

Programmieren I und II

Unit 12

Parallele Programmierung (Multithread)

(unter Nutzung von Vorlesungsunterlagen von Prof. Dr. Uwe Krohn)



Prof. Dr. rer. nat. Nane Kratzke

*Praktische Informatik und
betriebliche Informationssysteme*


- Raum: 17-0.10
- Tel.: 0451 300 5549
- Email: kratzke@fh-luebeck.de



@NaneKratzke

Updates der Handouts auch über Twitter #prog_inf
und #prog_itd


Units




Unit 1 Einleitung und Grundbegriffe	Unit 2 Grundelemente imperativer Programme	Unit 3 Selbstdefinierbare Datentypen und Collections	Unit 4 Einfache I/O Programmierung
Unit 5 Rekursive Programmierung und rekursive Datenstrukturen	Unit 6 Einführung in die objektorientierte Programmierung und UML	Unit 7 Konzepte objektorientierter Programmiersprachen	Unit 8 Testen (objektorientierter) Programme
Unit 9 Generische Datentypen	Unit 10 Objektorientierter Entwurf und objektorientierte Designprinzipien	Unit 11 Graphical User Interfaces	Unit 12 Parallele Programmierung
Unit 13 Funktionale Programmierung			

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme **3**

Abgedeckte Ziele dieser UNIT



Kennen existierender Programmierparadigmen und Laufzeitmodelle	Sicheres Anwenden grundlegender programmiersprachlicher Konzepte (Datentypen, Variable, Operatoren, Ausdrücke, Kontrollstrukturen)	Fähigkeit zur problemorientierten Definition und Nutzung von Routinen und Referenztypen (insbesondere Liste, Stack, Mapping)	Verstehen des Unterschieds zwischen Werte- und Referenzsemantik
Kennen und Anwenden des Prinzips der rekursiven Programmierung und rekursiver Datenstrukturen	Kennen des Algorithmusbegriffs, Implementieren einfacher Algorithmen	Kennen objektorientierter Konzepte Datenkapselung, Polymorphie und Vererbung	Sicheres Anwenden programmiersprachlicher Konzepte der Objektorientierung (Klassen und Objekte, Schnittstellen und Generics, Streams, GUI und MVC)
Kennen von UML Klassendiagrammen, sicheres Übersetzen von UML Klassendiagrammen in Java (und von Java in UML)	Kennen der Grenzen des Testens von Software und erste Erfahrungen im Testen (objektorientierter) Software	Sammeln erster Erfahrungen in der Anwendung objektorientierter Entwurfprinzipien	Sammeln von Erfahrungen mit weiteren Programmiermodellen und -paradigmen, insbesondere Multithread Programmierung sowie funktionale Programmierung



Am Beispiel der Sprache JAVA

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme **4**

Kleines Quiz



Apple MacBook Air

1.3 GHz Dual Core Intel i5

Anzahl Kerne?

2

Anzahl simultane Threads?

4

Wieviel Prozent nutzen sie mit
der Programmierung die sie
bislant kennen gelernt haben?

25%



Apple MacPro

Intel Xeon E5 8-Core 3 GHz

Anzahl Kerne?

8

Anzahl simultane Threads

16

6.25%

Zum Nachlesen ...



Kapitel 18

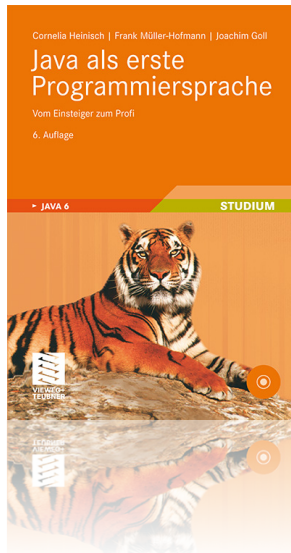
Parallele Programmierung mit Threads

18.2 Threads in Java

18.3 Wissenswertes über Threads

18.4 Thread-Synchronisation und
Kommunikation

Noch mehr zum Nachlesen ...



Kapitel 19

Threads

- 19.2 Zustände und Zustandsübergänge von Threads
- 19.3 Programmierung von Threads
- 19.4 Scheduling von Threads
- 19.5 Zugriff auf gemeinsame Ressourcen

Themen dieser Unit



Nebenläufigkeit

- Prozesse und Threads
- Threads in Java
- Zustände von Threads
- Effekte

Thread Safeness

- Leser/Schreiber Problem
- Erzeuger/Verbraucher Problem
- Thread-sichere Objekte

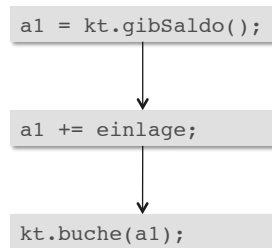
Verklemmungen

- Philosophenproblem
- Entstehung von Verklemmungen
- Vermeidung von Verklemmungen

Bisher: Sequentielle Verarbeitung (Single Thread)

Bisher haben wir (bzw. das imperative Programmiermodell) immer stillschweigend angenommen, dass eine gegebene Folge von Anweisungen Schritt für Schritt abgearbeitet.

Eine derartige Folge von sequentiell ausgeführten Anweisungen nennt man auch **Kontrollfluss** oder **Ablaufpfad** (engl. **control flow / thread**).



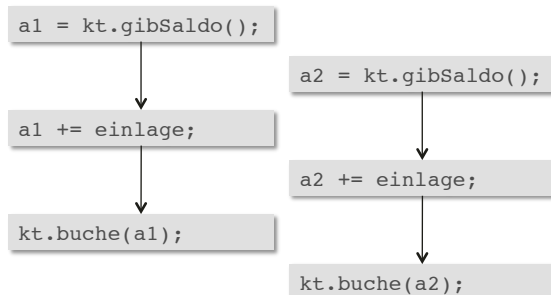
kt sei ein Objekt der Klasse Konto.

Nun: Nebenläufige Verarbeitung (Multi Thread)

In der realen Welt geschehen viele Abläufe gleichzeitig. Z.B. können mehrere Bankangestellte Buchungen auf Konten gleichzeitig vorgenommen werden.

Dies wird in der Informatik als **Nebenläufigkeit** (engl. **concurrency**) bezeichnet.

Nebenläufigkeit ist immer dann problematisch, wenn Ablaufpfade von einander abhängig sind.



Das Ergebnis der ersten Buchung wird durch das der zweiten überschrieben und geht dadurch verloren.

Erforderlich: Synchronisation

Lösung des Problems mit Hilfe eines Sperr- oder Synchronisationsmechanismus (hier nur Pseudocode).

```
kt.pruefeUndSperr();
```

Beim Start des Ablaufpfades wird das Objekt `kt` für weitere Ablaufpfade gesperrt.

```
a1 = kt.gibSaldo();
a1 += einlage;
kt.buche(a1);
```

Kritischer Abschnitt (wird nur durch maximal einen Thread zur Zeit ausgeführt)

```
kt.entsperr();
```

Nach Abschluss der Operation (die durch mehr als eine Anweisung besteht) wird das Objekt `kt` wieder für weitere Ablaufpfade freigegeben.

Eine solche unteilbare Anweisungsfolge wird auch als **Transaktion** bezeichnet.

Wirkungsweise einer Synchronisation

[1] Vor Zugriff auf das Objekt, wird geprüft ob dieses gesperrt ist. Wenn nein, wird Objekt gesperrt und die Transaktion kann beginnen.

```
kt.pruefeUndSperr();
```

```
a1 = kt.gibSaldo();
a1 += einlage;
kt.buche(a1);
```

```
kt.entsperr();
```

[3] Nach der Transaktion wird das Objekt für weitere Threads freigegeben.

[2] Nun ist das Objekt gesperrt. Es wird auf die Freigabe gewartet.

```
kt.pruefeUndSperr();
```

[4] (Der erste wartende Thread wird benachrichtigt.

```
a2 = kt.gibSaldo();
a2 += einlage;
kt.buche(a2);
```

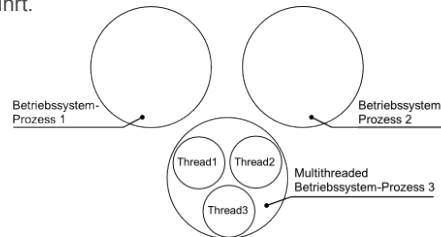
```
kt.entsperr();
```

Threads

Threads sind sogenannte leichtgewichtige Prozesse. Threads haben gemeinsam Zugriff auf den Speicherbereich eines Betriebssystem-Prozesses.

Ein Betriebssystem-Prozess kann mehrere Threads haben (mindestens einen).

Wenn ein Rechner mehrere Prozessoren hat, können Threads tatsächlich gleichzeitig ausgeführt werden, ansonsten werden sie in irgendeiner nicht vorhersagbaren Reihenfolge ausgeführt.



Bei Multithread Programmierung bietet es sich an, keine impliziten Annahmen über die Ausführungs-/Startreihenfolge von Threads zu machen!

!!! Achtung !!! Der Mensch ist nicht gut darin parallel zu denken!

Threads in Java

Threads werden in Java durch Objekte der Klasse **Thread** repräsentiert.

Die Klasse **Thread** implementiert das Interface **Runnable**, das die Methode **run()** vereinbart.

```
public interface Runnable {  
    public void run();  
}
```

Die Methode **run()** enthält die im Thread auszuführenden Anweisungen.

Erzeugen und Starten von Threads in Java

Variante 1:

Erweitern der Klasse Thread

```
// Erweitern durch anonyme Klasse
// (klassisch mittels extends geht auch)
Thread t = new Thread() {
    public void run() {
        System.out.println("New Thread");
    }
};

t.start();
```

Variante 2:

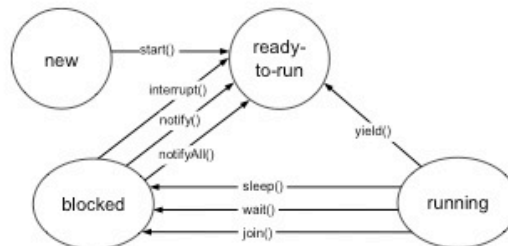
Implementieren von Runnable

```
// Neue Klasse die Runnable implementiert
// und den auszuführenden Code kapselt
class ParallelCode implements Runnable {
    public void run() {
        System.out.println("Runnable code");
    }
}

Thread t = new Thread(new ParallelCode());
t.start();
```

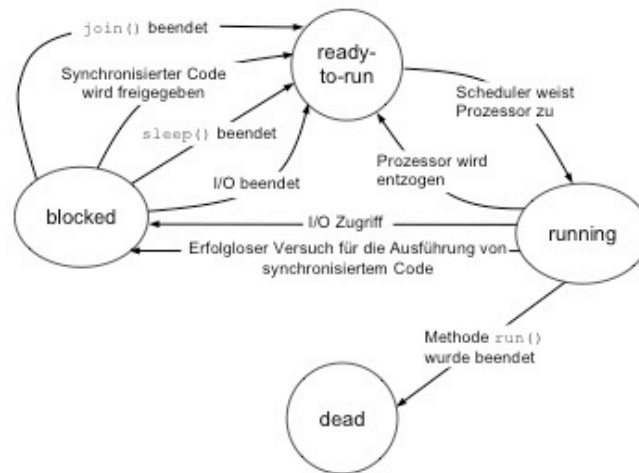
Ein Thread wird gestoppt, wenn die Methode `run()` fertig abgearbeitet wurde.

Zustandsübergänge von Threads



- start()** Startet einen Thread (Ausführung seiner `run()`-Methode)
- sleep(long m)** Entzieht einem Thread für `m` Millisekunden den Prozessor (Thread legt sich „schlafen“).
- yield()** Thread gibt freiwillig Prozessor ab. Fortsetzung nach Entscheidung Scheduler.
- join()** Thread gibt Prozessor ab, um auf Beendigung eines anderen Threads zu warten. Wird dann nach Entscheidung Scheduler fortgesetzt.
- interrupt()** Überführt einen blockierten Thread in einen ausführbaren Thread.

Durch den Scheduler erzeugte Zustandsübergänge



Beispiel: Berechnung von Fibonacci Folgen beschleunigen

Es soll nun die Fibonacci Folge von 1 bis n (z.B. 45) berechnet werden, und zwar mit folgender rekursiven Methode.

```
long fib(int n) {
    if (n == 0) { return 0; }
    if (n == 1) { return 1; }
    return fib(n-1) + fib(n-2);
}
```

Die rekursive Berechnung der Fibonacci Folge ist recht zeitaufwändig. So können zwar die Fibonacci Zahlen bis ca. 15 in kaum messbarer Zeit (Java VM Zeitauflösung) berechnet werden. Größere Zahlen dauern aber schon deutlich länger:

- fib(20) = 6765 (benötigt ca. 2 ms)
- fib(30) = 832040 (benötigt ca. 12 ms)
- fib(40) = 102334155 (benötigt aber bereits 2316 ms) und
- fib(45) = 1134903170 (benötigt sogar schon 11626 ms)

Beispiel: Berechnung von Fibonacci Folgen beschleunigen

Der Ansatz

```
for (int i = 0; i <= 45; i++) {
    System.out.println("fib(" + i + ") = " + fib(i));
}
```

wird also zu erheblichen Laufzeiten führen.

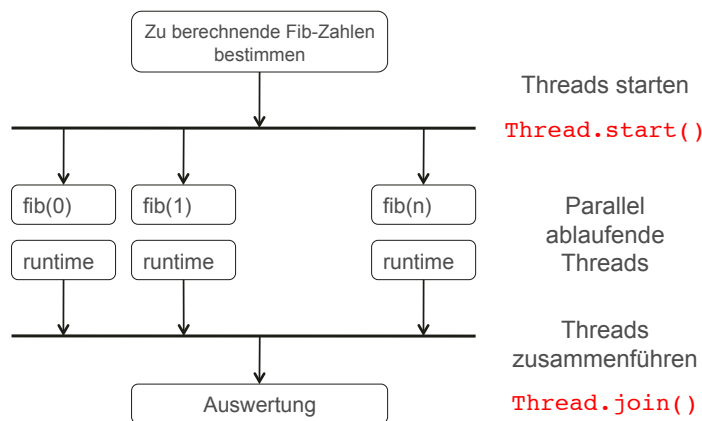
$$runtime_{seq} = \sum_{i=1}^n runtime(fib(i))$$

Mittels Threads kann man die $n=45$ Berechnungen aber in einer Zeitspanne durchführen, die **in etwa** so lang ist, wie die Berechnung der größten zu bestimmenden Fibonacci Zahl.

$$runtime_{parallel} \approx \max_{i=1}^n (runtime(fib(i)))$$

Parallelisierung mittels Threads

UML Aktivitätsdiagramm



FibonacciThread



```
class FibonacciThread extends Thread {
    private byte n;
    private long fibn;
    private long procTime;

    public FibonacciThread(byte n) { this.n = n; }

    private long fib(int n) {
        if (n == 0) { return 0; }
        if (n == 1) { return 1; }
        return fib(n-1) + fib(n-2);
    }

    public void run() {
        long start = System.currentTimeMillis();
        this.fibn = fib(this.n);
        long end = System.currentTimeMillis();
        this.procTime = end - start;
    }
}
```

```
public byte getN() {
    return this.n;
}

public long getFibN() {
    return this.fibn;
}

public long getProcTime() {
    return this.procTime;
}
}
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

21

FibonacciThreads starten



```
// Zu berechnende Fibonaccizahlen bestimmen
List<FibonacciThread> threads = new ArrayList<FibonacciThread>();
for (byte i = 0; i <= 45; i++) { threads.add(new FibonacciThread(i)); }

long start = System.currentTimeMillis();
for (FibonacciThread t : threads) { t.start(); } // Threads starten
for (FibonacciThread t : threads) { t.join(); } // Threads zusammenführen
long runtime = System.currentTimeMillis() - start;

// Auswertung
long total = 0; long max = 0;
for (FibonacciThread t : threads) {
    byte n = t.getN(); long fibn = t.getFibN(); long proc = t.getProcTime();
    System.out.println("fib(" + n + ") = " + fibn + " (berechnet in " + proc + " ms)");
    max = proc > max ? proc : max;
    total += proc;
}

System.out.println("Gemessene Laufzeit: " + runtime + " ms");
System.out.println("Längste Laufzeit: " + max + " ms");
System.out.println("Addierte Laufzeit: " + total + " ms");
System.out.println("Speedup von: " + ((double)total / runtime));
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

22

Auswertung (Fibonacci bis 45)



University of Applied Sciences

```
fib(0) = 0 (berechnet in 0 ms)
fib(1) = 1 (berechnet in 0 ms)
fib(2) = 1 (berechnet in 0 ms)
fib(3) = 2 (berechnet in 0 ms)
...
fib(20) = 6765 (berechnet in 5 ms)
...
fib(30) = 832040 (berechnet in 9 ms)
...
fib(43) = 433494437 (berechnet in 5303 ms)
fib(44) = 701408733 (berechnet in 7311 ms)
fib(45) = 1134903170 (berechnet in 9883 ms)
```

Gemessene Laufzeit: 10067 ms
Längste Laufzeit: 9883 ms
Addierte Laufzeit: 35206 ms
Speedup von: 3.4971689679149

Die sequentielle Laufzeit hätte ca. 35 sec gedauert (addierte Laufzeit).

Tatsächlich wurde etwa 10 sec gerechnet (gemessene Laufzeit).

Die längste Berechnung für eine Fibonacci Zahl betrug ca. 9.8 sec (längste Laufzeit).

Parallelisierung hat das Programm somit etwa 3.5 mal schneller gemacht als die rein sequentielle Berechnung (Speedup).

Ist Multithreading immer eine gute Idee?

Was passiert, wenn wir einfachere Probleme parallelisieren (z.B. nur die Fibonacci Folge bis 25 berechnen)?

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

23

Auswertung (Fibonacci bis 25)



University of Applied Sciences

```
fib(0) = 0 (berechnet in 0 ms)
fib(1) = 1 (berechnet in 0 ms)
fib(2) = 1 (berechnet in 0 ms)
fib(3) = 2 (berechnet in 0 ms)
...
fib(25) = 6765 (berechnet in 1 ms)
```

Gemessene Laufzeit: 18 ms
Längste Laufzeit: 4 ms
Addierte Laufzeit: 16 ms
Speedup von: 0.888888

Wir erhalten einen Speedup von < 1.

D.h. parallelisierter Code ist langsamer als es sequentieller Code wäre!

Warum? Das Anlegen, Starten und Zusammenführen von Threads erzeugt Aufwand. Muss pro Thread nur wenig gerechnet werden, ist der Aufwand der Thread-Verwaltung ggf. größer als der Nutzen der Parallelisierung.

Es sollten also vor allem rechenintensive Aufgaben parallelisiert werden.

Hinweis: Zu Performance Messung und Tuning existieren interessante Wahlpflichtveranstaltungen!

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

24

Wann wird ein Prozess beendet?

Ein Java Programm wird beendet, sobald der letzte noch laufende Thread beendet wurde.

In Java kann man aber spezielle Threads (Dämon-Threads) deklarieren, die die Fähigkeit haben „ewig“ zu laufen und, dennoch die Terminierung eines Programms nicht verhindern.

In diesen **Dämon-Threads** laufen normalerweise Hintergrundaufgaben ab, die Service Leistung für Vordergrund Threads anbieten.

Der Java Garbage Collector oder die Java Swing GUI laufen bspw. in solcher Art Thread.

Es muss also eigentlich genauer heißen:

Ein Java Programm wird also beendet, sobald der letzte noch laufende Nichtdämon-Thread beendet wurde.



```
Thread t = new Thread() {  
    public void run() {  
        while (true) {  
            System.out.println(  
                „Ich bin ein Dämon!“ +  
                „Ich laufe bis zum Ende.“  
            );  
        }  
    }  
};  
t.setDaemon(true);  
t.start();
```

Themen dieser Unit

Nebenläufigkeit

- Prozesse und Threads
- Threads in Java
- Zustände von Threads
- Effekte

Thread Safeness

- Leser/Schreiber Problem
- Erzeuger/Verbraucher Problem
- Thread-sichere Objekte

Verklemmungen

- Philosophenproblem
- Entstehung von Verklemmungen
- Vermeidung von Verklemmungen

Das Leser/Schreiber Problem

Wenn in einem Programm mehrere Threads eingesetzt werden, muss stets bedacht werden, dass diese **gleichzeitig** Zugriff auf denselben Speicherbereich haben.

Beim Zugriff auf Variablen, Datenfelder sind wir bislang jedoch immer implizit davon ausgegangen, dass nur ein Thread gleichzeitig auf den Hauptspeicher zugreift. Diese Annahme muss fallengelassen werden, wenn multithreaded programmiert wird.

Ein typisches Problem dabei ist, dass sogenannte Leser/Schreiber Problem.

Zwei (oder mehrere) Threads greifen gleichzeitig lesend und schreibend auf denselben Speicherbereich zu.

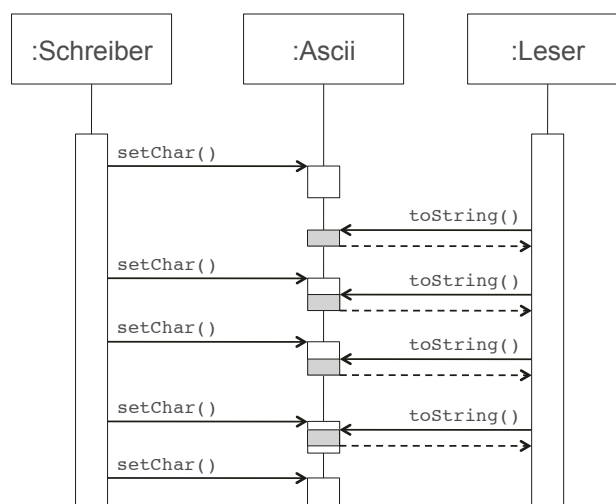
Dabei auftretende Probleme wollen wir nun beleuchten. Wir schaffen uns hierzu eine Klasse `Ascii`, die für einen gegebenen Character seine Ascii Nummer zurückgibt.

```
class Ascii {
    protected char x = 'A';
    protected int value = (int) x;

    public void setChar(char x) {
        this.x = x;
        this.value = (int) x;
    }

    public String toString() {
        return "(" + x + " = " + value + ")";
    }
}
```

Das Leser/Schreiber Problem



Schreiber und Leser

```
class Schreiber extends Thread {
    Ascii asc;

    public Schreiber(Ascii f) {
        this.asc = f;
    }

    public void run() {
        while (true) {
            int z = (int)(Math.random() * 6);
            asc.setChar((char)('A' + z));
        }
    }
}
```

```
class Leser extends Thread {
    Ascii asc;

    public Leser(Ascii f) {
        this.asc = f;
    }

    public void run() {
        for (int i = 1; i <= 25; i++) {
            System.out.print(asc + " ");
            try { Thread.sleep(10); }
            catch (InterruptedException e) {}
            if (i % 5 == 0 {
                System.out.println();
            }
        }
    }
}
```

Resultat

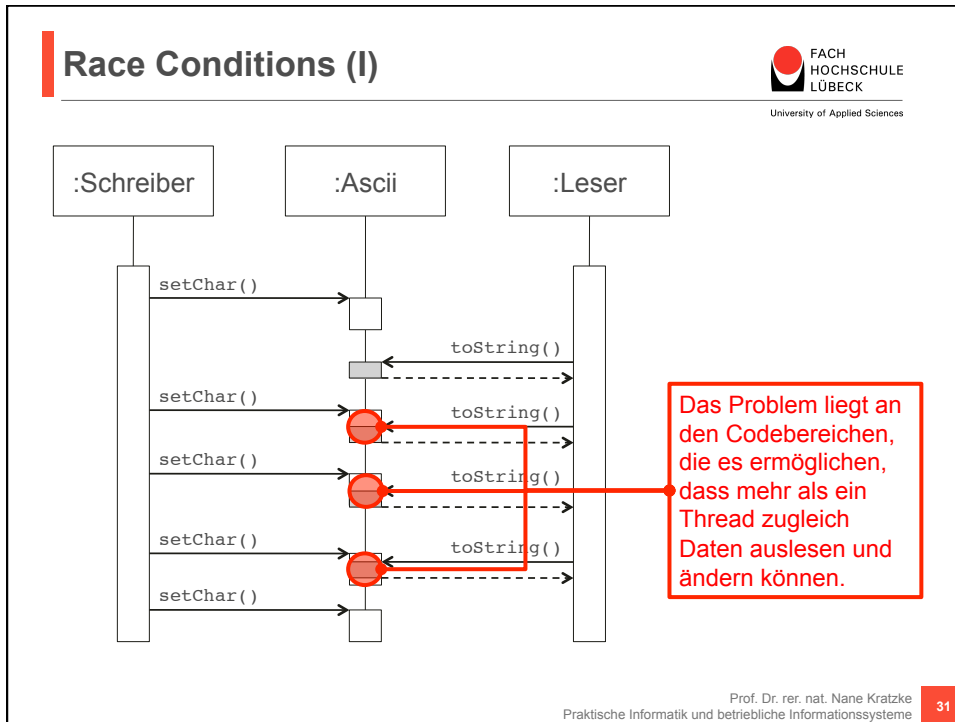
```
Ascii a = new Ascii();
Schreiber s = new Schreiber(a); s.setDaemon(true);
Leser l = new Leser(a);

s.start();
l.start();
```

Ergebnisse
eines exem-
plarischen
Testlaufs

```
(A == 65) (C == 65) (F == 69) (E == 69) (F == 69)
(A == 68) (C == 67) (F == 70) (C == 70) (F == 67)
(D == 70) (F == 67) (D == 67) (B == 66) (D == 68)
(D == 67) (E == 69) (E == 66) (C == 67) (A == 66)
(B == 70) (E == 67) (E == 70) (C == 69) (E == 69)
```

Rot markierte Ausgaben sind falsch (Mehr als 50%) !!!



Race Conditions (II)

FACH HOCHSCHULE LÜBECK
 University of Applied Sciences

In die Methoden `setChar()` und `toString()` können Threads zeitgleich vollkommen unsynchronisiert eintreten.

An der Methode `setChar()` sieht man aber bspw. das zwei Anweisungen ausgeführt werden (Anweisungen in Java selber müssen ferner nicht atomar sein).

So könnte es bspw. vorkommen, dass nachdem die Anweisung `this.x = x;` in `setChar()` ausgeführt wurde, der Leser-Thread die `toString()` Methode ausführt. Dann bekommt er nur die Hälfte des Outputs der `setChar()` Methode mit.

So ergeben sich die gezeigten Inkonsistenzen (obwohl der Code grundsätzlich korrekt ist).

```
class Ascii {
    protected char x = 'A';
    protected int value = (int) x;

    public void setChar(char x) {
        this.x = x;
        this.value = (int) x;
    }

    public String toString() {
        return "(" + x + " = " + value + ")";
    }
}
```

Solcher Art Code, ist daher nicht **threadsicher** (engl. **thread-safe**) programmiert.

Code, der berücksichtigt, dass mehr als ein Thread ihn ausführt und sicherstellt, das Inkonsistenzen aufgrund von Race Conditions nicht auftreten, nennt man hingegen thread-safe programmiert.

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 32

Race Conditions (III)



Um diesen Effekt zu vermeiden, können Methoden in Java mit dem Schlüsselwort **synchronized** gekennzeichnet werden.

Für eine synchronisierte Methode stellt die Java VM sicher, dass zu einem Zeitpunkt nur ein Thread Code des zugehörigen Objekts ausführt.

```
class Ascii {  
    protected char x = 'A';  
    protected int value = (int) x;  
  
    public synchronized void setChar(char x) {  
        this.x = x;  
        this.value = (int) x;  
    }  
  
    public synchronized String toString() {  
        return "(" + x + " = " + value + ")";  
    }  
}
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

33

Objektübergreifende Synchronisierung (I)



Das **synchronized** Schlüsselwort kann auch anders eingesetzt werden, indem es auf Objekten eingesetzt wird. Damit lassen sich dann objektübergreifende Synchronisierungen realisieren, sogar von Objekten, die aus nicht thread-safe programmierten Klassen instanziiert wurden.

```
class TSSchreiber extends Thread {  
    NTSAscii asc;  
  
    public TSSchreiber(NTSAscii f) {  
        this.asc = f;  
    }  
  
    public void run() {  
        while (true) {  
            int z = (int)(Math.random() * 6);  
            synchronized (asc) {  
                asc.setChar((char)('A' + z));  
            }  
        }  
    }  
}
```

```
class TSLeser extends Thread {  
    NTSAscii asc;  
  
    public TSLeser(NTSAscii f) { this.asc = f; }  
  
    public void run() {  
        for (int i = 1; i <= 25; i++) {  
            synchronized (asc) {  
                System.out.print(asc + " ");  
            }  
            try { Thread.sleep(10); }  
            catch (InterruptedException e) { }  
            if (i % 5 == 0) {  
                System.out.println();  
            }  
        }  
    }  
}
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

34

Objektübergreifende Synchronisierung (II)

Das `synchronized` Schlüsselwort auf Methoden ist eigentlich nur eine Kurznotation für:

```
class TAscii {
    protected char x = 'A';
    protected int value = (int) x;

    public synchronized void setChar(char x) {
        this.x = x;
        this.value = (int) x;
    }

    public synchronized String toString() {
        return "(" + x + " = " + value + ")";
    }
}
```

```
class TAscii {
    protected char x = 'A';
    protected int value = (int) x;

    public void setChar(char x) {
        synchronized (this) {
            this.x = x;
            this.value = (int) x;
        }
    }

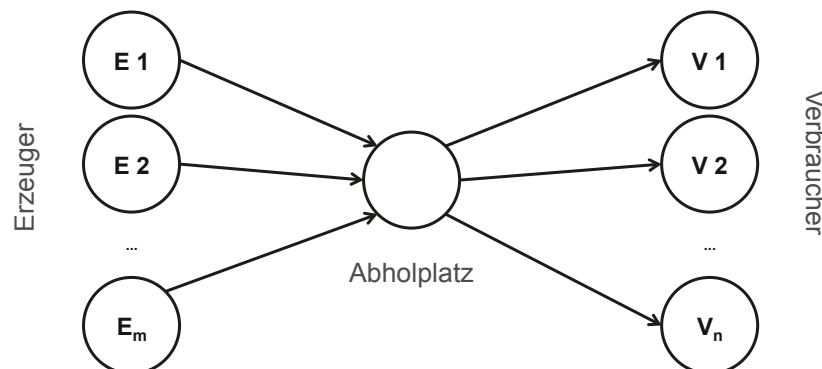
    public String toString() {
        synchronized (this) {
            return "(" + x + " = " + value + ")";
        }
    }
}
```

Das Erzeuger/Verbraucher Problem

Wenden wir uns nun einem weiteren Problem der parallelen Programmierung zu.

Es soll nun mehrere Erzeuger und mehrerer Verbraucher geben. Erzeuger erzeugen Waren und legen diese auf einem Abholplatz ab. Der Abholplatz hat aus Gründen der Einfachheit nur eine Ablage (einelementige Warteschlange). Verbraucher holen diese Waren von diesem Abholplatz ab.

Wenn der Abholplatz belegt ist, sollen Erzeuger keine weiteren Waren mehr produzieren und ablegen.
 Wenn der Abholplatz leer ist, sollen Verbraucher warten, bis der



Erzeuger/Verbraucher Problem

Hier die Singlethreaded Implementierung des Abholplatzes und der Ware.

```
class Abholplatz {
    protected Ware ware;

    public Ware get() {
        Ware ret = this.ware;
        this.ware = null;
        System.out.println("Ware " + ret + " abgeholt.");
        return ret;
    }

    public void put(Ware w) {
        this.ware = w;
        System.out.println("Ware " + w + " abgelegt.");
    }
}
```

```
class Ware {
    private String name;

    public Ware(String n) {
        this.name = n;
    }

    public String toString() {
        return this.name;
    }
}
```

Erzeuger Thread

```
class Erzeuger extends Thread {
    String name;
    Abholplatz platz;
    int anzahl;

    public Erzeuger(String name, Abholplatz p, int n) {
        this.name = name;
        this.platz = p;
        this.anzahl = n;
    }

    public void run() {
        for (int i = 1; i <= this.anzahl; i++) {
            Ware w = new Ware(this.name + " Produkt " + i);
            platz.put(w);
        }
    }
}
```

Ein Erzeuger Thread erzeugt n Waren und legt diese auf dem Abholplatz ab.

Es kann mehrere Erzeuger Threads geben.

Verbraucher Thread

```
class Verbraucher extends Thread {  
    String name;  
    Abholplatz platz;  
    int anzahl;  
  
    public Verbraucher(String name, Abholplatz p, int n) {  
        this.name = name;  
        this.platz = p;  
        this.anzahl = n;  
    }  
  
    public void run() {  
        for (int i = 1; i <= this.anzahl; i++) {  
            Ware w = platz.get();  
        }  
    }  
}
```

Ein Verbraucher Thread konsumiert n Waren und holt diese vom Abholplatz ab.

Es kann mehrere Verbraucher Threads geben.

Zusammenspiel von Erzeuger und Verbraucher

```
Abholplatz p = new Abholplatz();  
  
Erzeuger e1 = new Erzeuger("Erzeuger 1", p, 5); e1.start();  
Erzeuger e2 = new Erzeuger("Erzeuger 2", p, 5); e2.start();  
Erzeuger e3 = new Erzeuger("Erzeuger 3", p, 10); e3.start();  
  
Verbraucher v1 = new Verbraucher("Verbraucher 1", p, 10); v1.start();  
Verbraucher v2 = new Verbraucher("Verbraucher 2", p, 10); v2.start();
```

In diesem Beispiel werden drei Erzeuger e_1 , e_2 und e_3 , die 5, 5 und 10 Waren erzeugen gestartet.

Den Verbrauch übernehmen zwei Verbraucher v_1 und v_2 , die jeweils 10 Waren konsumieren.

Alle Verbraucher und Erzeuger sind über einen Abholplatz p verbunden.

Zusammenspiel von Erzeuger und Verbraucher

Werden diese Threads gestartet, ergibt sich z.B. folgende Ausgabe!

```
Ware Erzeuger 1 Produkt 1 abgelegt.  
Ware Erzeuger 1 Produkt 2 abgelegt.  
Ware Erzeuger 1 Produkt 3 abgelegt.  
...  
Ware null abgeholt.  
Ware null abgeholt.  
Ware null abgeholt.
```

Waren werden auf Waren abgelegt!

Nicht existente Waren werden
abgeholt!



Vollkommenes Chaos!

Synchronisierter Abholplatz

```
class Abholplatz {  
    protected Ware ware;  
  
    public synchronized Ware get() {  
        Ware ret = this.ware;  
        this.ware = null;  
        System.out.println("Ware " + ret + " abgeholt.");  
        return ret;  
    }  
  
    public synchronized void put(Ware w) {  
        this.ware = w;  
        System.out.println("Ware " + w + " abgelegt.");  
    }  
}
```

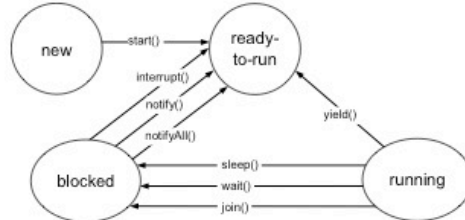
Auch der Einsatz von
synchronized alleine löst
das Problem nicht.

Dies Ausgabe bleibt im
wesentlichen erhalten.

Der Abholplatz ist also
nicht im mindesten **thread-**
safe!

Das ist auch kein Wunder,
denn die Threads
kommunizieren in keinsten
Weise miteinander.

Methoden zur Thread Kommunikation



Jedes Objekt bietet in Java hierzu zwei wesentliche Methoden an, damit sich Threads gegenseitig benachrichtigen können und bis zur Benachrichtigung „schlafen“ (d.h. keine Prozessorressourcen beanspruchen):

- **void wait() throws InterruptedException** veranlasst den Thread, der gerade ausgeführt wird, mit seiner weiteren Ausführung zu warten, bis ein anderer Thread die `notifyAll()` Methode auf dem Objekt ausführt. Der Thread gibt dazu die Objekt-Sperre ab und muss sie nach dem Wartevorgang wieder erwerben.
- **void notifyAll()** reaktiviert alle Threads, die sich im Wartezustand bezüglich des Objekts befinden.

Synchronisierter und kommunizierender Abholplatz

```

class Abholplatz {
    protected Ware ware;

    public synchronized Ware get() {
        while (ware == null) { wait(); }
        Ware ret = this.ware;
        this.ware = null;
        System.out.println("Ware " + ret + " abgeholt.");
        notifyAll();
        return ret;
    }

    public synchronized void put(Ware w) {
        while (this.ware != null) { wait(); }
        this.ware = w;
        System.out.println("Ware " + w + " abgelegt.");
        notifyAll();
    }
}
    
```

Warte solange bis Abholplatz belegt ist.

Benachrichtige alle wartenden Threads (dass Abholplatz wieder frei ist).

Warte solange bis Abholplatz frei ist.

Benachrichtige alle wartenden Threads (dass Abholplatz nun belegt ist).

Sicherheit paralleler Programme

Ein nebenläufiges Programm gilt als sicher (thread safe), wenn keine fehlerhaften Zustände in einem Programm eintreten (vgl. auch Unit 8).

Ein objektorientiertes paralleles Programm ist thread safe, wenn alle Objekte innerhalb des Programms sicher sind.

Ein Objekt ist **thread safe**, wenn es

- **unveränderlich** ist oder
- **korrekt synchronisiert** ist oder
- **korrekt gekapselt** ist.



Unveränderliche Objekte

Ein Objekt ist unveränderlich, wenn es niemals seinen Status ändert.

Ist ein Objekt veränderlich, so darf man nur die statuslosen Methoden des Objekts nebenläufig verwenden.

Eine statuslose Methode ist eine Methode, die kein Datenfeld ihres Objekts verändert.

```
class ImmutablePerson {  
    protected String vorname;  
    protected String nachname;  
  
    public ImmutablePerson(String vn, String nn) {  
        this.vorname = vn;  
        this.nachname = nn;  
    }  
  
    public String getVorname() {  
        return this.vorname;  
    }  
  
    public String getNachname() {  
        return this.nachname;  
    }  
  
    public String toString() {  
        return getVorname() + " " + getNachname();  
    }  
}
```

Korrekt synchronisierte Objekte

Ein korrekt synchronisiertes Objekt ist ein Objekt, dessen statusbehafteten Methoden synchronisiert sind (und ggf. das Erzeuger/Verbraucher Problem behandeln).

Ein Objekt dessen statusbehafteten Methoden alle synchronisiert sind, heißt **vollständig synchronisiert**.

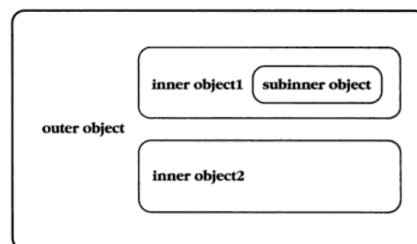
```
class MutablePerson extends ImmutablePerson {  
  
    public MutablePerson(String vn, String nn) {  
        super(vn, nn);  
    }  
  
    public synchronized void setName(String vn, String nn) {  
        this.vorname = vn;  
        this.nachname = nn;  
    }  
  
}
```

Korrekt gekapseltes Objekt

Statt der Synchronisation, die dynamisch den Zugriff auf Objekte begrenzt, kann auch der Zugriff auf Objekte durch Kapselung (containment) strukturell begrenzt werden. Objekte werden hierbei logisch gekapselt.

Folgende Regeln sind dabei einzuhalten:

- Das äußere Objekt muss alle inneren Objekte innerhalb seines eigenen Konstruktors erzeugen.
- Das äußere Objekt darf keine Referenzen des inneren Objekts an andere Objekte weitergeben.
- Das äußere Objekt muss vollständig synchronisiert oder in ein anderes vollständig synchronisiertes Objekt eingebettet sein.



Themen dieser Unit

FACH HOCHSCHULE LÜBECK
 University of Applied Sciences

Nebenläufigkeit

- Prozesse und Threads
- Threads in Java
- Zustände von Threads
- Effekte

Thread Safeness

- Leser/Schreiber Problem
- Erzeuger/Verbraucher Problem
- Thread-sichere Objekte

Verklemmungen

- Philosophenproblem
- Entstehung von Verklemmungen
- Vermeidung von Verklemmungen

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 49

Das Philosophenproblem und die Entstehung von Deadlocks

FACH HOCHSCHULE LÜBECK
 University of Applied Sciences

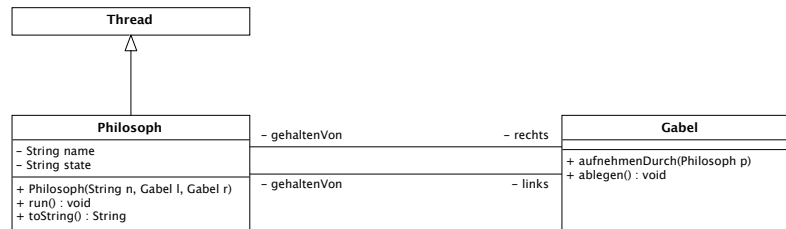
Fünf Philosophen sitzen an einem runden Tisch und jeder hat einen Teller mit Spaghetti vor sich. Zum Essen von Spaghetti benötigt jeder Philosoph zwei Gabeln. Allerdings sind nur fünf Gabeln vorhanden, die nun zwischen den Tellern liegen. Die Philosophen können also nicht gleichzeitig speisen.

Die Philosophen sitzen am Tisch und denken über philosophische Probleme nach. Wenn einer hungrig wird, greift er zuerst die Gabel links von seinem Teller, dann die auf der rechten Seite und beginnt zu essen. Wenn er satt ist, legt er die Gabeln wieder zurück und beginnt wieder zu denken. Sollte eine Gabel nicht an ihrem Platz liegen, wenn der Philosoph sie aufnehmen möchte, so wartet er, bis die Gabel wieder verfügbar ist.

Wenn sich alle fünf Philosophen gleichzeitig entschließen, zu essen, ergreifen also alle gleichzeitig ihre linke Gabel und nehmen damit dem jeweils links von ihnen sitzenden Kollegen seine rechte Gabel weg. Nun warten alle fünf darauf, dass die rechte Gabel wieder auftaucht. Das passiert aber nicht, da keiner der fünf seine linke Gabel zurücklegt. Die Philosophen verhungern.

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 50

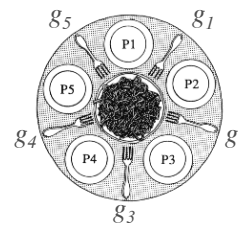
Objektorientierte Formulierung des Philosophenproblems



```

Gabel g1 = new Gabel();
Gabel g2 = new Gabel();
Gabel g3 = new Gabel();
Gabel g4 = new Gabel();
Gabel g5 = new Gabel();

Philosoph p1 = new Philosoph("P1", g1, g5);
Philosoph p2 = new Philosoph("P2", g2, g1);
Philosoph p3 = new Philosoph("P3", g3, g2);
Philosoph p4 = new Philosoph("P4", g4, g3);
Philosoph p5 = new Philosoph("P5", g5, g4);
    
```



Implementierung der Philosophen

```

class Philosoph extends Thread {
    [...]
    public void run() {
        while (true) {
            this.state = "denkt";
            Thread.sleep((int)(Math.random() * DENKZEIT));

            this.state = "wartet auf linke Gabel";
            this.links.aufnehmenDurch(this);
            this.state = "wartet auf rechte Gabel";
            this.rechts.aufnehmenDurch(this);

            this.state = "isst";
            Thread.sleep((int)(Math.random() * ESSZEIT));

            this.state = "legt linke Gabel ab";
            this.links.ablegen();
            this.state = "legt rechte Gabel ab";
            this.rechts.ablegen();
        }
    }
    [...]
}
    
```



Hinweis: Datenfelder, Konstruktor und Exception Handling aus Gründen der Anschaulichkeit weggelassen.

Implementierung der Gabeln (konkurrierend genutzte Ressource)

```
class Gabel {  
  
    private Philosoph gehaltenVon;  
  
    public synchronized void aufnehmenDurch(Philosoph p) {  
        while (this.gehaltenVon != null) f wait();  
        this.gehaltenVon = p;  
        notifyAll();  
    }  
  
    public synchronized void ablegen() {  
        this.gehaltenVon = null;  
        notifyAll();  
    }  
}
```



Hinweis: Exception Handling aus Gründen der Anschaulichkeit weggelassen.

Ausführen des Philosophenproblems ...

```
Gabel g1 = new Gabel(); Gabel g2 = new Gabel();  
Gabel g3 = new Gabel(); Gabel g4 = new Gabel();  
Gabel g5 = new Gabel();  
  
Philosoph p1 = new Philosoph("P1", g1, g5);  
Philosoph p2 = new Philosoph("P2", g2, g1);  
Philosoph p3 = new Philosoph("P3", g3, g2);  
Philosoph p4 = new Philosoph("P4", g4, g3);  
Philosoph p5 = new Philosoph("P5", g5, g4);  
  
while (true) { // Jede Sekunde den Philosophenzustand ausgeben  
    Thread.sleep(1000);  
    System.out.println(p1); System.out.println(p2);  
    System.out.println(p3); System.out.println(p4);  
    System.out.println(p5);  
    System.out.println();  
}
```

Hinweis: Exception Handling aus Gründen der Anschaulichkeit weggelassen.

... führt irgendwann zu dieser Ausgabe

```
[...]  
P1 wartet auf rechte Gabel.  
P2 denkt.  
P3 isst.  
P4 denkt.  
P5 isst.  
  
P1 denkt.  
P2 denkt.  
P3 isst.  
P4 wartet auf rechte Gabel.  
P5 denkt.
```

```
P1 wartet auf rechte Gabel.  
P2 wartet auf rechte Gabel.  
P3 wartet auf rechte Gabel.  
P4 wartet auf rechte Gabel.  
P5 wartet auf rechte Gabel.  
[...]  
P1 wartet auf rechte Gabel.  
P2 wartet auf rechte Gabel.  
P3 wartet auf rechte Gabel.  
P4 wartet auf rechte Gabel.  
P5 wartet auf rechte Gabel.
```

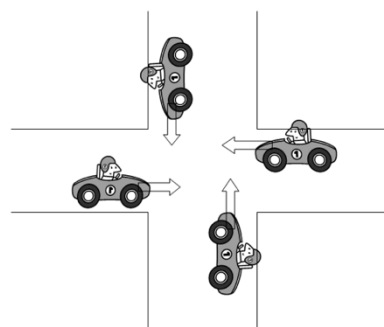
Der Zustand eines Deadlocks kann folgendermaßen definiert werden: Eine Menge von Prozessen oder Threads befindet sich in einem Deadlock, wenn jeder dieser Prozesse/Threads auf ein Ereignis wartet, das nur ein anderer Prozess/Thread aus dieser Menge verursachen kann.

Dieser Zustand wird ewig bestehen bleiben. Alle Philosophen warten aufeinander (und werden verhungern).
Dies nennt man einen Deadlock!

Verklemmungen / Deadlock

Deadlocks/Verklemmungen können im allgemeinen nicht verhindert werden. Sie sind ein statistisches Phänomen, welches Auftritt wenn folgende vier Kriterien erfüllt sind:

- **No Preemption:** Die Betriebsmittel werden ausschließlich durch die Prozesse/Threads freigegeben (**Nur die Philosophen legen Gabeln ab**).
- **Hold and Wait:** Die Prozesse/Threads fordern Betriebsmittel an, behalten aber zugleich den Zugriff auf andere (**Philosophen nehmen erst rechte, dann die linke Gabel auf**).
- **Mutual Exclusion:** Der Zugriff auf die Betriebsmittel ist exklusiv (**Eine Gabel kann nur durch genau einen Philosophen gehalten werden**).
- **Circular Wait:** Mindestens zwei Prozesse besitzen bezüglich der Betriebsmittel eine zirkuläre Abhängigkeit (**Philosophen sind über die zwischen ihnen liegenden Gabeln von einander abhängig**).



Beispiel einer Verklemmung im Straßenverkehr (jeder muss auf den von rechts kommenden Verkehrsteilnehmer warten).

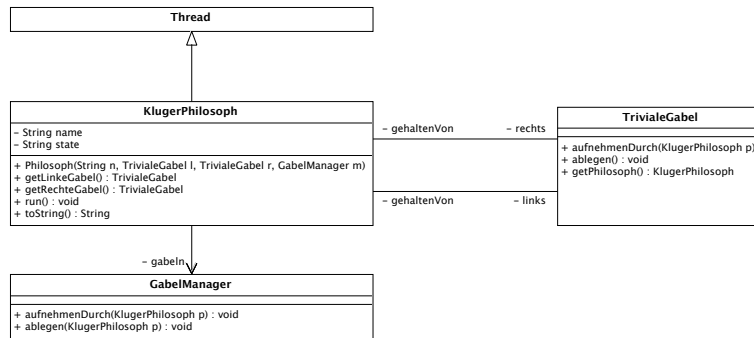
Verhinderung von Verklemmungen / Deadlocks

Deadlocks lassen sich verhindern, wenn die Synchronisierung bei einem Problem derart gestaltet werden kann, dass mindestens einer der vier Deadlock Kriterien nicht erfüllt ist.

In der Praxis gelingt dies häufig mittels der Vermeidung des „Hold and Wait“, in dem für eine Operation erforderlichen Ressourcen alle auf einmal angefordert werden.

Wir wollen das Philosophenproblem nun derart umformulieren, dass ein Philosoph, wenn er essen möchte, beide Gabeln gleichzeitig aufnimmt (also ein Hold und Wait nicht eintreten kann).

Hierzu führen wir einen GabelManager ein, der die gleichzeitige Aufnahme der Gabeln sicherstellen soll.



Implementierung des Klugen Philosophen

```

class KlugerPhilosoph extends Thread {
    [...]
    public void run() {
        while (true) {
            this.state = "denkt";
            Thread.sleep((int)(Math.random() * DENKZEIT));

            this.state = "wartet auf Gabeln";
            this.gabeln.aufnehmen(this);

            this.state = "isst";
            Thread.sleep((int)(Math.random() * ESSZEIT));

            this.state = "legt Gabeln ab";
            this.gabeln.ablegen(this);
        }
    }
    [...]
}
    
```

Hinweis: Datenfelder, Konstruktor und Exception Handling aus Gründen der Anschaulichkeit weggelassen.



Dieser wendet sich an den Gabel Manager, um linke und rechte Gabel gleichzeitig zu erhalten und abzulegen.

So vermeidet der Philosoph das „Hold and Wait“.

Implementierung des Gabel Managers

```
class GabelManager {  
  
    public synchronized void aufnehmen(KlugerPhilosoph p) {  
        while (p.getRechteGabel().getPhilosoph() != null &&  
            p.getLinkeGabel().getPhilosoph() != null) { wait(); }  
        p.getRechteGabel().aufnehmenDurch(p);  
        p.getLinkeGabel().aufnehmenDurch(p);  
        notifyAll();  
    }  
  
    public synchronized void ablegen(KlugerPhilosoph p) {  
        p.getRechteGabel().ablegen();  
        p.getLinkeGabel().ablegen();  
        notifyAll();  
    }  
  
}
```



Der Gabel Manager stellt sicher, dass ein Philosoph immer zwei Gabeln gleichzeitig erhält.

Möchte der Philosoph essen, sind aber nicht beide Gabeln verfügbar, wird der Philosoph wieder schlafen geschickt.

Hinweis: Exception Handling aus Gründen der Anschaulichkeit weggelassen.

Implementierung der trivialen Gabeln (konkurrierend genutzte Ressource)

```
class TrivialeGabel {  
  
    private KlugerPhilosoph gehaltenVon;  
  
    public void aufnehmenDurch(KlugerPhilosoph p {  
        this.gehaltenVon = p;  
    }  
  
    public void ablegen() {  
        this.gehaltenVon = null;  
    }  
  
    public KlugerPhilosoph getPhilosoph() {  
        return this.gehaltenVon;  
    }  
  
}
```



Hinweis: Die Implementierung der Gabeln ist dann nicht komplizierter als im Single Threaded Fall.

The Downfall of Imperative Programming



„If programmers were electricians, parallel programmers would be bomb disposal experts. Both cut wires [...]“

Bartosz Milewski, „The Downfall of Imperative Programming“

Quelle (letzter Zugriff am 22.12.2013):

<https://www.fpcomplete.com/business/blog/the-downfall-of-imperative-programming/>

Zusammenfassung

- **Nebenläufigkeit**
 - Thread vs. Prozess
 - Zustandsübergänge von Threads
 - (Grenzen der) Beschleunigung durch Parallelisierung
- **Thread Safeness**
 - Leser/Schreiber Problem (Synchronisierung)
 - Erzeuger/Verbraucher Problem (wait/notify)
 - Kriterien Thread-sicherer Objekte (Unveränderlichkeit oder korrekte Synchronisierung oder korrekte Kapselung)
- **Deadlocks/Verklemmungen**
 - Philosophenproblem
 - Kriterien zur Entstehung von Deadlocks (no preemption, hold and wait, mutual exclusion, circular wait)
 - Vermeidung von Verklemmungen (Ressourcen auf einmal zuteilen vermeidet bspw. hold and wait)

