die Anforderung.
- Üblicherweise arbeitet ein Server gleichzeitig für mehrere Clients.









- Server-seitig:
  1. Server-Socket erstellen.
  2. Binden des Sockets an einen Port, über den ankommende Anfragen akzeptiert werden.
  3. Auf Anfragen warten.
  4. Ankommende Anfrage akzeptieren und damit ein neues Socket-Paar für diesen Client erstellen.
  5. Bearbeiten der Client-Anfrage auf dem neu erstellten Client-Socket.
  6. Client-Socket wieder schließen.



```
public class ServerEcho {  
    public static void main(String[] args) {  
        ServerSocket server = null;  
        Socket s = null;  
        BufferedReader in = null;  
        try {  
            System.out.println("Server starting...");  
            server = new ServerSocket(1234);  
            s = server.accept();  
            in = new BufferedReader(new InputStreamReader(s.getInputStream()));  
            String text = in.readLine();  
            System.out.println("incoming message: "+text);  
            System.out.println("Server terminated.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



## 1. Der einfachste TCP Socket Server: Auf jeden Fall wieder schließen

```
finally{  
    try {  
        in.close();  
    } catch (Exception e) {}  
    try {  
        server.close();  
    } catch (IOException e) {}  
}  
}  
}
```



## Ablauf der Socket-Kommunikation bei TCP Stream Sockets

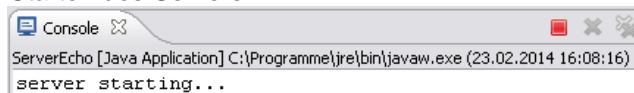
- Client-seitig:
  1. Socket erstellen.
  2. Den erstellten Socket mit der Server-Adresse verbinden, von welcher Daten angefordert werden sollen.
  3. Senden und Empfangen von Daten.
  4. Evtl. Socket herunterfahren.
  5. Verbindung trennen und Socket schließen.



```
public class ClientEcho {
    public static void main(String[] args) {
        Socket s = null;
        BufferedWriter out = null;
        try{
            System.out.println("client starting...");
            String x = "Hello?";
            s = new Socket("localhost",1234);
            out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));
            System.out.println("Client sends '"+x+"'...");
            out.write(x);
            out.newLine();
            out.flush();
            System.out.println("client terminated.");
        } catch (IOException e) {
            e.printStackTrace();
        }
        finally{
            try {
                out.close();
            } catch (Exception e) {}
        }
    }
}
```

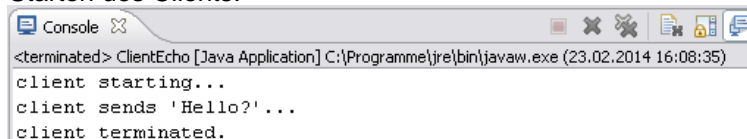


## 1. Starten des Servers:



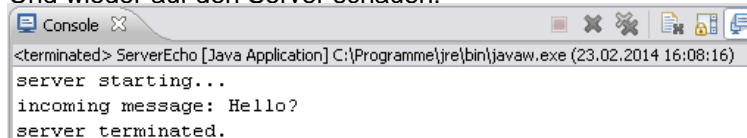
```
Console [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:08:16)
server starting...
```

## 2. Starten des Clients:

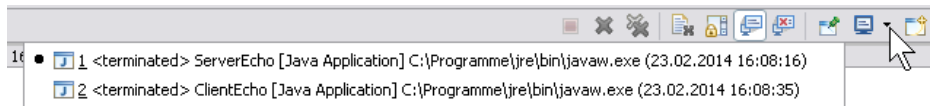


```
Console [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:08:35)
<terminated> ClientEcho [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:08:16)
client starting...
client sends 'Hello?'...
client terminated.
```

## 3. Und wieder auf den Server schauen:



```
Console [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:08:35)
<terminated> ServerEcho [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:08:16)
server starting...
incoming message: Hello?
server terminated.
```



Es wäre schön, wenn der Server antworten könnte...

```
public class ServerEcho {  
    public static void main(String[] args){  
        ServerSocket server = null;  
        Socket s = null;  
        BufferedReader in = null;  
        BufferedWriter out = null;  
        try{  
            System.out.println("server starting...");  
            server = new ServerSocket(1234);  
            s = server.accept();  
            in = new BufferedReader(new InputStreamReader(s.getInputStream()));  
            String text = in.readLine();  
            System.out.println("incoming message: "+text);  
            out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));  
            String text2 = "SERVER: "+text;  
            System.out.println("outgoing message: "+text2);  
            out.write(text2);  
            out.newLine();  
            out.flush();  
            System.out.println("server terminated.");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



... und der Client die Antwort lesen könnte.

```
public class ClientEcho {
    public static void main(String[] args) {
        Socket s = null;
        BufferedReader in = null;
        BufferedWriter out = null;
        try{
            System.out.println("client starting...");
            String x = "Hello?";
            s = new Socket("localhost", 1234);
            out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));
            System.out.println("client sends '"+x+"'...");
            out.write(x);
            out.newLine();
            out.flush();
            in = new BufferedReader(new InputStreamReader(s.getInputStream()));
            String text = in.readLine();
            System.out.println(text);
            System.out.println("client terminated.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```



... und der Client die Antwort lesen könnte.

```
finally{
    try {
        in.close();
    } catch (Exception e) {}
    try {
        out.close();
    } catch (Exception e) {}
}
}
```



## 2. TCP Socket Client: Ablauf der Kommunikation

```
Console X
<terminated> ServerEcho [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:14:48)
server starting...
incoming message: Hello?
outgoing message: SERVER: Hello?
server terminated.
```

```
Console X
<terminated> ClientEcho [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:14:50)
client starting...
client sends 'Hello?'...
SERVER: Hello?
client terminated.
```



## 3. TCP Socket Server endlos

Der Server terminiert bislang nach dem ersten Request:  
Es wäre gut, wenn der Server endlos Anfragen abwickeln könnte...

```
public class ServerEndless {
    private ServerSocket server = null;
    private Socket s = null;
    private BufferedReader in = null;
    private BufferedWriter out = null;
```



Der Server terminiert bislang nach dem ersten Request:

Es wäre gut, wenn der Server endlos Anfragen abwickeln könnte...

```
private ServerEndless() {  
    try {  
        while (true) {  
            System.out.println("server starting...");  
            server = new ServerSocket(1234);  
            s = server.accept();  
            in = new BufferedReader(new InputStreamReader(s.getInputStream()));  
            String text = in.readLine();  
            System.out.println("incoming message: "+text);  
            out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));  
            String text2 = "SERVER: "+text;  
            System.out.println("outgoing message: "+text2);  
            out.write(text2);  
            out.newLine();  
            out.flush();  
            terminate();  
        }  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    finally {  
        terminate();  
    }  
}
```



Der Server terminiert bislang nach dem ersten Request:

Es wäre gut, wenn der Server endlos Anfragen abwickeln könnte...

```
private void terminate() {  
    System.out.println("server terminated.");  
    try {  
        in.close();  
    } catch (Exception e) {}  
    try {  
        out.close();  
    } catch (Exception e) {}  
    try {  
        server.close();  
    } catch (IOException e) {}  
}  
  
public static void main(String[] args) {  
    new ServerEndless();  
}
```



hochschule mannheim

### 3. TCP Socket Server endlos: Starten von Server und Client(s)

```
Console [X]
ServerEndless [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:44:29)
server starting...
incoming message: Hello?
outgoing message: SERVER: Hello?
server terminated.
server starting...
incoming message: Hello?
outgoing message: SERVER: Hello?
server terminated.
server starting...
incoming message: Hello?
outgoing message: SERVER: Hello?
server terminated.
server starting...
incoming message: Hello?
outgoing message: SERVER: Hello?
server terminated.
server starting...
incoming message: Hello?
outgoing message: SERVER: Hello?
server terminated.
server starting...
```

```
Console [X]
<terminated> ClientEcho [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 17:04:32)
client starting...
client sends 'Hello?'...
SERVER: Hello?
client terminated.
```



hochschule mannheim

### 3. TCP Socket Server endlos: Manuelles Terminieren des Servers

```
Console [X]
ServerEndless [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 16:44:29)
server starting...
```





Der Server kann bislang nur einen Request zu einem Zeitpunkt abwickeln:  
Es wäre gut, wenn er gleichzeitig mehr könnte...

```
public class ServerEndlessThreading {  
    private ServerSocket server = null;  
  
    private ServerEndlessThreading() {  
        try {  
            System.out.println("server starting...");  
            server = new ServerSocket(1234);  
            while (true) {  
                ServerThread th=new ServerThread(server.accept());  
                th.start();  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        finally {  
            System.out.println("server terminated.");  
            try {  
                server.close();  
            } catch (IOException e) {}  
        }  
    }  
}
```



Der Server kann bislang nur einen Request zu einem Zeitpunkt abwickeln:  
Es wäre gut, wenn er gleichzeitig mehr könnte...

```
public static void main(String[] args) {  
    new ServerEndlessThreading();  
}
```



Der Server kann bislang nur einen Request zu einem Zeitpunkt abwickeln:  
Es wäre gut, wenn er gleichzeitig mehr könnte...

```
public class ServerThread extends Thread{  
    private Socket s = null;  
    private BufferedReader in = null;  
    private BufferedWriter out = null;  
  
    public ServerThread(Socket s) {  
        System.out.println("connection established.");  
        this.s=s;  
    }  
}
```



Der Server kann bislang nur einen Request zu einem Zeitpunkt abwickeln:  
Es wäre gut, wenn er gleichzeitig mehr könnte...

```
@Override  
public void run() {  
    try{  
        System.out.println("serving incoming request...");  
        in = new BufferedReader(new InputStreamReader(s.getInputStream()));  
        String text = in.readLine();  
        System.out.println("incoming message: "+text);  
        out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));  
        String text2 = "SERVER: "+text;  
        System.out.println("outgoing message: "+text2);  
        out.write(text2);  
        out.newLine();  
        out.flush();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
    finally{  
        terminate();  
    }  
}
```



Der Server kann bislang nur einen Request zu einem Zeitpunkt abwickeln:  
Es wäre gut, wenn er gleichzeitig mehr könnte...

```
private void terminate(){  
    System.out.println("connection terminated.");  
    try {  
        in.close();  
    } catch (Exception e) {}  
    try {  
        out.close();  
    } catch (Exception e) {}  
}  
}
```



Der Server kann bislang nur einen Request zu einem Zeitpunkt abwickeln:  
Es wäre gut, wenn er gleichzeitig mehr könnte...

```
Console  
<terminated> ServerEndlessThreading [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 17:23:40)  
server starting...  
connection established.  
serving incoming request...  
incoming message: Hello?  
outgoing message: SERVER: Hello?  
connection terminated.  
connection established.  
serving incoming request...  
incoming message: Hello?  
outgoing message: SERVER: Hello?  
connection terminated.
```

```
Console  
<terminated> ClientEcho [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 17:23:48)  
client starting...  
client sends 'Hello?'...  
SERVER: Hello?  
client terminated.
```



- Warum kann dieser Server mehrere Anfragen über einen einzelnen TCP Port beantworten?
- Gibt es da keine Datenkollisionen?
- Ein Socket ist ja eine Verknüpfung zwischen IP Adresse und Portnummer.
- Kommunizierende lokale und entfernte Sockets werden als Socketpaare bezeichnet.
- Jedes Socketpaar wird durch ein einzigartiges 4-Tupel beschrieben, bestehend aus:
  - IP-Adresse : Portnummer des Rechners 1
  - IP-Adresse : Portnummer des Rechners 2
- Somit sind die Kommunikationskanäle zwischen den beiden Anwendungen auf dem Server und Client eindeutig definiert.



- Client-seitig:
  1. Socket erstellen.
  2. An die Ziel-Adresse senden.
- Server-seitig:
  1. Socket erstellen.
  2. Socket binden.
  3. Auf ankommende Pakete warten.



```
public class ServerEcho {
    public static void main(String[] args){
        DatagramSocket server = null;
        try{
            System.out.println("server starting...");
            DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
            server = new DatagramSocket(1234);
            server.receive(packet);
            System.out.println("incoming packet: "+new String(packet.getData()));
        } catch (IOException e) {
            e.printStackTrace();
        }
        finally{
            System.out.println("server terminated.");
            try {
                server.close();
            } catch (Exception e) {}
        }
    }
}
```



```
public class ClientEcho {
    public static void main(String[] args) {
        DatagramSocket s = null;
        try{
            System.out.println("client starting...");
            String x = "Hello?";
            DatagramPacket packet = new DatagramPacket(x.getBytes(), x.length(),
                InetAddress.getByName("localhost"), 1234);
            s = new DatagramSocket();
            System.out.println("client sends '"+x+"'");
            s.send(packet);
            System.out.println("client terminated.");
        } catch (IOException e) {
            e.printStackTrace();
        }
        finally{
            try {
                s.close();
            } catch (Exception e) {}
        }
    }
}
```



```
Console 
<terminated> ServerEcho (1) [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 18:30:18)
server starting...
incoming packet: Hello?
server terminated.
```

```
Console 
<terminated> ClientEcho (1) [Java Application] C:\Programme\jre\bin\javaw.exe (23.02.2014 18:30:21)
client starting...
client sends 'Hello?'
client terminated.
```



- Basierend auf einem TCP Socket Server können Sie ihren eigenen Anwendungsserver für die OSI Schicht 5-7 programmieren.
- Sie müssen eingehende Anfragen akzeptieren, das Protokoll interpretieren und im korrekten Protokollformat antworten.
- In diesem Beispiel nehmen wir unseren 3. TCP Socket Server als Basis für den eigenen HTTP Server...



Wir beginnen mit einem simplen Socket Server, der an Port 80 lauscht...

```
public class HTTPserver {
    private ServerSocket server = null;
    private Socket s = null;
    private BufferedReader in = null;
    private BufferedWriter out = null;

    private HTTPserver() throws IOException{
        try{
            System.out.println("HTTP server starting...");
            server = new ServerSocket(80);

            while (true){
                s = server.accept();
                System.out.println("New incoming request...");
                in = new BufferedReader(new InputStreamReader(s.getInputStream()));
```



Jetzt lesen wir die HTTP Anfrage (request) aus und bereiten die Antwort (response) vor...

```
String request = "";
int buffersize=2000;
char[] buffer = new char[buffersize];
int amount=in.read(buffer,0,buffersize);
request=new String(buffer);
request=request.substring(0, amount);

out = new BufferedWriter(new OutputStreamWriter(s.getOutputStream()));
String content="";
out.write("HTTP/1.1 200 OK\n");
out.write("Server: Franks Server\n");
if (request.contains("GET / ")){
    System.out.println("HTTP GET request on main site.");
    content+="<!DOCTYPE html>";
    content+="<html>";
    content+="<head></head>";
    content+="<body>";
    content+="<h1>Hello everybody!</h1>";
    content+="</body>";
    content+="</html>";
}
```



Und einige weitere Antworten...

```
else if (request.contains("GET /frank.html ")){
    System.out.println("HTTP GET request on frank.html.");
    content+="<!DOCTYPE html>";
    content+="<html>";
    content+="<head></head>";
    content+="<body>";
    content+="<h1>Hello Frank!</h1>";
    content+="</body>";
    content+="</html>";
}
else{
    content+="<!DOCTYPE html>";
    content+="<html>";
    content+="<head></head>";
    content+="<body>";
    content+="<h1>What?</h1>";
    content+="</body>";
    content+="</html>";
}
```



Absenden einer gültigen HTTP Antwort...

```
out.write("Content-Length: "+content.getBytes().length+"\n");
out.write("Content-Language: de\n");
out.write("Connection: close\n");
out.write("Content-Type: text/html\n");
out.write("\n");
out.write(content);

out.flush();
in.close();
out.close();
System.out.println("Incoming request solved.");
}
}
finally{
    terminate();
}
}
```





Der Rest ist identisch zum TCP Socket Server...

```
private void terminate() {
    System.out.println("server terminated.");
    try {
        in.close();
    } catch (Exception e) {}
    try {
        out.close();
    } catch (Exception e) {}
    try {
        server.close();
    } catch (IOException e) {}
}

public static void main(String[] args) throws IOException {
    new HTTPserver();
}
```



Es ist an der Zeit, den Server zu testen...

```
Console
HTTPserver (1) [Java Application] C:\Programme\jre\bin\javaw.exe (16.03.2014 14:48:10)
HTTP server starting...
```

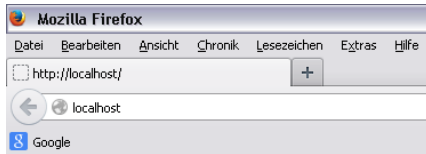
```
Console
HTTPserver (1) [Java Application] C:\Programme\jre\bin\javaw.exe (16.03.2014 14:48:10)
HTTP server starting...
New incoming request...
HTTP GET request on main site.
Incoming request solved.
```



hochschule mannheim

## Der erste eigene HTTP Server

Der eingehende Request kommt von einem beliebigen HTTP Browser...



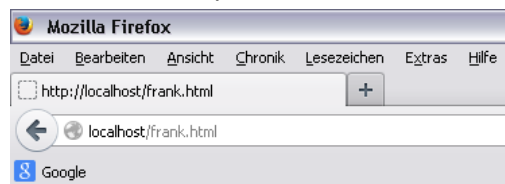
# Hello everybody!



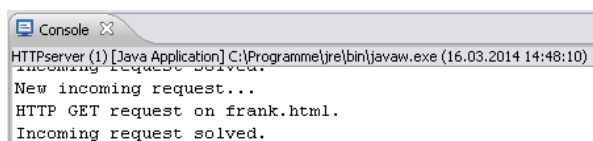
hochschule mannheim

## Der erste eigene HTTP Server

Und noch ein Request...

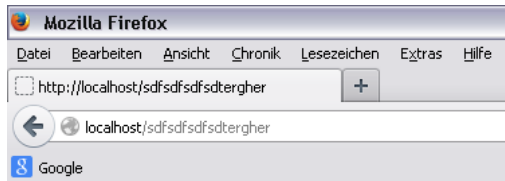


# Hello Frank!





Und noch ein Request...



## What?



- Programmieren Sie einen TCP Client, der eine Liste von Zahlen an den Server sendet.
- Programmieren Sie einen TCP Server, welcher auf der Basis der Anfrage einen der folgenden Dienste bietet:
  - addiere alle Zahlen,
  - multipliziere alle Zahlen,
  - suche das Minimum,
  - suche das Maximum oder
  - berechne den arithmetischen Mittelwert.
- Das Ergebnis soll dann wieder an den Socket Client zurück gesendet werden.



- Programmieren Sie einen TCP Client, der den folgenden Dienst bietet:
  - Empfangen einer HTTP Seite von einem HTTP Server wie Google und
  - Anzeigen des HTTP Responses in der Systemkonsole.
- Dazu müssen Sie einen gültigen HTTP Request an den HTTP Server absetzen.



- Programmieren Sie
  - einen einfachen HTTP Server basierend auf TCP Sockets und
  - fügen Sie eine Java Swing GUI zur Verwaltung des Servers hinzu.
- Beschreiben Sie alle Unterschiede zwischen UDP und TCP Sockets
  - anhand der verschiedenen Konzepte der Datenübertragung und
  - anhand ihrer Implementierung und Nutzung in Java.
  - In welchen Fällen würden Sie welche Art der Übertragung verwenden?