HOCHSCHULE KONSTANZ
TECHNIK, WIRTSCHAFT UND GESTALTUNG

Verteilte Systeme

Kapitel 2: Threads

Prof. Dr. Rainer Mueller

Version: 6

Stand: 06.04.16

Version	Veröffentlicht
3	21.10.2015
4	16.03.2016
5	23.03.2016
6	06.04.2016

KAPITEL 2

Threads, Kooperation und Kommunikation

Übersicht

2.1.1 Eigenschaften

GRUNDLAGEN VON THREADS

- **Threads**: Leichtgewichtige Prozesse innerhalb „normaler“ (schwergewichtiger) Prozesse
  - Mehrere Ausführungsfäden (engl. Threads) innerhalb eines Prozesses
- Ausführung: Nebenläufig → Quasi parallel

- Kommunikation & Kooperation: Zugriff auf gemeinsame Daten/Objekte
  - Problem: Inkonsistenzen
  - Lösung: Synchronisierungsmechanismen
    - **Sperrobjekte**: Lock, Mutex (Mutally exclusive lock), Monitor, Spinlock, Semaphor
    - **Kritische Abschnitte** oder **atomare Operationen**: Synchronisierte Programmblöcke

THREADS IN PROGRAMMIERSPRACHEN

- Java und .NET sind inhärent multithreading-fähig
- Entwicklung und Synchronisierung von Threads mit Bordmitteln: Keine zusätzlichen Bibliotheken erforderlich
  - JVM und .NET CLR : Automatische Ausführung von Threads als nebenläufige Prozesse unabhängig von unterliegendem Betriebssystem
    → Ggf. Nutzung der nativen Thread-Realisierung des Betriebssystems

2.1.1 Eigenschaften

THREADS AUF BETRIEBSSYSTEMEBENE

- UNIX/LINUX
  - Ursprünglich keine Thread-Unterstützung
  - Lange Zeit keine Standard-API für Threads
  - Später **POSIX Threads** (Pthreads): API für Thread-Verwaltung (Erzeugen, Manipulieren, Kontrolle, Beenden)
    - IEEE-Standard: POSIX. 1c, Threads Extensions (IEEE Std 1003.1c-1995)
    - Verwendung in C- und C++-Programmen
    - Implementierungen auf verschiedensten Unix-Derivaten (inkl. Linux und Mac OS X)
      - Realisierung sehr unterschiedlich
    - Implementierung unter Windows: pthreads-w32
- Windows NT und Folgeversionen
  - Thread-Unterstützung auf Kernel-Ebene
  - WinSDK enthält umfangreichen Funktionen für Thread-Nutzung in C/C++

2.1.1 Eigenschaften

THR

- U

- W

Beispiel: PThreads in C++

```
01 #include <pthread.h>
02 #include <stdio.h>
03 #include <stdlib.h>
04 #include <assert.h>
05
06 #define THREAD_COUNT 20
07
08 void *threadMain(void *argument)
09 {
10   int tid;
11
12   tid = *((int *) argument);
13   printf(„Inside thread %i!\n", tid);
14   return 0;
15 }
16
17 int main()
18 {
19   pthread_t threads[THREAD_COUNT];
20   int thread_args[THREAD_COUNT];
21   int rc, i;
22
23   // Create threads
24   for (i=0; i<THREAD_COUNT; ++i) {
25      thread_args[i] = i;
26      printf(„Create thread %i\n", i);
27      rc = pthread_create(&threads[i], NULL, threadMain, (void *) &thread_args[i]);
28      assert(0 == rc);
29   }
30
31   // Wait for threads to end
32   for (i=0; i<THREAD_COUNT; ++i) {
33      rc = pthread_join(threads[i], 0);
34      assert(0 == rc);
35   }
36
37   exit(0);
38 }
```

2.1.2 Anwendungsszenarien

GUI UND ANWENDUNGSLOGIK

- Anwendungen mit grafischen Benutzerschnittstellen
  - Parallele Abarbeitung
    - Anwendungslogik: Hintergrundberechnungen → Thread 1
    - Änderung in der GUI-Darstellung → Thread 2
    - Reaktion auf Anwenderinteraktionen → Thread 3 (evtl. auch Thread 2)
- Lösung: Threads (insb. zur Bearbeitung von Events durch Anwenderinteraktion)
- Inhärente Realisierung in Systemen zur GUI-Realisierung
  - Thread-Bearbeitung von grafischen Ausgaben und Anwender-Events
  - *Beispiel: Java (Swing), Microsoft MFC, Nokia Qt Development Frameworks*

KOMMUNIKATION (MULTI-CLIENT SERVER)

- Server
  - Parallele Bearbeitung von empfangenen Daten/Nachrichten und Empfang neuer Nachrichten
  - Parallele Bedienung mehrerer Clients
- Client: Asynchrone Kommunikation (kein Warten auf Server-Antworten)
- Realisierung in Java
  - RMI: Server inhärent, Client manuell
  - Socket: manuell bei Client und Server

2.1.3 Realisierung

GENERIERUNG VON THREADS IN JAVA

- Klasse `Thread`: Ableitung erforderlich
- Schnittstelle `Runnable`: Implementierung erforderlich
  - Hilfreich bei Verwendung weiterer Superklassen (Mehrfachableitung in Java nicht möglich)

KLASSE THREAD

- Schritt 1 „Ableiten“: Neue Klasse als Ableitung von `Thread`


```
public class MyClass extends Thread { … }
```

- Schritt 2 „Überschreiben“: Methode `run` überschreiben


```
public void run ()
```

  - Startpunkt für Thread-Ausführung: `run` wird nach Start des Threads automatisch zuerst ausgeführt

- Schritt 3 „Starten“: Thread-Objekt instanziieren und starten


```
Thread thread = new MyClass();
thread.start ();
```

2.1.3 Realisierung

SCHNITTSTELLE RUNNABLE

- Schritt 1 „Implementieren“: Neue Klasse als Implementierung von `Runnable`


```
public class MyClass implements Runnable { … }
```

- Schritt 2 „Überschreiben“: Methode `run` überschreiben


```
public void run ()
```

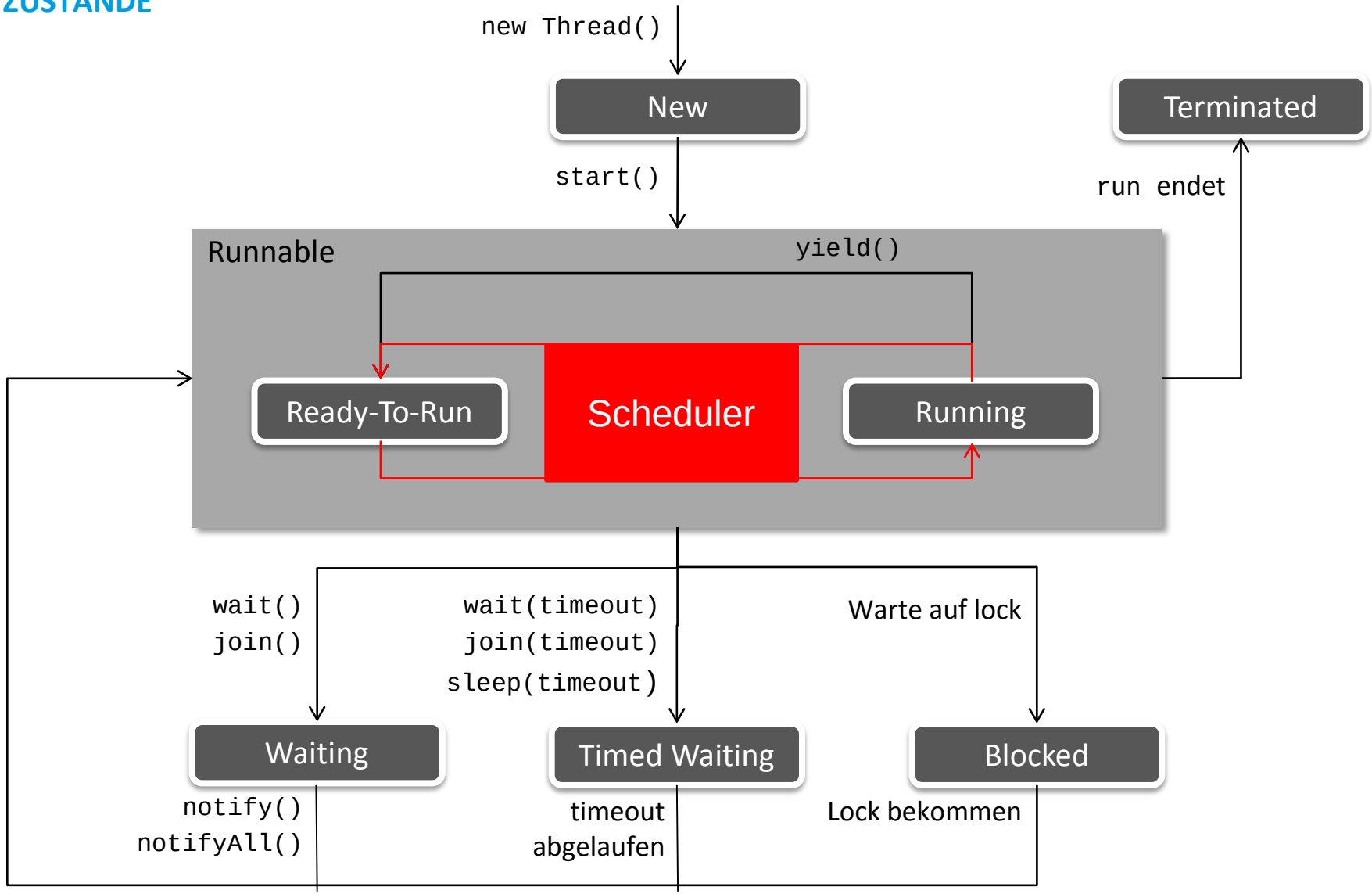
  - Startpunkt für Thread-Ausführung: `run` wird nach Start des Threads automatisch zuerst ausgeführt

- Schritt 3 „Starten“: Thread-Objekt instanziieren und starten


```
Thread thread = new Thread(new MyClass());
thread.start ();
```

2.1.3 Realisierung

ZUSTÄNDE



Übersicht

2.1 THREADS

- 2.1.1 Eigenschaften
- 2.1.2 Anwendungsszenarien
- 2.1.3 Realisierung

2.2 SYNCHRONISIERUNG

- 2.2.1 Motivation
- 2.2.2 Synchronisationsmechanismus: Kritische Bereiche und Sperrobjekte
- 2.2.3 Synchronisationsmechanismus: Monitore
- 2.2.4 Deadlocks & Starvation

2.3 POOLING UND EXECUTOR

- 2.3.1 Grad der Nebenläufigkeit
- 2.3.2 Executor

2.4 SERIALISIERUNG

- 2.4.1 Motivation
- 2.4.2 Serialisierbare Klassen
- 2.4.3 Verwendung von Streams
- 2.4.4 Versionierung
- 2.4.5 Formatdefinition

2.2.1 Motivation

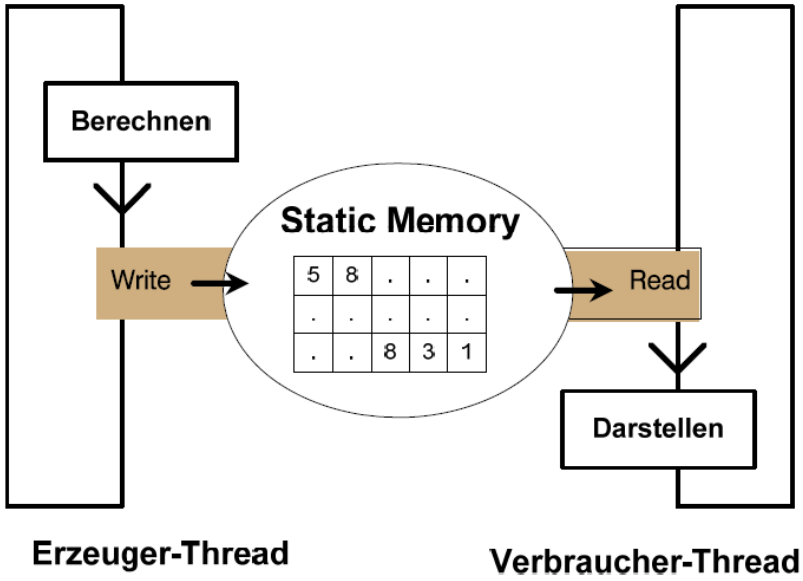
ABHÄNGIGE THREADS

- Szenario: Aufgaben von Threads sind nicht unabhängig
  - Zugriff auf dieselben Ressourcen
  - Ggf. Kooperation zur Bewältigung einer gemeinsamen Aufgabe
- **Konkurrierende Threads**: Threads greifen auf dieselben Ressourcen zu

  → Synchronisierung des Zugriffs erforderlich zur Vermeidung inkonsistenter Daten
- **Kooperierende Threads**: Abfolge der Thread-Ausführung entscheidend
  - Erzeuger-Verbraucher-Beziehung zwischen Threads
    - Verbraucher-Thread nur dann erfolgreich, wenn Aufgabe von Erzeuger-Thread vollständig

ERZEUGER-/VERBRAUCHER-SITUATION

- Teilaufgaben: Erzeugen und Verbrauchen werden in separaten Threads ausgeführt
  - *Beispiel: Erzeugen – Berechnung einer Tabelle; Verbrauchen – Grafische Darstellung einer Tabelle*

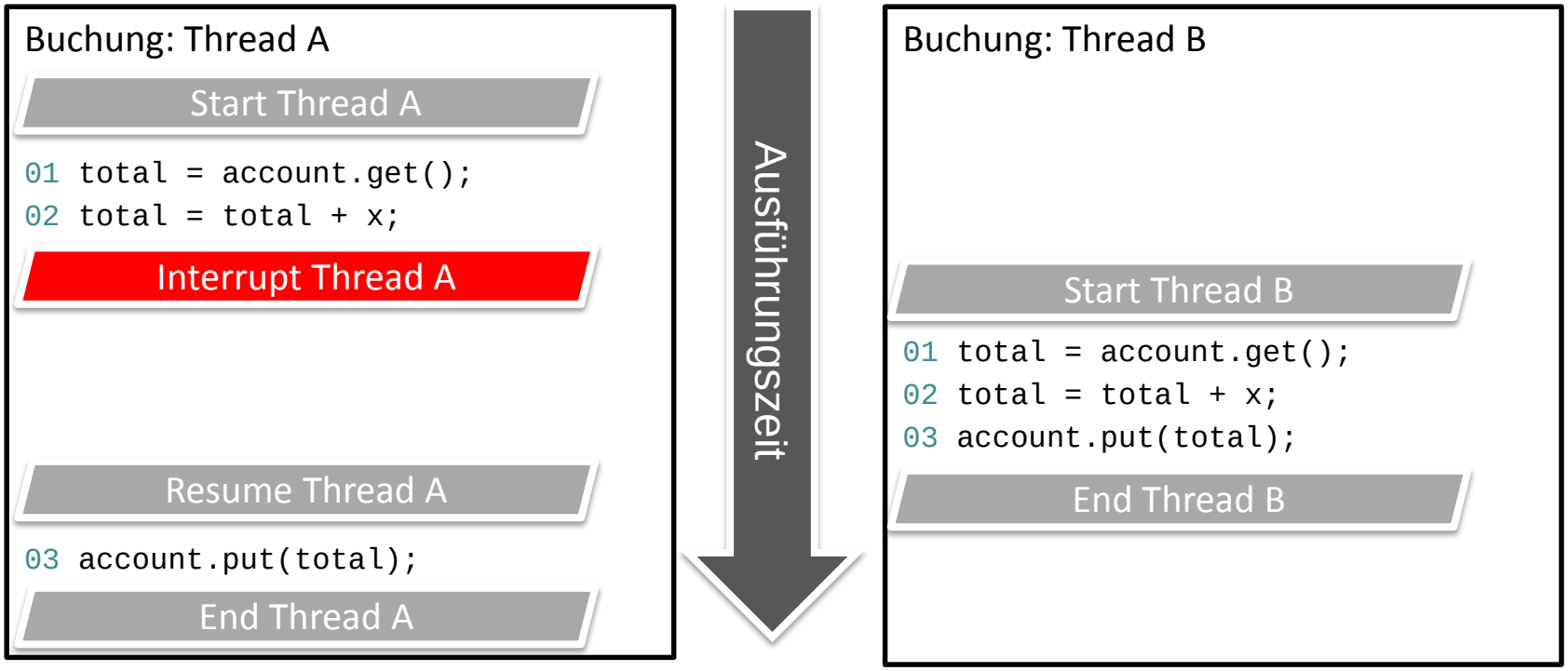


2.2.1 Motivation

BEISPIEL: SIMULTANE KONTOBUCHUNGEN

- Mehrere Anwender greifen auf dasselbe Konto zu

  → Mehrere Threads buchen gleichzeitig auf dasselbe Konto



- **Race Condition (Wettlaufsituation, kritischer Wettlauf)**: Ergebnis der beiden Threads hängt von Verteilung der CPU-Zeit auf Threads ab
  - Im Beispiel: Buchung B geht verloren;
    - Kontostand ist Ressource, um die beide Threads konkurrieren

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

UNTERBRECHUNGSFREIE OPERATIONEN

- **Atomare Operation**: Ausführung der Blöcke (Folgen von Anweisungen) oder Methoden ohne Unterbrechung durch andere Threads
- **Kritischer Abschnitt**: Programmcode innerhalb einer atomaren Operation Critical section
- Verantwortung des Entwicklers: Kenntnis der Anwendung und Programmstruktur zur Definition von atomaren Operationen und kritischen Abschnitten erforderlich
- Verantwortung der Laufzeitumgebung: Bereitstellung der Mittel zur unterbrechungsfreien Ausführung von kritischen Abschnitten
  - Virtuelle Maschine/Betriebssystem, ggf. in Zusammenarbeit mit Prozessor

SPERROBJEKTE (LOCKS)

- Besitz des Sperrobjekts zum Durchlauf eines kritischen Abschnitts erforderlich
- Verfahren
  (1) Prüfung des Sperrobjekts vor Eintritt in kritischen Abschnitt
    (a) Sperrobjekt frei → Belegung des Sperrobjekts und Ausführung des kritischen Abschnitts
    (b) Sperrobjekt belegt → Thread wartet auf Freigabe
  (2) Kritischer Abschnitt durchlaufen → Freigabe Sperrobjekt

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

… SPERROBJEKTE

- Voraussetzung: Abprüfen und Belegen des Sperrobjekts sind atomare Operationen
  - In Verantwortung der Laufzeitumgebung (VM, Betriebssystem, ggf. Prozessor)
    - Windows od UNIX/LINUX: System Calls des Betriebssystems oder SDK-Funktionen
    - Prozessor: Maschinenbefehl „Test and Set“ (TAS)
- Bezeichnungen für Sperrobjekte: **Lock**, **Mutex** (Mutally exclusive lock), **Monitor**, **Spinlock**, **Semaphor**
- Allgemeine Bezeichnung des Sperrverfahrens: **Wait and Signal**
- Historie: Frühe Lösung (1965) für Synchronisierungsproblem mit Semaphoren durch Dijkstra (holländischer Mathematiker/Informatiker, 1930-2002)
- Sichere Programmierung von kritischen Abschnitten ist fundamentale Aufgabenstellung der angewandten Informatik und insb. des Fachgebiets „Verteilte Systeme“
- Frameworks zur Softwareentwicklung: Java und .NET enthalten Funktionen zur Threadsynchronisierung
  - Verwenden idR System Calls des unterliegenden Betriebssystems

Bayrische Zugspitzbahn, eingleisig, Juni 2000

Quelle: DPA

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

ZIEL

- Ja: Vermeidung von Race Conditions: Realisierung durch Sperrobjekte *Zeitlich*
  → In Java: **Locks**
- Nein: Festlegung der Ausführungsreihenfolge für kooperierende Threads *Inhaltlich*
  → In Java: Realisierung durch **Monitore** (vgl. nächster Abschnitt 2.2.3)

LOCKS IN JAVA

- Umsetzung mit Schlüsselwort `synchronized`
- Ziel: Kapselt Programmblock, der auf gemeinsame Ressource(n) zugreift
  → Programmblock kann nicht durch anderen Thread unterbrochen werden
- Auswirkung: Belegt Objekt oder Array mit Lock für aktuell ausgeführten Thread
  - Lock bereits belegt → Thread wechselt in Zustand „Blocked“
- Syntax

```
synchronized (expression)
{
  statements // critical section
}
```

  - `expression`: Referenziert auf zu „lockendes“ Objekt oder Array
  - `statements`: Kritischer Abschnitt
- Synchronisierung von Methoden: Modifier `synchronized` bei Methoden-Deklaration
  → Verwendetes Objekt der Klasse beim Methodenaufruf dient als Sperrobjekt

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

BEISPIEL: PRIMZAHL

```
import java.io.*;

public class PrimeMain {
    private static BufferedReader reader;
    private static PrimThread[] threads;
    private static int threadCount;

    private static void createThread() throws IOException {
        int initialValue,count;
        // Read initial thread values
        System.out.print("Initial value> ");
        initialValue=Integer.parseInt(reader.readLine());
        System.out.print("Count> ");
        count=Integer.parseInt(reader.readLine());
        // (end) Read initial thread values

        threads[threadCount++]=new PrimThread(initialValue,count);
        System.out.println("Thread created with initial value "+initialValue+" and count "+count+".");
    }

    private static String getMenuCommand() throws IOException {
        System.out.print("> ");

        return reader.readLine();
    }
```

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

… BEISPIEL: PRIMZAHL

```
private static void printMenu() {
    System.out.println("--------------------");
    System.out.println("(c)reate thread");
    System.out.println("show (v)alues");
    System.out.println("(s)tart threads");

    System.out.println("\nprint (m)enu");
    System.out.println("e(x)it");
    System.out.println("--------------------");
}

private static void showValues() {
    for(int i=0;i<threadCount;i++) {
        System.out.println("Thread "+i+":");
        synchronized(threads[i].found) {
            for(int j=0;j<10;j++) {
                System.out.print ("  "+threads[i].found[j]);
              }
              System.out.println("\n\n");
        }
    }
}
```

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

... BEISPIEL: PRIMZAHL

```
    public static void main(String[] args) throws IOException {
        String command;
        boolean doExit=false;

        reader=new BufferedReader(new InputStreamReader(System.in ));
        threads=new PrimThread[10];
        threadCount=0;

        printMenu();

        while(!doExit) {
            command=getMenuCommand();

            switch(command) {
                case "c":   createThread(); break;
                case "s":   for(int i=0;i<threadCount;i++) {
                                threads[i].start();
                                System.out.println("Thread "+i+" started.");
                            }
                            break;
                case "v":   showValues(); break;

                case "m":   printMenu(); break;
                case "x":   doExit=true; break;
            }
        }
    }
}
```

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

... BEISPIEL: PRIMZAHL

```
public class PrimThread extends Thread {
    private int initialValue, count;
    public int found[];

    public PrimThread(int initialValue, int count) {
        this.initialValue=initialValue;
        this.count=count;

        found=new int[10];
    }

    public void run() {
        boolean prim;
        int k=-1, j=0;

        for (int i=initialValue; ; i++) {
            try {sleep(100);} catch (Exception e) {}
            prim = true;
            for (int h = (i - 1); h > 1; h--) {
              if ( (i % h) == 0) {prim = false; break;}
            }
            synchronized(found) {
                if (prim) {
                    k = k + 1;
                    j = k % 10;
                    found[j] = i;
                }
            }
```

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

… BEISPIEL: PRIMZAHL

```
            if(k==count) break;
        }
    }
}
```

2.2.2 Synchronisationsmechanismen: Kritische Bereiche und Sperrobjekte

FAZIT

- Locks vermeiden Race Conditions konkurrierender Threads

  → Kritische Abschnitte (ggf. ganze Methode) sind **thread safe**

BEISPIEL „SIMULTANE KONTENBUCHUNGEN“ MIT LOCKS

```
class account  {
     private float total;
     public synchronized boolean put (float t) {
          if(total+t < 0) {
               total+=t;
               return true;
          }
          else return false;
     }
     public synchronized float get() {
          return total;
     }
}
```

Übersicht

2.1 THREADS

- 2.1.1 Eigenschaften

- 2.1.2 Anwendungsszenarien

- 2.1.3 Realisierung

2.2 SYNCHRONISIERUNG

- 2.2.1 Motivation

- 2.2.2 Synchronisationsmechanismus: Kritische Bereiche und Sperrobjekte
- 2.2.3 Synchronisationsmechanismus: Monitore
- 2.2.4 Deadlocks & Starvation

2.3 POOLING UND EXECUTOR

- 2.3.1 Grad der Nebenläufigkeit
- 2.3.2 Executor

2.4 SERIALISIERUNG

- 2.4.1 Motivation
- 2.4.2 Serialisierbare Klassen
- 2.4.3 Verwendung von Streams
- 2.4.4 Versionierung
- 2.4.5 Formatdefinition

2.2.3 Synchronisationsmechanismus: Monitore

MONITORE

- Locks lösen Problem kooperierender Threads nicht → Alternative Synchronisierung gefragt
  - Grund: Kritische Bereiche sichern nicht die Reihenfolge der Ausführung beteiligter Threads
- Monitore: Methoden `wait()` und `notify()` zu jedem beliebigen Objekt einer Klasse

```
public final void notify()
```

  - Weckt einen auf den Monitor wartenden Thread → Versetzt Thread in Status „Runnable“
  - Bei mehreren wartenden Threads: Auswahl des zu weckenden Threads implementationsabhängig

```
public final void wait ([long timeout])
```

  - Versetzt Thread in Wartestatus „Waiting“ bzw. „Timed Waiting“
  - Thread wartet auf `notify()` oder `notifyAll()` oder Erreichen des Timeouts

2.2.3 Synchronisationsmechanismus: Monitore

BEISPIEL: PRIMZAHL MIT MONITOR

```
import java.io.*;

public class PrimeMain {
    private static BufferedReader reader;
    private static PrimThread[] threads;
    private static int threadCount;

    private static void createThread() throws IOException {
        int initialValue,count;
        boolean monitored;
        // Read initial thread values
        System.out.print("Initial value> ");
        initialValue=Integer.parseInt(reader.readLine());
        System.out.print("Count> ");
        count=Integer.parseInt(reader.readLine());
        System.out.print("Monitored> ");
        monitored=Boolean.parseBoolean(reader.readLine());
        // (end) Read initial thread values

        threads[threadCount++]=new PrimThread(initialValue,count,monitored);
        System.out.println("Thread created with initial value "+initialValue+
                ", count "+count+" and monitor "+monitored+".");
    }

    private static String getMenuCommand() throws IOException {
        System.out.print("> ");

        return reader.readLine();
    }
```

2.2.3 Synchronisationsmechanismus: Monitore

… BEISPIEL: PRIMZAHL MIT MONITOR

```
private static void printMenu() {
    System.out.println("--------------------");
    System.out.println("(c)reate thread");
    System.out.println("show (v)alues");
    System.out.println("(s)tart threads");

    System.out.println("\nprint (m)enu");
    System.out.println("e(x)it");
    System.out.println("--------------------");
}

private static void showValues() {
    for(int i=0;i<threadCount;i++) {
        System.out.println("Thread "+i+":");
        synchronized(threads[i].found) {
            try {
                threads[i].found.notify(); threads[i].found.wait();
            }
            catch(InterruptedException e) {
                e.printStackTrace();
            }
            for(int j=0;j<10;j++) {
                System.out.print ("  "+threads[i].found[j]);
              }
              System.out.println("\n\n");
        }
    }
}
```

2.2.3 Synchronisationsmechanismus: Monitore

… BEISPIEL: PRIMZAHL MIT MONITOR

```
    public static void main(String[] args) throws IOException {
        String command;
        boolean doExit=false;

        reader=new BufferedReader(new InputStreamReader(System.in ));
        threads=new PrimThread[10];
        threadCount=0;

        printMenu();

        while(!doExit) {
            command=getMenuCommand();

            switch(command) {
                case "c":   createThread(); break;
                case "s":   for(int i=0;i<threadCount;i++) {
                                threads[i].start();
                                System.out.println("Thread "+i+" started.");
                            }
                            break;
                case "v":   showValues(); break;

                case "m":   printMenu(); break;
                case "x":   doExit=true; break;
            }
        }
    }
}
```

2.2.3 Synchronisationsmechanismus: Monitore

… BEISPIEL: PRIMZAHL MIT MONITOR

```
public class PrimThread extends Thread {
    private int initialValue, count;
    private boolean monitored;
    public int found[];

    public PrimThread(int initialValue, int count, boolean monitored) {
        this.initialValue=initialValue;
        this.count=count;
        this.monitored=monitored;

        found=new int[10];
    }

    public void run() {
        boolean prim;
        int k=-1, j=0;

        for (int i=initialValue; ; i++) {
            try {sleep(100);} catch (Exception e) {}
            prim = true;
            for (int h = (i - 1); h > 1; h--) {
              if ( (i % h) == 0) {prim = false; break;}
            }
            synchronized(found) {
                if (prim) {
                    k = k + 1;
                    j = k % 10;
                    found[j] = i;
                }
```

2.2.3 Synchronisationsmechanismus: Monitore

… BEISPIEL: PRIMZAHL MIT MONITOR

```
                try {
                    if(monitored) {
                        if (j == 9) {
                            found.notify(); found.wait();
                        }
                    }
                    else found.notify();
                }
                catch(InterruptedException e) {
                    e.printStackTrace();
                }
            }

            if(k==count) break;
        }
    }
}
```

Übersicht

2.2.4 Deadlocks & Starvation

BEISPIEL

```
public static int [] a = new int[10];
public static int [] b = new int[10];

public class Thread1 extends Thread {
    public void run() {
        synchronized(a) {
            synchronized(b) {
                doSomething();
            }
        }
    }
}
public class Thread2 extends Thread {
    public void run() {
        synchronized(b) {
            synchronized(a) {
                doSomething();
            }
        }
    }
}
```

2.2.4 Deadlocks & Starvation

PROBLEM

- Erfolgreiche Synchronisierung von Threads mit Locks und Monitoren
  - → Voraussetzung: Genaue Kenntnis von
    - Anwendungsproblem
    - Software-Entwurf
    - Funktionsweise von Locks und Monitoren
  - → Drohende Konsequenz: Deadlocks oder Starvation

BEISPIELE VON DEADLOCKS

- Deadlock (Verklemmung)
  - Thread T1 hat Ressource A reserviert und wartet auf Ressource B
  - Thread T2 hat Ressource B reserviert und wartet auf Ressource A
- Deadlock bei `wait()` und `notify()`
  - Falsche Reihenfolge

2.2.4 Deadlocks & Starvation

STARVATION

- Unzureichender, unfairer Anteil an Rechenleistung für mindestens einen Thread
  → Thread „verhungert“ (Starvation)
- Mögliche Ursache: Fehlerhafte Vergabe von Prioritäten

ALLTAGSBEISPIELE FÜR DEADLOCKS

- Kreuzung mit Rechts-vor-Links-Vorfahrtsregelung
  - Gleichzeitige Annäherung der Verkehrsteilnehmer von allen Seiten
- Speisende Philosophen: 5 Philosophen an einem Tisch
  - Ein Teller Spaghetti und eine Gabel (oder Stäbchen) auf der rechten Seite des Tellers pro Philosoph
  - Philosoph kann nachdenken oder (wenn er hungrig wird) essen
  - Philosoph braucht zum essen zusätzlich Gabel des linken Nachbarn
- Deadlock?
- Starvation?

PROBLEM BEI GROBGRANULAREN KRITISCHEN BEREICHEN

- Wahrscheinlichkeit für Deadlocks und Starvation steigt
- Nebenläufigkeit der Anwendung reduziert

2.2.4 Deadlocks & Starvation

STARVATION

- Unzureichender, unfaire
  → Thread „verhungert“
- Mögliche Ursache: Fehle

ALLTAGSBEISPIELE FÜR DE

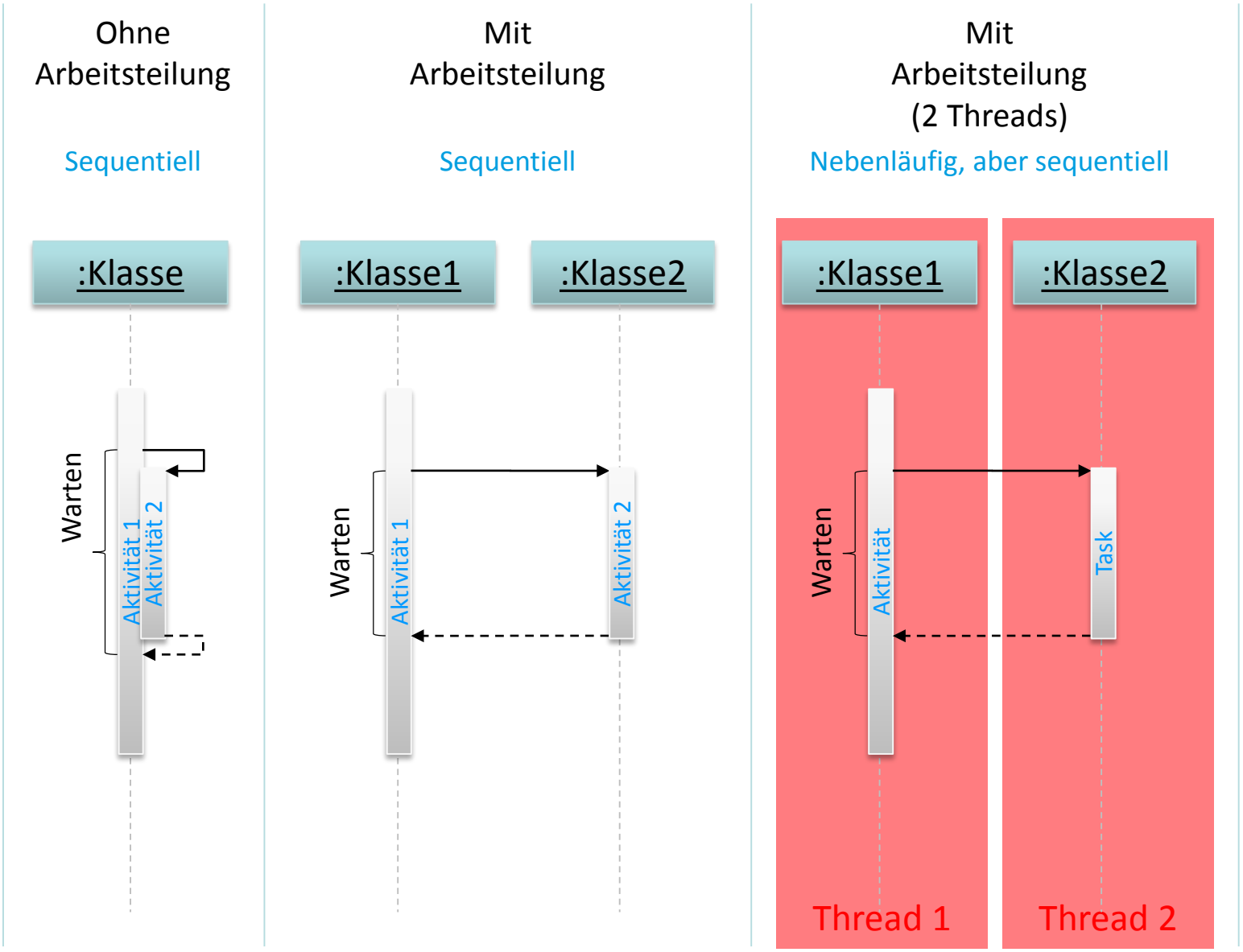
- Kreuzung mit Rechts-vor
  - Gleichzeitige Annähe
- Speisende Philosophen:
  - Ein Teller Spaghetti u
  - Philosoph kann nach
  - Philosoph braucht zu
- Deadlock?
- Starvation?
- Schöne Simulation: http

PROBLEM BEI GROBGRANULAREN KRITISCHEN BEREICHEN

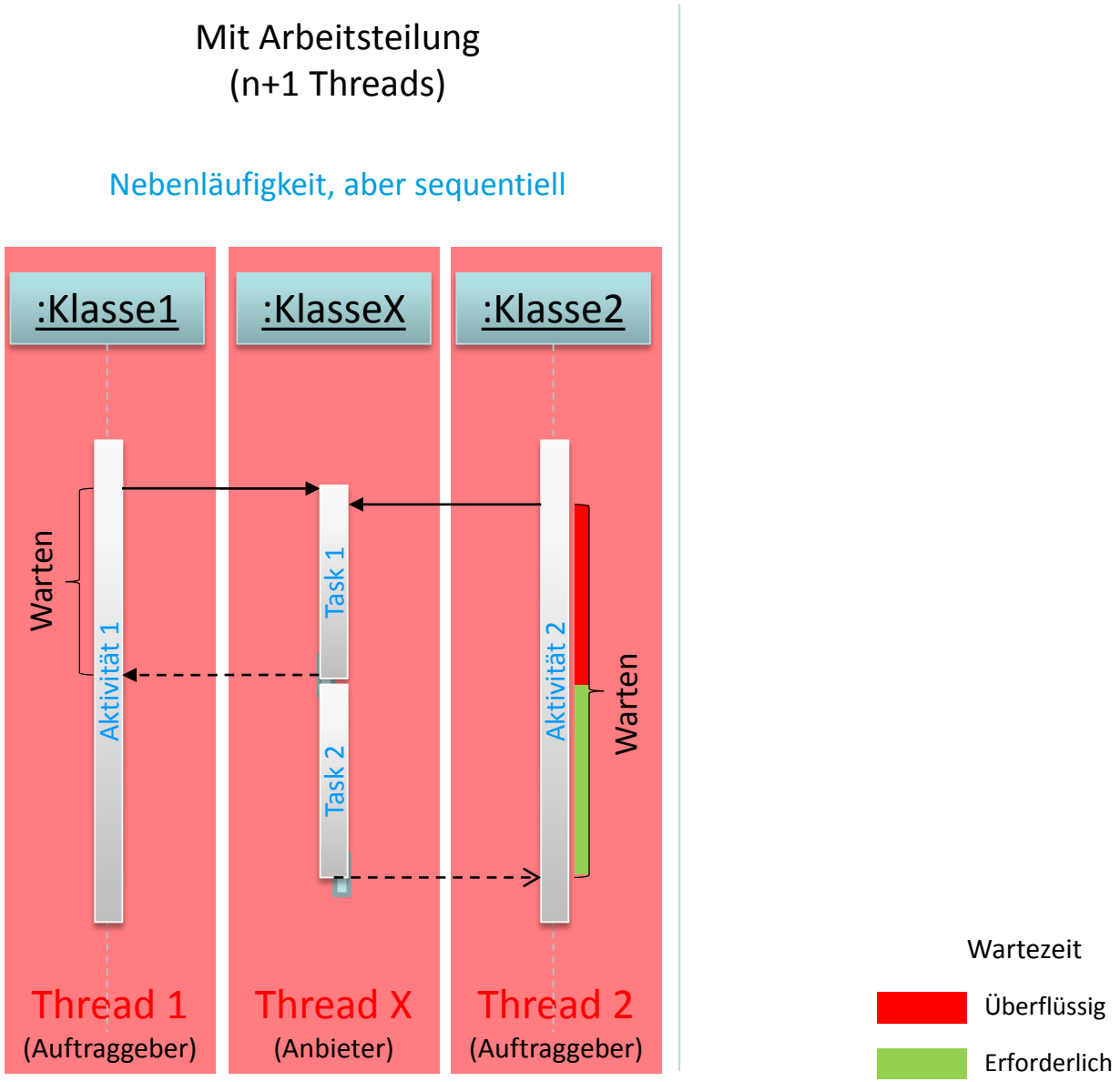
- Wahrscheinlichkeit für Deadlocks und Starvation steigt
- Nebenläufigkeit der Anwendung reduziert

Übersicht

2.3.1 Grad der Nebenläufigkeit



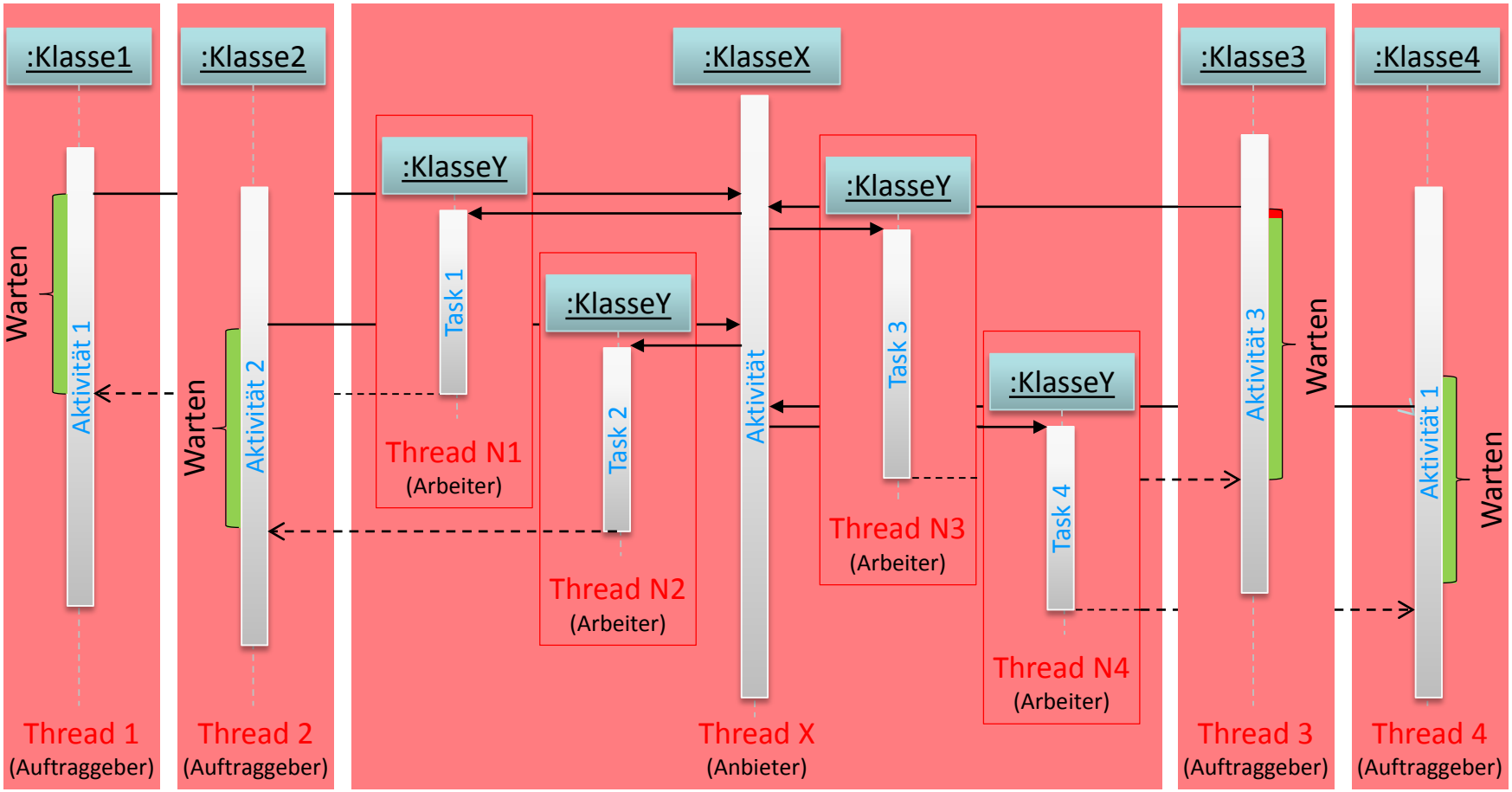
2.3.1 Grad der Nebenläufigkeit



2.3.1 Grad der Nebenläufigkeit

Mit Arbeitsteilung
(n+n+1 Threads)

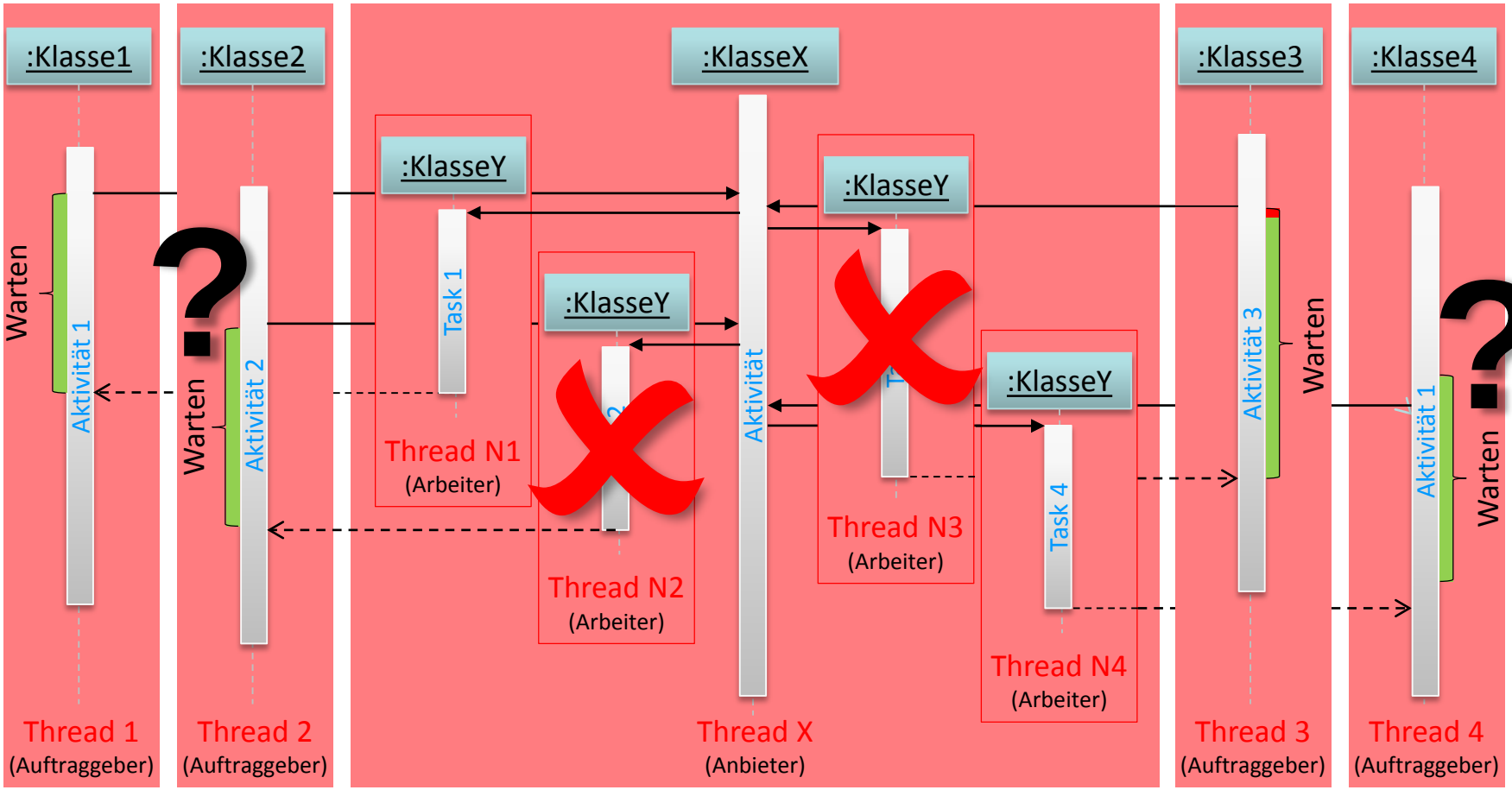
Nebenläufig
(1 Thread pro Task)



2.3.1 Grad der Nebenläufigkeit

Mit Arbeitsteilung
(n+n+1 Threads)

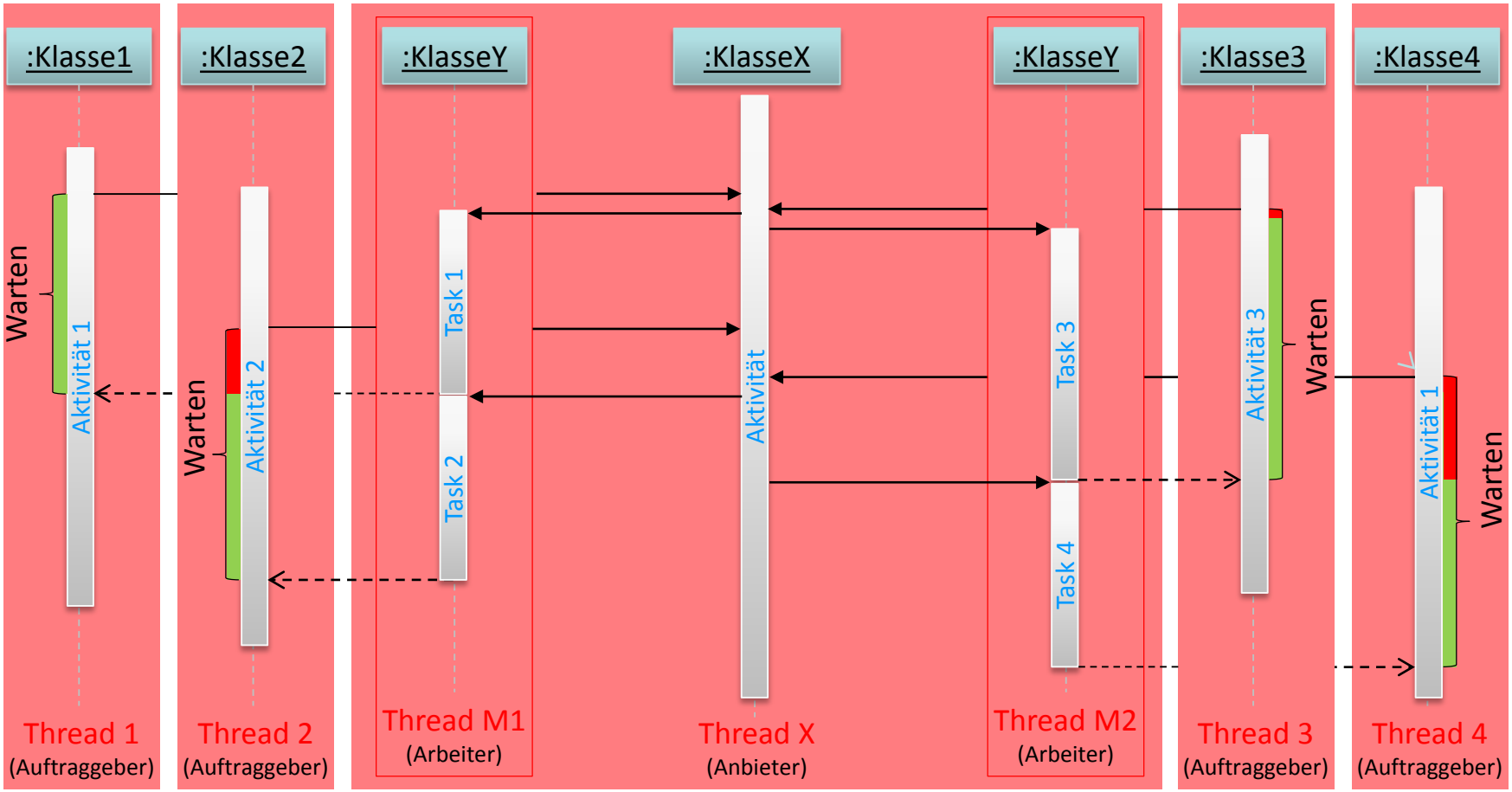
Nebenläufig
(1 Thread pro Task)



2.3.1 Grad der Nebenläufigkeit

Mit Arbeitsteilung
(n+m+1 Threads, m << n)

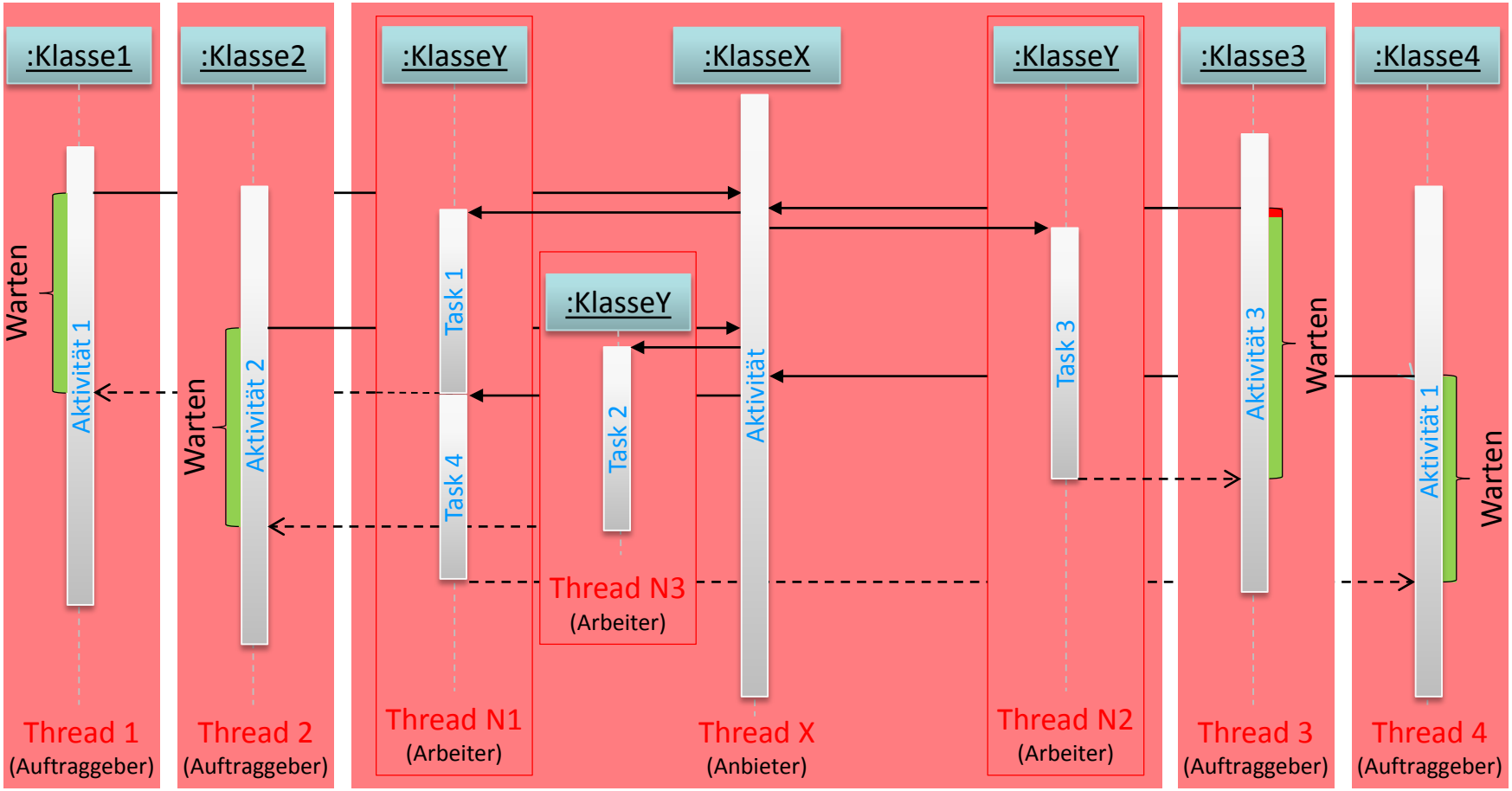
Nebenläufig
(Konstanter Thread-Pool mit m Threads)



2.3.1 Grad der Nebenläufigkeit

Mit Arbeitsteilung
(n+m+k+1 Threads, m << n, k ≤ n-m)

Nebenläufig
(Dynamischer Thread-Pool mit m Initial-Threads)



2.3.2 Pool-Größen und Bewertungsmaßstäbe

LAUFZEITEN, WARTEZEITEN, LASTVERHALTEN UND RESSOURCENAUSNUTZUNG

Pool		Konsequenzen großer Anfragemengen für				
Verhalten	Größe	Tasks			Gesamtsystem	
		Neu		Laufend	Last	Ressourcen-ausnutzung
		Wartezeit	Laufzeit	Laufzeit		
Konstant	1	hoch	schnell	gleich	niedrig	schlecht
Dynamisch	∞	Keine	langsam	langsamer	zu hoch	optimal - hoch
Konstant	m	mittel-hoch	langsam - schnell	gleich - langsamer	niedrig - mittel	schlecht - optimal
Dynamisch	m + ∞	keine	langsam	langsamer	zu hoch	optimal - hoch

2.3.2 Executor

EXECUTOR FRAMEWORK IN JAVA

- Seit Java 5 keine manuelle Thread-Pool-Erstellung erforderlich
- Automatisierung eines Thread-Pools durch `ExecutorService`-Klasse
- **ExecutorService:** Funktionsweise
  - Realisierung einer Task Queue
    - → Freie oder freiwerdende Threads übernehmen nächsten Tasks der Queue
- **Task:** Erstellung
  - Implementierung von `Runnable`: Task-Ausführung von ExecutorService durch Aufruf von `run`-Methode
  - Implementierung von `Callable`: Task-Ausführung von ExecutorService durch Aufruf von `call`-Methode
    - Zusätzlich zu `Runnable` Rückgabewerte und Exceptions

EXECUTORSERVICE: VERWENDUNG

- Erstellung des `ExecutorService` durch Methoden von `Executors`
  - Konstanter Thread-Pool: `public static ExecutorService newFixedThreadPool(int nThreads)`
  - Dynamischer Thread-Pool: `public static ExecutorService newCachedThreadPool()`

- Übergabe von Tasks an Thread-Pool:
  `<executorService-Objekt>.execute(<runnable-Objekt>)`

- Beenden des Thread-Pools: `<executorService-Objekt>.shutdown()`

2.3.2 Executor

BEISPIEL

```
public class Task implements Runnable {
…
  public void run() {
        // Work to be done
  }
…
}

public class MyThreadPool {
  public static void main(String[] args){
        ExecutorService executor = Executors.newFixedThreadPool(MAX_THREADS);

        while (! done) {
                executor.execute(new Task());
        }
        executor.shutdown();
  }
}
```

Übersicht

2.4.1 Motivation

BEGRIFF SERIALISIERUNG

- Ja → „Sequentielle Darstellungsform“: Serialisierung als Sequenz von Bytes in der Repräsentation eines Objekts
- Nein → „Serialisierung vs. Parallelisierung“: Serialisierung in der Sequenz der Zugriffe auf ein Objekt

PRINZIP

- Serialisierung: Konvertierung eines Objekts in Bytestrom
- Deserialisierung: Konvertierung eines serialisierten Objekts (Bytestrom) in Kopie des Objekts

ANWENDUNGSSZENARIEN

- Persistente Objektspeicherung
- Netzwerkkommunikation
  - *Beispiel Java RMI: Transport von Objekten: Parameter und Ergebnisse*
  - *Beispiel Socket-Programmierung: Kein eigenes Format für Kommunikation erforderlich*

2.4.2 Serialisierbare Klassen und Transienz

SERIALISIERUNG EINER KLASSE

- Alt. A: Implementierung der Schnittstelle `Serializable`
  - `class MyClass implements Serializable`
  - Schnittstelle `Serializable` hat keine Methoden
    - → Keine weitere Implementierung erforderlich
    - → Automatische Serialisierung durch Laufzeitsystem
  - Reihenfolge der Serialisierung
    - Header mit Objekttyp
    - Instanzvariablen: `public, private, protected, package-private` (Verwendung der `write`- oder `writeObjekt`-Methode)
  - Verweisduplikate: Mehrfach referenzierte Objekte nur einmal im Bytestrom
    - → Reduktion von Zyklen und Redundanz
- Alt. B: Superklasse ist serialisierbar
  - Rekursive Serialisierung der enthaltenen Objekte

TRANSIENZ

- Kennzeichnung der nicht zu serialisierenden Instanzvariablen mit `transient`
- Voraussetzung: Wert nach Deserialisierung unnötig oder wiederherstellbar
- Vorteile
  - Bandbreitenreduktion, Speicherplatzreduktion, schnellere Deserialisierung
  - Umgehen nicht serialisierbarer Objekte: Sonst `NotSerializableException` für gesamtes Objekt

2.4.3 Verwendung von Streams

OUTPUTSTREAM

- Konkrete Serialisierung durch Schreiben in `ObjectOutputStream`
  - Wrappen der Klasse `Outputstream: OutputStream oStream`

```
ObjectOutputStream myStream = new ObjectOutputStream (oStream);
```

  - Schreiben auf Stream durch Methode `writeObject`

```
myStream.writeObject(myObject);
```

INPUTSTREAM

- Konkrete Deserialisierung durch Lesen aus `ObjectInputStream`
  - Wrappen der Klasse `Inputstream: InputStream iStream`

```
ObjectInputStream myStream = new ObjectInputStream (iStream);
```

  - Lesen aus Stream durch Methode `readObject`

```
MyClass myObject = (MyClass) myStream.readObject();
```

    - `readObject` liefert Objekt vom Typ `Object` → Expliziter Typecast erforderlich

2.4.4 Versionierung

PROBLEMSTELLUNG

- Unterschiedliche Klassenversionen bei Serialisierung und Deserialisierung
  → `InvalidClassException`, wenn Instanzvariable `serialVersionUID` verschieden

SERIALVERSION UID

- `serialVersionUID`
  - Alt. A: Wird automatisch von System gesetzt
  - Alt. B: Manuelle Deklaration und Definition

```
private static final long serialVersionUID = 1234;
```

- Empfehlung: Manuelle Definition (Java Object Serialization Specification)
- Eclipse: Übernimmt manuelles Setzen automatisch

2.4.5 Formatdefinition

EXTERNALIZABLE

- Eigenständige Definition des Serialisierungsformats
  - → Schnittstelle `Externalizable` (Ableitung von `Serializable`)
  - Definition der Methoden

```
public void readExternal (ObjectInput);
public void writeExternal (ObjectOutput);
```

- Vorteil: Flexibler
- Nachteil: Fehleranfälliger

Übersicht