Nebenläufigkeit in Java

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

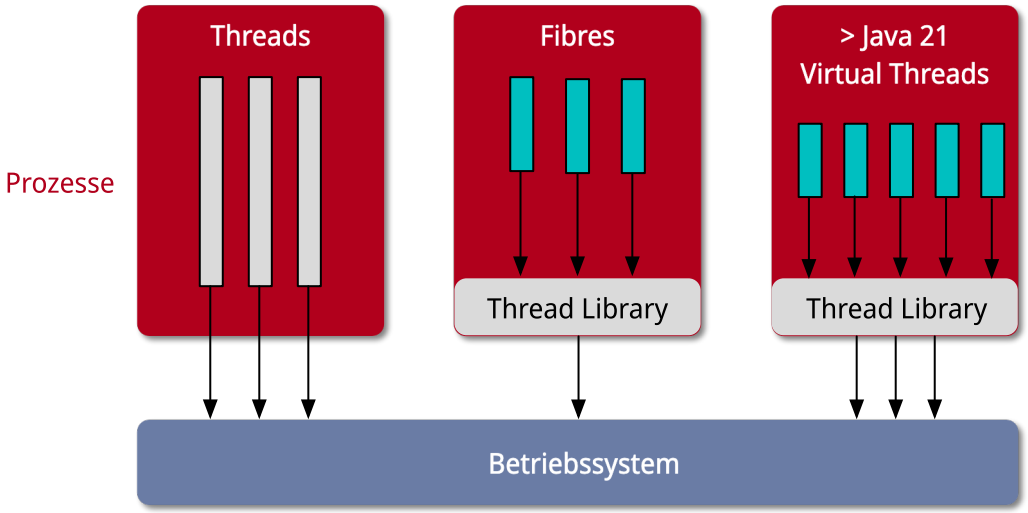
Concurrency in Java

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de
Version: 1.0.1.2

Folien: https://delors.github.io/ds-nebenlaeufigkeit-in-java/folien.de.md.html
https://delors.github.io/ds-nebenlaeufigkeit-in-java/folien.de.md.html.pdf
Fehler melden: https://github.com/Delors/delors.github.io/issues
KI Verwendung: Bei der Erstellung der Unterlagen wurden KI Assistenten (insbesondere Claude aber ggf. auch ChatGPT, Ollama mit Gemma/LLama/Qwen oder OpenCode mit Kimi/Qwen/Deepseek...) unterstützend eingesetzt. Dies erfolgte insbesondere zur Unterstützung bei der Generierung von Grafiken (d. h. SVG Dateien), oder um sich Übersichtstabellen generieren zu lassen. Weiterhin wurde KI zur allgemeinen Qualitätssicherung eingesetzt. Inhalte, die ggf. von der KI vorgeschlagen wurden, wurden im Falle der Übernahme explizit validiert und angepasst.

Ein gutes Verständnis von nebenläufiger Programmierung ist für die Entwicklung von verteilten Anwendungen unerlässlich, da Server immer mehrere Anfragen gleichzeitig bearbeiten.

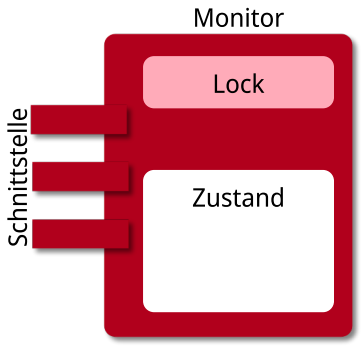
Prozesse vs. Threads



- Prozesse sind voneinander isoliert und können nur über explizite Mechanismen miteinander kommunizieren; Prozesse teilen sich nicht denselben Adressraum.
- Alle Threads eines Prozesses teilen sich denselben Adressraum. *Native Threads* sind vom Betriebssystem unterstützte Threads, die direkt vom Betriebssystem verwaltet werden. Standard Java Threads sind *Native Threads*.
- *Fibres* (auch *Coroutines*) nutzen immer kooperatives Multitasking. D. h. ein Fibre gibt die Kontrolle an eine andere Fibre explizit ab. (Früher auch als *Green Threads* bezeichnet.) Diese sind für das Betriebssystem unsichtbar.
- Ab Java 21 unterstützt Java nicht nur klassische (native) Threads sondern zusätzlich auf Virtual Threads. Letztere erlauben insbesondere eine sehr natürliche Programmierung von Middleware, die sich um die Parallelisierung/Nebenläufigkeit kümmert.

Kommunikation und Synchronisation mit Hilfe von *Monitoren*

Ein *Monitor* ist ein Objekt, bei dem die Methoden im wechselseitigen Ausschluss (engl. *mutual exclusion*) ausgeführt werden.



Bedingungs-Synchronisation

- drückt eine Bedingung für die Reihenfolge der Ausführung von Operationen aus.
- z. B. können Daten erst dann aus einem Puffer entfernt werden, wenn Daten in den Puffer eingegeben wurden.
- Java unterstützt pro Monitor nur eine (anonyme) Bedingungs-Variable, mit den klassischen Methoden `wait` und `notify` bzw. `notifyAll`.

⚠ Warnung

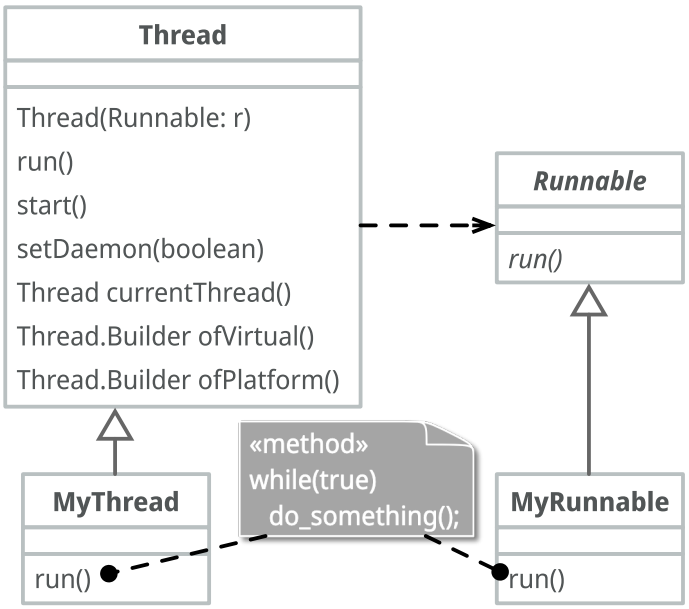
In Java findet der wechselseitige Ausschluss nur zwischen solchen Methoden statt, die explizit als `synchronized` deklariert wurden.

Monitore sind nur ein Modell (Alternativen: *Semaphores*, *Message Passing*), das die Kommunikation und Synchronisation von Threads ermöglicht. Es ist das Standardmodell in Java und wird von der Java Virtual Machine (JVM) unterstützt.

Kommunikation zwischen Threads mit Hilfe von Monitoren

- Durch Lesen und Schreiben von Daten, die in gemeinsamen Objekten gekapselt sind, die durch Monitore geschützt werden.
- Jedes Objekt ist implizit von der Klasse `Object` abgeleitet, welche eine gegenseitige Ausschluss-sperre definiert.
- Methoden in einer Klasse können als `synchronized` gekennzeichnet werden. Die Methode wird erst dann ausgeführt, wenn die Sperre vorliegt. Bis dahin wird gewartet. Dieser Prozess geschieht automatisch.
- Die Sperre kann auch über eine `synchronized` Anweisung erworben werden, die das Objekt benennt.
- Ein Thread kann auf eine einzelne (anonyme) Bedingungsvariable warten und diese benachrichtigen.

Nebenläufigkeit in Java



- Threads werden in Java über die vordefinierte Klasse java.lang.Thread bereitgestellt.
- Alternativ kann das Interface:

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

  implementiert werden und an ein Thread-Objekt übergeben werden.
- Threads beginnen ihre Ausführung erst, wenn die `start`-Methode in der Thread-Klasse aufgerufen wird. Die `Thread.start`-Methode ruft die `run`-Methode auf. Ein Aufruf der `run`-Methode direkt führt nicht zu einer parallelen Ausführung.
- Der aktuelle Thread kann mittels der statischen Methode `Thread.currentThread()` ermittelt werden.
- Ein Thread wird beendet, wenn die Ausführung seiner Run-Methode entweder normal oder als Ergebnis einer unbehandelten Ausnahme endet.
- Java unterscheidet *User*-Threads und *Daemon*-Threads.

  Daemon-Threads sind Threads, die allgemeine Dienste bereitstellen und normalerweise nie beendet werden.

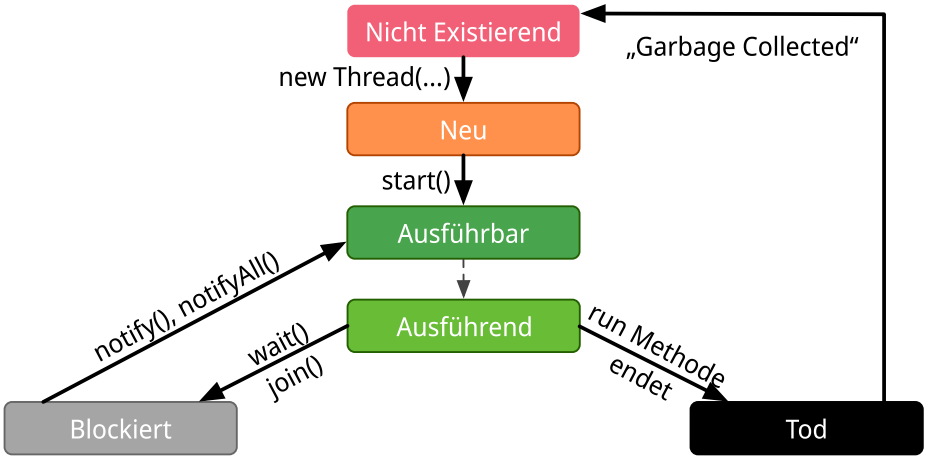
  Wenn alle Benutzer-Threads beendet sind, werden die Daemon-Threads von der JVM beendet, und das Hauptprogramm wird beendet.

  Die Methode `setDaemon` muss aufgerufen werden, bevor der Thread gestartet wird.

Inter-Thread-Kommunikation bzw. Koordination

- Ein Thread kann (mit oder ohne Zeitüberschreitung) auf die Beendigung eines anderen Threads (des Ziels) warten, indem er die `join`-Methode für das Thread-Objekt des Ziels aufruft.
- Mit der Methode `isAlive` kann ein Thread feststellen, ob der Ziel-Thread beendet wurde.

Java Thread States



`synchronized`-Methoden und `synchronized`-Blöcke

- Jedem Objekt ist eine gegenseitige Ausschlusssperre zugeordnet. Auf die Sperre kann von der Anwendung nicht explizit zugegriffen werden. Dies geschieht implizit, wenn:
  - eine Methode den Methodenmodifikator `synchronized` verwendet
  - Blocksynchronisierung mit dem Schlüsselwort `synchronized` verwendet wird
- Wenn eine Methode als synchronisiert gekennzeichnet ist, kann der Zugriff auf die Methode nur erfolgen, wenn das System die Sperre erhalten hat.
- Daher haben synchronisierte Methoden einen sich gegenseitig ausschließenden Zugriff auf die vom Objekt gekapselten Daten, wenn auf diese Daten nur von anderen synchronisierten Methoden zugegriffen wird.
- Nicht-synchronisierte Methoden benötigen keine Sperre und können daher *jederzeit* aufgerufen werden.

Beispiel: Synchronisierte Methode

```
public class SynchronizedCounter {

  private int count = 0;

  public synchronized void increment() {
    count++;
  }

  public synchronized int getCount() {
    return count;
  }
}
```

```
public class SharedLong {

  private long theData; // reading and writing longs is not atomic

  public SharedLong(long initialValue) {
    theData = initialValue;
  }

  public synchronized long read() { return theData; }

  public synchronized void write(long newValue) { theData = newValue; }

  public synchronized void incrementBy(long by) {
    theData = theData + by;
  }
}

  SharedLong myData = new SharedLong(42);
```

```
public class SynchronizedCounter {

  private int count = 0;

  public void increment() {
    synchronized(this) {
      count++;
    }
  }

  public int getCount() {
    synchronized(this) {
      return count;
    }
  }
}
```

⚠ Warnung

Wenn `synchronized` in seiner ganzen Allgemeinheit verwendet wird, kann er einen der Vorteile von klassischen Monitoren untergraben: Die Kapselung von Synchronisationseinschränkungen, die mit einem Objekt verbunden sind, an einer einzigen Stelle im Programm!

Dies liegt daran, dass es nicht möglich ist, die mit einem bestimmten Objekt verbundene Synchronisation zu verstehen, indem man sich nur das Objekt selbst ansieht. Andere Objekte können bgzl. des Objekts eine `synchronized`-Block verwenden.

Komplexe Rückgabewerte

```
1  public class SharedCoordinate {
2
3    private int x, y;
4
5    public SharedCoordinate(int initX, int initY) {
6      this.x = initX; this.y = initY;
7    }
8
9    public synchronized void write(int newX, int newY) {
10     this.x = newX; this.y = newY;
11   }
12
13   /*⚠*/ public /* synchronized irrelevant */ int readX() { return x; } /*⚠*/
14   /*⚠*/ public /* synchronized irrelevant */ int readY() { return y; } /*⚠*/
15
16   public synchronized SharedCoordinate read() {
17     return new SharedCoordinate(x, y);
18 } }
```

Die beiden Methoden: `readX` und `readY` sind nicht synchronisiert, da das Lesen von `int`-Werten atomar ist. Allerdings erlauben sie das Auslesen eines inkonsistenten Zustands! Es ist denkbar, dass direkt nach einem `readX` der entsprechende Thread unterbrochen wird und ein anderer Thread die Werte von x und y verändert. Wird dann der ursprüngliche Thread fortgesetzt, und ruft `readY` auf, so erhält er den neuen Wert von y und hat somit ein paar x, y vorliegen, dass in dieser Form nie existiert hat.

Ein konsistenter Zustand kann nur durch die Methode `read` ermittelt werden, die die Werte von x und y in einem Schritt ausliest und als Paar zurückgibt.

Kann sichergestellt werden, dass ein auslesender Thread die Instanz in einem `synchronized` Block benennt, dann kann die Auslesung eines konsistenten Zustands auch bei mehreren Methodenaufrufen hintereinander sichergestellt werden.

```
1 SharedCoordinate point = new SharedCoordinate(0,0);
2 synchronized (point1) {
3   var x = point1.readX();
4   var y = point1.readY();
5 }
6 // do something with x and y
```

Diese „Lösung“ muss jedoch als sehr kritisch betrachtet werden, da die Wahrscheinlichkeit von Programmierfehlern *sehr hoch* ist und es dann entweder zur *Race Conditions* oder zu *Deadlocks* kommen kann.

Bedingte Synchronisation

Zum Zwecke der bedingten Synchronisation können in Java die Methoden `wait`, `notify` und `notifyAll` verwendet werden. Diese Methoden erlauben es auf bestimmte Bedingungen zu warten und andere Threads zu benachrichtigen, wenn sich die Bedingung geändert hat.

- Diese Methoden können nur innerhalb von Methoden verwendet werden, die die Objektsperre halten; andernfalls wird eine `IllegalMonitorStateException` ausgelöst.
- Die `wait`-Methode blockiert immer den aufrufenden Thread und gibt die mit dem Objekt verbundene Sperre frei.
- Die `notify`-Methode weckt *einen* wartenden Thread auf. Welcher Thread aufgeweckt wird, ist nicht spezifiziert.

  `notify` gibt die Sperre nicht frei; daher muss der aufgeweckte Thread warten, bis er die Sperre erhalten kann, bevor er fortfahren kann.
- Um alle wartenden Threads aufzuwecken, muss die Methode `notifyAll` verwendet werden.

  Warten die Threads aufgrund unterschiedlicher Bedingungen, so ist immer `notifyAll` zu verwenden.
- Wenn kein Thread wartet, dann haben `notify` und `notifyAll` keine Wirkung.

!! Wichtig

Wenn ein Thread aufgeweckt wird, kann er nicht davon ausgehen, dass seine Bedingung erfüllt ist!

Die Bedingung ist immer in einer Schleife zu prüfen und der Thread muss ich ggf. wieder in den Wartezustand versetzen.

Beispiel: Synchronisation mit *Condition Variables*

Wenn ein Thread auf eine Bedingung wartet, kann kein anderer Thread auf die andere Bedingung warten.

Mit den bisher vorgestellten Primitiven ist eine direkte Modellierung dieses Szenarios so nicht möglich. Stattdessen müssen immer alle Threads aufgeweckt werden, um sicherzustellen, dass auch der intendierte Thread aufgeweckt wird. Deswegen ist auch das Überprüfen der Bedingung in einer Schleife notwendig.

Ein *BoundedBuffer* hat z. B. traditionell zwei Bedingungsvariablen: *BufferNotFull* und *BufferNotEmpty*.

```
1  public class BoundedBuffer {
2    private final int buffer[];
3    private int first;
4    private int last;
5    private int numberInBuffer = 0;
6    private final int size;
7
8    public BoundedBuffer(int length) {
9      size = length;
10     buffer = new int[size];
11     last = 0;
12     first = 0;
13   };
14   public synchronized void put(int item) throws InterruptedException {
15     while (numberInBuffer == size)
16       wait();
17     last = (last + 1) % size;
18     numberInBuffer++;
19     buffer[last] = item;
20     notifyAll();
21   };
22   public synchronized int get() throws InterruptedException {
23     while (numberInBuffer == 0)
24       wait();
25     first = (first + 1) % size;
26     numberInBuffer--;
27     notifyAll();
28     return buffer[first];
29   }
30 }
```

Fehlersituation, die bei der Verwendung von `notify` statt `notifyAll` auftreten könnte:

```
1 BoundedBuffer bb = new BoundedBuffer(1);
2 Thread g1,g2 = new Thread(() => { bb.get(); });
3 Thread p1,p2 = new Thread(() => { bb.put(new Object()); });
4 g1.start(); g2.start(); p1.start(); p2.start();
```

	Aktionen	(Änderung des) Zustand(s) des Buffers	Auf die Sperre (*Lock*) wartend	An der Bedingung wartend
1	**g1:bb.get()** g2:bb.get(), p1:bb.put(), p2:bb.put()	empty	{g2,p1,p2}	{g1}

	Aktionen	(Änderung des) Zustand(s) des Buffers	Auf die Sperre (*Lock*) wartend	An der Bedingung wartend
2	**g2:bb.get()**	empty	{p1,p2}	{g1,g2}
3	**p1:bb.put()**	empty → not empty	{p2,g1}	{g2}
4	**p2:bb.put()**	not empty	{g1}	{g2,p2}
5	**g1:bb.get()**	not empty → empty	{g2}	{p2}
6	**g2:bb.get()**	empty	∅	{g2,p2}

In Schritt 5 wurde von der VM - aufgrund des Aufrufs von `notify` durch `g1` - der Thread `g2` aufgeweckt - anstatt des Threads `p2`. Der aufgeweckte Thread `g2` prüft die Bedingung (Schritt 6) und stellt fest, dass der Buffer leer ist. Er geht wieder in den Wartezustand. Jetzt warten sowohl ein Thread, der ein Wert schreiben möchte als auch ein Thread, der einen Wert lesen möchte.

1. Fortgeschrittene Synchronisationsmechanismen, -primitive und -konzepte.

Java API für nebenläufige Programmierung

java.util.concurrent:

Bietet verschiedene Klassen zur Unterstützung gängiger nebenläufiger Programmierparadigmen, z. B. Unterstützung für *BoundedBuffers* oder Thread-Pools.

java.util.concurrent.atomic:

Bietet Unterstützung für sperrfreie (*lock-free*), thread-sichere Programmierung auf einfachen Variablen — wie zum Beispiel atomaren Integern — an.

java.util.concurrent.locks:

Bietet verschiedene Sperralgorithmen an, die die Java-Sprachmechanismen ergänzen, z. B. Schreib-Lese-Sperren und Bedingungsvariablen. Dies ermöglicht zum Beispiel: „Hand-over-Hand“ oder „Chain Locking“.

Beispiel: Synchronisation mit *ReentrantLock*s.

Ein *BoundedBuffer* hat z. B. traditionell zwei Bedingungsvariablen: *BufferNotFull* und *BufferNotEmpty*.

```
1  public class BoundedBuffer<T> {
2
3    private final T buffer[];
4    private int first;
5    private int last;
6    private int numberInBuffer;
7    private final int size;
8
9    private final Lock lock = new ReentrantLock();
10   private final Condition notFull = lock.newCondition();
11   private final Condition notEmpty = lock.newCondition();
12   public BoundedBuffer(int length) { /* Normaler Constructor. */
13     size = length;
14     buffer = (T[]) new Object[size];
15     last = 0;
16     first = 0;
17     numberInBuffer = 0;
18   }
19   public void put(T item) throws InterruptedException {
20     lock.lock();
21     try {
22
23       while (numberInBuffer == size) { notFull.await(); }
24       last = (last + 1) % size;
25       numberInBuffer++;
26       buffer[last] = item;
27       notEmpty.signal();
28
29     } finally {
30       lock.unlock();
31     }
32   }
33   public T get() ... {
34     lock.lock();
35     try {
36
37       while (numberInBuffer == 0) { notEmpty.await(); }
38       first = (first + 1) % size;
39       numberInBuffer--;
40       notFull.signal();
41       return buffer[first];
42
43     } finally {
44       lock.unlock();
45     }
46   }
47 }
```

Thread Prioritäten

- Obwohl den Java-Threads Prioritäten zugewiesen werden können (`setPriority`), dienen sie dem zugrunde liegenden Scheduler nur als Richtschnur für die Ressourcenzuweisung.
- Sobald ein Thread läuft, kann er die Prozessorressourcen explizit aufgeben, indem er die `yield`-Methode aufruft.
- `yield` setzt den Thread an das Ende der Warteschlange für seine Prioritätsstufe.
- Die Scheduling- und Prioritätsmodelle von Java sind jedoch schwach:
  - Es gibt keine Garantie dafür, dass immer der Thread mit der höchsten Priorität ausgeführt wird, der lauffähig ist.
  - Threads mit gleicher Priorität können in Zeitscheiben unterteilt sein oder auch nicht.
  - Bei der Verwendung nativer Threads können unterschiedliche Java-Prioritäten auf dieselbe Betriebssystempriorität abgebildet werden.

Best Practices

!! Wichtig

- `synchronized` Code sollte so kurz wie möglich gehalten werden.
- Verschachtelte Monitoraufrufe sollten vermieden werden, da die äußere Sperre nicht freigegeben wird, wenn der innere Monitor wartet. Dies kann leicht zum Auftreten eines Deadlocks führen.

⚠ Warnung

Wenn zwei (oder mehr) Threads auf die gleichen Ressourcen in unterschiedlicher Reihenfolge zugreifen, kann es zu einem Deadlock kommen.

!! Wichtig

Ressourcen sind immer in der gleichen Reihenfolge zu sperren, um Deadlocks zu vermeiden.

2. Thread Safety

Threadsicherheit

Thread Safety - Voraussetzung

Damit eine Klasse thread-sicher ist, muss sie sich in einer single-threaded Umgebung korrekt verhalten.

D. h. wenn eine Klasse korrekt implementiert ist, dann sollte keine Abfolge von Operationen (Lesen oder Schreiben von öffentlichen Feldern und Aufrufen von öffentlichen Methoden) auf Objekten dieser Klasse in der Lage sein:

- das Objekt in einen ungültigen Zustand versetzen,
- das Objekt in einem ungültigen Zustand zu beobachten oder
- eine der Invarianten, Vorbedingungen oder Nachbedingungen der Klasse verletzen.

Die Klasse muss das korrekte Verhalten auch dann aufweisen, wenn auf sie von mehreren Threads aus zugegriffen wird.

- Unabhängig vom *Scheduling* oder der Verschachtelung der Ausführung dieser Threads durch die Laufzeitumgebung,
- Ohne zusätzliche Synchronisierung auf Seiten des aufrufenden Codes.

⚑ Bewertung

Dies hat zur Folge, dass Operationen auf einem thread-sicheren Objekt für alle Threads so erscheinen als ob die Operationen in einer festen, global konsistenten Reihenfolge erfolgen würden.

Thread Safety Level

Immutable *Unveränderlich*:
: Die Objekte sind konstant und können nicht geändert werden.

Thread-sicher:
: Die Objekte sind veränderbar, unterstützen aber nebenläufigen Zugriff, da die Methoden entsprechend synchronisiert sind.

Bedingt Thread-sicher:
: All solche Objekte bei denen jede einzelne Operation thread-sicher ist, aber bestimmte Sequenzen von Operationen eine externe Synchronisierung erfordern können.

Thread-kompatibel:
: Alle Objekte die keinerlei Synchronisierung aufweisen. Der Aufrufer kann die Synchronisierung jedoch ggf. extern übernehmen.

Thread-hostile „Thread-schädlich“:
: Objekte, die nicht thread-sicher sind und auch nicht thread-sicher gemacht werden können, da sie zum Beispiel globalen Zustand manipulieren.

✎ Übung

2.1. Virtueller Puffer

Implementieren Sie einen virtuellen Puffer, der Tasks (Instanzen von `java.lang.Runable`) entgegennimmt und nach einer bestimmten Zeit ausführt. Der Puffer darf währenddessen nicht blockieren bzw. gesperrt sein.

Nutzen Sie ggf. virtuelle Threads, um auf ein explizites Puffern zu verzichten. Ein virtueller Thread kann zum Beispiel mit: `Thread.ofVirtual()` erzeugt werden. Danach kann an die Methode `start` ein `Runnable` Objekt übergeben werden.

Verzögern Sie die Ausführung (`Thread.sleep()`) im Schnitt um 100ms mit einer Standardabweichung von 20ms. (Nutzen Sie `Random.nextGaussian(mean,stddev)`)

Starten Sie 100 000 virtuelle Threads. Wie lange dauert die Ausführung? Wie lange dauert die Ausführung bei 100 000 platform (*native*) Threads.

Nutzen Sie ggf. die Vorlage.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class VirtualBuffer {

  private final Random random = new Random();

  private Thread runDelayed(int id, Runnable task) {
    // TODO
  }

  public static void main(String[] args) throws Exception {
    var start = System.nanoTime();
    VirtualBuffer buffer = new VirtualBuffer();
    List<Thread> threads = new ArrayList<>();
    for (int i = 0; i < 100000; i++) {
      final var no = i;
      var thread = buffer.runDelayed(
          i,
          () -> System.out.println("i'm no.: " + no));
      threads.add(thread);
    }
    System.out.println("finished starting all threads");
    for (Thread thread : threads) {
      thread.join();
    }
    var runtime = (System.nanoTime() - start)/1_000_000;
    System.out.println(
      "all threads finished after: " + runtime + "ms"
    );
  }
}
```

✎ Übung

2.2. Thread-sichere Programmierung

Implementieren Sie eine Klasse `ThreadsafeArray` zum Speichern von nicht-`null` Objekten (*java.lang.Object*) an ausgewählten Indizes — vergleichbar mit einem normalen Array. Im Vergleich zu einem normalen Array sollen die Aufrufer jedoch ggf. blockiert werden, wenn die Zelle belegt ist. Die Klasse soll folgende Methoden bereitstellen:

get`(int index)`: Liefert den Wert an der Position `index` zurück. Der aufrufende Thread wird ggf. blockiert, bis ein Wert an der Position `index` gespeichert wurde. (Die `get`-Methode entfernt den Wert nicht aus dem Array.)

set`(int index, Object value)`: Speichert den Wert `value` an der Position `index`. Falls an der Position `index` bereits ein Wert gespeichert wurde, wird der aufrufende Thread blockiert, bis der Wert an der Position `index` gelöscht wurde.

delete`(int index)`: Löscht ggf. den Wert an der Position `index` wenn ein Wert vorhanden ist. Andernfalls wird der Thread blockiert, bis es einen Wert gibt, der gelöscht werden kann.

a. Implementieren Sie die Klasse `ThreadsafeArray` nur unter Verwendung der Standardprimitive: `synchronized`, `wait`, `notify` und `notifyAll`. Nutzen Sie die Vorlage.

b. Können Sie sowohl `notify` als auch `notifyAll` verwenden?

c. Implementieren Sie die Klasse `ThreadsafeArray` unter Verwendung von `ReentrantLock`s und `Condition`s. Nutzen Sie die Vorlage.

d. Welche Vorteile hat die Verwendung von `ReentrantLock`s?

Sie können sich die Klasse `ThreadsafeArray` auch als ein Array von *BoundedBuffers* mit der Größe 1 vorstellen.

```
1  public class ThreadsafeArray {
2
3    private final Object[] array;
4
5    public ThreadsafeArray(int size) {
6      this.array = new Object[size];
7    }
8
9    // Methodensignaturen ggf. vervollständigen
10   // und Implementierungen ergänzen
11   Object get(int index)
12   void set(int index, Object value)
13   void remove(int index)
14
15   public static void main(String[] args) throws Exception {
16     final var ARRAY_SIZE = 2;
17     final var SLEEP_TIME = 1; // ms
18     var array = new ThreadsafeArray(ARRAY_SIZE);
19     for (int i = 0; i < ARRAY_SIZE; i++) {
20       final var threadId = i;
21
22       final var readerThreadName = "Reader";
```

```
23        var t2 = new Thread(() -> {
24          while (true) {
25            int j = (int) (Math.random() * ARRAY_SIZE);
26            try {
27              out.println(readerThreadName + "[" + j + "]" );
28              var o = array.get(j);
29              out.println(readerThreadName +
30                  "[" + j + "] ⇒ #" + o.hashCode());
31              Thread.sleep(SLEEP_TIME);
32            } catch (InterruptedException e) {
33              e.printStackTrace();
34            }
35          }
36        }, readerThreadName);
37        t2.start();
38
39        // One Thread for each slot that will eventually
40        // write some content
41        final var writerThreadName = "Writer[" + threadId + "]";
42        var t1 = new Thread(() -> {
43          while (true) {
44            try {
45              var o = new Object();
46              out.println(writerThreadName + " = #" + o.hashCode());
47              array.set(threadId, o);
48              out.println(writerThreadName + " done");
49              Thread.sleep(SLEEP_TIME);
50            } catch (InterruptedException e) {
51              e.printStackTrace();
52            }
53          }
54        }, writerThreadName);
55        t1.start();
56
57        // One Thread for each slot that will eventually
58        // delete the content
59        final var deleterThreadName = "Delete[" + threadId + "]";
60        var t3 = new Thread(() -> {
61          while (true) {
62            try {
63              out.println(deleterThreadName);
64              array.delete(threadId);
65              Thread.sleep(SLEEP_TIME);
66            } catch (InterruptedException e) {
67              e.printStackTrace();
68            }
69          }
70        }, deleterThreadName);
71        t3.start();
72      }
73    }
74  }
```