Universität Bremen

# Praktische Informatik 1
# Nebenläufigkeit


Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

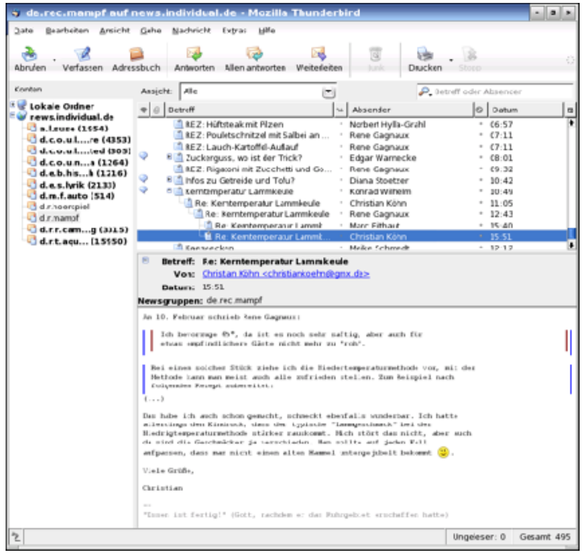


Bild: ©Michael Ihle

# Nebenläufigkeit: Motivation

**GUI**

E-Mail lesen




1.

E-Mail-Client

2.

**Netzwerk**

Nachrichten abrufen/senden

1. und 2. dürfen sich nicht gegenseitig blockieren

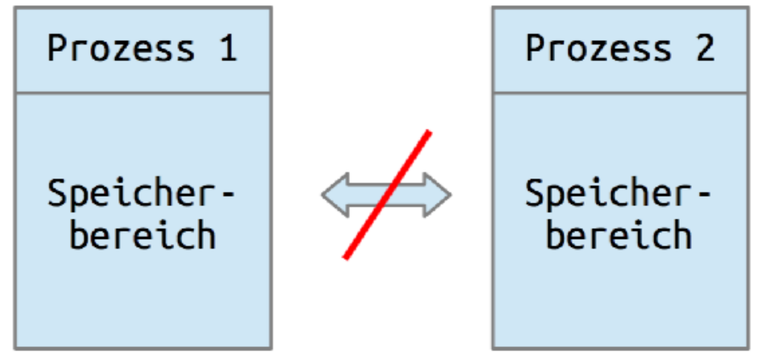
Universität Bremen

# Nebenläufigkeit

- Moderne Betriebssysteme
  - Illusion parallel ausgeführter Programme
  - Mischung aus **quasi-parallel** und **real-parallel** (bei Multi-Core-/Multi-CPU-Systemen)
  - **Scheduler**: verzahnte Abarbeitung durch Umschaltung
- Pro ausgeführtem Programm mindestens ein **Thread**
  - Scheduler schaltet also Threads um
- Multithreading: mehrere Threads pro ausgeführtem Programm

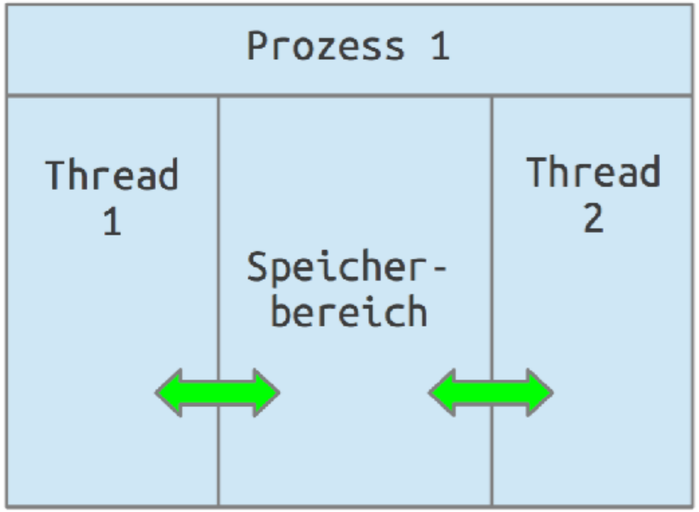
# Prozess

- Programm in Ausführung
  - plus Daten und Ressourcen (offene Dateien, …)
- Prozesse laufen parallel zueinander
- Jeder Prozess hat eigenen Speicherbereich
- Betriebssystem schützt Speicherbereiche: Allgemeine Schutzverletzung/ Segmentation Fault
- Betriebssystem gibt nach Ende Ressourcen wieder frei



# Thread („Ausführungsfaden“)

- Mehrere Threads pro Prozess, einzeln vom Scheduler des Betriebssystems kontrolliert
- Laufen parallel zueinander
- Teilen sich Speicherbereich des Prozesses



Universität Bremen

# Scheduler

- Teilt Prozessen/Threads Rechenzeit zu
- Speichert Zustand und stellt ihn wieder her
- Prioritäten
  - Höhere Priorität: Bevorzugte Zuteilung von Rechenzeit
  - Unterbrechung niedrig priorisierter Prozesse/Threads für höher priorisierte
- Designregel
  - Höhere Priorität: Möglichst geringe Rechenzeit in Anspruch nehmen
  - Sonst: Verhungern der anderen Prozesse/Threads

Universität Bremen

# Multitasking

- Programmteile reagieren auf externe Ereignisse
  - Mausklick, Lesen von Festplatte, Netzwerkdaten …
- Varianten
  - **Polling**: Jedes Ereignis reihum auf den Eintritt testen und behandeln
  - **select()-Operation**: Dem Betriebssystem mitteilen, auf welche Menge von Ereignissen der Prozess wartet: Prozess schläft, bis irgendeines dieser Ereignisse eingetreten ist
  - **Multithreading**: Jedes Ereignis durch einen separaten Thread auf Eintritt überwachen: Thread „schläft“ bis zum nächsten Eintritt des Ereignisses

Universität Bremen

# Threads in Java

Von **Thread** erben

```
class C extends Thread
{
    C()
    {
        start();
    }

    public void run()
    { // Läuft parallel zum Rest
    }
}
```

**Runnable** implementieren

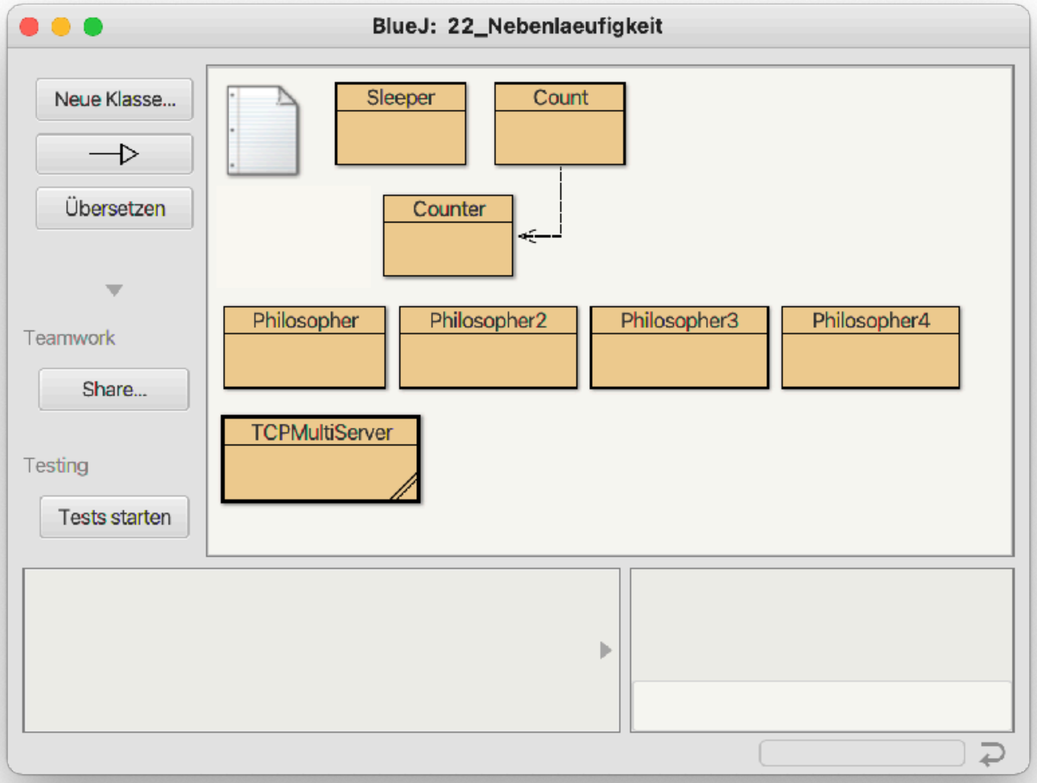
```
class C implements Runnable
{
    C()
    {
        new Thread(this).start();
    }

    public void run()
    { // Läuft parallel zum Rest
    }
}
```

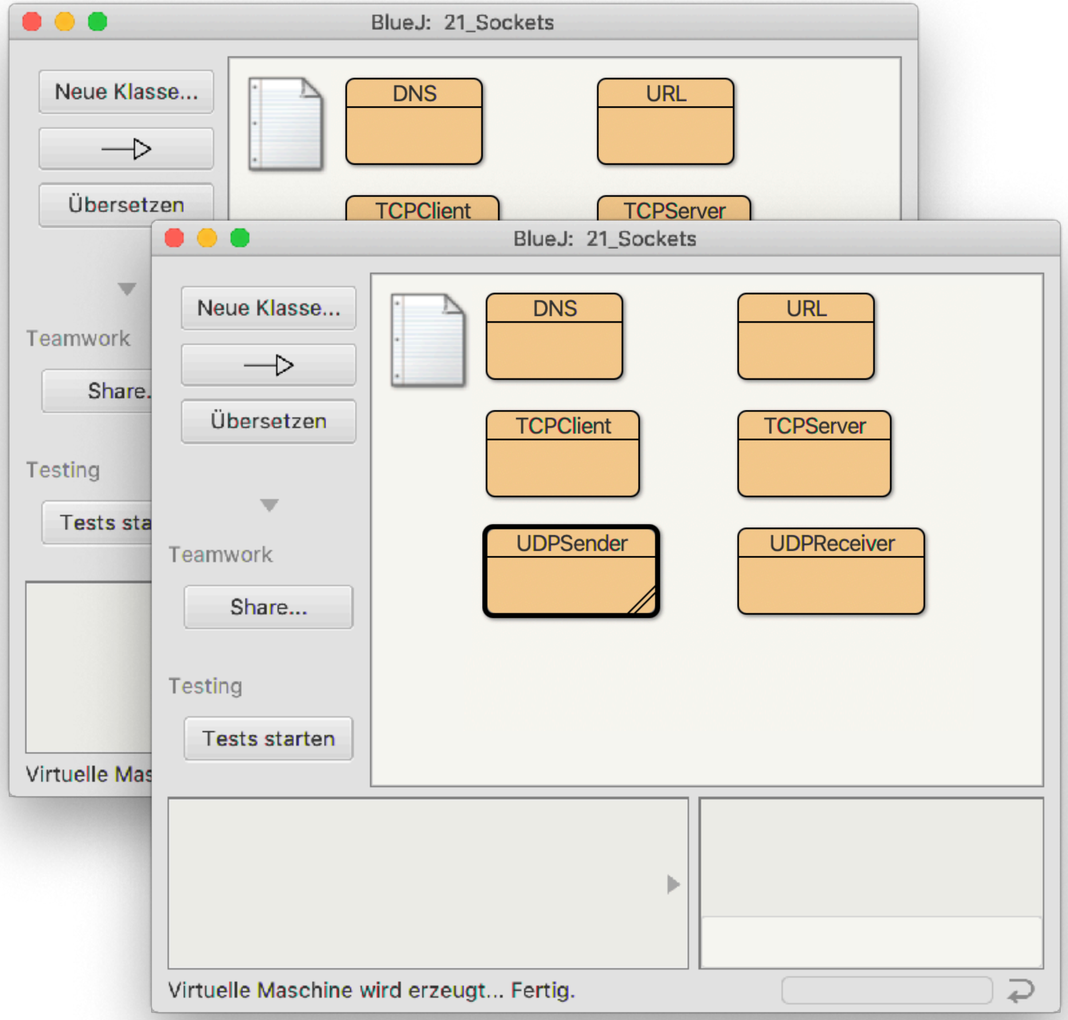
- **start()** führt **run()** in eigenem Thread aus

**Runnable** als Lambda-Ausdruck

```
new Thread(() -> {
        // Läuft parallel zum Rest
    }).start();
```

# TCPMultiServer: Demo



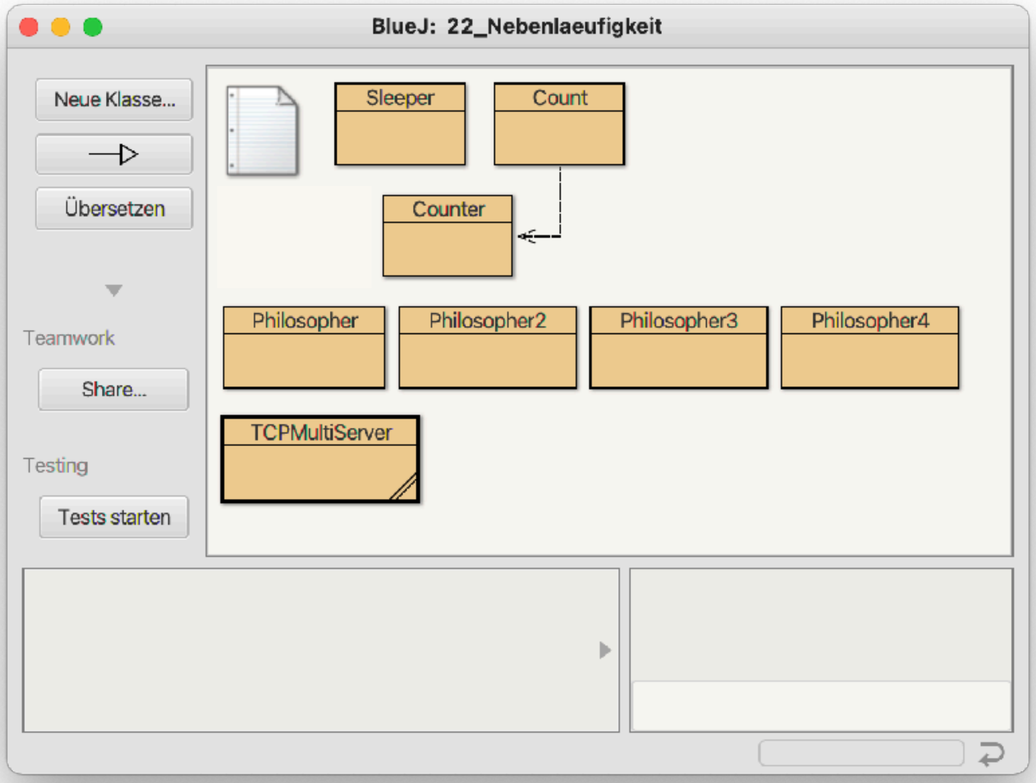
Universität Bremen

# Ende eines Threads

- **run()**-Methode wird fehlerfrei beendet (Endlosschleife → unendlich laufender Thread)
- **RuntimeException** in **run()**-Methode: Beendet nur betroffenen Thread
- Virtuelle Maschine wird beendet: Alle Threads werden beendet
- Abbruch von außen
  - **stop()**: Veraltet und sollte nicht benutzt werden („This method is inherently unsafe“)
  - **interrupt()**: Thread muss hierzu als Schleife **while (!isInterrupted()) {…}** laufen
  - **join()**: Warten, bis sich der Thread beendet hat

Universität Bremen

# Demo: Sleeper



Universität Bremen

# Threads ordnungsgemäß unterbrechen

- **interrupt()**: Setzt Unterbrechungssignal
- **isInterrupted()**: Fragt Signal ab
- **interrupted()**: Fragt Signal ab und löscht es wieder
- **sleep(ms)**: Legt Thread schlafen
  - **InterruptedException**: **interrupt()** hat Schlaf unterbrochen (Signal wurde wieder gelöscht)

```
try {
  Thread.sleep(2000); // 2 Sekunden
}
catch (final InterruptedException e) { //…
}
```

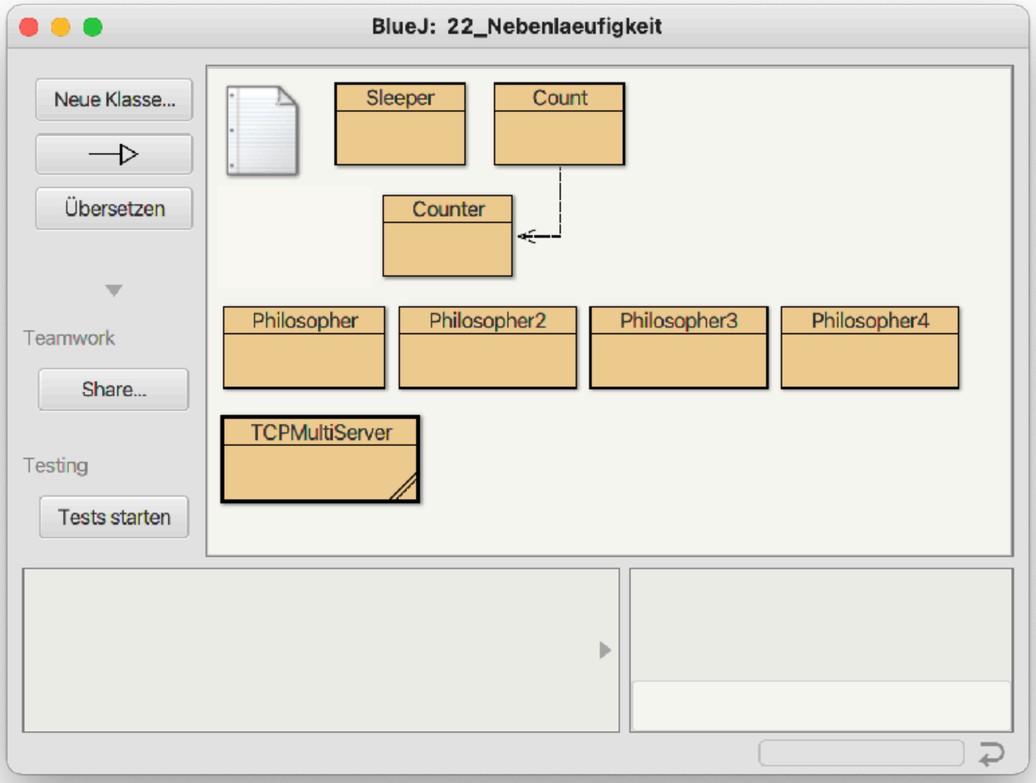
Universität Bremen

# Threads und Speicher

- Gemeinsam
  - Objekte
    - Nur relevant, wenn in mehreren Threads Referenzen auf sie bekannt sind
  - Klassenvariablen
- Pro Thread
  - Position im Programm: Aktuelle Anweisung, Aufruffolge der Methoden
  - Lokale Variablen und Parameter aller aufgerufenen Methoden

Universität Bremen

# Count: Demo



# Kritischer Abschnitt

```
public void inc()
{
    final int v = value;
    value = v + 1;
}
```

- Wann immer mehrere Threads (mindestens einer schreibend) auf dieselben Daten zugreifen, kann es zu Inkonsistenzen kommen

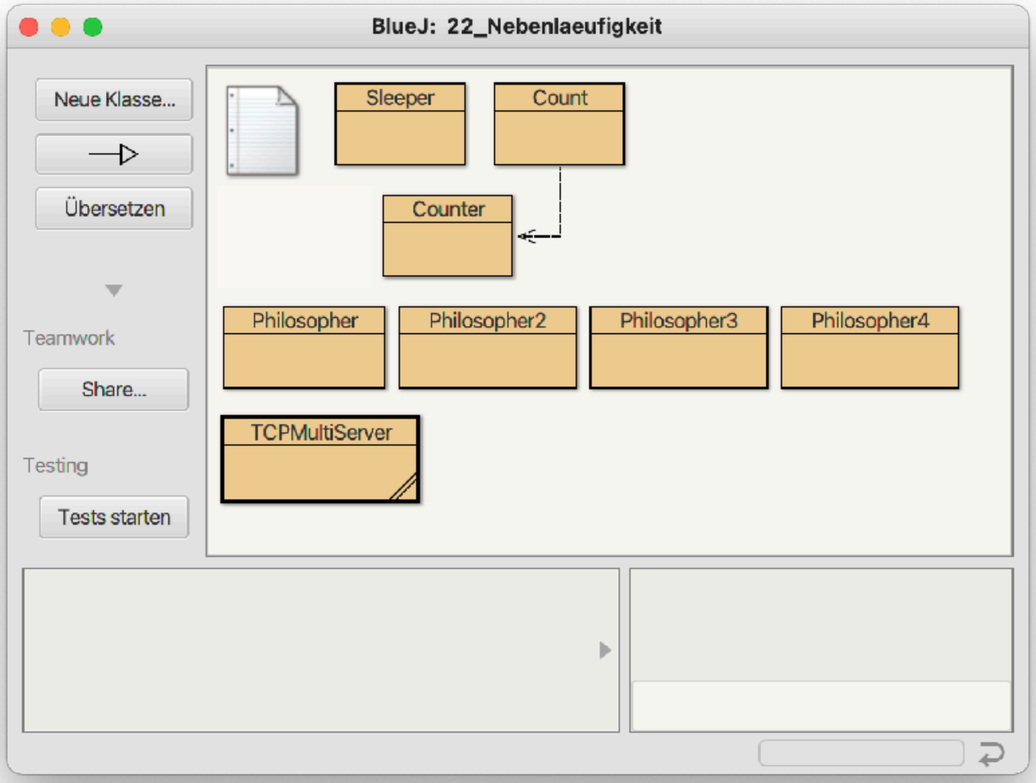
- Beispiel

| Zeitpunkt | Thread 1 | Thread 2 |
|---|---|---|
| 1 | v = value; // v = 0 | |
| 2 | | v = value; // v = 0 |
| 3 | | value = v + 1; // value = 1 |
| 4 | | v = value; // v = 1 |
| 5 | | value = v + 1; // value = 2 |
| 6 | value = v + 1; // value = 1 | |

- Klassen der Java-Laufzeitbibliothek können normalerweise **nicht** mit Nebenläufigkeit umgehen

Universität Bremen

# Count synchronisiert: Demo



Universität Bremen

# Monitor

- Monitor schützt kritische Abschnitte und verwendet dazu Lock (Schloss)
- Thread tritt in kritischen Abschnitt ein
  - Monitor wird „abgeschlossen“ (Thread bekommt Lock)
  - Andere Threads müssen warten
- Thread verlässt kritischen Abschnitt
  - Monitor wird „aufgeschlossen“
  - Nächster Thread darf eintreten

Universität Bremen

# Monitor: **synchronized**-Methoden

- Java-Klassen haben implizit Monitore
- Schlüsselwort **synchronized**
- komplette Methoden synchronisieren
  - Objektmethoden: **this**
  - Klassenmethoden: zugehöriges **Class**-Objekt
- Methode mit **synchronized**
  - Nur ein Thread zur Zeit (pro Objekt)

```
synchronized void inc()
{
    final int v = value;
    value = v + 1;
}
```

Universität Bremen

# Monitor: **synchronized**-Block

- Problem von **synchronized** Methoden
  - Ganze Methode muss abgearbeitet werden
  - Parallelität eingeschränkt
- Alternative
  - Bestimmte Anweisungen schützen
  - Durch Block einrahmen und Block synchronisieren
  - Beliebiges Objekt verwendbar (üblicherweise **this**)

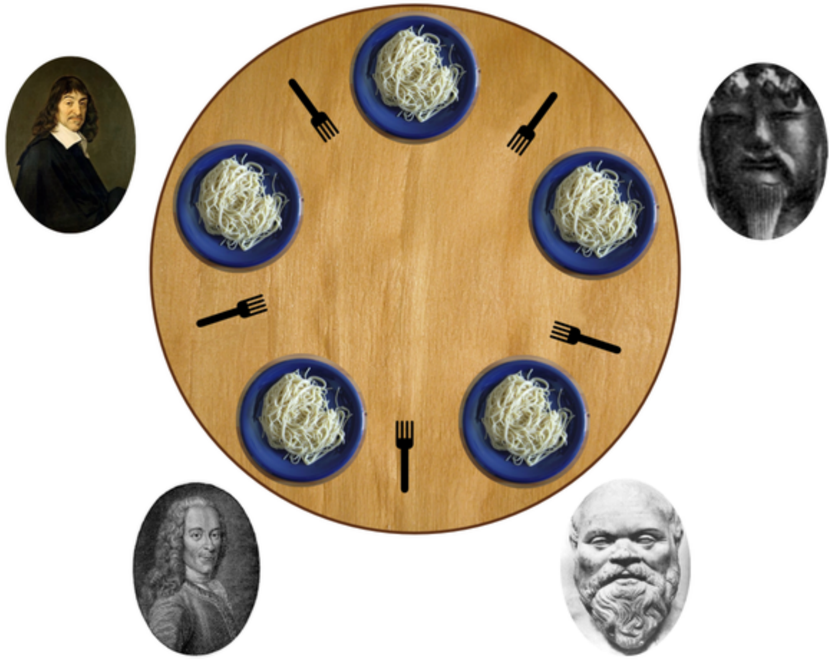
```
void inc()
{
    // …
    synchronized (this) {
        final int v = value;
        value = v + 1;
    }
    // …
}
```

# Deadlock: Demo

- Zwei oder mehr Threads warten gegenseitig darauf, dass der jeweils andere eine Ressource freigibt


© 2005 by Benjamin D. Esham / Wikimedia Commons

Speisende Philosophen



C. Ullenboom. Java ist auch eine Insel, 10. Auflage.

Universität Bremen

# Explizite Locks (Paket **java.util.concurrent.locks**)

- Schnittstelle **Lock** (implementiert z.B. durch **ReentrantLock**)
- **lock()**: Wartet, bis kritischer Abschnitt betretbar und betritt ihn dann
- **lockInterruptibly()**: Wie **lock()**, aber mit mit **interrupt()** unterbrechbar
- **tryLock()**: Wie **lock()**, wenn Abschnitt frei (Rückgabe **true**), kehrt sonst ohne Warten mit Rückgabe **false** zurück
- **tryLock(long time, TimeUnit unit)**: Wie **tryLock()**, wartet aber maximal angegebene Zeit und kann mit **interrupt()** unterbrochen werden
- **unlock()**: Verlässt kritischen Abschnitt (**lock()** ohne **unlock()** → Deadlock)

```
lock.lock();
try {
    //....
}
finally {
    lock.unlock();
}
```

Universität Bremen

# Synchronisierte Datenstrukturen

- Manche Datenstrukturen der Laufzeitbibliothek gibt es in einer synchronisierten Fassung, z.B. **StringBuilder** (unsynchronisiert) vs. **StringBuffer** (synchronisiert)

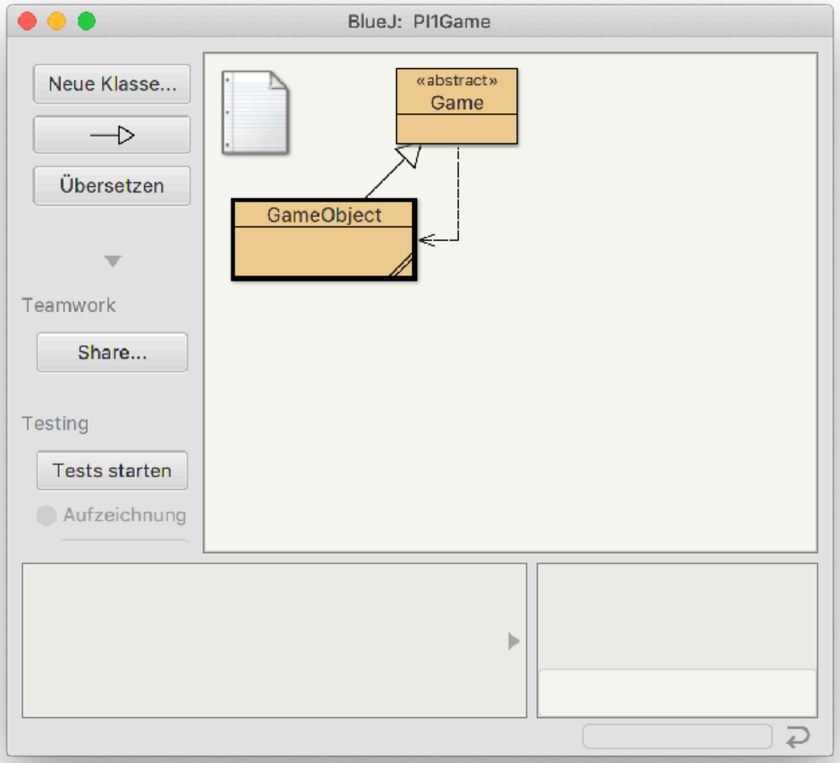
- Für manche Datenstrukturen gibt es synchronisierte Wrapper, z.B. **Collections.synchronizedMap(Map<K, V>)**

- Synchronisierung gilt aber immer nur für einzelne Methodenaufrufe, d.h. eine Folge von synchronisierten Methodenaufrufen ist als Ganzes nicht synchronisiert

```
final static Map<Integer, Integer> map =
        Collections.synchronizedMap(
        new TreeMap<Integer, Integer>()); // …
if (map.get(x) != null) {
    final int i = map.get(x); // könnte null liefern!
}
```

- Synchronisierung nur einsetzen, wenn notwendig (Laufzeit!)

# Bewachte Blöcke: Demo



Universität Bremen

# Bewachte Blöcke (Guarded Blocks)

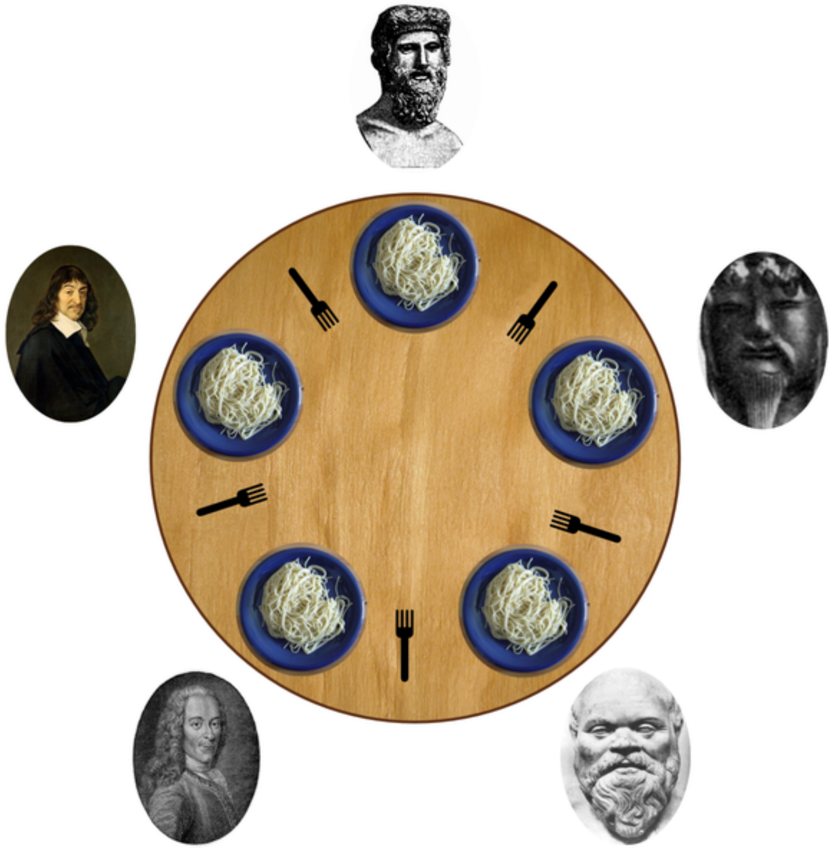
- Ausführung von Anweisungen ist von Daten abhängig, die ein anderer Thread liefert
- Zugriff auf Daten innerhalb eines durch einen Monitor abgesicherten kritischen Abschnitts
- **wait**: Wartet auf Benachrichtigung und entriegelt den kritischen Abschnitt während des Wartens
  - Es gibt auch Überladungen, die maximal eine bestimmte Zeit warten
- **notify**/**notifyAll**: Benachrichtigt einen/alle Threads, die gerade auf diesen Monitor warten

```
synchronized (keys) {
    keys.offer(event.getKeyCode());
    keys.notify();
}
```

```
try {
    synchronized (keys) {
        while (keys.isEmpty()) {
            keys.wait();
        }
        return keys.poll();
    }
}
catch (final InterruptedException e) {
    return KeyEvent.VK_ESCAPE;
}
```

# Zusammenfassung der Konzepte

- **Prozess** und **Thread**
- **Kritischer Abschnitt**
- **Monitor**
- **synchronized** und **Lock**
- **Deadlock**
- **wait**/**notify**

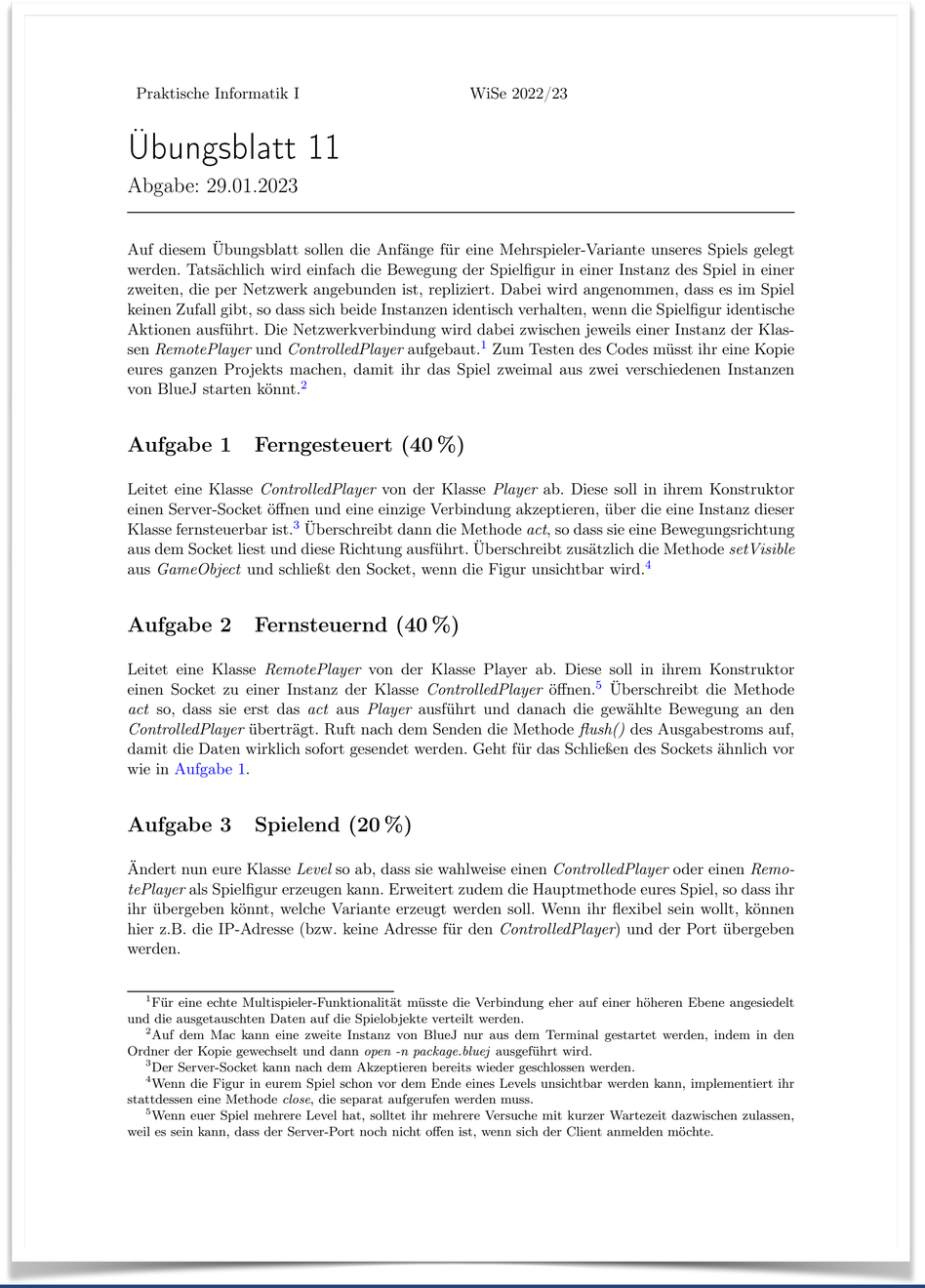


C. Ullenboom. Java ist auch eine Insel, 10. Auflage.

Universität Bremen

# Übungsblatt 11

- Aufgabe 1: Ferngesteuerte Spielfigur
- Aufgabe 2: Fernsteuernde Spielfigur
- Aufgabe 3: Figuren in Level integrieren
- Bonusaufgabe 4: Spiel veröffentlichen (bis 02.02, nachmittags)
- Aufgabe 5: Spiele ausprobieren und Likes vergeben (ab 02.02, abends)
  - Wahl des besten Spiels in letzter Vorlesung!



Praktische Informatik I WiSe 2022/23

## Übungsblatt 11

Abgabe: 29.01.2023

Auf diesem Übungsblatt sollen die Anfänge für eine Mehrspieler-Variante unseres Spiels gelegt werden. Tatsächlich wird einfach die Bewegung der Spielfigur in einer Instanz des Spiel in einer zweiten, die per Netzwerk angebunden ist, repliziert. Dabei wird angenommen, dass es im Spiel keinen Zufall gibt, so dass sich beide Instanzen identisch verhalten, wenn die Spielfigur identische Aktionen ausführt. Die Netzwerkverbindung wird dabei zwischen jeweils einer Instanz der Klassen *RemotePlayer* und *ControlledPlayer* aufgebaut.[1] Zum Testen des Codes müsst ihr eine Kopie eures ganzen Projekts machen, damit ihr das Spiel zweimal aus zwei verschiedenen Instanzen von BlueJ starten könnt.[2]

### Aufgabe 1 Ferngesteuert (40 %)

Leitet eine Klasse *ControlledPlayer* von der Klasse *Player* ab. Diese soll in ihrem Konstruktor einen Server-Socket öffnen und eine einzige Verbindung akzeptieren, über die eine Instanz dieser Klasse fernsteuerbar ist.[3] Überschreibt dann die Methode *act*, so dass sie eine Bewegungsrichtung aus dem Socket liest und diese Richtung ausführt. Überschreibt zusätzlich die Methode *setVisible* aus *GameObject* und schließt den Socket, wenn die Figur unsichtbar wird.[4]

### Aufgabe 2 Fernsteuernd (40 %)

Leitet eine Klasse *RemotePlayer* von der Klasse Player ab. Diese soll in ihrem Konstruktor einen Socket zu einer Instanz der Klasse *ControlledPlayer* öffnen.[5] Überschreibt die Methode *act* so, dass sie erst das *act* aus *Player* ausführt und danach die gewählte Bewegung an den *ControlledPlayer* überträgt. Ruft nach dem Senden die Methode *flush()* des Ausgabestroms auf, damit die Daten wirklich sofort gesendet werden. Geht für das Schließen des Sockets ähnlich vor wie in Aufgabe 1.

### Aufgabe 3 Spielend (20 %)

Ändert nun eure Klasse *Level* so ab, dass sie wahlweise einen *ControlledPlayer* oder einen *RemotePlayer* als Spielfigur erzeugen kann. Erweitert zudem die Hauptmethode eures Spiel, so dass ihr ihr übergeben könnt, welche Variante erzeugt werden soll. Wenn ihr flexibel sein wollt, können hier z.B. die IP-Adresse (bzw. keine Adresse für den *ControlledPlayer*) und der Port übergeben werden.

[1] Für eine echte Multispieler-Funktionalität müsste die Verbindung eher auf einer höheren Ebene angesiedelt und die ausgetauschten Daten auf die Spielobjekte verteilt werden.
[2] Auf dem Mac kann eine zweite Instanz von BlueJ nur aus dem Terminal gestartet werden, indem in den Ordner der Kopie gewechselt und dann *open -n package.bluej* ausgeführt wird.
[3] Der Server-Socket kann nach dem Akzeptieren bereits wieder geschlossen werden.
[4] Wenn die Figur in eurem Spiel schon vor dem Ende eines Levels unsichtbar werden kann, implementiert ihr stattdessen eine Methode *close*, die separat aufgerufen werden muss.
[5] Wenn euer Spiel mehrere Level hat, solltet ihr mehrere Versuche mit kurzer Wartezeit dazwischen zulassen, weil es sein kann, dass der Server-Port noch nicht offen ist, wenn sich der Client anmelden möchte.