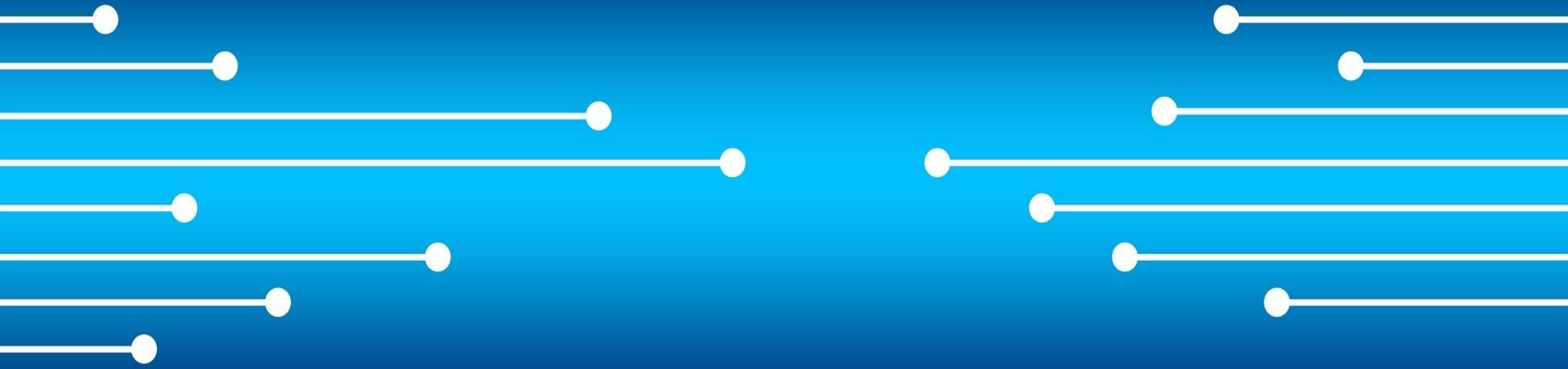
Datenverarbeitung

Teil des Moduls 5CS-DPDL-20



Prof. Dr. Deweß

Thema 12

Schnell, schneller ... „multi“

Ist Parallelverkehr möglich?

Ist Parallelverkehr sinnvoll?

Manchmal sollte man sich ein paar Gedanken machen ...

„Threads“

als Anbieter einer Form der konkurrenten Ausführung von Quellcode

- Überblick
- Thread-Erstellung
- Thread-Lebenszyklus
- Threads: Konflikte, Interaktion, Sicherheits-überlegungen

Überblick - Abstraktionsebenen

weitere Abstraktinsmöglichkeiten sind z.B. Prozess-Gruppen als Behälter für Prozesse und Sessions als Behälter für Prozess-Gruppen (Stichwort: POSIX)

Prozess

- repräsentiert die Ausführung eines Programms im Betriebssystem
- jeder Prozess hat seinen eigenen Speicherbereich und seine eigene Ausführungsumgebung
- bilden daher voneinander unabhängige Aufgabenbereiche ab, z.B. das Arbeiten in Eclipse, das Hören einer Musikdatei oder das Stöbern im Internet
- eigene Prozesse für Aufgabenbereiche bieten also eine starke Isolation der Aufgabenbereiche voneinander und eignen sich daher für unabhängige Aufgabenbereiche, die z.B. aus Sicherheits- oder Stabilitätsgründen getrennt voneinander ausgeführt werden sollen

Überblick - Abstraktionsebenen

weitere Abstraktinsmöglichkeiten sind z.B. Prozess-Gruppen als Behälter für Prozesse und Sessions als Behälter für Prozess-Gruppen (Stichwort: POSIX)

Prozess

Thread

- repräsentieren einzelnen Arbeitsablauf innerhalb eines Prozesses, der auf einem Prozessor sequentiell ausgeführt wird
- teilt sich Speicher und Ausführungsumgebung des Prozesses mit anderen Threads
- Threads eines Prozesses können daher leicht miteinander kommunizieren
- Ressourcenteilung bedingt aber auch, dass ein fehlerhafter Thread den Gesamtprozess mit seinen ggf. weiteren Threads leicht negativ beeinflussen kann
- besitzt daneben eigenen Ressourcen (eigener Satz von Registern, eigener Stack für lokale Variablen)
- eignen sich gut für Situationen, in denen es um kooperative oder synchronisierte Arbeit innerhalb derselben Anwendung geht

Überblick - Abstraktionsebenen

weitere Abstraktinsmöglichkeiten sind z.B. Prozess-Gruppen als Behälter für Prozesse und Sessions als Behälter für Prozess-Gruppen (Stichwort: POSIX)

Prozess

Thread

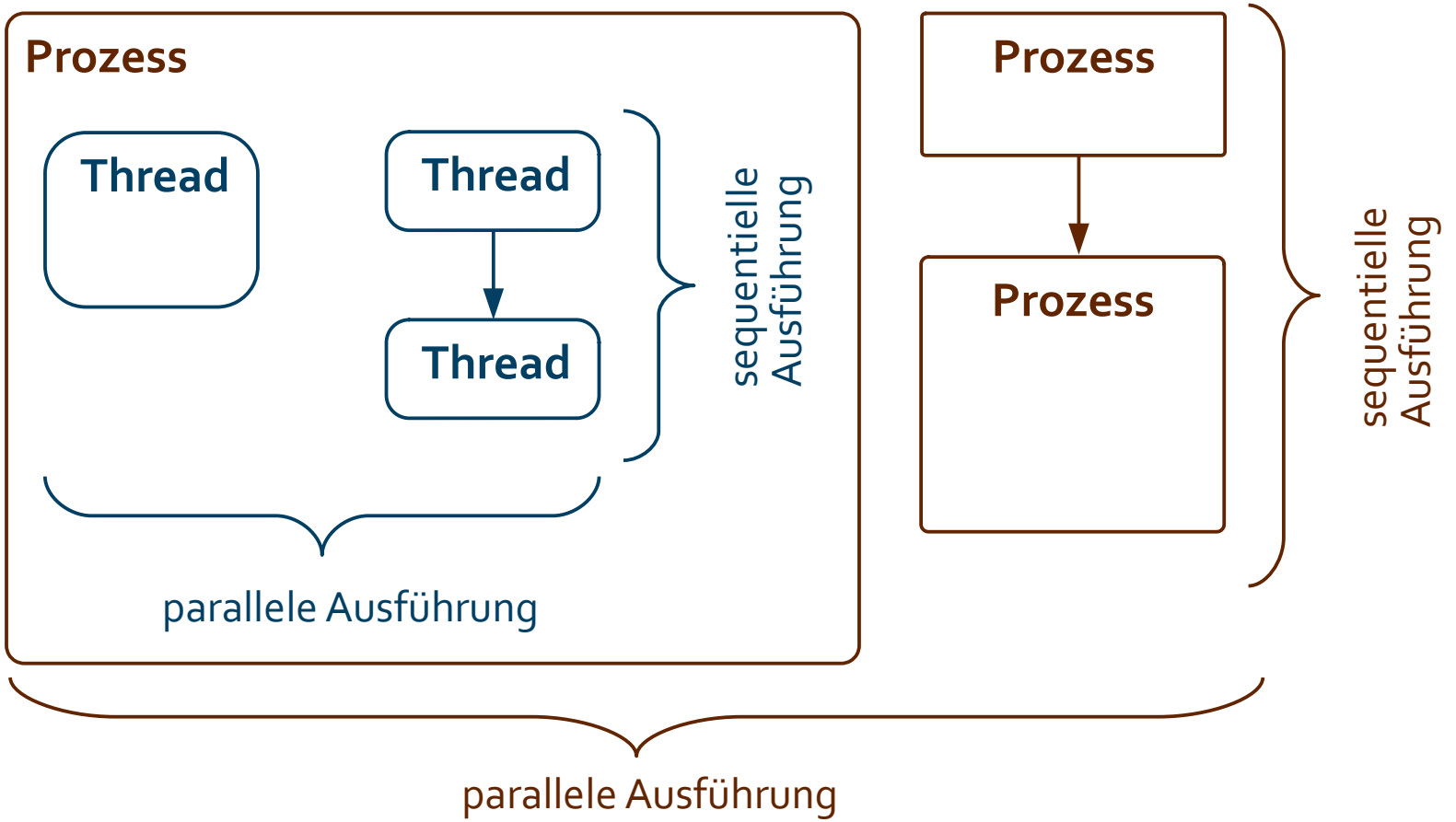
atomare Operation

- einzelne, unteilbare Einheit eines Threads
- kann während der Ausführung von keinem anderen Thread unterbrochen oder anderweitig beeinflusst werden

atomare Operation

atomare Operation

Überblick

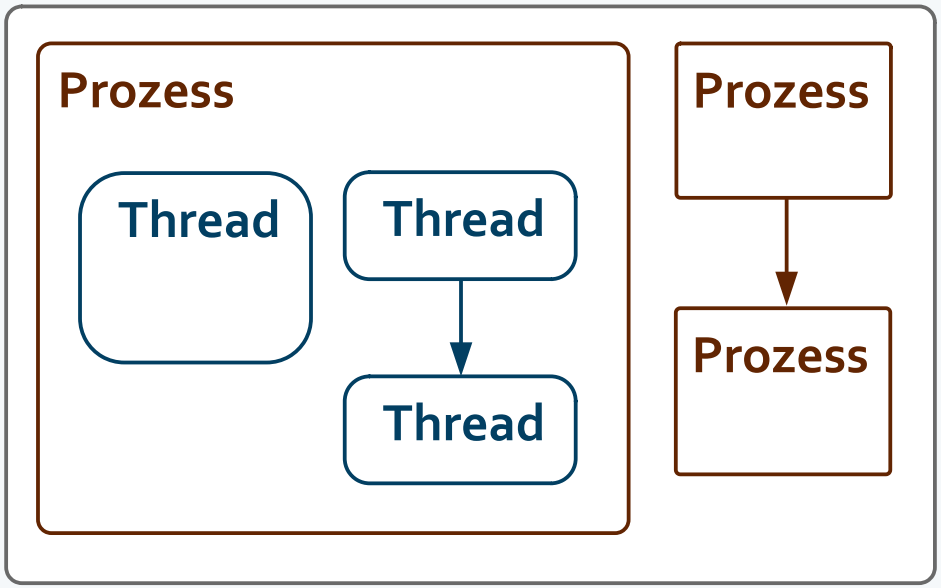


Prozesse und Threads sind meist zerlegbar, so dass neben einer rein parallelen oder rein sequentiellen Ausführung auch eine verzahnte Ausführung „Stück von dem einen Prozess/Thread, Stück von dem anderen Prozess/Thread" möglich ist.

Überblick - Multi-Tasking

Multi-Tasking

- Fähigkeit eines Betriebssystems, mehrere Dinge gleichzeitig zu tun, also die CPU-Zeit zwischen diesen Dingen zu wechseln, um die Ausführung dieser Dinge zu koordinieren
- ermöglicht effiziente Nutzung der CPU-Ressourcen und kann die Reaktionsfähigkeit des Systems verbessern



- wird erreicht durch:
  - **Multi-Threading:**
    - parallele oder nebenläufige Ausführung mehrerer Threads eines Prozesses
    - damit gleichzeitige Bearbeitung mehrerer Aufgaben innerhalb einer Anwendung
  - **Multi-Processing:**
    - parallele oder nebenläufige Ausführung mehrerer Prozesse
    - damit gleichzeitige Ausführung mehrerer Anwendungen möglich

Überblick - Multi-Tasking

Multi-Tasking

- bei der Programmierung interessiert uns also „Multi-Threading“ (während „Multi-Processing“ mehr auf Betriebssystemebene relevant ist)
- beim Start eines Java-Programmes wird durch die Ausführung der „main“-Funktion automatisch ein Thread gestartet („single-threaded“) und die eigentliche Frage ist, ob wir noch mehr Threads als diesen benötigen („multi-threaded“) – diese müssten wir dann aktiv anlegen und starten (als direkte Threads oder Anforderungen für Parallelverarbeitung, die intern dazu führen), während der „normale Thread“ bei der Programmierung nicht bewusst angelegt und gestartet werden muss

- wird erreicht durch:
  - **Multi-Threading:**
    - parallele oder nebenläufige Ausführung mehrerer Threads eines Prozesses
    - damit gleichzeitige Bearbeitung mehrerer Aufgaben innerhalb einer Anwendung
  - **Multi-Processing:**
    - parallele oder nebenläufige Ausführung mehrerer Prozesse
    - damit gleichzeitige Ausführung mehrerer Anwendungen möglich

„Threads“

als Anbieter einer Form der konkurrenten Ausführung von Quellcode

- Überblick
- **Thread-Erstellung**
- Thread-Lebenszyklus
- Threads: Konflikte, Interaktion, Sicherheits-überlegungen

Threads in Java

- für Threads gibt es in Java die **Klasse „java.lang.Thread"**
- Ausführung eines Java-Programms beginnt mit einer Instanz von Thread, die von der Java Virtual Machine (JVM) erstellt wird, die die „main()"-Methode des Programms ausführt
- zusätzliche Threads können durch die Erstellung zusätzlicher Instanzen der Klasse Thread eingeführt werden

- in Java wird ab Java SE 19 zwischen „Plattform-Threads" (die praktisch eins zu eins auf normale „Betriebssystem-Threads" abgebildet werden und daher nur in begrenzter Anzahl stabil eingesetzt werden können) und „virtuellen Threads" (hier werden praktisch viele Threads gemeinsam auf einen „Betriebssystem-Thread" abgebildet, so dass wesentlich mehr Threads stabil möglich sind) unterschieden
- diese Unterscheidung und die dazugehörigen Builder (sowohl für „Plattform-Threads" als auch für „virutelle Threads") sind allerdings noch ein Preview-Feature, die nur im Preview-Modus verfügbar sind und deren Fortbestand in weiteren Java-Versionen zwar anzunehmen, aber theoretisch noch ungewiss ist
- wir werden uns daher und weil wir hier noch in einer Grundlagenvorlesung sind auf die bisherigen „Threads" in Java, also die „Plattform-Threads", beschränken

Thread-Erstellung in Java

Grundsätzlich gibt es zwei Möglichkeiten zur Thread-Erstellung in Java

- Erben von der Klasse „java.lang.Thread“, mit überschriebener „run“-Methode und damit Erzeugung eines ggf. benannten Threads und starten des Threads
- Implementierung der „Runnable“-Schnittstelle (einer funktionalen Schnittstelle mit der Methode „run“), Erzeugung eines ggf. benannten Threads mit einem „Runnable“-Objekt und starten dieses Threads
- dabei können Threads auch als Mitglieder einer Thread-Gruppe erstellt werden (nicht klausurrelevant)

Constructors

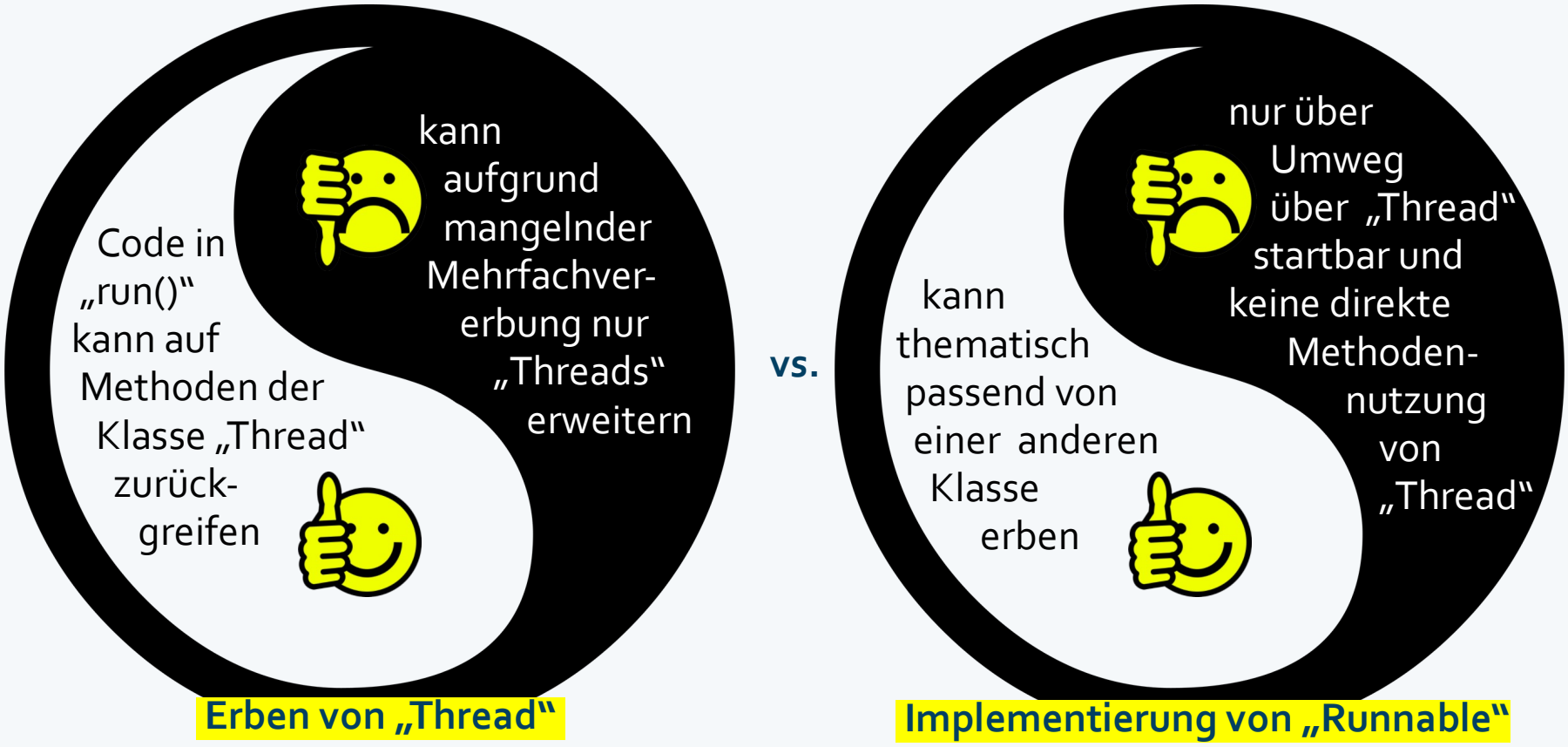
Constructor
`Thread()`
`Thread(Runnable task)`
`Thread(Runnable task, String name)`
`Thread(String name)`
`Thread(ThreadGroup group, Runnable task)`

.
.
.

Link zur Dokumentation:
https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/ThreadGroup.html

Thread-Erstellung in Java

Erben von „Thread“ im Vergleich zur Implementierung von „Runnable“



Erben von „Thread“ **vs.** **Implementierung von „Runnable“**

Thread-Erstellung: von „Thread“ erben

Ein Beispiel

```
package de.baleipzig.threads;
public class ThreadExtendingClass extends Thread {
  @Override
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println("ThreadExtendingClass i = " + i);
    }
  }
}
```

„run“ ist für eine Thread das, was „main“ für das Programm an sich ist

```
package de.baleipzig.threads;
public class ThreadDemo {
  public static void main(String[] args) {
    ThreadExtendingClass testClass = new ThreadExtendingClass();
    testClass.start();
  }
}
```

wir rufen nicht „run“ auf, sondern starten den Thread mit „start“, wodurch „run“ ausgeführt wird

Thread-Erstellung: von „Thread“ erben

Aufgabe 1

a) Übernehmen Sie das Beispiel auf der vorherigen Folie. Erzeugen Sie in Ihrer Klasse „ThreadDemo“ einen weiteren Thread analog zum bisherigen Thread und starten Sie auch diesen.

b) Damit Sie bei der Ausgabe beide Threads gut voneinander unterscheiden können, nutzen Sie nun bei der Erzeugung beider Threads zusätzlich jeweils einen sinnvollen Namen.

c) Erweitern Sie dafür die Klasse „ThreadExtendingClass“ um den nötigen Konstruktor, der den passenden Konstruktor von Thread aufruft.

d) Nutzen Sie nun eine Methode von „Thread“, um sich zusätzlich bei den Konsolenausgaben von „run“ in „ThreadExtendingClass“ jeweils noch den Namen des aktuellen Threads anzeigen zu lassen.

Thread-Erstellung: „Runnable“ implementieren

Ein Beispiel

```
package de.baleipzig.threads;
public class RunnableImplementingClass implements Runnable {
  @Override
  public void run() {
    for (int i = 0; i < 10; i++) {
      System.out.println("RunnableImplementingClass: Thread: " +
        Thread.currentThread().getName() + ", i = " + i);
    }
  }
}
```

„run“ ist für eine Thread das, was „main“ für das Programm an sich ist

```
package de.baleipzig.threads;
public class ThreadDemo {
  public static void main(String[] args) {
    //...
    RunnableImplementingClass implTestClass
      = new RunnableImplementingClass();
    Thread testImplementingThread
      = new Thread(implTestClass, "implTestClass");
    testImplementingThread.start();
  }
}
```

wir rufen auch hier nicht „run“ auf, sondern starten den Thread mit „start“, wodurch „run“ ausgeführt wird

Thread-Erstellung: „Runnable“ implementieren

Aufgabe 2

„Runnable“ ist eine funktionale Schnittstelle, so dass man dafür einen Lambda-Ausdruck nutzen kann.

Erzeugen Sie einen weiteren Thread (analog zu den bisherigen Threads, jedoch noch ohne Vergabe eines speziellen Namens), der mit Hilfe eines Lambda-Ausdrucks erzeugt wird.

Zusatzaufgabe

Benutzen Sie auch für diesen Thread einen eigenen Namen und geben Sie diesen jeweils mit aus.

„Threads“

als Anbieter einer Form der konkurrenten Ausführung von Quellcode

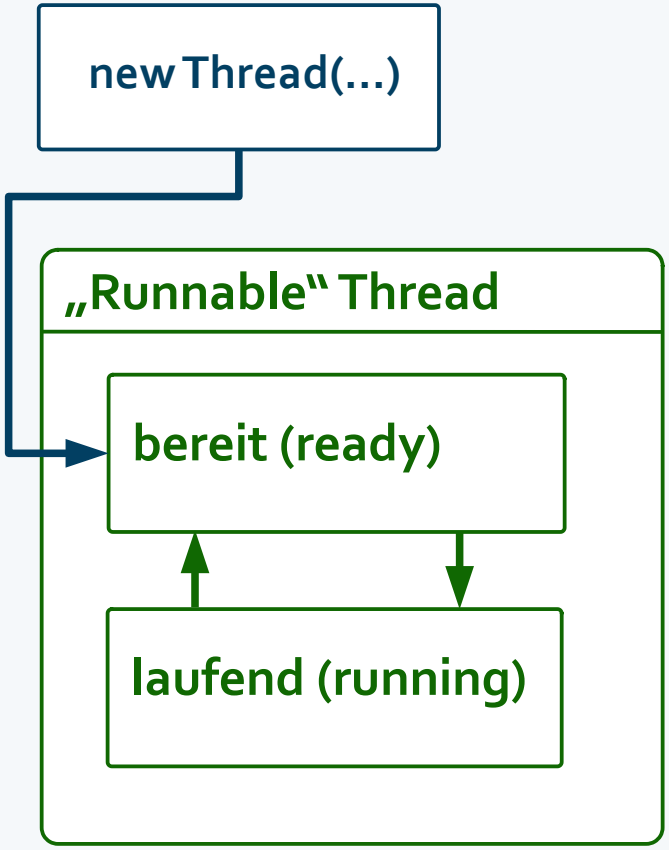
- Überblick
- Thread-Erstellung
- **Thread-Lebenszyklus**
- Threads: Konflikte, Interaktion, Sicherheits-überlegungen

Thread-Lebenszyklus

new Thread(...)

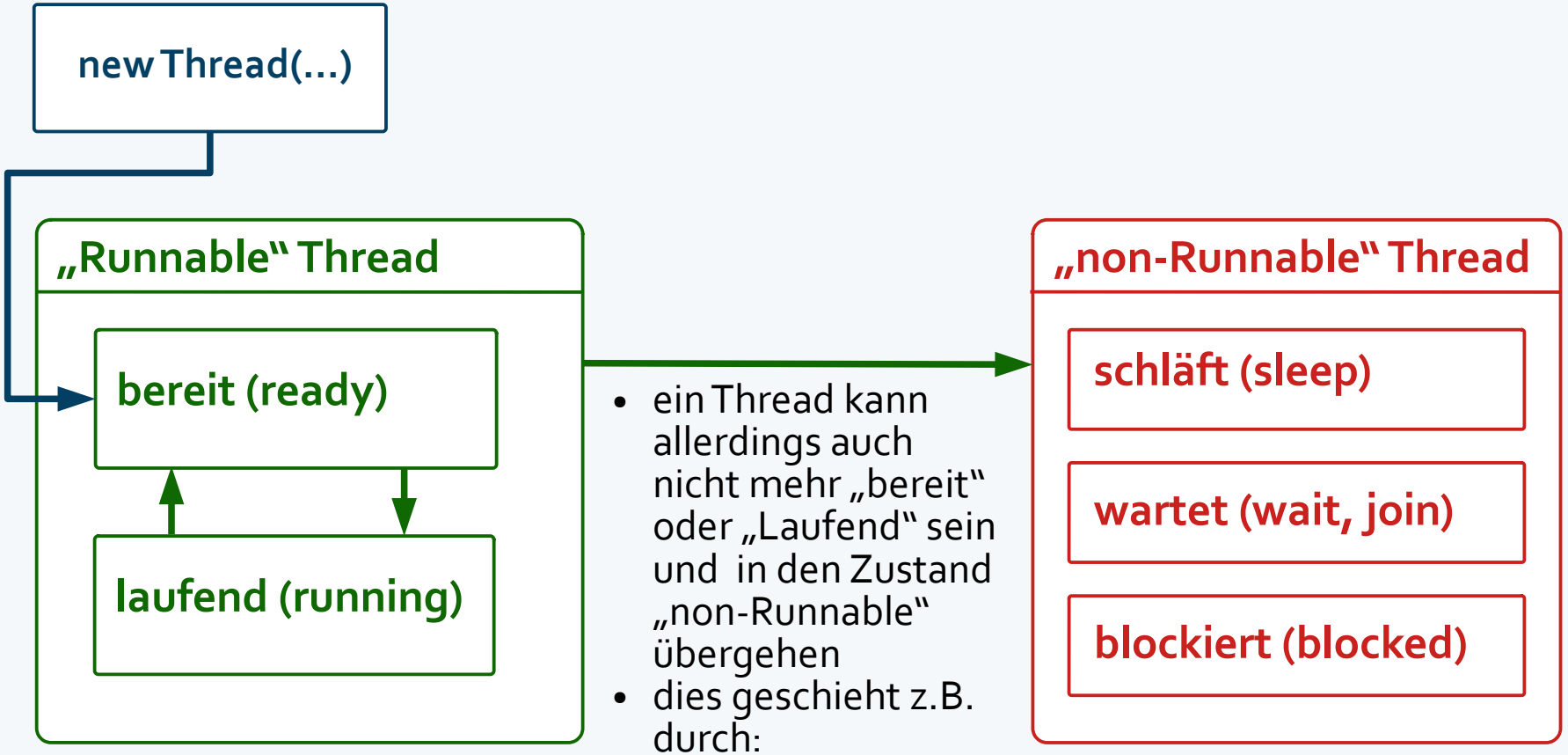
- zunächst muss überhaupt ein Thread erzeugt werden, bevor dieser gestartet werden kann

Thread-Lebenszyklus



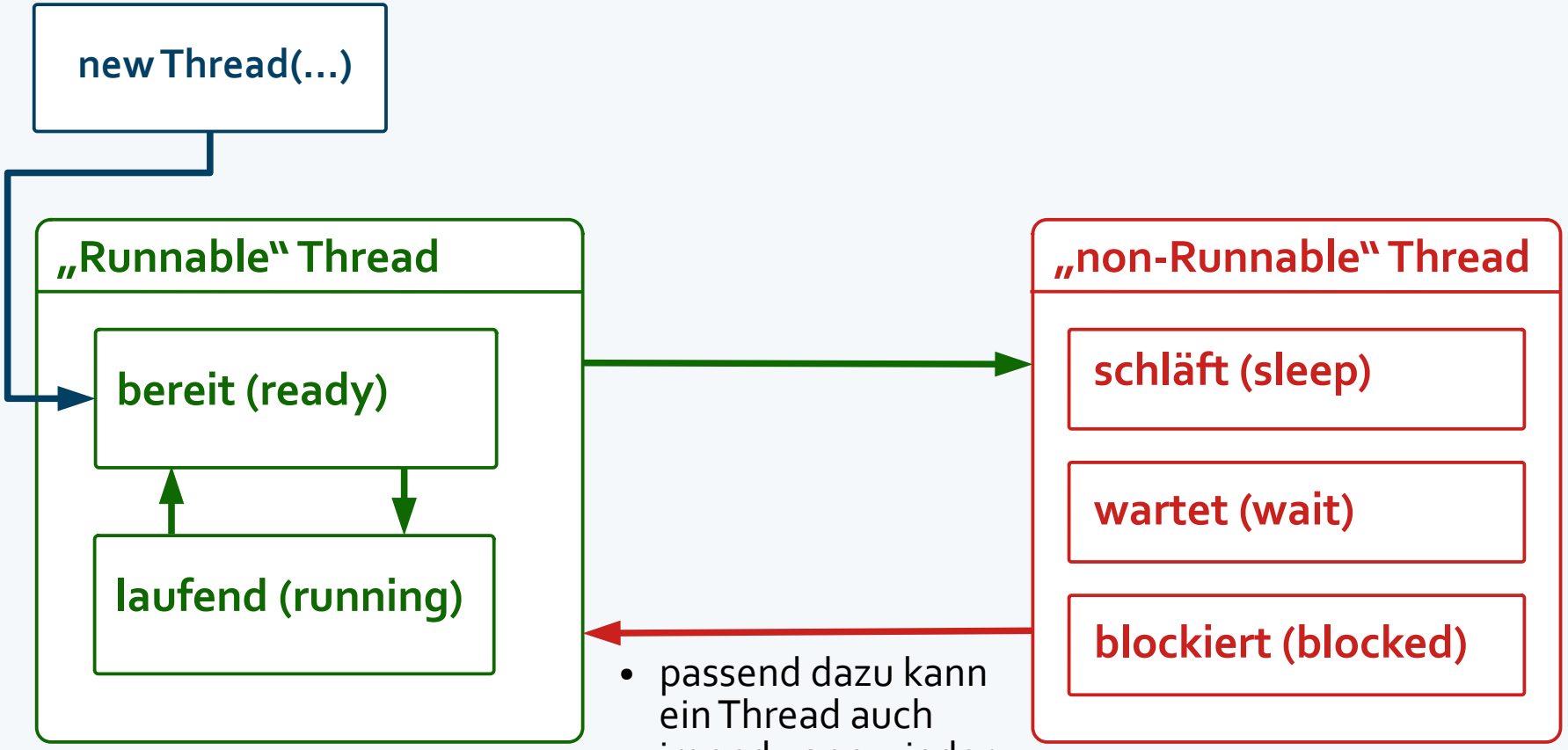
- wenn ein Thread gestartet wird („start"), dann reiht er sich zunächst unter die Threads ein, die potentiell ausgeführt werden können, ist also „Runnable" und dabei „bereit" für die Ausführung
- der Thread-Scheduler entscheidet dann, welcher Thread bzw. welche Threads aktuell ausgeführt werden sollen
- auf diese Weise kann ein Thread, wenn er zur Ausführung ausgewählt wird, ausgeführt werden und den Status „laufend" erhalten
- auch ohne sonstige Einflüsse kann ein Thread durch den Thread-Scheduler wieder angehalten und zurück in den Status „bereit" versetzt werden, wenn nun andere Threads mit der Ausführung dran sind

Thread-Lebenszyklus



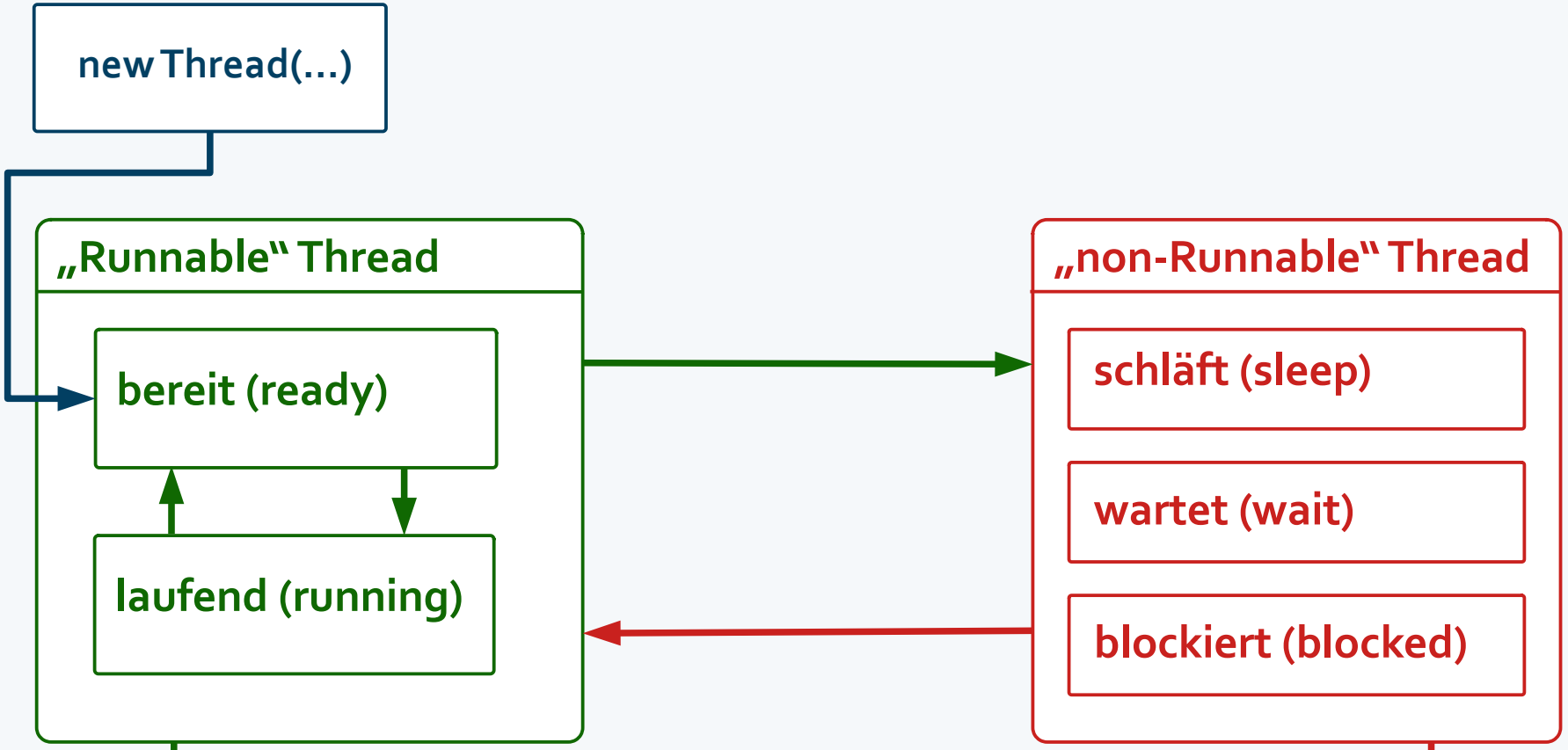
- ein Thread kann allerdings auch nicht mehr „bereit" oder „Laufend" sein und in den Zustand „non-Runnable" übergehen
- dies geschieht z.B. durch:
  - aktives Pausieren (z.B. sleep, wait, join) für eine gewisse Zeit, bis zu einer bestimmten Bedingung (z.B. Beendigung eines anderen Threads), bis zu einem Signal oder bis zur Verfügbarkeit einer bestimmten Ressource oder
  - durch eine anderweitige Blockade (z.B., wenn auf Eingabedaten gewartet werden muss und eine Weiterausführung ohne diese nicht möglich ist)

Thread-Lebenszyklus



- passend dazu kann ein Thread auch irgendwann wieder „Runnable" werden

Thread-Lebenszyklus



- ein Thread endet dann:
  - planmäßig nach Durchlaufen aller seiner Aufgaben oder
  - außerplanmäßig (z.B. bei Hauptprogrammende oder Abbruch, falls eine Bedingung für „Runnable" nicht rechtzeitig erfüllt wird)

Thread-Lebenszyklus

Aufgabe 3 – Vorbereitung

```
Thread testCycleThread = new Thread(() -> {
  System.out.println("testCycleThread state: running");
  try {
    Thread.sleep(5000);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
  System.out.println("testCycleThread state: completed");
});

testCycleThread.start();

try {
  Thread.sleep(1000);
} catch (InterruptedException e) {
  e.printStackTrace();
}

try {
  testCycleThread.join();
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

Thread-Lebenszyklus

Aufgabe 3

a) Schauen Sie in der Dokumentation nach, was die Methoden „sleep(x)“ und „join(x)“ bewirken.

b) Schauen Sie in der Dokumentation nach, ob Sie eine Methode finden, mit der Sie sich für unseren „testCycleThread“ den Status ausgeben lassen können. Nutzen Sie diese Methode und lassen Sie sich den Status mehrfach nach Erzeugung des Threads anzeigen.

„Threads“

als Anbieter einer Form der konkurrenten Ausführung von Quellcode

- Überblick
- Thread-Erstellung
- Thread-Lebenszyklus
- **Threads: Konflikte, Interaktion, Sicherheits-überlegungen**

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Frage: Wann ist Multi-Threading besonders problematisch?

Antwort: Wenn mehrere Threads auf die gleichen Ressourcen zugreifen.

"race condition" („Wettlaufsituation" [nicht als Fachbegriff])

- wird verwendet als Problembeschreibung: beschreibt das Problem, wenn das Ergebnis eines Programmdurchlaufs abhängig von der zeitlichen Abfolge oder dem Zusammentreffen von mehreren Threads oder Prozessen ist
- wird verwendet als Problemfall: tritt auf, wenn das korrekte Verhalten eines Programms von der Reihenfolge der Ausführung von Anweisungen abhängt und nicht durch Synchronisationsmechanismen oder andere Kontrollstrukturen gewährleistet ist, so dass Dateninkonsistenzen und andere unerwartete Effekte auftreten können
- Beispiel ist das sogenannte "Read-Modify-Write"-Problem, bei dem mehrere Threads gleichzeitig dieselbe Variable auslesen, modifizieren und zurückschreiben – die Threads führen bei der einzeiligen, scheinbar einfachen Aufgabe „x++" also mehrere atomaren (unteilbaren) Operationen aus, bei denen sie sich stören können

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Aufgabe 4 ("race condition")

```
package de.baleipzig.threads;
public class ThreadWithResource {
  private static int counter = 0;
  public static void main(String[] args) {
    try {
      aThread.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    try {
      anotherThread.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

a) Implementieren (mit Lambda-Ausdruck) und starten Sie die Threads „aThread“ und „anotherThread“.
Aufgabe der beiden Threads soll jeweils mindestens 1000 Mal „counter++“ sein.

b) Überlegen Sie sich, wie groß der „counter“ am Ende sein würde, wenn die Threads sequentiell ausgeführt werden würden.

c) Fügen Sie am Ende noch eine Ausgabe des „counter“ ein.

d) Lassen Sie Ihr Programm nun mehrfach laufen (bis Sie etwas bemerken ,-))

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

1. Konfliktvermeidung im Ganzen
   - Threads stören sich nicht
2. Konfliktvermeidung im Kleinen
   - Verwendung schon implementierter Sicherheitsmechanismen für kritische Thread-Abschnitte
   - ggf. aufwendige Synchronisation kritischer Thread-Abschnitte

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Ganzen

- Threads nur für isolierte Aufgabe, bei denen nicht auf gemeinsame Daten oder andere gemeinsame Ressourcen zugegriffen wird
- Threads greifen nur gemeinsam auf unveränderliche Daten/Datenstrukturen zu (Stichwort: „final“)
- Benutzung thread-lokaler Variabler, indem Threads nur Variablenkopien erhalten (Änderungen an der Kopie einer Variable in einem Thread haben keine Auswirkungen auf die Kopie einer Variable in einem anderen Thread)

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Konfliktvermeidung im Kleinen - „Monitore", „Locks", „Blocking Queues"

- um Ressourcen konkurrierende Threads nutzen einen gemeinsamen Speicher und darin gemeinsame Objekte, über deren Veränderung Sie auch ggf. ungewollt miteinander (indirekt) kommunizieren
- „kritischer Bereich" für eine korrekte Programmausführung ist also der gemeinsam genutzte Speicher, so dass der Zugriff darauf organisiert werden muss:
  Threads dürfen nur nacheinander auf diesen „kritischen Bereich" zugreifen, weil ein gleichzeitiger Zugriff offensichtlich problematisch ist.
- „Locks" (Sperren), ggf. unterstützt durch „Conditions", dienen dazu, den „kritischen Bereich" für andere Threads zu sperren, sobald ein Thread in diesem ist (Stichwort: „Mutex" [„mutual exclusion"])

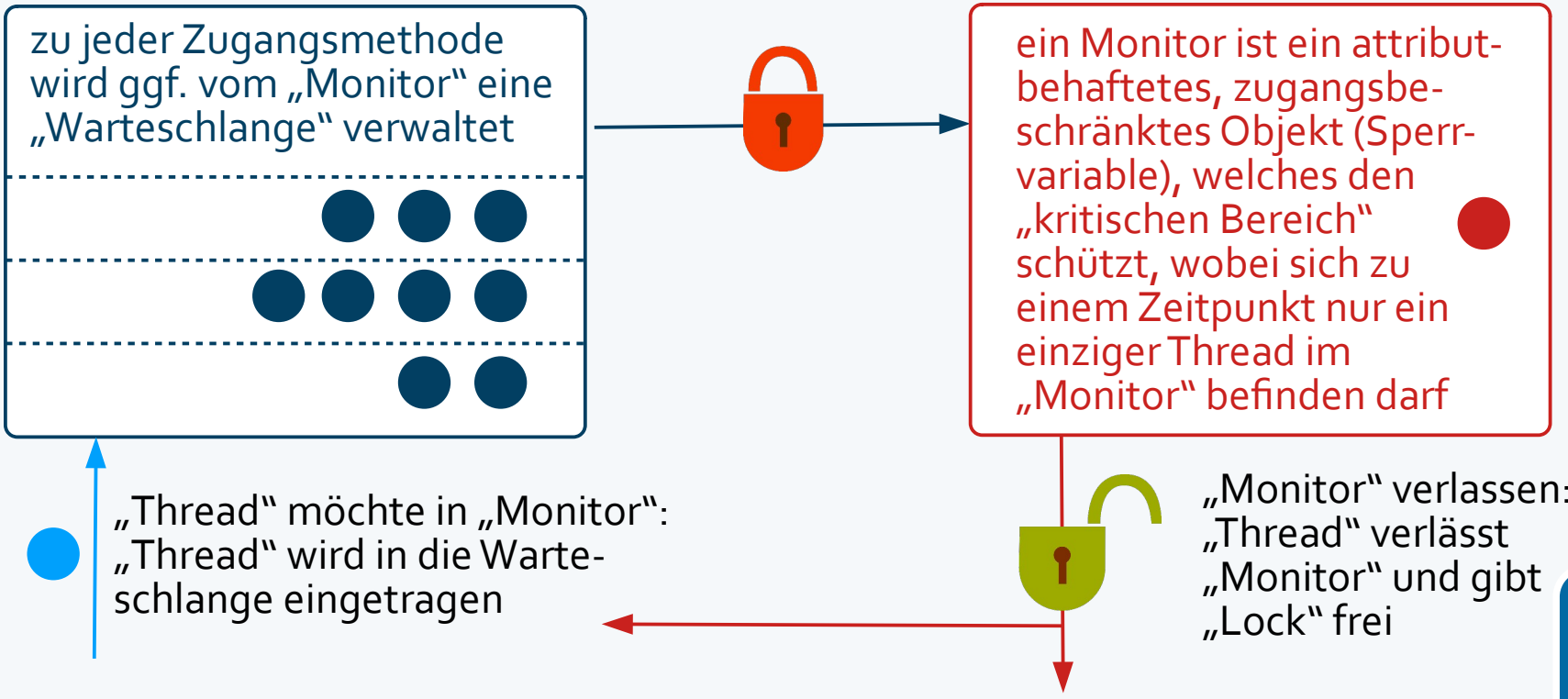
- nicht zu unterschätzen:
  - dieses Vorgehen muss gut überlegt sein, weil ein unsachgemäßer Einsatz obiger Methoden zu weiteren Problemen führt
  - nicht-funktionierende Programme mit mehreren Threads lassen sich nur sehr schwer debuggen

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Konfliktvermeidung im Kleinen - „Monitore", „Locks", „Blocking Queues"

„Monitor" (in dem Zusammenhang) ist ein Konzept zur Zugriffssteuerung auf Daten/Ressourcen nebenläufiger Prozesse oder Threads

„Monitor" betreten: wenn kein „Lock", betritt der erste „Thread" den „Monitor" durch Ausführung einer Zugangsmethode und setzt „Lock", um anzuzeigen, dass der „Monitor" besetzt ist



Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Kleinen

Verwendung schon implementierter Sicherheitsmechanismen

- Verwendung thread-sicherer Datenstrukturen, z.B. ConcurrentHashMap, ConcurrentLinkedQueue usw. (Paket: java.util.concurrent)
- Verwendung von Atomic-Variablen, bei denen mehrere Operationen als eine atomare Operation ausgeführt werden können, z.B. AtomicInteger usw. (Paket: java.util.concurrent.atomic)
- Benutzung thread-sicherer Bibliotheken für kritische Abschnitte
- Benutzung des Schlüsselwortes „volatile", um Variablenveränderungen in einem Thread in einem anderen sofort bekannt zu machen und basierend darauf Anwendung weiterer Synchronisationsverfahren

wir schauen uns in der Grundlagenvorlesung aus diesem Bereich nur den Teil an

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Kleinen

Beispiel: Verwendung von Atomic-Variablen

Aufgabe 5

Aufgabe 4, bei der es zu einer „race condition" kommen kann, soll zum Vergleich um eine sichere Alternative erweitert werden.

a) Deklarieren Sie eine weitere Variable „atomicCounter" vom Typ „AtomicInteger" (Hinweis: java.util.concurrent.atomic) und initialisieren Sie diese mit dem Wert 0 (Hinweis: Konstruktor benutzen).

b) Erhöhen Sie den Variablenwert analog zu „counter" in jedem Thread, indem Sie eine passende Methode von „AtomicInteger" nutzen.

c) Geben Sie zum Schluss analog zu „counter" den endgültigen Wert von „atomicCounter" zum Vergleich aus.

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Kleinen

ggf. aufwendige Synchronisation kritischer Thread-Abschnitte

- geschieht über die Benutzung von „Monitoren", „Locks" (Sperren) und dazugehörigen „Blocking Queues" (spez. Warteschlagen)
- sehr grobe Synchronisation durch Erzwingung sequentieller Ausführung durch das Warten auf das Ende anderer Threads („join")
- grobe Synchronisation durch „synchronized" Abschnitte oder „synchronized" Methoden
- verfeinerte Synchronisation durch Thread-Kommunikation über Ereignisse (wait, notify usw.)
- explizite Nutzung von „Locks" der Klasse „java.util.concurrent.locks.Lock"
- noch feinere Steuerung der „Locks" über „Conditions" der Klasse „java.util.concurrent.locks.Condition"

wir schauen uns in der Grundlagenvorlesung aus diesem Bereich nur den Teil an

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Kleinen

Beispiel: Erzwingung sequentieller Ausführung

Aufgabe 6a – Vorbereitung

Implementieren Sie eine Klasse „SequentialThreadDemo" mit

- einer statischen int-Variable „counter", die mit 0 initialisiert wird
- einer statischen Methode „addToCounter", die eine int-Variable entgegennimmt, um deren Wert „counter" erhöht werden soll, dann „counter" entsprechend erhöht und nichts zurück liefert
- einer statischen Methode „subtractFromCounter", die eine int-Variable entgegennimmt, um deren Wert „counter"gesenkt werden soll, dann „counter" entsprechend senkt und nichts zurück liefert
- der üblichen „main"-Methode

das sind Beispielmethoden, die in der Praxis kompliziertere Methoden repräsentieren, z. B. Einzahlungen auf ein und Auszahlungen von einem Konto

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Kleinen

Beispiel: Erzwingung sequentieller Ausführung

Aufgabe 6b – sequentiell ausgeführte Threads

Implementieren Sie nun in der „main"-Methode von „SequentialThreadDemo"

- einen Thread „addThread", der tausendmal „addToCounter(2)" aufruft
- einen Thread „subtractThread", der tausendmal „subtractToCounter(2)" aufruft
- nutzen Sie die Methoden „start" und „join" für die beiden Threads so, dass beide Threads nacheinander ausgeführt werden
- lassen Sie sich den Wert des „counter" am Ende anzeigen

hier sind die Threads an sich dadurch sinnlos; das würde Sinn machen, wenn es noch mehr Threads gäbe und manche davon mit manchen parallel ausgeführt werden könnten und nur bei bestimmten Konstellationen auf diese Art manuell eingegriffen werden müsste

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Kleinen

Beispiel: Nutzung von „synchronized"

Aufgabe 7a – Vorbereitung

Implementieren Sie eine Klasse „SynchronizedThreadDemo" mit

- einer statischen int-Variable „counter", die mit 0 initialisiert wird
- einer statischen Methode „addToCounter", die eine int-Variable entgegennimmt, um deren Wert „counter" erhöht werden soll, dann „counter" entsprechend erhöht und nichts zurück liefert
- einer statischen Methode „subtractFromCounter", die eine int-Variable entgegennimmt, um deren Wert „counter"gesenkt werden soll, dann „counter" entsprechend senkt und nichts zurück liefert
- der üblichen „main"-Methode

das sind Beispielmethoden, die in der Praxis kompliziertere Methoden repräsentieren, z. B. Einzahlungen auf ein und Auszahlungen von einem Konto

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Kleinen

Beispiel: Nutzung von „synchronized"

Aufgabe 7b – zunächst Threads mit problematischer „race condition"

Implementieren Sie nun in der „main"-Methode der Klasse

- einen Thread „addThread", der tausendmal „addToCounter(3)" aufruft
- einen Thread „subtractThread", der tausendmal „subtractToCounter(3)" aufruft
- nutzen Sie die Methoden „start" und „join" für die beiden Threads so, dass beide Threads parallel ausgeführt werden können
- lassen Sie sich den Wert des „counter" am Ende anzeigen

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Wie kann man mit Konflikten zwischen Threads umgehen?

Konfliktvermeidung im Kleinen

Beispiel: Nutzung von „synchronized"

Aufgabe 7c – Threads über „synchronized" sichern

Versuchen Sie, die Methoden „addToCounter" und „subtractFromCounter" mit Hilfe des Schlüsselwortes „synchronized" abzusichern.

- Finden Sie mit Hilfe Ihrer Entwicklungsumgebung heraus, an welcher Stelle bei der Methoden-Deklaration das Schlüsselwort „synchronized" eingebaut werden kann.
- Sichern Sie zunächst testweise nur eine der Methoden mit „synchronized" ab und schauen Sie sich zunächst an, inwieweit das ausreichend ist (und überlegen Sie natürlich auch, warum).
- Sichern Sie ggf. auch die andere der beiden Methoden mit „synchronized" ab.

Threads: Konflikte, Interaktion, Sicherheitsüberlegungen

Gefahrenüberblick bei der Verwendung von Threads

- Sicherheitsrisiko
  - Programm läuft mit mehreren Threads ggf. nicht korrekt
  - Stichwort „race condition"
- Laufzeitrisiko
  - Programm mit mehreren Threads gerät in einen ausweglosen Zustand ohne weitere Programmfortsetzung:
    - „Deadlock": alle warten gegenseitig auf unerfüllbare Bedingungen (im Stillstand gefangen)
    - „Livelock": es findet ein permanenter Wechsel zwischen mehreren Zuständen statt (im Kreis gefangen)
    - „Starvation" (Aushungern): andere Threads werden immer bevorzugt, so dass ein Thread nicht vorankommt
- Performancerisiko
  - Programm funktioniert korrekt, Performance ist aber trotz (wegen?) mehrere Threads schlecht

Threads: zu (fast) guter Letzt

das war nur ein kleiner Einblick zum Thema „Threads“, sowohl auf theoretischer Seite als auch auf praktischer Seite gibt es noch viel zu entdecken

- die Interaktion zwischen „Threads“ mittels „wait“, „notify“, „notifyAll“ bietet noch sehr viele Möglichkeiten
- „Threads“ haben noch veränderliche Prioritäten, können sich selbst als pausierend vorschlagen und sich selbst über Konstruktoren starten, …
- interessant sind „Threads“ inbesondere bei der GUI-Programmierung (bekommt jedes Fenster einen eigenen Thread?) und natürlich bei verteilten Systemen (die Probleme im Großen sind nicht anders als die Probleme im Kleinen – geteiltes Leid kann halbes oder doppeltes Leid sein)

JAVA

Threads: zu guter Letzt

Aufgabe 8 – ohne explizite Lösung

Schreiben Sie ein Java-Programm, in dem zwei Threads benutzt werden.

a) Ein Thread soll in einer Klasse durch Vererbung von „Thread" definiert werden und die Aufgabe des Threads ist es, innerhalb einer Schleife die Zahlen von 1 bis einschließlich 26 auszugeben.

b) Ein Thread soll in einer Klasse durch Implementierung von „Runnable" definiert werden und die Aufgabe des Threads ist es, innerhalb einer Schleife die Buchstaben (Zeichen) von ‚A' bis ‚Z' auszugeben.

c) Das Programm soll Instanzen beider Threads erzeugen und starten. Schauen Sie sich die dazugehörigen Ausgaben an.

d) Das Programm soll sicherstellen, dass erst alle Zahlen und danach die Buchstaben ausgegeben werden. Schauen Sie sich wieder alle Ausgaben an.

Vielen Dank für Ihre Aufmerksamkeit.