

Übungsaufgaben Gesamtsicht

Kapitel 1: Programmiermodell und Konzept der Virtual-Threads

- [V-T 1: Virtual-Threads starten und auf sie warten](#)
- [V-T 2: Blocking-Call gibt Carrier-Thread frei](#)

Einfach

Mittel

Einfach

Mittel

Schwer

Kapitel 2: Perfomance und Skalierbarkeit

- [V-T 3: Skalierbarkeit bei Blocking-Calls](#)
- [V-T 4: Performance bei Blocking-Calls](#)
- [V-T 15: Server-Skalierbarkeit](#)

Kapitel 3: Technisches Fundament, Blocking, Realesing

- [V-T 6: Lock.lock Blocking Verhalten](#)
- [V-T 8: Queue: Blocking-Verhalten](#)

Einfach

Mittel

Mittel

Kapitel 4: Anwendungsarchitektur und Technologieauswahl

- [V-T 10 ForkJoinTasks vs. Virtual-Threads](#)

Einfach

Mittel

Schwer

Einfach

Mittel

Mittel

Kapitel 5: Structured Concurrency, ScopedValues

- [S-C 01 Scope based Structured Concurrency](#)
- [S-C 02 Sub-Scopes](#)
- [S-C 03 Joiner](#)
- [S-C 00 Scope Values Einfaches Beispiel](#)
- [S-V 01 Scoped Values Basic API](#)
- [S-V 02 Scoped Values Threading](#)

Kapitel 6: Architektur und Probleme der Virtual-Threads

- [V-T 13: Preemption und Programmsemantik](#)
- [V-T 14: Thread-Pool Konfiguration](#)

Übung V-T 1

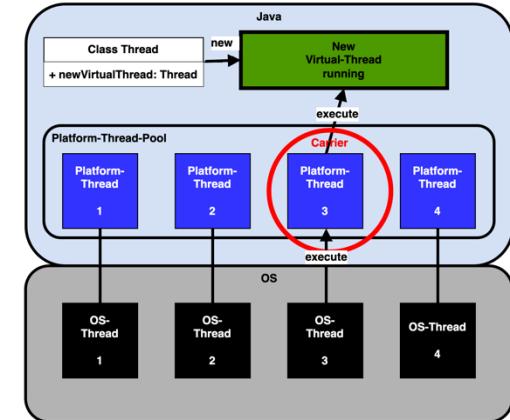
Virtual-Threads starten und auf sie warten

Einfach

SIEMENS

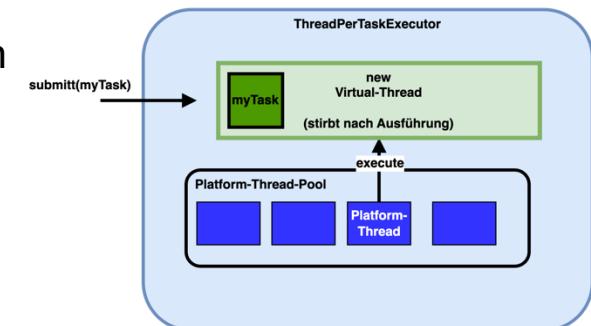
1. Starte einen einzelnen Virtual-Thread

- Nutze eine statische Methode der Klasse Thread, um einen Virtual-Thread zu starten
- Schreibe aus dem Virtual-Thread heraus dessen ID auf die Konsole
- Warte auf das Ende des Virtual-Threads



2. Nutze einen Virtual-Thread mithilfe eines ExecutorService

- Verwende Executors.newVirtualThreadPerTaskExecutor(), um einen geeigneten ExecutorService zu erhalten
- Starte im ExecutorService einen Virtual-Thread und schreibe dessen String- Repräsentation auf die Konsole
- Warte auf das Ende der Ausführung des Threads, nutze dafür ein Future Objekt



- **Lernziel:** Virtual-Threads starten, auf sie warten und sie identifizieren können
- **Musterlösung:** [src/exercises/vt_01_api](#)

Übung V-T 2

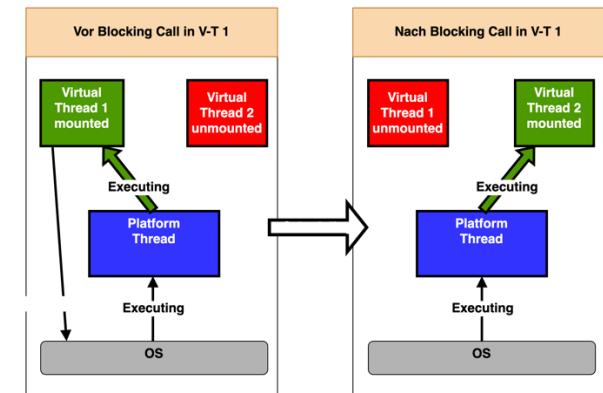
Blocking-Call gibt Carrier-Thread frei

Mittel

SIEMENS

Zeige, dass ein Virtual-Thread, der einen blockierenden Aufruf ausführt, den unterliegenden Platform-Thread („Carrier“) freigibt

- Nutze blockierenden Aufruf: Thread.sleep()
- Identifiziere Carrier-Thread durch: Thread.currentThread()
- Zeige, dass ein Virtual-Thread verschiedene Carrier haben kann, indem Du den Carrier vor und nach sleep() vergleichst



```
Virtual Thread ID and underlying Carrier ID before sleep: VirtualThread[#26]/Runnable@ForkJoinPool-1-worker-1
Virtual Thread ID and underlying Carrier ID before sleep: VirtualThread[#28]/Runnable@ForkJoinPool-1-worker-2
Virtual Thread ID and underlying Carrier ID before sleep: VirtualThread[#30]/Runnable@ForkJoinPool-1-worker-3
Virtual Thread ID and underlying Carrier ID before sleep: VirtualThread[#34]/Runnable@ForkJoinPool-1-worker-4
Virtual Thread id and underlying Carrier ID after sleep: VirtualThread[#34]/Runnable@ForkJoinPool-1-worker-2
Virtual Thread id and underlying Carrier ID after sleep: VirtualThread[#26]/Runnable@ForkJoinPool-1-worker-1
Virtual Thread id and underlying Carrier ID after sleep: VirtualThread[#28]/Runnable@ForkJoinPool-1-worker-4
Virtual Thread id and underlying Carrier ID after sleep: VirtualThread[#30]/Runnable@ForkJoinPool-1-worker-3
```

- **Lernziel:** Verstehen, dass blockierende Calls aus Virtual-Threads den unterliegenden Platform-Thread nicht blockieren
- **Musterlösung:** src/exercises/vt_02_blocking

Übung V-T 3

Skalierbarkeit bei Blocking

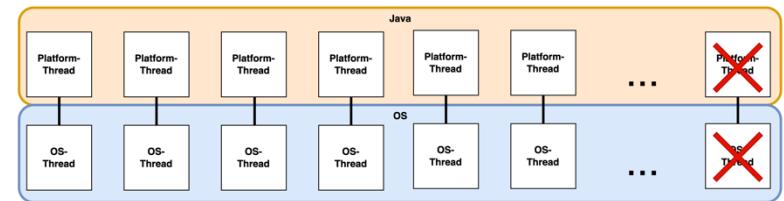
Einfach

SIEMENS

Zeige, dass eine Anwendung mit Virtual-Threads besser skalieren kann als mit Platform-Threads, wenn die Threads blockierende Aufgaben haben

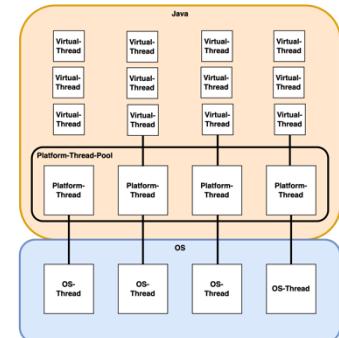
1. Starte so viele Platform Threads, dass das Programm abstürzt
 - o Nutze dafür z.B. Thread.sleep() um die Threads blockierend am Leben zu erhalten

```
Thread thread = new Thread(() -> {
    // ...
});
thread.start();
```



2. Zeige, dass wesentlich mehr virtuelle Threads gleichzeitig existieren können

```
Thread virtualThread =
    Thread.startVirtualThread(() -> {
        // ...
    });
)
```



3. Erkläre den Unterschied

Lernziel: Erkennen, dass Virtual-Threads das Skalierbarkeitsproblem der Platform-Threads lösen

Musterlösung: rc/exercises/vt_03_scalability_blk

Übung V-T 4

Performance durch Virtual-Threads bei Blocking-Calls

Finde einen Use Case, bei dem die parallele Ausführung mit Virtual-Threads schneller ist, als die parallele Ausführung mit Platform-Threads

1. (Ist in der Klasse Performance_BASE bereits implementiert)
 - Definiere eine Aufgabe, die mit einem blockierenden OS-Aufruf arbeitet
 - Starte so viele dieser Aufgaben mit Platform-Threads, dass das Programm noch stabil läuft
 - Messe die Laufzeit
2. Zeige, dass dieses Programm mit virtuellen Threads beschleunigt werden kann, da mehr virtuelle Threads gestartet werden können als Platform Threads.
3. Erkläre wie der Performance Vorteil zustande kommt

Lernziel: Verstehen, dass der Hauptnutzen der Virtual-Threads in der verbesserten Performance von Server-Applikationen liegt

Musterlösung: src/exercises/vt_04_performance_blkg

Zeige, dass die Methode lock() der expliziten Lock Klassen aus java.util.concurrent den Carrier Thread freigibt, wenn lock() in einem virtuellen Thread aufgerufen wird.

Programmiere folgendes Szenario:

- Starte viele (mehr als Grad der Parallelität des Pools) virtuelle Threads, die eine Funktion aufrufen, die eine Zeit lang auf der CPU läuft, z.B. eine Sekunde
- Zeige, dass nicht alle Virtual-Threads sofort starten, sondern nur so viele, wie es Platform Threads im Pool gibt
- Baue dann eine sogenannte Critical Section mit ReentrantLock.lock() und ReentrantLock.unlock() in die langlaufende Funktion ein
- Zeige, dass nun weitere virtuelle Threads starten, sobald die vorhergehenden virtuellen Threads die lock() Methode aufrufen.

Lernziel: Blocking Verhalten Virtual-Threads am Beispiel von Lock kennenlernen

Musterlösung: [src/exercises/vt_06_lock](#)

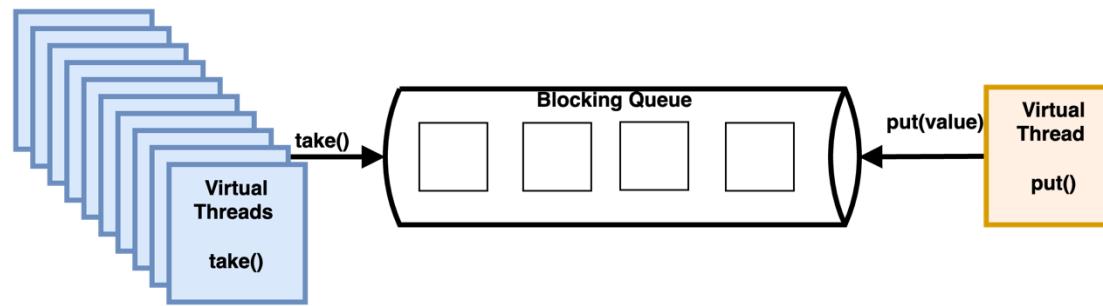
The screenshot shows a Java application window titled "Console". The output text is as follows:

```
<terminated> Lock_BASE [Java Application] /Library/Java/JavaVirtualMachines/jdk-25.jdk/Contents/Home/bin/java (25 Oct 2025, 21:26:30)
virtual thread running 11 at 6 Milliseconds VirtualThread[#50]/runnable@ForkJoinPool-1-worker-12
virtual thread running 5 at 5 Milliseconds VirtualThread[#37]/runnable@ForkJoinPool-1-worker-5
virtual thread running 8 at 6 Milliseconds VirtualThread[#43]/runnable@ForkJoinPool-1-worker-10
virtual thread running 3 at 5 Milliseconds VirtualThread[#34]/runnable@ForkJoinPool-1-worker-4
virtual thread running 11 at 642 Milliseconds VirtualThread[#54]/runnable@ForkJoinPool-1-worker-8
virtual thread running 15 at 642 Milliseconds VirtualThread[#55]/runnable@ForkJoinPool-1-worker-6
```

A red box highlights the first two lines of output: "virtual thread running 11 at 6 Milliseconds VirtualThread[#50]/runnable@ForkJoinPool-1-worker-12" and "virtual thread running 5 at 5 Milliseconds VirtualThread[#37]/runnable@ForkJoinPool-1-worker-5". A red line connects this box to the text "Zwei Threads starten später!" located in the bottom-left corner of the slide.

Zeige, dass ein Virtual-Thread, der einen blockierenden Aufruf einer Queue benutzt, im Falle des Blockierens seinen unterliegenden Carrier Thread freigibt

- Nutze dafür eine `java.util.concurrent.BlockingQueue`
- Starte viele virtuelle Threads, die `queue.take()` aufrufen (mehr als Pool-Threads vorhanden sind)
- Diese `take()`-threads blockieren, da die Queue leer ist.
- Starte dann einige virtuelle Threads, die mit `queue.put()` einen Wert in die Queue einfügen.
- Zeige, dass es zu keinem Deadlock kommt, Obwohl alle diese virtuellen Threads einen gemeinsamen Threadpool beschränkter Größe benutzen.



Lernziel: Verhalten von Virtual-Threads am Beispiel von Producer-Consumer Szenarien verstehen

Musterlösung: src/exercises/vt_09_queue_blocking

V-T 10 ForkJoinTasks vs. Virtual-Threads

Mittel

Zeige, dass ForkJoinTasks anders mit blockierenden Aufrufen umgehen als Virtual-Threads, da ForkJoinTasks Ihren unterliegenden Platform-Thread nicht freigeben.

Baue dafür ein Szenario, bei dem ForkJoinTasks in eine blockierende Queue Elemente mit put() einfügen und mit take() Elemente entnehmen. Dies beides sind blockierende Aufrufe.

A) Zeige, dass dabei mit ForkJoinTasks ein Deadlock entstehen kann (Hinweis: Viele ForkJoinTasks starten)

B) Zeige, dass sich dieses Problem mit Virtual-Threads lösen lässt

Lernziel: Unterschied zwischen Virtual-Threads und ForkJoinTasks erkennen

Musterlösung: JavaVirtualThr src/exercises/vt_10_forkjoin

Bei Virtual-Threads gibt es keine Preemption, das heißt, ein Virtual-Thread wird nicht von einem anderen Virtua-Thread verdrängt. Der laufende Thread muss von sich aus die Kontrolle abgeben, indem er endet oder blockiert.

Zeige, dass daher das Ergebnis eines Programmes ein anderes sein kann, je nachdem ob es mit Platform-Threads oder mit Virtual-Threads parallelisiert ist.

Lösungshinweis:

- Nutze Virtual-Threads die unendlich lange laufen, z.B. einen Zähler unendlich oft inkrementieren
- Starte dann $P+2$ solche Threads, wobei P die Parallelität im Threadpool ist
- Zeige, dass das Verhalten mit Platform Threads anders ist

Default Einstellung für P ist Anzahl der logischen CPUs

```
P = Runtime.getRuntime().availableProcessors();
```

Lernziel: Verstehen, dass die Migration von Platform-Threads zu Virtual-Threads die Semantik eines Programmes verändern kann

Musterlösung: src/exercises/vt_13_preemption_semantics

Zeige, wie der Threadpool konfiguriert werden kann, und wie sich das auf das Verhalten der Virtual-Threads auswirkt

VM Parameter mit denen der Threadpool konfiguriert werden kann:

- -Djdk.virtualThreadScheduler.parallelism=X
- -Djdk.virtualThreadScheduler.maxPoolSize=Y
- -Djdk.virtualThreadScheduler.minRunnable=Z

A) Konfiguriere den Threadpool so, dass nur ein Virtual-Thread gleichzeitig laufen kann und zeige an einem Beispiel, dass dann andere Virtual-Threads warten müssen

B) Konfiguriere den Pool so, dass im Falle vieler pinning Threads der Pool auf 100 anwächst. Zeige dass damit auch bei Pinning 100 Virtual-Threads starten.

Lernziel: Erkennen, dass der unterliegende Threadpool das Verhalten der Virtual-Threads beeinflusst

Musterlösung: src/exercises/vt_14_threadpool_config

Baue einen einfachen Server, der mit Platform Threads bei vielen Anfragen in kurzer Zeit abstürzt. Verbessere ihn durch den Einsatz von Virtual-Threads.

Die Basis der Musterlösung implementiert bereits folgendes Szenario. Repariere es durch Einsatz von Virtual-Threads im Server

- Der Server startet für jeden Client-Request einen eigenen Thread, der den Request entgegennimmt und eine Antwort zurück sendet.
- Der Client setzt viele (z.B. 10.000) Requests ab, ohne dazwischen zu warten. So entsteht Last auf dem Server
- Der Server stürzt ab einer gewissen Anzahl Requests ab (OutOfMemoryError bei der Thread-Erzeugung)

Lernziel: Erkennen, dass Server mit Virtual Threads besser skalieren

Musterlösung: src/exercises/vt_15_client_server_scalability

Grundlagen des Arbeitens mit StructuredTaskScope

- 1. Scope aufbauen:** Definiere einen Scope mithilfe der Klasse StructuredTaskScope
- 2. Subtasks erzeugen:** Starte in diesem Scope zwei Subtasks unter Verwendung der Methode StructuredTaskScope.fork() und zeige, dass der Scope erst dann beendet wird, wenn beide Subtasks abgeschlossen sind (join())
- 3. Exception Handling:** Werfe aus der ersten Subtask eine Exception. Sorge dafür, dass die zweite Subtask dadurch beendet wird.

Hinweise

- scope.join(); wartet auf alle Subtasks
- Jede Subtask muss ihren Interrupted Zustand regelmäßig prüfen:
Thread.currentThread().isInterrupted()

Lernziel: Benutzung von StructuredTaskScope erlernen und das Exceptionhandling darin verstehen

Musterlösung: src/exercises/struct_conc/sc_01_structured_scope

Es gibt mehrere Varianten von Sub-Scopes in einem StructuredTaskScope.

- Variante 1 Sub Scope innerhalb des umschließenden Scopes
- Variante 2 Sub-Scope innerhalb eines Subtasks des umschließenden Scopes

A) Variante 1 im Scope: Baue einen äusseren Scope unter Verwendung von StructuredTaskScope.open. Erweitere den äusseren Scope um einen Subscope (Variante 1). Starte einen Task-1 in diesem Subscope.

B) Variante 2 im Subtask: Erweitere das Ergebnis von A) indem Du in dem äusseren Scope einen Task-2 forkst, und erzeuge innerhalb dieses Task-2 einen Subscope. Darin startest Du dann wiederum eine Task-3.

C) Nebenläufigkeit: Wie lange müsste das Programm laufen, wenn Task-1 4 Sekunden, Task-2 3 Sekunden und Task-3 2 Sekunden lang laufen?

Lernziel: Varianten von Sub-Scopes kennenlernen. Nebenläufigkeit von Scopes verstehen.

Musterlösung: src/exercises/struct_conc/sc_02_sub_scopes

Joiner definieren das Verhalten eines StructuredTaskScope. Es gibt mehrere vordefinierte Joiner, Man kann aber auch selbst definierte Joiner schreiben.

- **A)** Baue einen Scope, der drei unterschiedlich lang laufende Subtasks umfasst.
 - **1)** Nutze einen Joiner, der dazu fuehrt, dass das Ergebnis derjenigen Subtask, die als erste fertig ist, zurueckgegeben wird. (Hinweis: anySuccessfulResultOrThrow)
 - **2)** Stelle sicher, dass sich die anderen Tasks beenden.
- **B)**
 - **1)** Baue einen eigenen Joiner, indem du das Interface StructuredTaskScope.Joiner<T, R> implementierst. Dieser soll das Maximum der Ergebnisse der Subtasks berechnen (Integer).
 - **2)** Baue einen Scope, der mehrere Tasks hat, die Integer Werte zurueckgeben. Benutze darin den Joiner aus 1) und zeige, dass er das Maximum der Rueckgabewerte findet.

Lernziel: Verhalten und Impelemntierung von Joinern verstehen

Musterlösung: src/exercises/struct_conc/sc_03_joiners

Aufgabe

Zeige, dass ein einzige ScopedValue Instanz in verschiedenen Virtual-Threads gleichzeitig unterschiedliche Werte haben kann

Hinweis:

- Eine ScopedValue Instanz anlegen, die von allen beteiligten Threads genutzt werden kann
- Mit ScopedValue.where(...) der ScopedValue Instanz einen Thread-Spezifischen Wert zuweisen
- Mit ScopedValue.where(...).run(() -> {...}) den innerhalb des Scopes auszuführenden Code übergeben

Lernziel: Verstehen dass ein ScopedValue in jedem Thread einen eigenen spezifischen Wert hat

Musterlösung: src/exercises/scoped_values/sv_00_simple_example

Diese Aufgabe demonstriert das grundlegende API von Scoped Values: where() und run().

Es geht dabei um das Binden von Werten innerhalb von Scopes und den Zugriff auf diese Werte. Alles findet im Main Thread statt(!).

A) Binding und Scope

- Weise eine Variable vom Typ ScopedValue mit where() einen Wert zu
- Führe mit run() ein Lambda im diesem Scope aus
- Greife aus dem Lambda auf den Wert zu und schreibe ihn auf die Console.

B) Chainig

- Instanziere mehrere Variablen vom Typ Scoped Value und weise ihnen mit verkettete where() Aufrufe Werte zu
- Starte mit run() ein Lambda, das auf diese Werte zugreifen kann, schreibe sie auf die Console

C) Rebinding

- Eröffne innerhalb einer Scoped-Value Scopes mit where() einen Sub-Scope
- Binde im Sub-Scope einen neuen Wert an den Scoped-Value.
- Zeige, dass innerhalb des Sub-Scopes der neue Wert gilt, und dass nach dem Verlassen des Sub-Scopes wieder der alte Wert gültig ist.

Lernziel: Scoped-Values API kennen: Scope, Binding, Rebinding, Chaining

Musterlösung: src/exercises/scoped_values/sv_01_basic_api

Der Sinn von Scoped Values ist es, diese in Threads zur Verfügung zu stellen. Dafuer gibt es unterschiedliche Herangehensweisen.

A) Inheritance identischer Werte im StructuredTaskScope

Nutze eine ScopedValue Instanz um mehreren Threads einen indentischen Wert zur Verfuegung zu stellen. Verwende dafuer einen StructuredTaskScope.

B) Unterschiedliche Werte im StructuredTaskScope

Weise mehreren Threads innerhalb eines StructuredTaskScope jeweils unterschiedliche Werte zu

C) Unterschiedliche Werte in unstrukturierten Threads

Nutze Bindings (where()) in unstrukturierten Threads um jedem einen eigenen Wert zuzuweisen

Lernziel: Scoped-Values Verahlten bei Multithreading verstehen

Musterlösung: [rc/exercises/scoped_values/sv_02_threading](#)