Synchronisation

Programmiermethodik 2

Zum Nachlesen:
Christian Ullenboom: Java ist auch eine Insel, Kapitel 14.5/14.6
http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_14_005.htm

Wiederholung

- Parallelität
- Erzeugen
- Beenden
- Weitere Methoden
- Timer

Ausblick

Agenda

- Probleme paralleler Programmausführung
- Kritischer Abschnitt
- Monitor-Mechanismus
- Reihenfolge-Beschränkungen
- Deadlocks

Zum Nachlesen

- Kathy Sierra, Bert Bates: Java von Kopf bis Fuß, Kapitel 15 ab Abschnitt "Multithreading", O'Reilly-Verlag
- zu `wait()/notify()`: Oracle Java Tutorial "Guarded Blocks": *http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html*, abgerufen am 11.04.2014
- Christian Ullenboom: Java ist auch eine Insel, 9., aktualisierte Auflage 2011, Galileo Computing, Kapitel 14.6: Synchronisation über Warten und Benachrichtigen, auch online verfügbar

Probleme paralleler Programmausführung

Das „Lost Update“-Problem

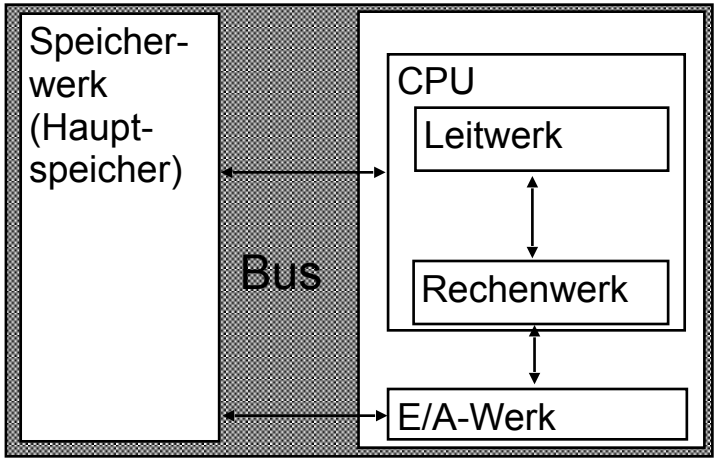
- Eine „harmlose“ Zeile Java-Quelltext:

```
zaehler = zaehler + 1;
```

- Warum ist diese Zeile nicht so harmlos?
  - Weil zaehler eine Objektvariable ist und deshalb potentiell von mehrere parallelen Threads manipuliert werden kann.
  - Weil die Java-Anweisung (eine Zuweisung) nur in der sequentiellen Programmierung atomar ist.
  - Weil Java-Programme (wie fast alle Programme) auf der von Neumann-Architektur ausgeführt werden.
- Was genau geht denn schief?

von Neumann-Rechner

- Rechner besteht aus 4 Werken
- Rechnerstruktur ist unabhängig vom bearbeiteten Problem
- Programme und Daten stehen im selben Speicher
- Der Hauptspeicher ist in Zellen gleicher Größe unterteilt, die durchgehend adressierbar sind
- Das Programm besteht aus Folgen von Befehlen, die generell nacheinander ausgeführt werden.
- Von der sequenziellen Abfolge kann durch Sprungbefehle abgewichen werden
- Die Maschine benutzt Binärcodes für die Darstellung von Programm und Daten



Imperative Programme

- elementaren Operationen eines von Neumann-Rechners:
  - CPU führt Maschinenbefehle aus
  - über den sog. Bus werden Befehle und Daten vom Speicher in die CPU übertragen und die Ergebnisse zurück übertragen
- imperative Programmiersprachen abstrahieren von diesen elementaren Operationen:
  - Anweisungen (engl.: statements) fassen Folgen von Maschinenbefehlen zusammen
  - Variablen (engl.: variables) abstrahieren vom physischen Speicherplatz

Eine Anweisung – mehrere Maschinenbefehle

- Die Java-Anweisung:

```
zaehler = zaehler + 1;
```

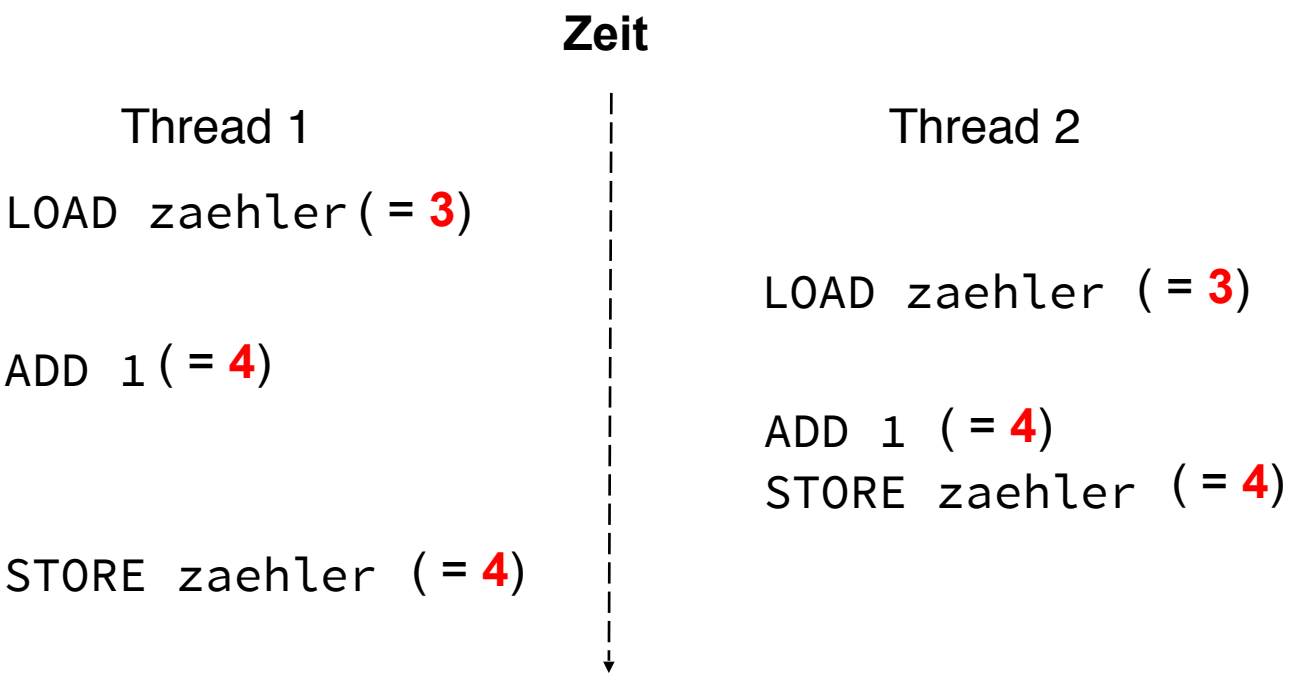
  wird (vereinfacht dargestellt) in mehrere Maschinenbefehle übersetzt:

```
LOAD _zaehler
ADD 1
STORE _zaehler
```

- Und warum ist das ein Problem?

Verzahnung von Threads

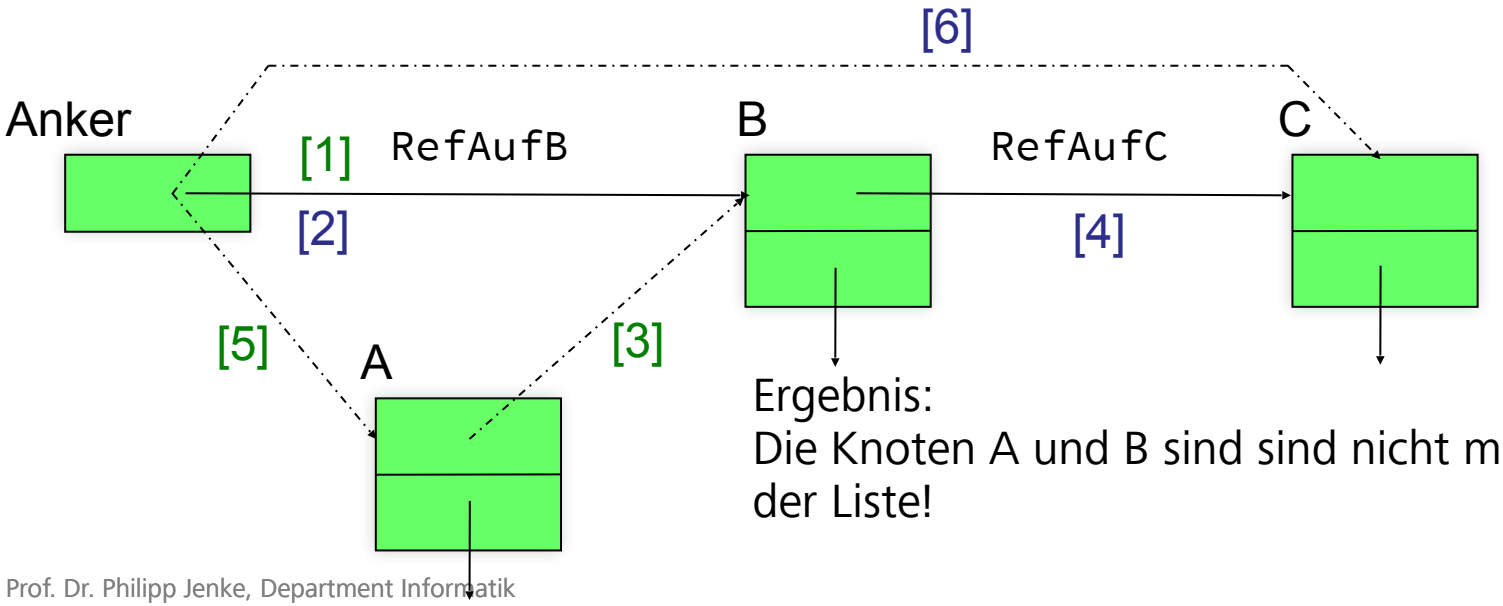
```
zaehler = zaehler + 1;
```



Ergebnis in `zaehler`: 4 statt 5!

Paralleler Zugriff auf verkettete Liste

- Thread 1: Einfügen von Knoten A
  - [1] Lesen des Ankers: RefAufB
  - [3] Setzen: NextRefA = RefAufB
  - [5] Setzen: Anker = RefAufA
- Thread 2: Entfernen von Knoten B
  - [2] Lesen des Ankers: RefAufB
  - [4] Lesen: NextRefB:RefAufC
  - [6] Setzen: Anker = RefAufC



Ergebnis:
Die Knoten A und B sind sind nicht mehr in der Liste!

Synchronisation von Prozessen/Threads

- Offensichtlich kann es zu Problemen kommen, wenn mehrere Prozesse/ Threads auf denselben Daten/Ressourcen arbeiten

- Wir müssen uns also ansehen, wie wir das nebenläufige Verhalten mehrerer Threads geeignet synchronisieren können

- Durch die Synchronisation von Prozessen soll gewährleistet werden, dass diese auch bei Nebenläufigkeit korrekt arbeiten

- Aber was heißt nochmal korrekt?

Zitate „Concurrent Java“ von B. Goetz

- Correctness
  - „Correctness means that a class conforms to its specification. A good specification defines invariants constraining an object‘s state and postconditions describing the effects of operations.“

- Thread-Safe
  - „A class ist thread-safe when it continues to behave correctly when accessed from multiple threads.“

- „When designing thread-safe classes, good object-oriented techniques – encapsulation, immutability and clear specifications of invariants – are your best friends.“

Mechanismen zur Synchronisation

- Wir betrachten im folgenden die Mechanismen zur Synchronisation paralleler Prozesse mit gemeinsamem Speicher auf drei Ebenen:
    - Auf der Maschinen-Ebene (Hardware)
    - Als Dienstleistungen des Betriebssystems
    - Auf Ebene einer Programmiersprache
- Minimale Voraussetzung:
    - Normalerweise kann man davon ausgehen, dass Lade- und Speicherbefehle unteilbar sind.

Kritischer Abschnitt

Thread-Synchronisation

- Thread-Synchronisation
    - Herstellen einer zeitlichen Reihenfolge zwischen parallel ablaufenden Threads
- grundsätzliche Probleme
    - Zugriff auf gemeinsam benutzte Objekte
        - Wechselseitiger Ausschluss
    - Einhalten von notwendigen Reihenfolgebedingungen
        - Ablaufsteuerung durch Reihenfolgebeschränkungen

Kritischer Abschnitt

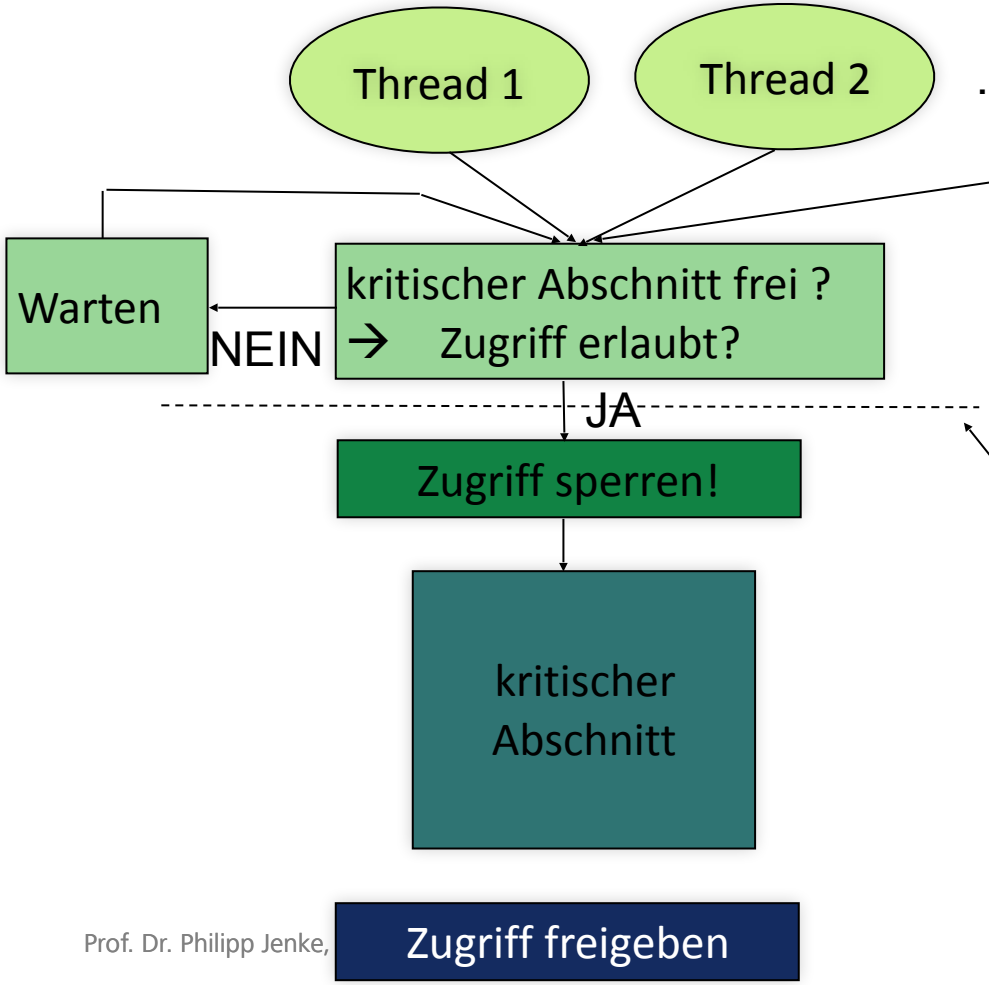
- Nebenläufigkeit führt zu der Notwendigkeit, Zugriff auf gemeinsam verwendete Objekte zu reglementieren
- Bereiche in denen Konflikte durch parallelen Zugriff auftreten können nennt man "Kritische Abschnitte" (engl. critical sections)
    - problematisch nur, wenn dort der Zustand verändert werden kann
    - kein Problem bei unveränderlichen Objekten
- Konsequenz
    - Befehlsfolgen in kritischen Abschnitten dürfen nicht unterbrochen werden
    - nur ein Thread zur Zeit darf einen kritischen Abschnitt betreten
        - z.B. Veränderung einer Liste oder eines Zählers

Lösung: Wechselseitiger Auschluss

- engl. mutual exclusion
- Anforderung, dass keine zwei Prozesse oder Threads parallel eine kritische Sektion betreten
- Problemstellung wurde 1965 von Edsger W. Dijkstra beschrieben [3]

Thread-Synchronisation

- Allgemeine Synchronisationslösung für wechselseitigen Ausschluss



Problem beim Abfragen / Sperren des Zugriffs:

- Ein Programm kann nach **jedem** Maschinenbefehl unterbrochen werden
- Unterbrechung zwischen Abfragen und Sperren führt evtl. zu Fehlern → ist eigener kritischer Abschnitt!

Aktives Warten

- eigene Lösung für wechselseitigen Ausschluss
- engl. busy waiting
    - ein Thread prüft ständig, ob er einen kritischen Abschnitt betreten darf
    - z.B. in einer “while”-Schleife
    - Beispiel:

```
while (istBesetzt) {
}; // Warten (leerer Block)
istBesetzt = true; // Selbst Sperre setzen
... // Code für kritischen Abschnitt
istBesetzt = false; // Sperre freigeben
```

Beispiel

- zwei Threads, die gemeinsame Zählervariable verändern
- eigentlich: +1
- Vorgehen
    - +0.5
    - Pause
    - +0.5
    - Ausgabe
- Wunsch: Ausgabe immer ganzzahlig
- aber: klappt nicht
    - immer wieder x.5-Werte

Beispiel

```
public class Zaehler {
  private double zaehler = 0.0;
  public void inkrement() {
    zaehler = zaehler + 0.5;
    try {
      Thread.sleep((int) (10 *
          Math.random()));
    } catch (InterruptedException e) {
    }
    zaehler = zaehler + 0.5;
    System.err.println(
        "Aktueller Zählerwert (" +
        Thread.currentThread().getName()
        + "): " + zaehler + " (keine
        Synchronisation)");
  }
  public double getZaehlerStand() {
    return zaehler;
  }
}
public static void main(String[] args) {
  Zaehler zaehler = new Zaehler();
  new VeraendererThread(zaehler).start();
  new VeraendererThread(zaehler).start();
}
```

```
public class VeraendererThread extends
    Thread {
  private final Zaehler zaehler;
  public VeraendererThread(
      Zaehler zaehler) {
    this.zaehler = zaehler;
  }

  @Override
  public void run() {
    while (zaehler.getZaehlerStand()
        < 100.0) {
      zaehler.inkrement();
    }
  }
}
```

Übung: Aktives Warten

- Verändern Sie den Beispielcode so, dass aktives Warten verwendet wird
- Die Ausgabe des Zählerstandes soll dann immer ganzzahlig sein.

```
public class Zaehler {
  private double zaehler = 0.0;
  public void inkrement() {
    zaehler = zaehler + 0.5;
    try {
      Thread.sleep((int) (10 *
          Math.random()));
    } catch (InterruptedException e) {
    }
    zaehler = zaehler + 0.5;
    System.err.println(
        "Aktueller Zählerwert (" +
        Thread.currentThread().getName()
        + "): " + zaehler + " (keine
        Synchronisation)");
  }
  public double getZaehlerStand() {
    return zaehler;
  }
}
```

Aktives Warten

- Probleme
  - mögliche Unterbrechung zwischen Abfrage und Sperren
  - Synchronisation klappt manchmal nicht!
  - eine saubere Programmierlösung mit aktivem Warten ist aufwändig
  - in der Praxis häufig: zweiter Thread kommt gar nicht zum Zug
  - wartender Thread (in while-Schleife) verbraucht CPU-Zeit!

- also: Aktives Warten ist keine Lösung!

Monitor-Mechanismus

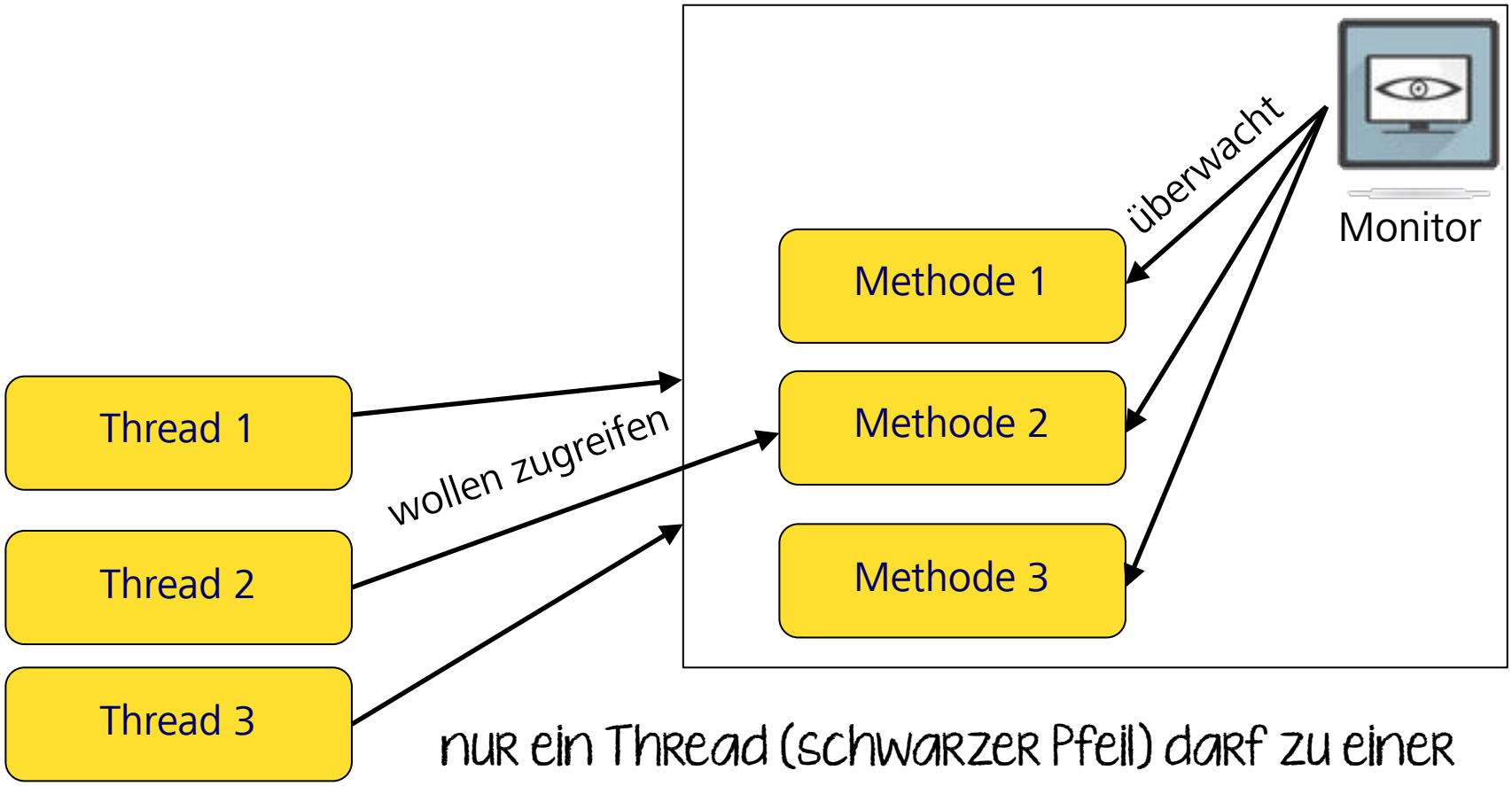
Synchronisation in Java

- Java bietet verschiedene Mechanismen zur Synchronisation
    - Monitor
    - Semaphore

Monitor-Mechanismus

- Ein Monitor überwacht den Aufruf bestimmter Methoden
- Ein Thread „betritt“ den überwachten Monitorbereich durch den Aufruf einer der Methoden und „verlässt“ ihn mit dem Ende dieser Methode
- Nur ein Thread zur Zeit kann sich innerhalb der überwachten Monitor-Methoden aufhalten (Sperre des kritischen Abschnitts), die übrigen müssen warten
- Die wartenden Threads werden von dem Monitor in einer Monitor-Warteschlange verwaltet (sind blockiert)
- Es gibt zusätzliche Synchronisationsfunktionen innerhalb des Monitors
- falls ein Monitor mehrere Methoden überwacht, darf nur eine zur Zeit betreten werden

Monitor-Mechanismus



nur ein Thread (schwarzer Pfeil) darf zu einer Zeit den überwachten Bereich betreten, alle anderen müssen warten (graue Pfeile)

Monitore in Java

- jedes Java-Objekt besitzt einen eigenen Monitor
  - ist sein eigener Monitor
- falls Monitorbereich eines Objektes gesperrt
  - kein anderer Thread eine synchronisierte Methode dieses Objektes ausführen
  - ab in die Monitor-Warteschlange!
  - jede unsynchronisierte Methode lässt sich dagegen ausführen!

Synchronized

- Monitor eines Objekts überwacht alle Methoden / Blöcke des Objekts, die mit synchronized bezeichnet sind
- Eintritt in den Monitorbereich über Aufruf irgendeiner synchronized – Methode des Objekts
- Eintritt in eine synchronized-Methode
    - Monitorbereich des Objekts für andere Threads gesperrt
    - nach dem Austritt wieder freigegeben
- Syntax:

```
<Sichtbarkeit> synchronized <Rückgabetyp> <Bezeichner>
(<Paramater>) {
      ...
}
```

- Beispiel:

```
public synchronized void erhoeheZaehler() { ... }
```

Beispiel

```
public synchronized void inkrement() {
  zaehler = zaehler + 0.5;
  try {
    Thread.sleep((int) (10 * Math.random()));
  } catch (InterruptedException e) {
  }
  zaehler = zaehler + 0.5;
  System.err.println("Aktueller Zählerwert (" +
      Thread.currentThread().getName() + "): " + zaehler
      + " (Monitor zur Synchronisation)");
}
```

Geschachtelte Aufrufe

- Frage: Kann man aus einer synchronized-Methode heraus eine andere, ebenfalls als synchronized gekennzeichnete Methode des gleichen Objekts aufrufen?
- Beispiel:

```
class Termin {
  synchronized void aendereTermin(...) {
    ...  schreibeTermin(...);    ...
  }
  synchronized void schreibeTermin(...){
    ...
  }
}
```

Synchronisations-Varianten

- Synchronisation von Methoden einer Klasse
  - Schlüsselwort synchronized im Methodenkopf angeben
  - Wirkung ist identisch mit synchronized(this) { ... } am Anfang der Methode
- Synchronisation von Blöcken
  - Es können beliebige Code-Blöcke synchronisiert werden
  - Angabe eines Synchronisationsobjekts nötig
  - Syntax: synchronized (<Synchr.-Objekt>) { ... }
    - meist getClass() als Monitor verwendet
- Synchronisation über Klassen
  - Wie Objekte, besitzt auch jede Klasse genau einen Monitor
  - Eine Klassenmethode, die die Attribute static synchronized trägt, fordert somit den Monitor der Klasse an

Beispiel

```
public void inkrement() {
  synchronized (this) {
    zaehler = zaehler + 0.5;
    try {
      Thread.sleep((int) (10 * Math.random()));
    } catch (InterruptedException e) {
    }
    zaehler = zaehler + 0.5;
    System.err.println("Aktueller Zählerwert (" +
      Thread.currentThread().getName() + "): " + zaehler
        + " (Block-Synchronisation mit Monitor)");
  }
}
```

Übung: Synchronisierter Fußball

- Identifizieren Sie die kritische Methode in der Fußballsimulation.
- Verändern Sie den Quellcode so, dass der kritische Bereich synchronisiert wird
  - also nur von einem Thread gleichzeitig besucht wird

```
public class Spieler extends Thread {
  private final Keeper keeper;
   public Spieler(Keeper keeper, String name) {
    super(name);
    this.keeper = keeper;
  }

  @Override
  public void run() {
    for (int i = 0; i < 10; i++) {
      keeper.score();
      System.err.println(getName() +
        " hat ein Tor geschossen.");
    }
  }
}

public class Keeper {
  protected int anzahlTore = 0;
  public void score() {
    anzahlTore++;
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Reihenfolge-beschränkungen

Synchronisation im Monitorbereich

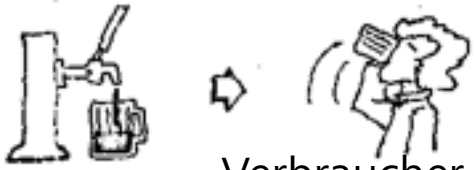
- bisher gelöst: Wechselseitiger Ausschluss
- neue Problemstellung: Einhalten von Reihenfolgebedingungen
- Ein Thread X befindet sich im kritischen Abschnitt
    - in einem synchronized-Block/ -Methode
    - im Monitorbereich
- Er kann den kritischen Abschnitt erst verlassen, nachdem ein anderer Thread im selben kritischen Abschnitt ein Ereignis ausgelöst hat

Beispiel: Erzeuger-Verbraucher-Problem

- ein oder mehrere Erzeuger-Threads generieren einzelne Datenpakete und speichern diese in einem Puffer
- ein oder mehrere Verbraucher-Threads entnehmen einzelne Datenpakete aus dem Puffer und verbrauchen diese
- Zu jedem Zeitpunkt darf nur ein Thread (Erzeuger oder Verbraucher) auf den Puffer zugreifen
    - Kritischer Abschnitt
- Erzeuger-Threads müssen auf einen Verbraucher warten, wenn der Puffer voll ist
- Verbraucher-Threads müssen auf einen Erzeuger warten, wenn der Puffer leer ist

Beispiel

- dann
  - Zapfhahn füllt nur, wenn mindestens ein Glas leer
  - Gast trinkt nur, wenn mindestens ein Glas voll

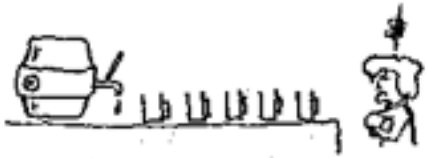


Erzeuger = Zapfhahn

Verbraucher = Gast

Idee: beschränkte Ressourcen: 5 Gläser

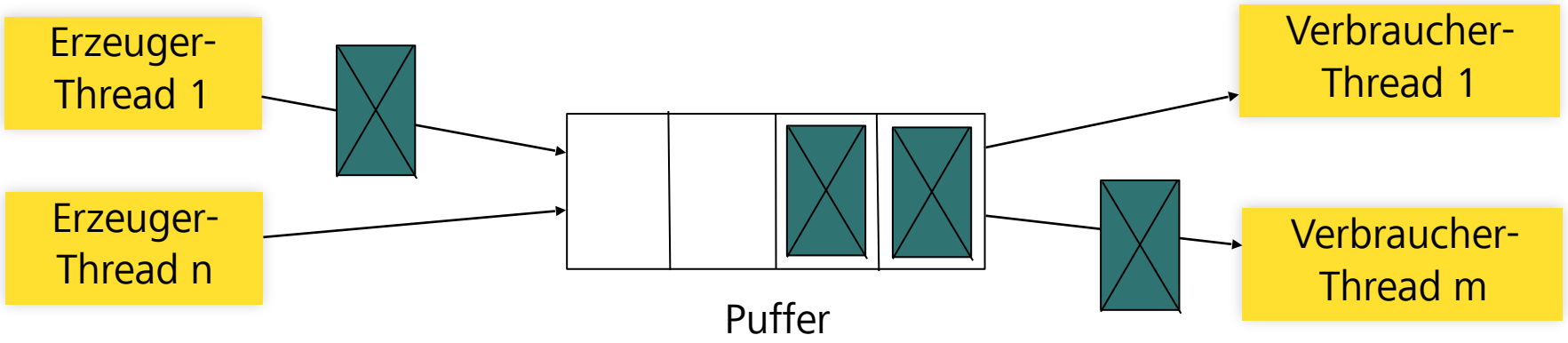
Produktion zu schnell: Stress für den Gast!



Produktion zu langsam: Gast verdurstet!

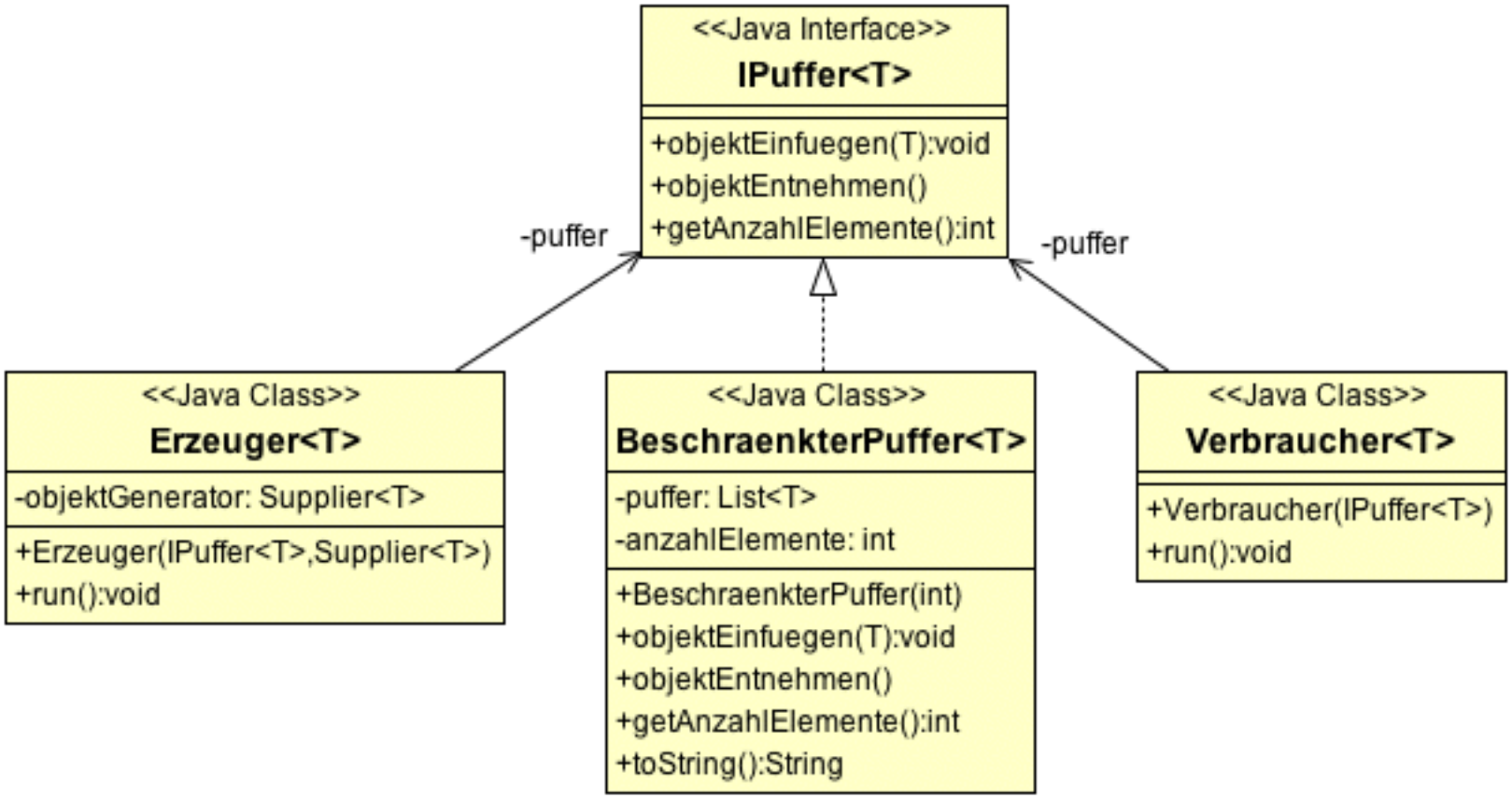
oder: Gast entfernt halbvolles Glas!

Erzeuger-Verbraucher-Problem



Umsetzung

- Wir versuchen, das Problem programmatisch zu lösen



Puffer

- Puffer kann Elemente aufnehmen
- und wieder abgeben
- gemeinsames Interface: `Puffer`

```
public interface IPuffer<T> {

  public void objektEinfuegen(T objekt);

  public T objektEntnehmen();

  public int getAnzahlElemente();

}
```

Erzeuger

- Erzeuger ist ein `Thread`
- fügt ein Objekt (generischer Typ `T`) in einen Puffer ein
- Objekterzeugung über Lambda-Ausdruck (SAM `Supplier<T>`

```
public class Erzeuger<T> extends Thread {
  private IPuffer<T> puffer;
  private final Supplier<T> objektGenerator;
  public Erzeuger(IPuffer<T> puffer, Supplier<T> objektGenerator) {
    this.puffer = puffer;
    this.objektGenerator = objektGenerator;
  }
  @Override
  public void run() {
    T objekt = objektGenerator.get();
    puffer.objektEinfuegen(objekt);
    System.err.println("Erzeuger hat Objekt " + objekt
        + " erzeugt und in den Puffer gelegt. ");
  }
}
```

Übung: Verbraucher

- Verbraucher
  - ist ein Thread
  - nimmt ein Element aus einem Puffer (T objektEntnehmen())
  - gibt Objekt auf Konsole aus (durch toString())
- Schreiben Sie die Klasse Verbraucher

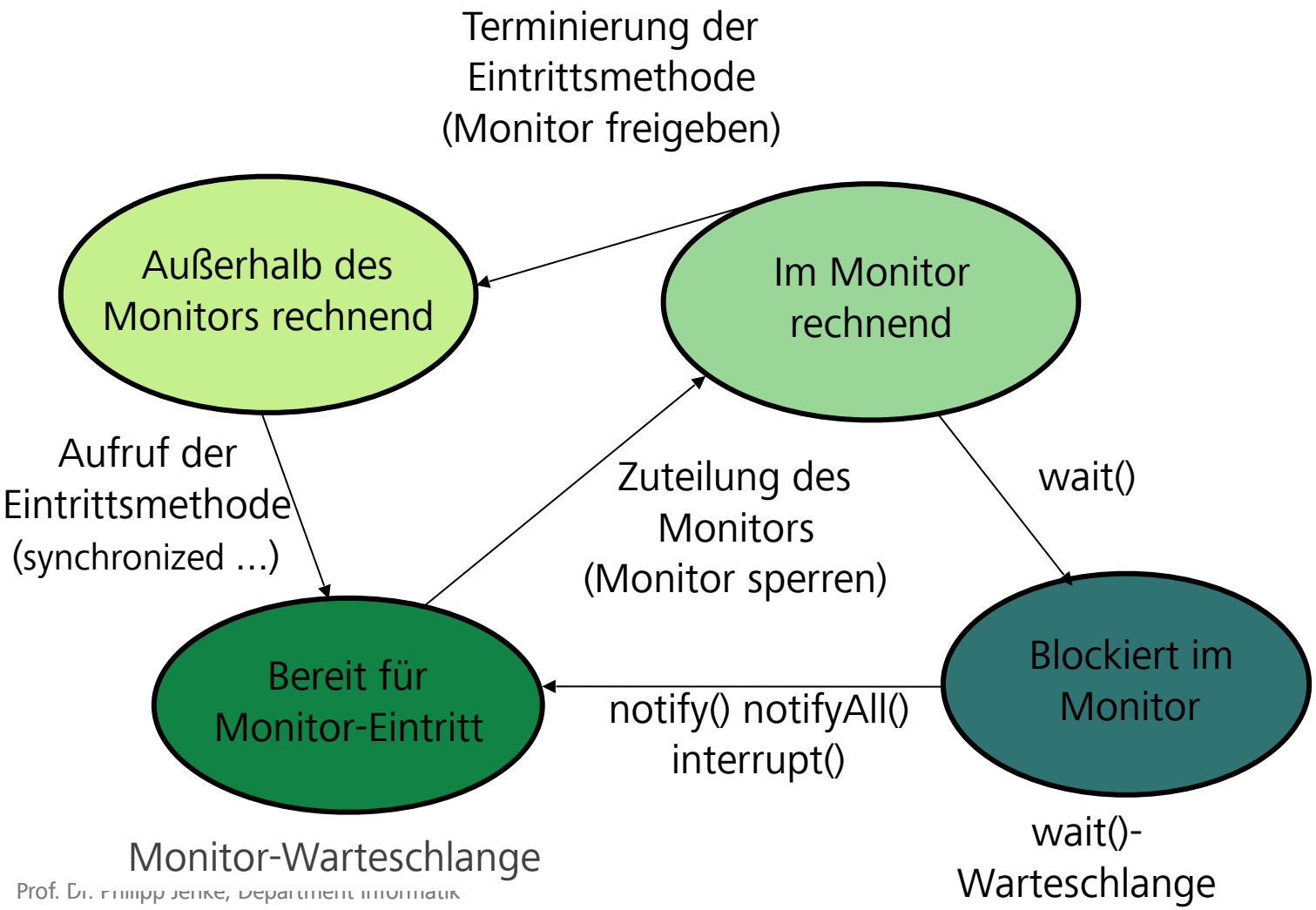
Mechanismen zur Reihenfolgensteuerung

- Ziel
  - nur ein Element einfügen, wenn Puffer nicht voll
  - nur ein Element entfernen, wenn mindestens eins im Puffer ist
- Umsetzung mit Threads (Wunsch)
  - Einfügen und Entfernen synchronisiert
- dann
  - Methode betreten
  - Prüfen, ob Bedingung erfüllt
  - falls ja: machen, Methode verlassen
  - falls nein: Warten (Parken), Monitor freigeben, später zurückkommen und wieder Bedingung prüfen

wait() und notify()

- Thread parken: wait()
  - Methode nicht weiter bearbeiten
  - Monitor freigeben für anderer Threads
  - Thread in einer Warteschlange parken
- Threads aus der Warteschlange zurückholen: notifyAll()
  - ein Thread aus der Warteschlange holen
  - Monitor zugriff erteilen
  - Thread darf weitermachen, wo er zuvor geparkt wurde

Thread-Zustandsdiagramm



wait() und notify()

- Monitor freigeben und in zusätzlicher wait()-Warteschlange warten
  - kann eine InterruptedException werfen

```
wait()
```

- einen (zufälligen) Thread in der wait()-Warteschlange wecken

```
notify()
```

- alle Threads in wait()-Warteschlange wecken

```
notifyAll()
```

- Der Aufruf dieser Methoden muss aus dem Monitor heraus erfolgen
  - innerhalb einer synchronized-Methode

Beschränkter Puffer

- Umsetzung in beschränktem Puffer
- Sicherstellung der Reihenfolge-Anforderungen

```
public class BeschraenkterPuffer<T> implements IPuffer<T> {

  private final List<T> puffer;

  private int anzahlElemente = 0;

  public BeschraenkterPuffer(int pufferGroesse) {
    puffer = new ArrayList<T>();
    anzahlElemente = 0;
    for (int i = 0; i < pufferGroesse; i++) {
      puffer.add(null);
    }

  }

  ...
```

Puffer-Methode: Objekt einfügen

```
@Override
public synchronized void objektEinfuegen(T objekt) {
  while (anzahlElemente == puffer.size()) {
    try {
      this.wait();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      return;
    }
  }
  puffer.set(anzahlElemente, objekt);
  anzahlElemente++;
  try {
    Thread.sleep(500);
  } catch (InterruptedException e) {
  }
  System.err.println("---\nNeuer Pufferinhalt: " + this);
  this.notifyAll();
}
```

solange "Puffer voll"

einfügenden Thread parken

erst wenn Platz im Puffer

Element einfügen

geparkte Threads aufwecken

Übung: Objekt aus Puffer entnehmen

- Implementieren Sie die Methode T objektEntnehmen() für die Klasse BeschraenkterPuffer:
  - Warten bis mindestens ein Element vorhanden
  - Element entnehmen
  - geparkte Threads informieren
  - Element zurückgeben

Weiteres Beispiel

- Zwei Zähler zählen abwechseln gemeinsamen Wert hoch

```
public class ZaehlerAbwechselnd extends Thread {
    private static double zaehler = 0.0;
    @Override
    public void run() {
        synchronized (getClass()) {

            while (zaehler < 1000.0) {
                zaehler = zaehler + 0.5;
                try {
                    Thread.sleep((int) (10 * Math.random()));
                } catch (InterruptedException e) {
                    return;
                }
                zaehler = zaehler + 0.5;
                System.err.format("%s: %.1f\n", Thread.currentThread()
                        .getName() + ": ", zaehler);
                getClass().notify();
                try {
                    getClass().wait();
                } catch (InterruptedException e) {
                    return;
                }
            }
            getClass().notify();
        }
    }
}
```

Ausgabe:

Thread-0: : 1,0
Thread-1: : 2,0
Thread-0: : 3,0
Thread-1: : 4,0
Thread-0: : 5,0
Thread-1: : 6,0
Thread-0: : 7,0
...
Thread-1: : 18,0
Thread-0: : 19,0
Thread-1: : 20,0

Deadlocks

Deadlocks

- mehrere Threads hängen voneinander ab
- es kann Situation entstehen in der kein Thread weitermachen kann
  - weil er auf einen anderen Thread wartet
- Deadlock!

Quelle: [4]

Beispiel: Philosophen-Problem

- fünf Philosophen
    - entweder denken oder essen
- fünf Gabeln je zwischen zwei Philosophen
- zum Essen zwei Gabeln benötigt

Quelle: [5]

Beispiel: Philosophen-Problem

- Philosoph = Thread
- entweder denken (= Warten)
- oder Essen
  - linke Gabel aufnehmen
  - rechte Gabel aufnehmen
  - Warten
  - linke Gabel zurücklegen
  - rechte Gabel zurücklegen
- möglicher Deadlock
  - jeder Philosoph hat eine Gabel aufgenommen
  - wartet, dass er zweite Gabel aufnehmen kann

- Auszug aus Gabel:

```
public synchronized void nimmAuf(
    Philosoph philosoph) {
  while (hatGabel != null) {
    try {
      wait();
    } catch (InterruptedException e) {
    }
  }
  System.err.println("Philosoph "
      + Thread.currentThread().getName()
      + " nimmt " + name + " auf");
  hatGabel = philosoph;
}

public synchronized void legeZurueck() {
  System.err.println("Philosoph " +
  Thread.currentThread().getName()
    + " legt Gabel " + name + " zurück.");
  hatGabel = null;
  notifyAll();
}
```

Zusammenfassung

- Probleme paralleler Programmausführung
- Kritischer Abschnitt
- Monitor-Mechanismus
- Reihenfolge-Beschränkungen
- Deadlocks

Quellen

- Die Folien basieren zum großen Teil auf den Folien von Prof. Dr. Martin Hübner, Hochschule für Angewandte Wissenschaften Hamburg und folgendem Buch: Elisabeth Freeman, Eric Freeman, Kathy Sierra, Bert Bates: *Head First Design Patterns*, O'Reilly Media, 2004
- [1] Valerijs Kostreckis, *123rf.com/*, Bild-Nummer : 14007058, abgerufen: 24.10.2013
- [2] Wikipedia: Mutual Exclusion: *http://en.wikipedia.org/wiki/Mutual_exclusion*, abgerufen am 31.10.2013
- [3] Dijkstra, E. W.: *"Solution of a problem in concurrent programming control"*. Communications of the ACM 8 (9): 569
- [4] Christian Ullenboom: Java ist auch eine Insel, Galileo Computing, ISBN 978-3-8362-1506-0
- [5] Wikipedia: Philosophenproblem, *http://de.wikipedia.org/wiki/Philosophenproblem*, abgerufen am 22.3.2014
- [6] Michael Vigneau, http://www.ccs.neu.edu/home/kenb/synchronize.html, abgerufen am 19.06.2015 (Texte überarbeitet)