DIE CHOREOGRAFIE DER THREADS

Parallele und nebenläufige Programmierung in Java meistern (Band 4)

Dietrich Boles

Impressum

Autor: Dietrich Boles

Kontakt: dietrich@boles.de

Urheberrechtlicher Hinweis:

Originalquelle (Ursprungsort): https://www.boles.de/programmierkurs-java/

Vorwort

Liebe Leserinnen und Leser,

willkommen in der Königsklasse der Softwareentwicklung, der nebenläufigen und parallelen Programmierung!

Wenn Sie dieses Buch aufschlagen, haben Sie die Grundlagen der Programmierung bereits gemeistert. Sie wissen, wie man Klassen entwirft, Schleifen schreibt und Algorithmen formuliert. Bisher folgte Ihr Code dabei einer tröstlichen, verlässlichen Regel: Eine Anweisung wird nach der anderen ausgeführt. Zeile für Zeile, Schritt für Schritt. Wenn Sie Ihr Programm zweimal mit denselben Eingabedaten starteten, erhielten Sie exakt dasselbe Ergebnis. Die Welt war deterministisch.

Mit dem Aufschlagen dieses Buches lassen Sie diese berechenbare Welt hinter sich.

Die Realität der modernen IT-Infrastruktur hat den sequenziellen Code längst überholt. Prozessoren werden nicht mehr nennenswert schneller, sie werden stattdessen immer breiter – sie erhalten mehr Rechenkerne. Wer heute Software schreibt, die nur einen einzigen Kern nutzt, verschwendet 90 Prozent der verfügbaren Hardwareleistung. Moderne Anwendungen – vom hochskalierenden Cloud-Microservice bis zur responsiven Smartphone-App – erfordern es, dass hunderte Dinge gleichzeitig passieren.

Doch Nebenläufigkeit (Concurrency) und Parallelität (Parallelism) sind berüchtigt dafür, selbst erfahrene Entwickler zur Verzweiflung zu treiben. Wenn plötzlich mehrere Ausführungsstränge (Threads) gleichzeitig auf dieselben Daten zugreifen, versagt unsere menschliche, sequenziell geprägte Intuition. Bugs treten scheinbar aus dem Nichts auf, lassen sich im Debugger nicht reproduzieren und verschwinden wieder, nur um beim Kunden auf dem Produktionsserver das gesamte System lahmzulegen.

Dieses Buch wurde geschrieben, um Ihnen die Angst vor diesem Chaos zu nehmen. Es ist nicht einfach nur eine Auflistung von Java-Klassen und Frameworks. Es ist ein strukturierter Leitfaden, der Ihnen eine völlig neue Art zu denken vermittelt. Wir werden Fehler schonungslos analysieren, verstehen, warum sie passieren, und lernen, wie wir unsere Softwarearchitektur so aufbauen, dass sie erst gar nicht entstehen können.

Ich verspreche Ihnen: Wenn Sie den anfänglichen Knoten im Kopf gelöst haben, wird Ihnen das Entwerfen von eleganten, hochgradig parallelen Systemen eine enorme

Freude bereiten. Sie werden Code schreiben, der nicht nur funktioniert, sondern der fliegt.

Viel Erfolg auf dieser Reise!

Dietrich Boles, Universität Oldenburg, März 2026

Inhaltsverzeichnis

1 Einleitung 10

1.1 Motivation für nebenläufige und parallele Programmierung 10

1.2 Voraussetzungen zum Lesen dieses Buches 11

1.3 Aufbau des Buches 11

1.4 Überblick über die einzelnen Themen des Buches 12

2 Grundlagen der Nebenläufigkeit und Parallelität 15

2.1 Motivation: Warum das Ganze? 15

2.2 Die Hardware-Basis: CPU und Kerne 15

2.3 Ausführungseinheiten: Prozesse und Threads 16

2.4 Die Brücke zu Java: Betriebssystem-Threads vs. Java-Threads 16

2.5 Scheduling: Wer darf wann auf die CPU? 17

2.6 Nebenläufigkeit vs. Parallelität 19

3 Nebenläufige Aktionen in Java starten – Ein Überblick 21

3.1 Ebene 1: Die klassischen Wege (Low-Level) 21

3.2 Ebene 2: Ergebnisse und Ressourcenmanagement (Mid-Level) 22

3.3 Ebene 3: Moderne und spezialisierte High-Level-Paradigmen 22

3.4 Ebene 4: Virtuelle Threads 23

3.5 Zusammenfassung 24

4 Der direkte Weg – Ableiten der Klasse Thread 25

4.1 Das Grundprinzip: Thread und run() 25

4.2 Beispiel 1: Einen einfachen Thread starten 25

4.3 Auf das Ende eines Threads warten: join() 27

4.4 Beispiel 2: Auf Threads warten mit join() 27

4.5 Kritische Würdigung dieses Ansatzes 28

5 Entkopplung von Aufgabe und Ausführung – Das Runnable-Interface 29

5.1 Das Konzept hinter Runnable 29

5.2 Beispiel 1: Der klassische Weg mit Runnable 29

5.3 Beispiel 2: Der moderne Weg mit Lambda-Ausdrücken 31

5.4 Zusammenfassung der Vorteile 31

6 Steuerung und Lebenszyklus von Threads 33

6.1 Identifikation und Kontext: currentThread(), Name und ID 33

6.2 Der Lebenszyklus: Status (state) abfragen 34

6.3 Zeitsteuerung: sleep und yield 34

6.4 Die große Frage: Wie beendet man einen anderen Thread? 35

6.5 Daemon-Threads und Prioritäten 37

6.6 Fazit 38
7 Ergebnisse aus Threads abrufen – Callable und Future 39
7.1 Das Problem mit Runnable 39
7.2 Das Callable-Interface: Die bessere Aufgabe 39
7.3 Das Future-Interface: Der Abholschein für das Ergebnis 40
7.4 Wie führt man ein Callable aus? 40
7.5 Beispiel 1: Eine komplexe Berechnung mit Rückgabewert 40
7.6 Beispiel 2: Exceptions sauber behandeln 42
8 Ressourcen intelligent verwalten – Thread-Pools und das Executor-Framework 43
8.1 Das Problem: Der Overhead der Thread-Erstellung 43
8.2 Die Lösung: Der Thread-Pool 43
8.3 Das Executor-Framework in Java 44
8.4 Beispiel: Arbeiten mit dem FixedThreadPool 45
8.5 Den Lebenszyklus managen: Das richtige Herunterfahren 46
8.6 Massenabfertigung: invokeAll und invokeAny 46
8.7 Ergebnisse sofort verarbeiten: Der CompletionService 49
9 Zeitgesteuerte und periodische Aufgaben – Der ScheduledExecutorService 52
9.1 Warum nicht einfach Thread.sleep() oder java.util.Timer? 52
9.2 Den ScheduledExecutorService erstellen 53
9.3 Einmalige Ausführung mit Verzögerung: schedule 53
9.4 Periodische Ausführung: Rate vs. Delay 54
9.5 Eine gefährliche Falle: Unbehandelte Exceptions 56
10 Teile und Herrsche – Das ForkJoin-Framework 57
10.1 Das Konzept: Divide and Conquer (Teile und Herrsche) 57
10.2 Die Magie unter der Haube: Work-Stealing 57
10.3 Die Kernklassen des Frameworks 58
10.4 Beispiel: Eine gigantische Summe berechnen 58
10.5 Die Anatomie der compute()-Methode 60
10.6 Wann Sie Fork/Join NICHT nutzen sollten 61
11 Deklarative Datenparallelität – Parallele Streams 62
11.1 Der Paradigmenwechsel: Deklarativ statt Imperativ 62
11.2 Wie aus sequenziell parallel wird 62
11.3 Beispiel: Massendatenverarbeitung 63
11.4 Wann lohnt sich ein paralleler Stream? (Die Aufwand-Nutzen-Frage) 64
11.5 Gefahren und Anti-Patterns 65
12 Asynchrone und Reaktive Pipelines – CompletableFuture 67

12.1 Das Dilemma des blockierenden Wartens 67
12.2 Die Lösung: Reagieren statt Blockieren 67
12.3 Der Startpunkt: supplyAsync und runAsync 67
12.4 Pipelines bauen: Transformation und Konsum 68
12.5 Aufgaben kombinieren 69
12.6 Fehlerbehandlung elegant gelöst 70
13 Der Paradigmenwechsel – Virtuelle Threads (Project Loom) 72
13.1 Die Rückkehr zum M:N-Mapping 72
13.2 Die Magie unter der Haube: Mounten und Unmounten 73
13.3 Virtuelle Threads erstellen 73
13.4 Der richtige Einsatzzweck (I/O vs. CPU) 75
14 Die neue Denkweise – Interleaving, Koordination und geteilter Zustand 77
14.1 Der Abschied vom sequenziellen Denken 77
14.2 Die Wurzel des Chaos: Interleaving (Verzahnung) 78
14.3 Die drei Säulen der Thread-Interaktion 79
14.4 Wo lauern die Gefahren? Das Dilemma von Safety vs. Liveness 80
14.5 Die Checkliste für Programmierer 80
14.6 Ausblick: Unser Fahrplan für Teil II 81
15 Der klassische Schutzschild – Monitore und das synchronized-Schlüsselwort 82
15.1 Das Geheimnis der Java-Objekte: Der Monitor 82
15.2 Synchronisierte Methoden (Instanz-Ebene) 83
15.3 Synchronisierte Blöcke (Feingranulare Sperren) 83
15.4 Statische Synchronisation (Klassen-Ebene) 84
15.5 Ein geniales Feature: Reentrancy (Wiedereintrittsfähigkeit) 85
16 Das Sichtbarkeitsproblem – Das Java Memory Model und volatile 86
16.1 Die Hardware-Illusion: CPU-Caches 86
16.2 Das Problem: Der Endlos-Schlaf 87
16.3 Die Lösung: Das volatile-Schlüsselwort 88
16.4 Das Java Memory Model (JMM) und "Happens-Before" 88
16.5 Die goldene Regel: volatile vs. synchronized 89
17 Klassische Thread-Kommunikation – wait(), notify() und notifyAll() 90
17.1 Das Problem: Aktives Warten (Busy Waiting) 90
17.2 Der Warteraum des Monitors 91
17.3 Das Paradigma: Erzeuger und Verbraucher (Producer-Consumer) 92
17.4 Die eiserne Regel: Das "Spurious Wakeup" 93
17.5 notify() vs. notifyAll(): Was ist besser? 94

18 Moderne und flexible Sperren – Das java.util.concurrent.locks-Paket 95
18.1 Die Grenzen von synchronized 95
18.2 Das ReentrantLock: Die objektorientierte Alternative 95
18.3 Fortgeschrittene Funktionen des ReentrantLock 96
18.4 Lese- und Schreib-Sperren: Das ReadWriteLock 97
18.5 Gezieltes Wecken: Condition-Objekte 98
18.6 Mehr als nur ein Türsteher: Der Aspekt der Sichtbarkeit 99
19 Thread-Choreografie – Barrieren, Zähler und Semaphoren 102
19.1 Der CountDownLatch: Ein Zähler, zwei Strategien 102
19.2 Der Treffpunkt: CyclicBarrier 105
19.3 Die Meisterklasse der Koordination: Der Phaser 106
19.4 Der Türsteher: Semaphore 108
19.5 Der direkte Tausch: Exchanger 110
20 Thread-sichere Datenstrukturen – Concurrent Collections 112
20.1 Die Katastrophe: Warum ArrayList und HashMap explodieren 112
20.2 Der traditionelle Schutzschild: Die synchronisierten Wrapper der Klasse Collections 113
20.3 Maßgeschneiderte Leistung: Die Concurrent-Klassen 115
20.4 Für Lesefans: CopyOnWriteArrayList 117
20.5 Die ultimative Lösung für Erzeuger & Verbraucher: BlockingQueue 118
20.6 Spezialisten für besondere Aufgaben: Weitere BlockingQueue-Klassen 119
20.7 Fazit: Das richtige Werkzeug für geteilte Datenstrukturen 121
21 Höchstleistung ohne Sperren – Lock-Free Programming und atomare Variablen 123
21.1 Die Hardware-Magie: Compare-And-Swap (CAS) 123
21.2 Die Endlosschleife des Optimisten 124
21.3 Das Paket java.util.concurrent.atomic 124
21.4 Eine kleine Schwäche: Das ABA-Problem 125
21.5 Der theoretische Stolperstein: Das ABA-Problem 126
21.6 Das Hochlast-Wunder: Adder und Accumulator 128
21.7 Speicherschlank optimieren: Die AtomicFieldUpdater 129
22 Wenn nichts mehr geht – Liveness-Probleme analysieren und vermeiden 131
22.1 Der absolute Stillstand: Deadlock (Verklemmung) 131
22.2 Strategien zur Deadlock-Vermeidung 132
22.3 Viel Lärm um nichts: Livelock 133
22.4 Das vergessene Kind: Starvation (Verhungern) 134
23 Praxis 137

24 Die Jagd nach dem Heisenbug – Testen und Debuggen 140
24.1 Die Heisenbug-Falle und warum der Debugger lügt 140
24.2 Testen mit Bordmitteln: CountDownLatch in JUnit 141
24.3 Elegantes Testen: Das Awaitility-Framework 142
24.4 Die Wahrheit ans Licht bringen: jcstress 143
24.5 Werkzeuge der Profis: Dem Stillstand auf der Spur 143
25 Performance messen, aber richtig – Microbenchmarking 146
25.1 Die Stoppuhr-Lüge: System.currentTimeMillis() 146
25.2 Die Streiche des JIT-Compilers 147
25.3 Die professionelle Lösung: JMH (Java Microbenchmark Harness) 147
25.4 Ein echter JMH-Benchmark im Code 148
25.5 Messergebnisse richtig lesen 149
26 Aus dem Nähkästchen – Best Practices, Nice-to-Haves und No-Gos 150
26.1 Die absolute Best Practice: Immutability (Unveränderlichkeit) 150
26.2 Die Geheimwaffe: ThreadLocal (Share Nothing) 151
26.3 Der Nachfolger für die Loom-Ära: Scoped Values 152
26.4 Der Friedhof der Thread-Methoden: stop(), suspend(), resume() 154
26.5 Das berüchtigte Anti-Pattern: Double-Checked Locking 155
26.6 Unsichtbare Gefahr: Thread Leaks in Server-Umgebungen 156
27 Über den Tellerrand – Alternative Paradigmen 157
27.1 Das Actor-Modell: Nachrichten statt Sperren 157
27.2 Reaktive Programmierung und die Java Flow-API 158
27.3 Wann wählt man welches Paradigma? 161
28 Die physikalischen Grenzen und Javas Zukunft 163
28.1 Die bittere Logik: Warum mehr Kerne nicht immer mehr Tempo bedeuten 163
28.2 Die Speichermauer (The Memory Wall) 164
28.3 Javas Antwort auf die Zukunft 165
29 Überblick über das Großküchen-Projekt 171
29.1 Die Rückkehr in die Großküche 171
29.2 Die 7 Iterationen auf unserem Weg zur Perfektion 171
29.3 Bereit für die Praxis? 173
30 Iteration 1 – Die Pizzeria eröffnet (Sequenziell & Erste Threads) 174
30.1 Der Flaschenhals: Die sequenzielle Pizzeria 174
30.2 Wir stellen Hilfsköche ein: Threads und Runnables 175
30.3 Was wir erreicht haben (und was nicht) 176
31 Iteration 2 – Schichtplan und Kellner (Thread-Pools & Futures) 178

31.1 Feste Posten: Der Thread-Pool ... 178
31.2 Abholscheine für die Kellner: Callable und Future ... 178
31.3 Die Umsetzung im Code ... 179
31.4 Analyse: Was wir aus Iteration 2 lernen ... 181
32 Iteration 3 – Chaos in der Speisekammer (Geteilter Zustand & Locks) ... 182
32.1 Die Speisekammer ohne Türsteher: Die Race Condition ... 182
32.2 Der klassische Schutz: Das synchronized-Schlüsselwort ... 183
32.3 Die Hochleistungs-Optimierung: Das ReentrantReadWriteLock ... 184
33 Iteration 4 – Die Ticket-Schiene (Producer-Consumer & Concurrent Collections) ... 186
33.1 Die Entkopplung: Das Producer-Consumer-Muster ... 186
33.2 Die Magie der BlockingQueue ... 186
33.3 Die Kasse ohne Schloss: AtomicInteger ... 187
33.4 Die Umsetzung im Code ... 187
33.5 Analyse: Die Kraft der Concurrent Collections ... 189
34 Iteration 5 – Meisterhafte Koordination (Barrieren & Semaphoren) ... 191
34.1 Der Türsteher für den Ofen: Das Semaphore ... 191
34.2 Der Sammelpunkt: Die CyclicBarrier ... 191
34.3 Die Umsetzung im Code ... 192
34.4 Analyse der Choreografie ... 194
35 Iteration 6 – Die landesweite Kette (Project Loom & CompletableFuture) ... 196
35.1 Die Speichermauer: Das C10K-Problem ... 196
35.2 Die Rettung: Virtuelle Threads (Project Loom) ... 196
35.3 Das Fließband: CompletableFuture ... 197
35.4 Die Umsetzung im Code ... 197
35.5 Analyse: Die Magie von Loom und funktionaler Ketten ... 199
36 Iteration 7 – Gesundheitsamt und Sterne-Bewertung (Testing & Benchmarking) ... 200
36.1 Das Problem asynchroner Tests (Heisenbugs) ... 200
36.2 Das Gesundheitsamt: Professionelles Testing mit Awaitility ... 201
36.3 Die Sterne-Bewertung: Benchmarking mit JMH ... 201
36.4 Fazit des Großküchen-Projekts ... 203

1 Einleitung

Der Übergang von der sequenziellen zur nebenläufigen Programmierung gleicht dem Wechsel vom Kochen in der heimischen Küche zum Leiten einer professionellen Großküche. Wenn Sie alleine kochen, bestimmen Sie das Tempo: Sie schneiden das Gemüse, dann braten Sie das Fleisch. Niemand nimmt Ihnen das Messer weg, niemand verstellt die Herdplatte.

In einer Großküche hingegen arbeiten zwanzig Köche gleichzeitig. Die Arbeit geht unglaublich schnell voran, aber nur, wenn die Koordination perfekt ist. Wenn zwei Köche gleichzeitig nach demselben Topf greifen oder einer auf den anderen wartet, bricht das System zusammen. Genau diese Koordination werden Sie in diesem Buch erlernen.

1.1 Motivation für nebenläufige und parallele Programmierung

Warum sollten wir unsere vertraute, sequenzielle Programmierwelt verlassen und uns freiwillig der Komplexität der Nebenläufigkeit stellen? Der Grund ist erstaunlich einfach: Die reale Welt, für die wir Software schreiben, ist von Natur aus nebenläufig.

Wenn Sie an einer Supermarktkasse stehen, friert nicht der restliche Supermarkt ein, bis Sie bezahlt haben. Überall passieren Dinge gleichzeitig, unabhängig voneinander und doch im selben System. Wenn wir Software entwickeln, die reale Prozesse steuert, abbildet oder unterstützt – sei es ein Webshop, ein Chat-System oder eine komplexe Simulation –, muss unsere Architektur diese Gleichzeitigkeit widerspiegeln können. Ein Programm, das immer nur eine einzige Aufgabe nach der anderen abarbeiten kann, wirkt in einer dynamischen Welt schnell starr und unnatürlich.

Ein zweiter, ebenso wichtiger Grund ist die pure Effizienz im Umgang mit Zeit. Moderne Anwendungen verbringen einen Großteil ihrer Laufzeit nicht mit Rechnen, sondern mit Warten: Warten auf die Antwort einer langsamen Datenbank, Warten auf Datenpakete aus dem Netzwerk oder Warten auf die Eingabe eines Benutzers.

Ein rein sequenzielles Programm verhält sich in solchen Momenten wie ein Handwerker, der Baumaterial bestellt und sich dann tagelang untätig auf die Baustelle setzt, bis die Lieferung eintrifft. Das ist pure Verschwendung. Nebenläufige Programmierung ermöglicht es uns, diese unvermeidlichen Wartezeiten intelligent zu nutzen. Während ein Teil unseres Programms auf Daten aus dem Internet wartet, kann ein anderer Teil bereits die Benutzeroberfläche aktualisieren oder andere Berechnungen im Hintergrund durchführen.

Es geht bei der Nebenläufigkeit also nicht nur darum, durch den Einsatz mehrerer Prozessorkerne schiere Rechenleistung zu erzwingen. Es geht vielmehr darum, intelligente, reaktionsschnelle und effiziente Systeme zu entwerfen, die ihre Ressourcen optimal auslasten und niemals stillstehen, nur weil sie auf ein externes Ereignis warten müssen.

1.2 Voraussetzungen zum Lesen dieses Buches

Wenn Sie dieses Buch aufschlagen, sollten Sie die Grundlagen der Java-Programmierung bereits gemeistert haben. Sie sollten wissen, wie man Klassen entwirft, Schleifen schreibt und Algorithmen formuliert.

Darüber hinaus sind folgende Konzepte hilfreich:

- Objektorientierung: Ein sicherer Umgang mit Konzepten wie Vererbung und Interfaces.
- Basiswissen der Java-API: Sie sollten mit Exceptions umgehen können und das Konzept der Generics verstanden haben.
- Lambda-Ausdrücke: Da moderne Java-APIs stark auf funktionale Programmierung setzen, sollten Sie Lambda-Ausdrücke (eingeführt in Java 8) lesen und schreiben können, da diese im Buch intensiv zur Definition von Aufgaben (etwa bei Runnable oder Callable) genutzt werden.

Sie benötigen hingegen kein Vorwissen im Bereich der Betriebssystemarchitektur oder des Multi-Threadings. Wir erarbeiten uns diese komplexe Thematik gemeinsam von Grund auf.

1.3 Aufbau des Buches

Um Sie sicher durch den Dschungel der Nebenläufigkeit zu führen, ist dieses Buch in vier große Teile gegliedert, die logisch aufeinander aufbauen:

- **Teil I: Grundlagen und Thread-Erstellung (Die Köche einstellen)** Im ersten Teil lernen Sie, wie Sie Aufgaben überhaupt parallelisieren und tausende Arbeiter losschicken, um Berechnungen durchzuführen.
- **Teil II: Wenn Threads aufeinandertreffen (Die Verkehrsregeln der Küche)** Hier betreten wir die Gefahrenzone und widmen uns der Koordination. Wir klären, was passiert, wenn mehrere Arbeiter gleichzeitig dieselben Daten verändern wollen.
- **Teil III: Praxis, Performance, Perspektiven – Nebenläufigkeit im realen Einsatz** Da in der Praxis handfeste Ergebnisse zählen, erlernen Sie im

dritten Teil das handwerkliche Rüstzeug eines Software-Ingenieurs, um Performance zu messen und hartnäckige Bugs zu beheben.

- **Teil IV: Die Choreografie in der Praxis – Das Großküchen-Projekt** Im vierten Teil des Buches führen wir alle zuvor behandelten Konzepte in einem großen, zusammenhängenden Praxisbeispiel zusammen: der Simulation einer hochskalierbaren Großküche.

1.4 Überblick über die einzelnen Themen des Buches

Damit Sie genau wissen, was Sie auf dieser Reise erwartet, hier ein detaillierter Ausblick auf die Inhalte:

Im ersten Teil klären wir zunächst die Hardware-Grundlagen sowie den fundamentalen Unterschied zwischen Prozessen und Threads und zwischen Nebenläufigkeit (Concurrency) und Parallelität (Parallelism). Wir arbeiten uns von historischen Low-Level-Konzepten wie der Klasse Thread und dem Interface Runnable schrittweise nach oben. Sie lernen, wie man Ergebnisse asynchron mit Callable und Future abruft und Systemressourcen mit Thread-Pools (ExecutorService) intelligent verwaltet. Anschließend behandeln wir moderne High-Level-Paradigmen: das ForkJoin-Framework, parallele Streams, asynchrone Pipelines mit CompletableFuture und als revolutionäres Highlight die ressourcenschonenden Virtuellen Threads (Project Loom).

Im zweiten Teil stellen wir uns den Tücken des geteilten Zustands. Wir beleuchten das Problem der Verzahnung (Interleaving) und das Java Memory Model. Sie erlernen Schutzmechanismen wie Monitore und das synchronized-Schlüsselwort, moderne ReentrantLock-Sperren sowie den Lock-Free-Ansatz mit atomaren Variablen. Wir choreografieren komplexes Thread-Verhalten mit Semaphoren und Barrieren und nutzen fertige, Thread-sichere Datenstrukturen aus dem Paket java.util.concurrent. Den Abschluss dieses Teils bildet die Analyse gefürchteter Liveness-Probleme wie Deadlocks und Starvation.

Im dritten Teil rüsten wir Sie für den Produktionsbetrieb. Wir machen Jagd auf "Heisenbugs" und besprechen, wie man nebenläufigen Code effektiv testet und debuggt. Wir zeigen auf, warum einfache Stoppuhren bei der Leistungsmessung lügen, und führen Sie in wissenschaftlich korrektes Benchmarking mit JMH ein. Aus dem Nähkästchen besprechen wir Best Practices (wie Unveränderlichkeit) und Werkzeuge wie ThreadLocal oder Scoped Values. Abschließend werfen wir einen Blick auf alternative Paradigmen wie das Actor-Modell, reaktive Programmierung und die unerbittlichen physikalischen Grenzen der Hardware (wie das Amdahlsche Gesetz).

Im vierten Teil führen wir viele der zuvor behandelten Konzepte in einem großen, zusammenhängenden Praxisbeispiel zusammen: der Simulation einer hochskalierbaren Großküche. In sieben iterativen Schritten transformieren wir eine einfache, überforderte Pizzeria in eine hocheffiziente, reaktive Cloud-Architektur. Wir beginnen mit der Einführung von Hilfsköchen (Threads), optimieren die Arbeitsabläufe durch Schichtpläne (Thread-Pools), sichern die Speisekammer gegen Chaos ab (Locks & Synchronisation) und entkoppeln die Prozesse über Ticket-Schienen (BlockingQueues). Den krönenden Abschluss bilden der Einsatz von virtuellen Threads (Project Loom) für maximale Skalierbarkeit sowie die Qualitätssicherung durch professionelles Testing und Benchmarking. Dieser Teil dient dazu, das Verständnis für das Zusammenspiel der verschiedenen Java-Concurrency-Werkzeuge in einer realitätsnahen Umgebung zu festigen.

Ein Ratschlag für Ihre Reise: Lesen Sie dieses Buch nicht einfach nur durch. Nebenläufigkeit kann man nicht durch reines Zusehen erlernen. Setzen Sie sich an Ihre Entwicklungsumgebung, tippen Sie die Beispiele ab, provozieren Sie absichtlich Race Conditions und schauen Sie zu, wie Ihr Code mit falschen Zahlen abstürzt. Nur wer das Chaos einmal selbst ausgelöst hat, versteht den wahren Wert der Struktur.

Lassen Sie uns beginnen!

Teil I:

Grundlagen und Thread-Erstellung

2 Grundlagen der Nebenläufigkeit und Parallelität

Willkommen in der Welt der parallelen und nebenläufigen Programmierung! Bisher haben Sie Programme geschrieben, die strikt sequenziell ablaufen: Eine Anweisung wird nach der anderen ausgeführt, Zeile für Zeile. In der modernen Softwareentwicklung reicht dieses Modell jedoch oft nicht mehr aus. Dieses Kapitel legt das theoretische Fundament, bevor wir in den folgenden Kapiteln in die praktische Umsetzung mit Java eintauchen.

2.1 Motivation: Warum das Ganze?

Warum sollten wir uns mit komplexeren Ausführungsmodellen beschäftigen, wenn sequenzieller Code doch so viel einfacher zu schreiben und zu verstehen ist? Dafür gibt es zwei Hauptgründe: Leistung und Reaktionsfähigkeit.

Lange Zeit verließen sich Softwareentwickler auf das Mooresche Gesetz: Prozessoren wurden mit jeder Generation automatisch schneller (höhere Taktraten). Um die Jahrtausendwende stießen die Chiphersteller jedoch an physikalische Grenzen (Abwärme, Stromverbrauch). Anstatt die Taktrate einzelner Rechenkerne weiter zu erhöhen, begannen sie, mehrere Rechenkerne auf einem Chip zu verbauen. Ein sequenzielles Programm nutzt jedoch immer nur einen Kern. Um die volle Leistung moderner Hardware auszuschöpfen, müssen wir unsere Programme so schreiben, dass sie Aufgaben aufteilen und gleichzeitig bearbeiten können.

Darüber hinaus erwarten Nutzer heute responsive Anwendungen. Eine Benutzeroberfläche darf nicht einfrieren, nur weil im Hintergrund eine große Datei heruntergeladen oder eine komplexe Datenbankabfrage durchgeführt wird. Auch Server-Anwendungen müssen Tausende von Anfragen gleichzeitig bedienen können, ohne dass ein Nutzer auf den anderen warten muss.

2.2 Die Hardware-Basis: CPU und Kerne

Um zu verstehen, wie Software gleichzeitig ausgeführt wird, müssen wir einen kurzen Blick auf die Hardware werfen.

Die CPU (Central Processing Unit) ist das Gehirn des Computers. Früher bestand eine CPU aus exakt einem Kern (Core). Dieser Kern konnte zu einem bestimmten Zeitpunkt genau einen Befehl ausführen.

Moderne Prozessoren sind Multi-Core-CPUs. Sie enthalten mehrere unabhängige Recheneinheiten (Kerne) auf einem einzigen physischen Chip. Ein Quad-Core-Prozessor hat beispielsweise vier Kerne und kann somit tatsächlich vier Instruktionen im exakt selben physikalischen Moment ausführen.

2.3 Ausführungseinheiten: Prozesse und Threads

Wie verwaltet das Betriebssystem (OS) die Programme, die auf diesen Kernen ausgeführt werden sollen? Hier kommen Prozesse und Threads ins Spiel.

Prozesse: Ein Prozess ist ein Programm in Ausführung. Wenn Sie Ihre Java-Anwendung starten, erzeugt das Betriebssystem einen neuen Prozess. Jeder Prozess ist stark isoliert und besitzt seinen eigenen, geschützten Speicherbereich (Address Space). Wenn Prozess A abstürzt, betrifft das Prozess B in der Regel nicht. Der Datenaustausch (Inter-Process Communication, IPC) zwischen zwei Prozessen ist relativ aufwendig und langsam.

Threads: Ein Thread (Ausführungsfaden) ist die kleinste Ausführungseinheit, die von einem Betriebssystem geplant (gescheduled) werden kann. Threads existieren innerhalb eines Prozesses.

Jeder Prozess hat mindestens einen Thread (den Haupt-Thread oder Main-Thread). Ein Prozess kann aber auch viele Threads beinhalten.

Gemeinsamkeiten und Unterschiede:

- Speicher: Während Prozesse strikt getrennte Speicherbereiche haben, teilen sich alle Threads eines Prozesses denselben Speicher (insbesondere den Heap-Speicher, in dem Java-Objekte leben). Jeder Thread hat jedoch seinen eigenen Stack (Call Stack) für lokale Variablen und Methodenaufrufe.
- Gewicht: Prozesse werden oft als "Heavyweight" bezeichnet, da ihre Erstellung und der Wechsel zwischen ihnen (Context Switch) ressourcenintensiv ist. Threads nennt man "Lightweight Processes". Sie lassen sich schneller erzeugen und der Wechsel zwischen Threads desselben Prozesses ist deutlich effizienter.
- Verbindung: Ein Thread kann nicht ohne einen Prozess existieren. Stirbt der Prozess, sterben alle seine Threads.

2.4 Die Brücke zu Java: Betriebssystem-Threads vs. Java-Threads

Da wir nun wissen, was Threads auf Betriebssystemebene sind, müssen wir klären, wie Java damit umgeht.

Wenn Sie in Java einen Thread erzeugen (z. B. mit der Klasse java.lang.Thread), erzeugen Sie zunächst ein ganz normales Java-Objekt im Speicher. Doch wie wird daraus eine tatsächliche Ausführungseinheit?

Die traditionelle Java Virtual Machine (JVM) verwendet ein 1:1-Mapping-Modell. Das bedeutet: Für jeden gestarteten Java-Thread fordert die JVM beim zugrunde liegenden Betriebssystem (Windows, Linux, macOS) einen echten OS-Thread an und bindet den Java-Thread fest an diesen.

Gemeinsamkeiten und Unterschiede:

- Lebenszyklus: Der Lebenszyklus des Java-Threads spiegelt den des OS-Threads wider. Wenn der Java-Thread startet, startet der OS-Thread. Endet die run()-Methode in Java, wird der OS-Thread an das System zurückgegeben.
- Scheduling: Die JVM entscheidet nicht selbst, welcher Thread wann auf der CPU ausgeführt wird. Sie überlässt das Scheduling komplett dem Betriebssystem. Die Prioritäten von Java-Threads sind lediglich "Hinweise" an das OS.
- Kosten: Da herkömmliche Java-Threads eigentlich OS-Threads sind, sind sie teuer. Ein OS-Thread verbraucht standardmäßig etwa 1 bis 2 MB Speicher außerhalb des Java-Heaps. Sie können auf einem normalen Server problemlos 1.000, aber nicht 1.000.000 dieser Threads starten, ohne dass der Speicher vollläuft.

2.5 Scheduling: Wer darf wann auf die CPU?

Nachdem wir nun wissen, dass Threads die grundlegenden Ausführungseinheiten sind, stellt sich in der Praxis sofort ein mathematisches Problem: Was passiert, wenn wir auf einem Computer mit 4 Rechenkernen 100 Threads gleichzeitig starten? Offensichtlich können nicht alle Threads im exakt selben physikalischen Moment berechnet werden.

Hier betritt eine der wichtigsten und mächtigsten Komponenten des Betriebssystems die Bühne: der Scheduler (zu Deutsch etwa Zuteiler oder Planer).

Der Scheduler ist vergleichbar mit einem Verkehrspolizisten an einer extrem stark frequentierten Kreuzung. Er entscheidet in jedem Bruchteil einer Sekunde neu, welcher Thread auf welchem Rechenkern arbeiten darf und welcher Thread warten muss.

2.5.1 Preemptives Multitasking (Verdrängendes Scheduling)

Moderne Betriebssysteme (wie Windows, Linux oder macOS) verwenden sogenanntes präemptives Scheduling. Das bedeutet, dass das Betriebssystem die absolute Kontrolle behält. Es weist einem arbeitsbereiten Thread ein winziges Zeitfenster (eine sogenannte Time-Slice oder Zeitscheibe) zu, das oft nur wenige Millisekunden lang ist.

Ist diese Zeit abgelaufen, unterbricht das Betriebssystem den Thread rücksichtslos – egal, was er gerade berechnet –, entzieht ihm den Rechenkern und gibt die CPU an den nächsten Thread in der Warteschlange weiter. Dieser ständige, rasante Wechsel erzeugt für uns Menschen die Illusion, dass hunderte Programme gleichzeitig laufen.

Ein Thread kann die CPU aber auch freiwillig räumen, bevor seine Zeitscheibe abgelaufen ist. Das passiert typischerweise, wenn er auf ein externes Ereignis warten muss (z. B. auf das Laden einer Datei von der Festplatte, auf eine Datenbankantwort oder auf das Aufheben einer Sperre). In diesem Moment wird der Thread "blockiert", und der Scheduler teilt die freigewordene CPU-Zeit sofort einem anderen, arbeitsbereiten Thread zu.

2.5.2 Der Preis der Magie: Der Context Switch

Dieser fliegende Wechsel zwischen Threads ist jedoch nicht kostenlos. Wenn der Scheduler Thread A stoppt und Thread B startet, muss er einen sogenannten Context Switch (Kontextwechsel) durchführen.

Stellen Sie sich vor, Sie lesen ein anspruchsvolles Fachbuch und jemand zwingt Sie, mitten im Satz aufzuhören und ein anderes Buch weiterzulesen. Sie müssen sich ein Lesezeichen legen, das neue Buch aus dem Regal holen, die richtige Seite aufschlagen und sich wieder in die Handlung hineindenken.

Exakt das muss die Hardware tun: Sie speichert den kompletten aktuellen Zustand von Thread A (alle CPU-Register, den Programmzähler, lokale Stack-Daten) im Arbeitsspeicher ab und lädt anschließend den zuvor gespeicherten Zustand von Thread B in die Recheneinheit. Dieser Vorgang dauert zwar nur Mikrosekunden, aber wenn ein System zehntausende Male pro Sekunde den Kontext wechselt, verbringt die CPU irgendwann mehr Zeit mit dem ständigen "Umblättern" (Verwaltungsaufwand) als mit der eigentlichen Arbeit (Ihrer Programmlogik). Das ist einer der Hauptgründe, warum unkontrolliertes Multi-Threading ein System stark verlangsamen kann (ein Problem, das wir in den späteren Kapiteln durch Thread-Pools lösen werden).

2.5.3 Was bedeutet das für uns Java-Entwickler?

Wie wir in Abschnitt 2.5 gelernt haben, bindet die traditionelle Java Virtual Machine (JVM) jeden Java-Thread fest an einen Betriebssystem-Thread. Das bringt für Sie als Entwickler eine bittere, aber enorm wichtige Erkenntnis mit sich: Sie haben in Java keine absolute Kontrolle über das Scheduling!

Sie können dem Betriebssystem zwar freundliche Hinweise geben (etwa indem Sie Thread-Prioritäten setzen), aber ob, wann und in welcher Reihenfolge der Scheduler Ihre Threads auf die CPU lässt, entscheidet das Betriebssystem völlig autonom anhand seiner eigenen, hochkomplexen Algorithmen.

Ein nebenläufiges Java-Programm darf sich daher niemals darauf verlassen, dass zwei Threads in einer bestimmten, vorhersehbaren Reihenfolge abgearbeitet werden. Genau diese Unvorhersehbarkeit (der sogenannte Non-Determinismus) ist der Grundstein für all das Chaos und die hartnäckigen Bugs, denen wir uns im zweiten Teil dieses Buches stellen werden.

2.6 Nebenläufigkeit vs. Parallelität

In der Umgangssprache werden diese beiden Begriffe oft synonym verwendet. In der Informatik beschreiben sie jedoch zwei unterschiedliche Konzepte.

2.6.1 Nebenläufige Programmierung (Concurrency)

Definition: Nebenläufigkeit ist die Eigenschaft eines Systems, mehrere Aufgaben (Tasks) im selben Zeitraum zu verwalten und zu bearbeiten, ohne dass diese zwingend exakt im selben physikalischen Moment ausgeführt werden müssen.

Bei der Nebenläufigkeit geht es um die Strukturierung eines Programms. Ein nebenläufiges System teilt die CPU-Zeit (Time-Slicing) zwischen verschiedenen Threads auf. Auf einer Single-Core-CPU wird zu einem Zeitpunkt immer nur ein Thread ausgeführt. Das Betriebssystem wechselt jedoch so schnell zwischen den Threads hin und her, dass es für den menschlichen Benutzer so aussieht, als würden sie gleichzeitig laufen. Es geht hier darum, mit mehreren Dingen gleichzeitig umzugehen.

2.6.2 Parallele Programmierung (Parallelism)

Definition: Parallelität ist die Eigenschaft eines Systems, mehrere Aufgaben im exakt selben physikalischen Moment auszuführen.

Bei der Parallelität geht es um die Ausführung. Dies erfordert zwingend Hardware mit mehreren Kernen (Multi-Core). Eine große Aufgabe wird in unabhängige Teilaufgaben

zerlegt, die dann physisch gleichzeitig auf unterschiedlichen Kernen berechnet werden, um das Endergebnis schneller zu erhalten. Es geht darum, mehrere Dinge gleichzeitig zu tun.

Hinweis: Ein System kann nebenläufig sein, ohne parallel zu sein (Time-Slicing auf einem Kern). Ein System kann parallel sein, ohne nebenläufig zu sein (Hardware-Vektor-Instruktionen für einen Datenstrom). Und natürlich kann ein System beides gleichzeitig sein – was das Ziel der meisten modernen Java-Anwendungen ist.

3 Nebenläufige Aktionen in Java starten – Ein Überblick

Nachdem wir im ersten Kapitel die theoretischen Grundlagen von Prozessen, Threads und Parallelität geklärt haben, stellt sich nun die praktische Frage: Wie teilen wir der Java Virtual Machine (JVM) mit, dass ein bestimmter Code-Block nicht sequenziell, sondern nebenläufig ausgeführt werden soll?

Java bietet dafür, historisch bedingt, eine ganze Reihe von Möglichkeiten. Sie bauen teilweise aufeinander auf oder dienen völlig unterschiedlichen Anwendungsfällen. In diesem Kapitel geben wir einen vollständigen Überblick über alle Mechanismen, bevor wir in den Folgekapiteln in die Details der einzelnen Ansätze abtauchen.

Wir können die Werkzeuge der Java-API grob in vier Abstraktionsebenen einteilen.

3.1 Ebene 1: Die klassischen Wege (Low-Level)

Diese Basis-Mechanismen existieren seit Java 1.0. Sie erfordern das manuelle Erstellen und Verwalten von Thread-Objekten.

3.1.1 Ableiten der Klasse java.lang.Thread

Der direkteste Weg ist die Erstellung einer eigenen Klasse, die von der Klasse Thread erbt. Sie überschreiben die Methode run(), in der die eigentliche Aufgabe definiert wird.

- Starten: Erstellen Sie ein Objekt Ihrer Klasse und rufen Sie die Methode start() (nicht run()) auf.
- Nachteil: Da Java keine Mehrfachvererbung bei Klassen unterstützt, kann Ihre Klasse von keiner anderen Klasse mehr erben.

3.1.2 Implementieren des Interfaces java.lang.Runnable

Dies ist die bevorzugte, klassische Methode. Sie kapseln nur die Aufgabe (den Task) in einer Klasse, die das Interface Runnable implementiert, und übergeben diese Aufgabe an ein separates Thread-Objekt.

- Starten: Erstellen Sie ein Thread-Objekt, übergeben Sie Ihre Runnable-Instanz an den Konstruktor und rufen Sie start() auf. (Oft auch kurz mit Lambda-Ausdrücken geschrieben: new Thread(() -> { ... }).start();).
- Vorteil: Saubere Trennung zwischen der eigentlichen Aufgabe (Runnable) und dem Ausführungsmechanismus (Thread).

3.2 Ebene 2: Ergebnisse und Ressourcenmanagement (Mid-Level)

Das direkte Arbeiten mit Thread-Objekten bringt zwei Probleme mit sich: Erstens können Runnable-Aufgaben keinen Rückgabewert liefern und keine Checked Exceptions werfen. Zweitens ist das ständige Erstellen neuer OS-Threads sehr ressourcenintensiv (wie in Kapitel 2 gelernt). Ab Java 5 wurde daher das java.util.concurrent-Paket eingeführt.

3.2.1 Werte zurückgeben: Callable<V> und Future<V>

Um Aufgaben mit einem Rückgabewert zu definieren, verwenden Sie das Interface Callable<V>. Dessen call()-Methode liefert ein Ergebnis vom Typ V zurück und darf Exceptions werfen. Da die Klasse Thread direkt nichts mit Callable anfangen kann, wird die Aufgabe meist an einen Executor übergeben (siehe nächster Punkt). Das Ergebnis wird in Form eines Future-Objekts zurückgeliefert. Ein Future ist eine Art "Gutschein" für ein Ergebnis, das in der Zukunft zur Verfügung stehen wird.

3.2.2 Thread-Pools: Das Executor Framework (ExecutorService)

Anstatt Threads für jede Aufgabe manuell zu erstellen und zu zerstören, verwendet man in professionellen Anwendungen Thread-Pools. Ein Pool hält eine bestimmte Anzahl von Threads am Leben. Wenn eine Aufgabe anliegt, wird sie von einem freien Thread abgearbeitet; danach wartet der Thread auf die nächste Aufgabe.

- Starten: Sie erstellen einen ExecutorService (z. B. über die Factory-Klasse Executors.newFixedThreadPool(4)). Mit der Methode .submit() oder .execute() übergeben Sie Ihre Runnable- oder Callable-Aufgaben an den Pool.

3.2.3 Zeitgesteuerte Ausführung: ScheduledExecutorService

Dies ist eine Spezialform des Executors. Er erlaubt es, Aufgaben nicht sofort, sondern mit einer bestimmten Verzögerung oder periodisch (z. B. "führe dies alle 5 Sekunden aus") nebenläufig auszuführen. (Dies löst die alte und fehleranfällige Klasse java.util.Timer ab).

3.3 Ebene 3: Moderne und spezialisierte High-Level-Paradigmen

Mit dem Aufkommen von Multi-Core-Prozessoren und dem Trend zu funktionaler und reaktiver Programmierung (ab Java 7 und 8) kamen abstraktere Konzepte hinzu, bei denen Sie sich um Threads fast gar nicht mehr kümmern müssen.

3.3.1 Divide and Conquer: Das ForkJoinPool Framework

Speziell für rechenintensive, parallele Aufgaben (Parallelism) gibt es den ForkJoinPool. Er ist darauf optimiert, große Aufgaben immer weiter in kleinere

Teilaufgaben aufzuspalten (fork), diese auf allen verfügbaren Kernen parallel zu berechnen und die Teilergebnisse am Ende wieder zusammenzufügen (join). Er nutzt einen "Work-Stealing"-Algorithmus zur optimalen Auslastung der Kerne.

3.3.2 Deklarative Datenparallelität: Parallele Streams

Mit der Stream-API (Java 8) können Collections auf sehr elegante Weise parallel verarbeitet werden.

- Starten: Sie rufen einfach die Methode .parallelStream() (statt .stream()) auf einer Collection auf.
- Mechanismus: Java kümmert sich komplett selbständig darum, die Daten in Chunks zu zerlegen und diese im Hintergrund (standardmäßig über den gemeinsamen ForkJoinPool.commonPool()) parallel zu verarbeiten. Sie beschreiben nur was getan werden soll, nicht wie die Threads verteilt werden.

3.3.3 Asynchrone/Reaktive Pipelines: CompletableFuture

Während normale Future-Objekte (aus Ebene 2) blockieren, wenn man das Ergebnis mit .get() abfragt, erlauben CompletableFuture (Java 8) den Aufbau von asynchronen, nicht-blockierenden Pipelines.

- Starten: Sie starten eine Aufgabe z. B. mit CompletableFuture.supplyAsync(() -> { ... }).
- Mechanismus: Sie können definieren: "Wenn Aufgabe A fertig ist, reiche das Ergebnis im Hintergrund an Aufgabe B weiter, und falls ein Fehler auftritt, führe Aufgabe C aus" – alles ohne den Haupt-Thread jemals zu blockieren.

3.4 Ebene 4: Virtuelle Threads

Lange Zeit dachte man, das Nonplusultra der Nebenläufigkeit läge in immer komplexeren asynchronen Frameworks. Doch diese machen den Code schwer lesbar und schwer zu debuggen (Stichwort: Callback Hell). Mit Projekt Loom (ausgereift in Java 21) ging Java einen neuen, revolutionären Weg zurück zur Einfachheit.

Virtuelle Threads brechen das 1:1-Mapping von Java-Threads auf OS-Threads auf. Sie werden von der JVM verwaltet (M:N-Mapping). Sie sind so extrem ressourcenschonend, dass eine normale Anwendung Millionen davon gleichzeitig starten kann.

- Starten: Über die Factory-Methoden wie Thread.ofVirtual().start(() -> { ... }); oder in Verbindung mit einem speziellen Executor: Executors.newVirtualThreadPerTaskExecutor().
- Vorteil: Sie schreiben simplen, blockierenden, sequenziell aussehenden Code (wie in Ebene 1), aber das System skaliert besser als die komplexesten asynchronen Pipelines aus Ebene 3. Wenn ein virtueller Thread blockiert (z. B. auf eine Datenbank wartet), löst die JVM ihn lautlos vom OS-Thread und lässt den OS-Thread einen anderen virtuellen Thread abarbeiten.

3.5 Zusammenfassung

Die Wahl des richtigen Werkzeugs hängt vom Anwendungsfall ab. Brauchen Sie nur eine einfache Hintergrundaufgabe? Müssen riesige Datenmengen parallel sortiert werden? Oder bauen Sie einen Webserver, der zehntausende Anfragen gleichzeitig bedient? In den folgenden Kapiteln werden wir jede dieser APIs im Detail beleuchten.

4 Der direkte Weg – Ableiten der Klasse Thread

Nachdem wir im vorherigen Kapitel einen Vogelperspektiven-Blick auf die verschiedenen Möglichkeiten der nebenläufigen Programmierung in Java geworfen haben, widmen wir uns nun der grundlegendsten Methode: der direkten Nutzung der Klasse java.lang.Thread.

Da Sie bereits mit den objektorientierten Konzepten von Java vertraut sind, ist dieser Ansatz sehr intuitiv: Ein Thread repräsentiert in Java ein Objekt, und um einen eigenen, spezialisierten Thread zu erstellen, erzeugen wir einfach eine Unterklasse davon.

4.1 Das Grundprinzip: Thread und run()

Die Klasse Thread bringt bereits alles mit, was das Java-Laufzeitsystem (JVM) benötigt, um mit dem Betriebssystem zu kommunizieren und einen neuen OS-Thread anzufordern.

Um festzulegen, was dieser neue Thread tun soll, müssen wir die Methode run() überschreiben. Die run()-Methode ist quasi das main()-Äquivalent für unseren neuen Ausführungsfaden. Sobald die Methode beendet ist (sei es regulär oder durch eine unbehandelte Exception), stirbt der Thread und wird vom Betriebssystem abgebaut.

Der entscheidende Unterschied: start() vs. run()

Ein klassischer Anfängerfehler ist es, die Methode run() direkt auf dem Thread-Objekt aufzurufen. Wenn Sie meinThread.run() aufrufen, führt Java den Code ganz normal sequenziell im aktuellen Thread (z. B. dem Main-Thread) aus. Es wird kein neuer Thread gestartet!

Um die JVM anzuweisen, einen echten, neuen Ausführungsfaden zu erzeugen, müssen Sie die Methode start() aufrufen. Die Methode start() erledigt die komplexe Kommunikation mit dem Betriebssystem, richtet den neuen Stack für den Thread ein und ruft dann selbstständig asynchron Ihre run()-Methode auf.

4.2 Beispiel 1: Einen einfachen Thread starten

Schauen wir uns das an einem konkreten Beispiel an. Wir erstellen einen Zähler-Thread, der von 1 bis 5 zählt und dazwischen kurz pausiert.

```
// 1. Eigene Klasse erstellen, die von Thread erbt
class ZaehlerThread extends Thread {
    private String name;

    public ZaehlerThread(String name) {
        this.name = name;
```

```
    }

    // 2. Die run()-Methode überschreiben: Hier steht die Aufgabe
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(name + " zählt: " + i);
            try {
                // Thread für 500 Millisekunden schlafen legen
                Thread.sleep(500);
            } catch (InterruptedException e) {
                System.out.println(name + " wurde unterbrochen.");
            }
        }
    }
}

public class ThreadBeispiel1 {
    public static void main(String[] args) {
        System.out.println("Main-Thread startet.");

        // 3. Instanzen unserer Thread-Klasse erzeugen
        ZaehlerThread threadA = new ZaehlerThread("Thread A");
        ZaehlerThread threadB = new ZaehlerThread("Thread B");

        // 4. Threads starten (Nicht run() aufrufen!)
        threadA.start();
        threadB.start();

        System.out
                .println("Main-Thread ist fertig und beendet sich.");
    }
}
```

Erläuterung des Beispiels:

- Wir definieren ZaehlerThread als Unterklasse von Thread.
- In der run()-Methode simulieren wir durch Thread.sleep(500) einen Zeitaufwand. (Da sleep() eine InterruptedException werfen kann, müssen wir diese mit try-catch abfangen).
- In der main()-Methode starten wir zwei dieser Threads.
- Beobachtung bei der Ausführung: Sie werden feststellen, dass die Ausgabe "Main-Thread ist fertig..." oft schon auf der Konsole erscheint, bevor Thread A und B fertig gezählt haben. Der Main-Thread läuft nach dem Aufruf von start() sofort weiter. Die Ausgaben von Thread A und B werden sich auf der Konsole vermischen, da das Betriebssystem (der Scheduler) entscheidet, welcher Thread wann Prozessorzeit erhält.

4.3 Auf das Ende eines Threads warten: join()

Im obigen Beispiel beendet sich der Main-Thread vor den beiden Worker-Threads. Oft ist das nicht erwünscht. Stellen Sie sich vor, zwei Threads berechnen jeweils ein Teilergebnis. Der Main-Thread benötigt beide Teilergebnisse, um das Endergebnis zu berechnen. Er muss also warten, bis die Worker-Threads fertig sind.

Hierfür existiert die Methode join(). Wenn Thread A threadB.join() aufruft, wird Thread A solange blockiert (angehalten), bis threadB seine run()-Methode vollständig beendet hat.

4.4 Beispiel 2: Auf Threads warten mit join()

Erweitern wir unser Wissen mit einem Szenario, in dem der Main-Thread die Kontrolle behält und auf den Abschluss seiner "Arbeiter" wartet.

```
class WorkerThread extends Thread {
    private String taskName;

    public WorkerThread(String taskName) {
        this.taskName = taskName;
    }

    @Override
    public void run() {
        System.out.println(taskName + " beginnt mit der Arbeit...");
        try {
            // Simuliert eine Aufgabe, die 2 Sekunden dauert
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out
                .println(taskName + " hat die Arbeit abgeschlossen.");
    }
}

public class ThreadBeispiel2 {
    public static void main(String[] args) {
        System.out.println("Main: Delegiere Aufgaben an Worker.");

        WorkerThread worker1 = new WorkerThread("Datenbank-Abfrage");
        WorkerThread worker2 = new WorkerThread("Datei-Download");

        worker1.start();
        worker2.start();

        System.out.println(
                "Main: Warte auf die Ergebnisse der Worker...");

        try {
            // Der Main-Thread blockiert hier, bis worker1 fertig ist
            worker1.join();
            // Danach wartet er auf worker2 (falls dieser nicht
            // ohnehin schon fertig ist)
            worker2.join();
```

```
        } catch (InterruptedException e) {
            System.out.println(
                    "Main-Thread wurde beim Warten unterbrochen!");
        }

        // Diese Zeile wird garantiert erst ausgeführt, wenn BEIDE
        // Worker fertig sind
        System.out.println(
                "Main: Alle Worker fertig. Setze Hauptprogramm fort.");
    }
}
```

Erläuterung des Beispiels:

- Die Aufrufe worker1.join() und worker2.join() zwingen den Main-Thread zur Pause.
- Da Threads beim Warten (ähnlich wie beim Schlafen mit sleep()) von außen unterbrochen werden können, deklariert join() eine InterruptedException, die wir per try-catch behandeln müssen.
- Wenn Sie diesen Code ausführen, sehen Sie, dass die finale Konsolenausgabe des Main-Threads immer als Letztes erscheint, unabhängig davon, wie das Betriebssystem die Threads im Hintergrund plant.

4.5 Kritische Würdigung dieses Ansatzes

Das Ableiten von der Klasse Thread ist einfach zu verstehen und schnell implementiert. In der professionellen Softwareentwicklung wird dieser Ansatz heute jedoch selten verwendet. Dafür gibt es zwei Hauptgründe:

1. Keine Mehrfachvererbung: In Java kann eine Klasse nur von einer anderen Klasse erben (ausgenommen Interfaces). Wenn Ihre Klasse ZaehlerThread bereits von Thread erbt, kann sie nicht mehr von einer anderen Basisklasse (z. B. einer fachlichen Klasse wie NetzwerkClient) erben. Sie haben Ihre Vererbungshierarchie "verbraucht".
2. Kopplung von Aufgabe und Ausführung: Objektorientiert betrachtet sollte die Definition einer Aufgabe (Was ist zu tun?) getrennt sein von dem Mechanismus, der sie ausführt (Wie wird sie abgearbeitet?). Beim Erben von Thread verschmelzen das Ausführungsobjekt und die Aufgabe untrennbar miteinander.

5 Entkopplung von Aufgabe und Ausführung – Das Runnable-Interface

Am Ende des letzten Kapitels haben wir festgestellt, dass das Ableiten der Klasse Thread zwei gravierende Nachteile hat: Es blockiert die einzige mögliche Vererbungshierarchie unserer Klasse und es vermischt das Was (die eigentliche Aufgabe) mit dem Wie (der Verwaltung des Betriebssystem-Threads).

Gutes Software-Design strebt jedoch nach "Separation of Concerns" (Trennung der Zuständigkeiten). Eine Klasse sollte idealerweise nur einen bestimmten Zweck erfüllen. Java bietet hierfür eine elegante Lösung: das Interface java.lang.Runnable.

5.1 Das Konzept hinter Runnable

Anstatt einen eigenen Thread zu bauen, definieren wir nur die Aufgabe (den "Task"), die erledigt werden soll.

Das Runnable-Interface in Java ist extrem simpel aufgebaut. Es enthält genau eine einzige, abstrakte Methode:

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

Wenn eine Klasse dieses Interface implementiert, verspricht sie der Java-Laufzeitumgebung lediglich: "Ich besitze eine run()-Methode, in der ausführbarer Code steht." Das Interface selbst weiß absolut nichts von Threads, Nebenläufigkeit oder dem Betriebssystem. Es ist ein reines Datenpaket mit Arbeitsanweisungen.

Um diese Arbeitsanweisung nun nebenläufig auszuführen, nutzen wir das Prinzip der Komposition statt Vererbung (Composition over Inheritance). Wir erzeugen ein Standard-Thread-Objekt und übergeben ihm unsere Runnable-Aufgabe über den Konstruktor.

5.2 Beispiel 1: Der klassische Weg mit Runnable

Schauen wir uns an, wie wir das Beispiel aus dem vorherigen Kapitel umschreiben können. Wir simulieren eine Klasse, die bereits von einer anderen fachlichen Klasse erbt und daher nicht mehr von Thread erben kann.

```
// Eine fiktive Basisklasse aus unserem System
class NetzwerkKomponente {
    public void verbinde() {
        System.out.println("Netzwerkverbindung hergestellt.");
    }
}
```

```java
// Unsere Aufgabe erbt von NetzwerkKomponente UND implementiert Runnable
class DownloadTask extends NetzwerkKomponente implements Runnable {
    private String dateiname;

    public DownloadTask(String dateiname) {
        this.dateiname = dateiname;
    }

    // Hier wird die eigentliche Arbeit definiert
    @Override
    public void run() {
        verbinde(); // Nutzung der Methode aus der Basisklasse
        System.out.println("Starte Download von: " + dateiname);
        try {
            Thread.sleep(1500); // Simuliere Download-Dauer
        } catch (InterruptedException e) {
            System.out.println("Download abgebrochen!");
        }
        System.out.println("Download abgeschlossen: " + dateiname);
    }
}

public class RunnableBeispiel1 {
    public static void main(String[] args) {
        // 1. Die Aufgabe (Task) erstellen
        DownloadTask meinTask = new DownloadTask("daten.zip");

        // 2. Den Arbeiter (Thread) erstellen und ihm die Aufgabe
        // übergeben
        Thread workerThread = new Thread(meinTask,
                "Download-Thread-1");

        // 3. Den Thread starten
        workerThread.start();

        System.out.println("Main-Thread macht andere Dinge...");

        try {
            // Auf das Ende warten funktioniert genau wie vorher!
            // Wir rufen join() auf dem THREAD auf, nicht auf dem
            // Task.
            workerThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main-Thread: Alles fertig.");
    }
}
```

Erläuterung:

1. Wir trennen die Logik: DownloadTask kümmert sich nur um die Fachlogik des Downloads.
2. Wir sind flexibel: DownloadTask erbt von NetzwerkKomponente. Wären wir von Thread abgeleitet, wäre das in Java unmöglich.

3. Warten mit join(): An der Art und Weise, wie wir den Thread starten (start()) oder auf ihn warten (join()), ändert sich absolut nichts. Diese Methoden gehören zur Klasse Thread, die nun als eine Art "Hülle" für unseren Task agiert.

5.3 Beispiel 2: Der moderne Weg mit Lambda-Ausdrücken

Da das Runnable-Interface nur eine einzige abstrakte Methode besitzt, gilt es in Java (seit Version 8) als sogenanntes Functional Interface (Funktionales Interface).

Das bedeutet für Sie als Entwickler einen enormen Komfortgewinn: Sie müssen nicht mehr zwingend eine eigene Klasse schreiben oder eine anonyme innere Klasse verwenden. Sie können die Aufgabe direkt als Lambda-Ausdruck definieren. Dies macht den Code für kurze, einmalige Hintergrundaufgaben extrem kompakt und lesbar.

```
public class RunnableBeispiel2 {
    public static void main(String[] args) {
        System.out.println("Main startet.");

        // Die Aufgabe wird direkt als Lambda-Ausdruck
        // (Arrow-Funktion) definiert
        Runnable schnelleAufgabe = () -> {
            System.out.println("Hallo aus dem Lambda-Thread!");
            for (int i = 0; i < 3; i++) {
                System.out.println("Verarbeite Schritt " + i);
            }
        };

        // Thread erstellen und starten
        Thread thread1 = new Thread(schnelleAufgabe);
        thread1.start();

        // Es geht sogar noch kürzer als Einzeiler (Inline):
        new Thread(() -> System.out
                .println("Ich bin ein anonymer Einzeiler-Thread!"))
                .start();

        System.out.println("Main beendet.");
    }
}
```

Erläuterung: Der Compiler erkennt automatisch, dass der Konstruktor von Thread ein Runnable erwartet. Der Lambda-Ausdruck () -> { ... } wird vom Compiler exakt so übersetzt, als hätten Sie eine Klasse geschrieben, deren run()-Methode den Code in den geschweiften Klammern enthält. Für kleine Nebenläufigkeiten ist dies heute der absolute Standard in Java.

5.4 Zusammenfassung der Vorteile

Warum Sie Runnable der Thread-Vererbung vorziehen sollten:

1. Flexibilität in der Vererbung: Ihre Klasse kann weiterhin von anderen Klassen erben.
2. Wiederverwendbarkeit: Sie können dasselbe Runnable-Objekt an mehrere verschiedene Threads übergeben (Vorsicht: Hierbei müssen wir uns später über "Thread-Sicherheit" unterhalten!).
3. Saubere Architektur: Die Logik der Aufgabe ist strikt getrennt von der Infrastruktur des Threads. Dies erleichtert auch automatisierte Tests (Unit Tests), da Sie die run()-Methode des Tasks einfach im Main-Thread testen können, ohne dass echte Nebenläufigkeit dazwischenfunkt.
4. Vorbereitung auf Thread-Pools: Wenn wir in späteren Kapiteln professionelle Thread-Pools (ExecutorService) nutzen, akzeptieren diese ausschließlich ausgelagerte Tasks wie Runnable (oder Callable). Eine von Thread abgeleitete Klasse könnten wir dort gar nicht sinnvoll einsetzen.

6 Steuerung und Lebenszyklus von Threads

In den vorangegangenen Kapiteln haben wir gelernt, wie wir Aufgaben definieren (Runnable) und sie in einem eigenen Ausführungsstrang (Thread) starten. Wir haben die Threads gestartet und sie dann mehr oder weniger sich selbst überlassen.

Doch in der Praxis reicht "Start and Forget" selten aus. Wir müssen wissen, in welchem Zustand sich ein Thread befindet, wir müssen ihn vielleicht kurz pausieren lassen (sleep) oder – und das ist das wichtigste Thema dieses Kapitels – wir müssen ihn von außen kontrolliert beenden können.

6.1 Identifikation und Kontext: currentThread(), Name und ID

Wenn Sie in einer komplexen Anwendung mehrere Threads starten und Log-Ausgaben in der Konsole betrachten, sehen Sie oft nur ein wildes Durcheinander von Meldungen. Um beim Debugging nachvollziehen zu können, wer gerade was tut, besitzt jeder Thread essenzielle Metadaten zur Identifikation.

Da wir unsere Aufgaben aber idealerweise als Runnable definieren (und ein Runnable selbst kein Thread ist), benötigen wir zunächst eine Brücke zum ausführenden Thread: Thread.currentThread(). Diese statische Methode gibt uns immer genau die Referenz auf das Thread-Objekt zurück, das in genau diesem Moment die aktuelle Codezeile auf der CPU ausführt.

Über dieses Objekt haben wir nun Zugriff auf die Identifikation:

- Die Thread-ID (threadId()): Die JVM weist jedem Thread bei seiner Erstellung automatisch eine eindeutige, fortlaufende Nummer zu. In älteren Java-Versionen nutzte man getId(), ab Java 19 verwendet man die moderne Methode threadId(). Diese ID ist unveränderlich.
- Der Thread-Name (getName() / setName()): Standardmäßig heißt Ihr erster Haupt-Thread "main". Alle selbst gestarteten Threads bekommen generische Namen wie "Thread-0", "Thread-1" usw.

Best Practice: Geben Sie Ihren Threads immer sprechende Namen! Wenn Sie später Ihr Programm analysieren, hilft Ihnen ein "Thread-42" nicht weiter. Ein Thread namens "Datenbank-Verbindungs-Pool-Worker-3" hingegen verrät Ihnen sofort, wo das Problem liegt.

```
public class MeineAufgabe implements Runnable {
    @Override
    public void run() {
        // 1. Welcher Thread führt mich gerade aus?
```

```
        Thread ausfuehrenderThread = Thread.currentThread();

        // 2. Metadaten auslesen
        System.out.println("Ich werde ausgeführt von: "
                + ausfuehrenderThread.getName());
        System.out.println("Die eindeutige ID ist: "
                + ausfuehrenderThread.threadId());
    }

    public static void main(String[] args) {

        // Beim Starten des Threads können wir den Namen direkt vergeben:
        Thread arbeiter = new Thread(new MeineAufgabe(),
                "Hintergrund-Sucher-1");
        // Oder nachträglich: arbeiter.setName("...");
        arbeiter.start();
    }
}
```

6.2 Der Lebenszyklus: Status (state) abfragen

Ein Thread durchläuft während seiner Existenz einen klar definierten Lebenszyklus, den Sie mit thread.getState() von außen abfragen können.

Das Ergebnis ist ein Enum (Thread.State) mit folgenden Werten:

1. NEW: Der Thread wurde erzeugt (new Thread()), aber start() wurde noch nicht gerufen.
2. RUNNABLE: Der Thread läuft auf der CPU oder ist bereit und wartet auf CPU-Zeit vom Betriebssystem.
3. BLOCKED: Der Thread wartet vor einem blockierten Monitor (z. B. einem synchronized-Block).
4. WAITING: Der Thread wartet zeitlich unbegrenzt auf ein Signal (z. B. join()).
5. TIMED_WAITING: Der Thread schläft für eine bestimmte Zeit (Thread.sleep()).
6. TERMINATED: Die run()-Methode ist beendet. Der Thread ist tot und kann niemals neu gestartet werden.

6.3 Zeitsteuerung: sleep und yield

Wie wir beim Thema Scheduling bereits gesehen haben, entscheidet grundsätzlich das Betriebssystem autonom darüber, wann und wie lange ein Thread die CPU nutzen darf. In den meisten Fällen arbeiten wir Entwickler einfach mit dieser Automatik und überlassen dem Scheduler die Arbeit.

Gelegentlich erfordert die Programmlogik jedoch, dass wir aktiv in diesen Ablauf eingreifen. Möglicherweise möchten wir eine künstliche Verzögerung einbauen (etwa bei einer Animation), das System drosseln (z. B. beim zyklischen Prüfen (Polling) eines Verzeichnisses, um die CPU nicht zu überlasten) oder dem Scheduler freundlich signalisieren, dass andere Threads gerade wichtiger sein könnten. Für diese grundlegende Form der zeitlichen Selbststeuerung stellt uns die Klasse Thread zwei statische Methoden zur Verfügung, mit denen der aktuell laufende Thread sein eigenes Ausführungsverhalten beeinflussen kann.

6.3.1 Pausieren mit Thread.sleep()

Mit Thread.sleep(Millisekunden) legen Sie den aktuell ausführenden Thread künstlich schlafen (Status TIMED_WAITING). Die CPU wird sofort für andere Threads freigegeben.

6.3.2 Höfliches Platzmachen mit Thread.yield()

Mit Thread.yield() sagt ein Thread dem Scheduler: "Ich habe gerade nichts Dringendes. Wenn jemand anderes arbeiten will, lass ich ihm den Vortritt." Der Thread bleibt jedoch im Status RUNNABLE. Es ist nur ein nett gemeinter Hinweis, den das Betriebssystem auch komplett ignorieren darf.

6.4 Die große Frage: Wie beendet man einen anderen Thread?

Stellen Sie sich vor, Sie haben einen Hintergrund-Thread gestartet, der ein riesiges Verzeichnis nach Dateien durchsucht. Der Benutzer klickt in der GUI plötzlich auf "Abbrechen". Wie stoppen Sie diesen Such-Thread von außen (z.B. aus dem Main-Thread)?

6.4.1 Der Mythos vom harten Abbruch (stop())

Die intuitive Antwort von Anfängern ist: "Ich rufe einfach suchThread.stop() auf!"

Das ist ein fataler Fehler! In Java (und fast allen modernen Sprachen) dürfen Sie einen Thread niemals von außen gewaltsam abschießen. Die Methode stop() existiert zwar noch aus den 90er Jahren, ist aber strengstens verboten (deprecated).

Warum? Wenn Sie einen Thread von außen "töten", stoppt er mitten in der Bewegung. Er könnte gerade dabei sein, eine Datei zu schreiben oder eine komplexe Datenstruktur im Arbeitsspeicher umzubauen. Wenn er stirbt, lässt er alle Sperren fallen und hinterlässt die Daten in einem völlig zerstörten, unvollständigen Zustand. Das gesamte Programm wird instabil.

6.4.2 Das Paradigma: Kooperativer Abbruch

Die eiserne Regel der Nebenläufigkeit lautet: Ein Thread darf nur von sich selbst beendet werden! Wir können einen Thread von außen nicht töten, wir können ihn nur höflich bitten, sich selbst zu beenden. Der Thread muss so programmiert sein, dass er diese Bitte regelmäßig überprüft und dann freiwillig und sauber seine run()-Methode verlässt (z.B. offene Dateien schließt).

Dafür gibt es zwei etablierte Wege:

Weg 1: Das eigene Flag (Die einfache Variante)

Wir bauen einen eigenen Schalter (boolean) in unsere Aufgabe ein.

```
public class SuchAufgabe implements Runnable {
    // WICHTIG: volatile garantiert, dass Änderungen von außen sofort
    // sichtbar sind!
    private volatile boolean sollStoppen = false;

    public void abbrechen() {
        this.sollStoppen = true; // Methode, die von außen aufgerufen
                                 // wird
    }

    @Override
    public void run() {
        while (!sollStoppen) {
            // Mache einen kleinen Teil der Arbeit
            System.out.println("Suche nach Dateien...");
        }
        System.out.println(
                "Abbruch empfangen. Räume auf und beende mich sauber.");
    }
}
```

Der Haken: Das funktioniert super, solange der Thread aktiv rechnet. Was passiert aber, wenn der Thread gerade Thread.sleep(10000) macht oder auf eine Datenbank-Antwort wartet? Er hängt fest und kann das sollStoppen-Flag minutenlang gar nicht abfragen!

Weg 2: Der professionelle Weg (interrupt())

Um auch schlafende oder wartende Threads sofort abzubrechen, nutzen wir den in Java eingebauten Interrupt-Mechanismus.

Wir setzen von außen das Interrupt-Flag des Threads:

```
    public static void main(String[] args) {
        // Main-Thread:
        Thread suchThread = new Thread(new SuchAufgabe());
        suchThread.start();
        // ... später ...
```

```
        suchThread.interrupt(); // Wirft den Interrupt-Anker!
    }
```

Innerhalb des Such-Threads müssen wir nun auf zwei Dinge achten:

1. Wenn der Thread arbeitet: Wir fragen das Flag mit Thread.currentThread().isInterrupted() ab.
2. Wenn der Thread schläft (sleep oder wait): Java weckt den Thread brutal aus dem Schlaf auf und wirft sofort eine InterruptedException! Genau deshalb zwingt uns Java bei jedem sleep() zu einem try-catch-Block.

So sieht ein professionell abbrechbarer Thread aus:

```
public class ProfiSuchAufgabe implements Runnable {
    @Override
    public void run() {
        // Prüfe bei jedem Schleifendurchlauf das eingebaute Flag
        while (!Thread.currentThread().isInterrupted()) {
            System.out.println("Suche...");

            try {
                // Simuliere Wartezeit. Hier könnte ein Interrupt
                // einschlagen!
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Der Thread wurde im SCHLAF unterbrochen!
                System.out
                        .println("Ich wurde im Schlaf unterbrochen!");
                // WICHTIG: Die Exception löscht das Flag! Wir setzen
                // es wieder, damit die while-Schleife oben beim
                // nächsten Mal false ergibt.
                Thread.currentThread().interrupt();
                break; // Schleife sofort verlassen
            }
        }
        System.out.println("Thread beendet sich geordnet.");
    }
}
```

Merksatz: Beenden Sie Threads niemals mit Gewalt. Nutzen Sie interrupt() und programmieren Sie die run()-Methode so, dass sie kooperativ auf dieses Signal reagiert.

6.5 Daemon-Threads und Prioritäten

Zum Abschluss betrachten wir zwei Einstellungen, die das Gewicht eines Threads im Gesamtsystem beeinflussen.

6.5.1 Daemon-Threads (setDaemon(true))

Normalerweise läuft eine Java-Anwendung so lange, bis auch der letzte gestartete Thread seine run()-Methode beendet hat. Manchmal wollen wir aber Hintergrunddienste (z.B. einen Garbage Collector oder einen Auto-Save-Prozess), die das Beenden der Hauptanwendung nicht blockieren sollen.

Diese markieren wir als Daemon-Threads (Schutzgeister im Hintergrund). meinThread.setDaemon(true); (Muss zwingend VOR start() aufgerufen werden). Sobald der Main-Thread und alle anderen normalen User-Threads enden, schaltet sich die JVM ab und reißt alle Daemon-Threads sofort mit sich (hier gibt es keinen kooperativen Abbruch, die JVM wird einfach hart beendet).

6.5.2 Thread-Prioritäten (setPriority())

Sie können Threads eine Priorität von 1 (Thread.MIN_PRIORITY) bis 10 (Thread.MAX_PRIORITY) geben. Die ungeschönte Wahrheit: Verlassen Sie sich niemals darauf! Ob und wie diese Prioritäten beachtet werden, hängt vollständig vom Scheduler des Betriebssystems (Windows, Linux, macOS) ab. Bauen Sie niemals eine Programm-Logik, die zwingend erfordert, dass ein Thread mit Priorität 10 schneller abgearbeitet wird als einer mit Priorität 1.

6.6 Fazit

Sie haben nun das vollständige Werkzeugset, um Threads zu identifizieren, ihren Zustand zu überwachen und sie – ganz wichtig – kooperativ und sicher von außen abzubrechen.

Doch die Arbeit mit bloßen Runnable-Objekten hat noch einen fundamentalen Haken: Was ist, wenn unsere Aufgabe ein Ergebnis berechnet, das wir an den aufrufenden Thread zurückgeben wollen? Die run()-Methode hat den Rückgabetyp void.

Dafür müssen wir unser Wissen auf das nächste Level heben. Schauen wir uns im nun folgenden Kapitel an, wie wir Werte aus asynchronen Berechnungen elegant extrahieren können.

7 Ergebnisse aus Threads abrufen – Callable und Future

In den vorherigen Kapiteln haben wir gelernt, wie wir Aufgaben definieren (Runnable) und sie von Threads im Hintergrund abarbeiten lassen. Das funktioniert hervorragend für Aufgaben, die einfach nur "gemacht" werden müssen, wie etwa das asynchrone Schreiben einer Log-Datei oder das Starten eines Downloads.

Aber was ist, wenn unsere Aufgabe etwas berechnen soll und wir das Ergebnis im Main-Thread benötigen?

7.1 Das Problem mit Runnable

Schauen wir uns das Runnable-Interface noch einmal genau an:

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

Es gibt hier zwei gravierende architektonische Einschränkungen:

1. Kein Rückgabewert: Die Methode run() hat den Rückgabetyp void. Sie kann dem aufrufenden Thread nicht einfach mit return ergebnis; einen Wert zurückgeben. Man müsste Umwege über geteilte Variablen (Shared Memory) gehen, was komplex und fehleranfällig ist (Thema Thread-Sicherheit, worüber wir später noch sprechen).

2. Keine Checked Exceptions: Die run()-Methode darf keine geprüften Ausnahmen (Checked Exceptions wie IOException oder SQLException) nach außen werfen. Tritt ein Fehler auf, muss er zwingend innerhalb von run() mit try-catch behandelt werden. Der Main-Thread erfährt im schlimmsten Fall gar nicht, dass der Worker-Thread wegen eines Fehlers abgebrochen ist.

Um diese Probleme zu lösen, führte Java ab Version 5 das Paket java.util.concurrent ein, und damit zwei neue, extrem wichtige Interfaces: Callable<V> und Future<V>.

7.2 Das Callable-Interface: Die bessere Aufgabe

Das Interface Callable ist das moderne, mächtigere Geschwisterchen von Runnable. Es ist generisch (das <V> steht für den Typ des Rückgabewerts) und sieht so aus:

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

Die Vorteile liegen auf der Hand:

- Die Methode heißt call() statt run().
- Sie liefert ein Objekt vom Typ V zurück.
- Sie deklariert throws Exception, darf also alle Arten von Fehlern an den Aufrufer weiterreichen.

7.3 Das Future-Interface: Der Abholschein für das Ergebnis

Wenn wir eine Callable-Aufgabe in einem anderen Thread starten, ist das Ergebnis natürlich nicht sofort da. Der Main-Thread läuft ja parallel weiter.

Hier kommt das Future-Interface ins Spiel. Wenn Sie eine Callable-Aufgabe an das System übergeben, erhalten Sie sofort ein Future-Objekt zurück. Stellen Sie sich das wie einen Abholschein in der Reinigung vor: Sie geben Ihre schmutzige Wäsche (die Aufgabe) ab, bekommen sofort den Abholschein (Future) und können später mit diesem Schein prüfen, ob die Wäsche fertig ist (isDone()), oder sich an den Tresen stellen und warten, bis sie Ihnen ausgehändigt wird (get()).

7.4 Wie führt man ein Callable aus?

Hier gibt es einen kleinen Haken: Die klassische Klasse Thread kennt nur Runnable. Man kann ihr kein Callable übergeben. Um Callables auszuführen, nutzen wir einen sogenannten ExecutorService (eine Art Manager für Threads, den wir im nächsten Kapitel noch viel genauer im Rahmen von Thread-Pools behandeln werden). Für den Moment reicht es zu wissen, dass wir uns vom System einen einfachen Executor geben lassen können, der unsere Aufgaben übernimmt.

7.5 Beispiel 1: Eine komplexe Berechnung mit Rückgabewert

Lassen Sie uns eine Aufgabe schreiben, die die Summe aller Zahlen bis zu einem bestimmten Limit berechnet und dieses Ergebnis zurückgibt.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableBeispiel {
    public static void main(String[] args) {
        System.out.println("Main: Erstelle Executor...");
        // Wir holen uns einen simplen Manager, der genau einen
        // Hintergrund-Thread besitzt
        ExecutorService executor = Executors
                .newSingleThreadExecutor();
```

```
        // 1. Die Aufgabe (Callable) als Lambda-Ausdruck definieren
        Callable<Long> summenRechner = () -> {
            System.out.println(
                    "Worker: Starte langwierige Berechnung...");
            long summe = 0;
            for (long i = 1; i <= 1_000_000_000L; i++) {
                summe += i;
            }
            Thread.sleep(1000); // Simuliere extra Zeitaufwand
            System.out.println("Worker: Berechnung fertig!");
            return summe; // Hier geben wir tatsächlich einen Wert
                          // zurück!
        };

        System.out.println("Main: Übergebe Aufgabe an den Executor.");
        // 2. Aufgabe einreichen und den "Abholschein" (Future)
        // entgegennehmen
        Future<Long> futureErgebnis = executor.submit(summenRechner);

        System.out.println(
                "Main: Mache andere Dinge, während der Worker rechnet...");

        try {
            // 3. Ergebnis abholen.
            // ACHTUNG: Die Methode get() BLOCKIERT den Main-Thread
            // solange, bis das Ergebnis tatsächlich vorliegt (ähnlich
            // wie join() bei Threads).
            Long result = futureErgebnis.get();
            System.out
                    .println("Main: Das Ergebnis ist da: " + result);

        } catch (InterruptedException e) {
            System.out.println(
                    "Main-Thread wurde beim Warten unterbrochen.");
        } catch (ExecutionException e) {
            System.out.println(
                    "Fehler INNERHALB der Callable-Aufgabe!");
        } finally {
            // Einen Executor muss man am Ende ordnungsgemäß
            // herunterfahren
            executor.shutdown();
        }
    }
}
```

Erläuterung:

- Mit executor.submit(...) wird der Hintergrund-Thread gestartet.
- Der Main-Thread läuft danach sofort weiter und druckt "Mache andere Dinge...".
- Sobald der Main-Thread futureErgebnis.get() aufruft, muss er warten (blockieren), bis der Worker-Thread das return summe; ausgeführt hat.

7.6 Beispiel 2: Exceptions sauber behandeln

Was passiert, wenn in unserem Callable ein Fehler auftritt (z.B. eine Division durch Null oder eine fehlende Datei)? Die Exception bringt nicht das ganze Programm sofort zum Absturz. Stattdessen fängt das Java-Framework die Exception ab, speichert sie im Future-Objekt und wirft sie erst dann, wenn der Main-Thread versucht, das Ergebnis mit get() abzuholen!

Sie wird dabei in eine ExecutionException verpackt.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableExceptionBeispiel {
    public static void main(String[] args) {
        ExecutorService executor = Executors
                .newSingleThreadExecutor();

        // Callable, das absichtlich einen Fehler provoziert
        Callable<Integer> fehlerhafteAufgabe = () -> {
            System.out.println(
                    "Worker: Versuche durch Null zu teilen...");
            return 10 / 0; // Wirft eine ArithmeticException
        };

        Future<Integer> future = executor.submit(fehlerhafteAufgabe);

        try {
            System.out.println("Main: Versuche Ergebnis zu holen...");
            Integer ergebnis = future.get(); // Hier knallt es!
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            // Mit getCause() holen wir die URSPRÜNGLICHE Exception
            // aus dem Worker-Thread
            System.out
                    .println("Main: Die Aufgabe ist fehlgeschlagen!");
            System.out.println(
                    "Ursache war: " + e.getCause().getMessage());
        }

        executor.shutdown();
    }
}
```

Erläuterung: Dieses Muster ist extrem mächtig. Der Main-Thread behält die volle Kontrolle. Er delegiert Aufgaben und kann später zentral und sauber entscheiden, wie er reagiert, falls einer der "Arbeiter" (Worker-Threads) ein Problem meldet.

8 Ressourcen intelligent verwalten – Thread-Pools und das Executor-Framework

In den vorangegangenen Kapiteln haben wir gelernt, wie wir Aufgaben (Runnable und Callable) definieren und diese in eigenen Ausführungsfäden (Threads) starten. Bisher haben wir dafür fast immer implizit oder explizit einen neuen Thread erzeugt.

Für kleine Skripte oder Demo-Anwendungen ist das völlig in Ordnung. In realen, großen Systemen – wie beispielsweise einem Webserver, der tausende Nutzeranfragen pro Sekunde verarbeitet – führt dieser Ansatz jedoch unweigerlich in eine Katastrophe. Warum das so ist und wie die Java-API dieses Problem löst, klären wir in diesem Kapitel.

8.1 Das Problem: Der Overhead der Thread-Erstellung

Erinnern wir uns an Kapitel 2: Ein klassischer Java-Thread ist direkt an einen Betriebssystem-Thread (OS-Thread) gekoppelt (1:1-Mapping).

Das Erstellen eines solchen OS-Threads ist teuer. Das Betriebssystem muss Speicherplatz für den Stack des Threads reservieren (oft 1 MB oder mehr) und interne Verwaltungsstrukturen aufbauen. Wenn unser Webserver für jede der 10.000 eingehenden Anfragen pro Sekunde ein new Thread(task).start() aufruft, verbringt die CPU mehr Zeit mit dem Erstellen und Zerstören von Threads als mit der eigentlichen Arbeit. Zudem würde der Arbeitsspeicher rasant volllaufen, was zu einem OutOfMemoryError führt und die Anwendung zum Absturz bringt.

Die Lösung für dieses Problem ist ein bewährtes Entwurfsmuster: das Pooling.

8.2 Die Lösung: Der Thread-Pool

Ein Thread-Pool (Faden-Becken) ist eine Sammlung von bereits erstellten, "aufgewärmten" und einsatzbereiten Threads.

Das Prinzip funktioniert wie bei einer Taxiflotte:

1. Anstatt für jeden Fahrgast (Aufgabe) ein neues Auto zu bauen (Thread) und es nach der Fahrt zu verschrotten, hält das Taxiunternehmen eine feste Flotte von Autos bereit.
2. Wenn ein Fahrgast anruft, steigt er in ein freies Taxi.

3. Ist kein Taxi frei, muss der Fahrgast in einer Warteschlange (Task Queue) warten.
4. Sobald ein Taxi eine Fahrt beendet hat, fährt es nicht auf den Schrottplatz, sondern holt den nächsten Fahrgast aus der Warteschlange ab.

Dieses Konzept senkt den Overhead dramatisch und begrenzt gleichzeitig die maximale Anzahl an Threads, sodass das System nicht überlastet wird.

8.3 Das Executor-Framework in Java

Seit Java 5 müssen (und sollten) wir Thread-Pools nicht mehr selbst programmieren. Das Paket java.util.concurrent stellt das Executor-Framework zur Verfügung. Es trennt die Erstellung von Aufgaben vollständig von deren Ausführung.

Die wichtigsten Schnittstellen (Interfaces) in diesem Framework sind:

- Executor: Ein sehr simples Interface mit nur einer Methode: execute(Runnable). Es definiert lediglich, dass eine Aufgabe ausgeführt wird, aber nicht wie.
- ExecutorService: Erweitert Executor um lebenswichtige Management-Funktionen. Es erlaubt das Ausführen von Callable-Aufgaben (via submit()), liefert Future-Objekte zurück und bietet Methoden, um den Pool kontrolliert herunterzufahren.

Die Factory-Klasse Executors

Um einen ExecutorService zu erstellen, rufen wir in der Regel nicht direkt einen Konstruktor auf (wie new ThreadPoolExecutor(...)), da dieser extrem viele komplexe Parameter verlangt. Stattdessen nutzen wir die Hilfsklasse Executors, die uns fertige Fabrikmethoden (Factory Methods) für die gängigsten Pool-Arten bietet:

1. Executors.newFixedThreadPool(int nThreads): Erstellt einen Pool mit einer festen, unveränderlichen Anzahl von Threads. Wenn mehr Aufgaben eingereicht werden als Threads vorhanden sind, werden diese in einer unbegrenzten Schlange zwischengespeichert, bis ein Thread frei wird. (Ideal für Server-Anwendungen mit vorhersehbarer Last).
2. Executors.newCachedThreadPool(): Ein extrem flexibler Pool. Er erstellt bei Bedarf neue Threads, wenn alle vorhandenen beschäftigt sind. Wenn ein Thread jedoch 60 Sekunden lang keine Aufgabe mehr bekommt, wird er abgebaut.

(Ideal für viele kleine, kurze Aufgaben, birgt aber die Gefahr von zu vielen Threads bei dauerhaft hoher Last).

3. Executors.newSingleThreadExecutor(): Ein Pool, der exakt einen Thread besitzt. Alle eingereichten Aufgaben werden streng sequenziell nacheinander (FIFO – First In, First Out) abgearbeitet. (Ideal, wenn Aufgaben auf keinen Fall parallel laufen dürfen, z.B. beim Schreiben in eine lokale Datei).

8.4 Beispiel: Arbeiten mit dem FixedThreadPool

Schauen wir uns an, wie wir 10 Aufgaben an einen Pool übergeben, der aber nur 3 Threads zur Verfügung hat.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolBeispiel {
    public static void main(String[] args) {
        System.out.println(
                "Main: Erstelle Thread-Pool mit 3 Arbeitern.");

        // Wir beauftragen 3 fest angestellte Arbeiter
        ExecutorService pool = Executors.newFixedThreadPool(3);

        // Wir haben 10 Aufgaben zu erledigen
        for (int i = 1; i <= 10; i++) {
            final int aufgabenNummer = i;

            // Aufgabe als Runnable (Lambda) definieren
            Runnable aufgabe = () -> {
                String threadName = Thread.currentThread().getName();
                System.out.println(threadName + " bearbeitet Aufgabe "
                        + aufgabenNummer);
                try {
                    // Simuliere Arbeitszeit (1 Sekunde)
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(threadName + " hat Aufgabe "
                        + aufgabenNummer + " beendet.");
            };

            // Aufgabe an den Pool übergeben. Der Pool entscheidet
            // selbst, welcher der 3 Threads sie abarbeitet.
            pool.execute(aufgabe);
        }

        System.out.println(
                "Main: Alle 10 Aufgaben wurden beim Pool eingereicht.");

        // WICHTIG: Den Pool am Ende schließen!
        pool.shutdown();
    }
}
```

Was passiert bei der Ausführung? Wenn Sie dieses Programm starten, werden Sie sehen, dass nur drei Threads (z.B. pool-1-thread-1, pool-1-thread-2, pool-1-thread-3) aktiv sind. Sie bearbeiten die Aufgaben 1, 2 und 3. Die Aufgaben 4 bis 10 warten in der Schlange. Nach einer Sekunde werden die ersten drei fertig, behalten aber ihr Leben und übernehmen sofort die Aufgaben 4, 5 und 6 – und so weiter. Die Threads werden recycelt!

8.5 Den Lebenszyklus managen: Das richtige Herunterfahren

Ein entscheidendes Detail im obigen Code ist der Aufruf von pool.shutdown().

Ein ExecutorService erzeugt sogenannte Non-Daemon-Threads. Solange mindestens ein solcher Thread in einer Java-Anwendung lebt, beendet sich die Java Virtual Machine (JVM) nicht. Selbst wenn der Main-Thread komplett fertig ist, würde Ihr Programm unendlich lange weiterlaufen und auf neue Aufgaben für den Pool warten.

Daher müssen Sie dem Pool explizit mitteilen, dass seine Arbeit beendet ist:

- shutdown(): Der "sanfte" Weg. Der Pool nimmt keine neuen Aufgaben mehr an (wirft eine RejectedExecutionException, falls man es doch versucht). Er arbeitet aber alle Aufgaben, die bereits eingereicht wurden oder in der Warteschlange liegen, noch gewissenhaft ab. Danach beenden sich die Threads und die JVM kann stoppen.
- shutdownNow(): Der "harte" Weg. Der Pool versucht, alle aktuell laufenden Threads sofort zu unterbrechen (via InterruptedException) und gibt eine Liste aller Aufgaben zurück, die noch in der Schlange lagen und nicht mehr gestartet wurden.
- awaitTermination(long timeout, TimeUnit unit): Blockiert den aufrufenden Thread (z.B. den Main-Thread), bis der Pool nach einem shutdown() wirklich alle verbleibenden Aufgaben erledigt hat, oder bis die Wartezeit (Timeout) abgelaufen ist (ähnlich wie join() bei einzelnen Threads).

8.6 Massenabfertigung: invokeAll und invokeAny

Bisher haben wir unsere Aufgaben immer einzeln mit den Methoden execute() oder submit() an den ExecutorService übergeben. In der Praxis stehen wir jedoch häufig vor der Situation, dass wir nicht nur eine einzelne Aufgabe, sondern eine ganze Liste gleichartiger Aufgaben haben, die wir als Paket verarbeiten wollen. Anstatt in einer Schleife dutzende Male submit() aufzurufen und die zurückgegebenen Future-Objekte mühsam in einer Liste zu sammeln, bietet uns das ExecutorService-Interface zwei

elegante Komfort-Methoden für die Massenverarbeitung (Bulk-Operationen): invokeAll und invokeAny.

Beide Methoden erwarten als Parameter eine Collection (z. B. eine List oder ein Set), die mit Callable-Objekten gefüllt ist. Doch in der Art und Weise, wie sie auf die Ergebnisse warten, könnten sie unterschiedlicher nicht sein.

8.6.1 Alle oder keiner: invokeAll()

Die Methode invokeAll() ist der geduldige Wächter. Sie nimmt Ihre Liste von Aufgaben entgegen, verteilt sie an die Threads im Pool und blockiert dann den aufrufenden Thread, bis ausnahmslos alle Aufgaben abgeschlossen sind.

Stellen Sie sich einen Lehrer vor, der eine Klausur schreiben lässt: Er verlässt den Raum erst, wenn auch der letzte Schüler seinen Test abgegeben hat.

```
List<Callable<String>> aufgaben = Arrays.asList(
        () -> ladeDatenVonServer("Server 1"),
        () -> ladeDatenVonServer("Server 2"),
        () -> ladeDatenVonServer("Server 3"));

ExecutorService executor = Executors.newFixedThreadPool(3);

try {
    // Der Haupt-Thread blockiert hier, bis alle 3 Aufgaben
    // fertig sind!
    List<Future<String>> ergebnisse = executor
            .invokeAll(aufgaben);

    for (Future<String> future : ergebnisse) {
        // future.get() blockiert hier nicht mehr, da wir
        // wissen,
        // dass bereits alle fertig sind.
        System.out.println("Ergebnis: " + future.get());
    }
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

Das Praktische an invokeAll ist, dass Sie nach der Rückkehr der Methode sicher sein können: Für jedes Future in der zurückgegebenen Liste liefert der Aufruf von isDone() den Wert true. Die Aufgaben können entweder erfolgreich beendet worden sein oder eine Exception geworfen haben – aber keine einzige Aufgabe läuft mehr.

Tipp aus der Praxis: invokeAll gibt es auch in einer überladenen Variante, der Sie einen Timeout (z. B. "warte maximal 5 Sekunden") mitgeben können. Werden nicht alle Aufgaben in dieser Zeit fertig, bricht der ExecutorService die noch laufenden Aufgaben gnadenlos ab.

8.6.2 Der Schnellste gewinnt: invokeAny()

Während invokeAll auf Vollständigkeit setzt, geht es bei invokeAny() nur um pure Geschwindigkeit. Diese Methode nimmt ebenfalls Ihre Liste von Aufgaben, wartet aber nur darauf, dass die allererste Aufgabe erfolgreich beendet wird.

Ein klassisches Beispiel aus dem echten Leben: Sie haben einen wichtigen Termin und rufen gleichzeitig drei verschiedene Taxi-Unternehmen an. In das Taxi, das als Erstes vor Ihrer Tür steht, steigen Sie ein. Die anderen beiden Bestellungen stornieren Sie.

Genau das macht invokeAny(): Sobald ein Callable ein Ergebnis liefert, wird dieses Ergebnis direkt zurückgegeben (es wird kein Future zurückgegeben, sondern direkt das fertige Objekt). Alle anderen, noch laufenden oder wartenden Aufgaben aus der übergebenen Liste werden vom ExecutorService automatisch abgebrochen (per interrupt()).

```
List<Callable<String>> redundanteAufgaben = Arrays.asList(
        () -> frageDatenAb("API Provider A"), // Dauert 200ms
        () -> frageDatenAb("API Provider B"), // Dauert 50ms
        () -> frageDatenAb("API Provider C") // Dauert 500ms
);

ExecutorService executor = Executors.newFixedThreadPool(3);

try {
    // Blockiert, bis die ERSTE Aufgabe fertig ist.
    // In unserem Fall wird Provider B gewinnen.
    String schnellstesErgebnis = executor
            .invokeAny(redundanteAufgaben);

    System.out.println("Das schnellste Ergebnis war: "
            + schnellstesErgebnis);
    // Die Abfragen an Provider A und C wurden automatisch
    // abgebrochen!

} catch (InterruptedException | ExecutionException e) {
    // Wird nur geworfen, wenn ALLE Aufgaben fehlgeschlagen
    // sind
    e.printStackTrace();
}
```

Wann brauchen wir das? invokeAny ist das perfekte Werkzeug für Redundanz. Wenn Sie beispielsweise einen DNS-Server oder eine verteilte Datenbank abfragen, können Sie denselben Request an drei verschiedene Knotenpunkte schicken. Der Knoten, der gerade am wenigsten ausgelastet ist, wird zuerst antworten. Sie nehmen diese schnelle Antwort und sparen Systemressourcen, indem die anderen, langsameren Abfragen sofort verworfen werden. Wenn eine der Aufgaben mit einer Exception fehlschlägt, ignoriert invokeAny diesen Fehler einfach und wartet auf das Ergebnis der nächsten

Aufgabe. Erst wenn alle Aufgaben fehlschlagen, wirft die Methode eine ExecutionException.

8.7 Ergebnisse sofort verarbeiten: Der CompletionService

Wir haben gerade zwei Extreme der Massenabfertigung kennengelernt: Mit invokeAll warten wir geduldig, bis restlos alle Aufgaben erledigt sind. Mit invokeAny schnappen wir uns das erste Ergebnis und werfen den Rest unbarmherzig weg. Doch in der Praxis gibt es sehr oft einen Mittelweg: Wir wollen zwar alle Ergebnisse haben, aber wir möchten jedes einzelne Ergebnis sofort verarbeiten, sobald es verfügbar ist – unabhängig davon, wann die anderen Aufgaben fertig werden.

Betrachten wir das klassische Problem: Wenn Sie Aufgaben einzeln mit submit() an einen ExecutorService übergeben und die zurückgegebenen Future-Objekte in einer Liste sammeln, müssen Sie diese Liste später in einer Schleife durchlaufen. Rufen Sie nun beim ersten Future in der Liste get() auf, blockiert Ihr Thread so lange, bis genau diese erste Aufgabe fertig ist. Was aber, wenn Aufgabe 1 extrem aufwendig ist und zehn Sekunden dauert, während Aufgabe 2 und 3 bereits nach einer halben Sekunde fertig sind? Ihr Haupt-Thread hängt bei Aufgabe 1 fest und die fertigen Ergebnisse von Aufgabe 2 und 3 liegen nutzlos herum.

Stellen Sie sich einen Kellner vor, der drei Bestellungen in der Küche abgibt: ein aufwendiges Steak (Tisch 1) und zwei schnelle Salate (Tisch 2 und 3). Wenn der Kellner stur am Tresen steht und darauf beharrt, zuerst das Steak mitzunehmen, werden die Salate inzwischen welk, obwohl er sie längst hätte servieren können.

Genau für dieses Problem bietet Java den CompletionService (und dessen Standardimplementierung ExecutorCompletionService). Er fungiert als intelligenter Zwischenspeicher, der die Abgabe von Aufgaben von der Abholung der Ergebnisse entkoppelt.

Der CompletionService stützt sich intern auf eine blockierende Warteschlange (BlockingQueue). Wenn ein Thread im Pool mit einer Aufgabe fertig ist, legt er das fertige Future sofort in diese Warteschlange. Sie als Entwickler holen die Ergebnisse dann nicht mehr aus Ihrer eigenen Liste ab, sondern ziehen sie direkt aus dem CompletionService.

So sieht das in der Praxis aus:

```
// Wir nutzen einen normalen ExecutorService als Motor
ExecutorService executor = Executors.newFixedThreadPool(3);

// Wir wickeln den Executor in einen CompletionService ein
```

```
CompletionService<String> completionService =
  new ExecutorCompletionService<>(executor);

// Wir übergeben Aufgaben mit extrem unterschiedlichen
// Laufzeiten
completionService.submit(() -> {
    Thread.sleep(5000); // Braucht 5 Sekunden
    return "Aufwendiges Steak (Tisch 1)";
});
completionService.submit(() -> {
    Thread.sleep(500); // Braucht nur 0.5 Sekunden
    return "Schneller Salat (Tisch 2)";
});
completionService.submit(() -> {
    Thread.sleep(1000); // Braucht 1 Sekunde
    return "Suppe (Tisch 3)";
});

try {
    // Wir wissen, dass wir 3 Aufgaben übergeben haben, also
    // fragen wir 3 mal ab
    for (int i = 0; i < 3; i++) {
        // take() blockiert, bis IRGENDEINE Aufgabe fertig ist
        Future<String> fertigesFuture = completionService
                .take();

        // Da wir das Future vom CompletionService bekommen
        // haben, ist es garantiert fertig. get() blockiert
        // hier also nicht mehr!
        String gericht = fertigesFuture.get();
        System.out.println("Serviere sofort: " + gericht);
    }
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
} finally {
    executor.shutdown();
}
```

Die Ausgabe dieses Programms verdeutlicht die Magie:

1. Nach 0.5 Sekunden: Serviere sofort: Schneller Salat (Tisch 2)
2. Nach 1.0 Sekunden: Serviere sofort: Suppe (Tisch 3)
3. Nach 5.0 Sekunden: Serviere sofort: Aufwendiges Steak (Tisch 1)

Obwohl das Steak als erstes bestellt (submit) wurde, hat der CompletionService dafür gesorgt, dass wir die schnellen Gerichte zuerst servieren konnten.

Wichtige Methoden des CompletionService:

- submit(Callable<V> task): Übergibt die Aufgabe an den internen Executor.
- take(): Holt das nächste fertige Future aus der internen Warteschlange und entfernt es. Wenn noch keine Aufgabe fertig ist, blockiert der aufrufende Thread, bis das erste Ergebnis eintrifft.

- poll(): Ähnlich wie take(), blockiert aber nicht. Wenn kein Ergebnis bereitliegt, liefert die Methode sofort null zurück. Es gibt auch eine überladene Variante poll(long timeout, TimeUnit unit), die für eine definierte Zeitspanne wartet.

Der CompletionService ist das Werkzeug der Wahl, wenn Sie Benutzeroberflächen reaktionsschnell halten wollen (z.B. Ladebalken für jedes einzelne fertige Element aktualisieren) oder wenn Sie eine große Menge an asynchronen I/O-Operationen (wie Netzwerkabfragen) starten und die Antworten so schnell wie möglich weiterverarbeiten möchten.

9 Zeitgesteuerte und periodische Aufgaben – Der ScheduledExecutorService

In dem vorherigen Kapitel haben wir gelernt, wie wir Aufgaben an einen Thread-Pool übergeben. Bisher galt dabei immer die Prämisse: Die Aufgabe soll so schnell wie möglich abgearbeitet werden. Sobald ein Thread im Pool frei ist, stürzt er sich auf die nächste Aufgabe in der Warteschlange.

Hin und wieder haben wir in der Softwareentwicklung jedoch andere Anforderungen:

- "Lösche temporäre Dateien, aber erst in 10 Minuten."
- "Sende alle 24 Stunden einen Statusbericht per E-Mail."
- "Prüfe alle 5 Sekunden, ob die Datenbankverbindung noch aktiv ist (Heartbeat)."

Für solche zeitgesteuerten (scheduled) Aufgaben bietet das Executor-Framework eine spezialisierte Erweiterung: den ScheduledExecutorService.

9.1 Warum nicht einfach Thread.sleep() oder java.util.Timer?

Bevor wir uns den modernen Weg ansehen, müssen wir kurz klären, warum wir nicht auf ältere oder simplere Mittel zurückgreifen sollten.

1. Thread.sleep() in einer Endlosschleife: Man könnte einen normalen Thread starten, der eine Aufgabe ausführt und sich danach mit Thread.sleep(5000) für 5 Sekunden schlafen legt, bevor die Schleife von vorn beginnt. Das Problem: Der Thread bleibt die ganze Zeit über an den teuren OS-Thread gebunden, auch wenn er nur "schläft" und nichts tut. Das verschwendet massiv Systemressourcen.
2. Die veraltete Klasse java.util.Timer: Vor Java 5 gab es die Klassen Timer und TimerTask. Diese gelten heute als veraltet (obsolete), da sie ein großes Design-Problem haben: Ein Timer nutzt im Hintergrund exakt einen einzigen Thread für alle seine Aufgaben. Wenn eine periodische Aufgabe zu lange dauert, verzögert sie alle anderen Aufgaben dieses Timers. Wirft eine Aufgabe eine Runtime-Exception, stürzt der gesamte Timer-Thread ab und keine der geplanten Aufgaben wird jemals wieder ausgeführt.

Der ScheduledExecutorService löst all diese Probleme, da er auf echten Thread-Pools basiert.

9.2 Den ScheduledExecutorService erstellen

Genauso wie den normalen ExecutorService erstellen wir auch diese spezialisierte Variante über die Factory-Klasse Executors.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;

// Erstellt einen Pool, der speziell für zeitgesteuerte
// Aufgaben optimiert ist
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(2);
```

Der Parameter 2 gibt die Core Pool Size an, also die Mindestanzahl an Threads, die im Pool am Leben gehalten werden, um die geplanten Aufgaben pünktlich abzuarbeiten.

Dieses Interface bietet uns nun drei zentrale Methoden, die wir uns im Detail ansehen.

9.3 Einmalige Ausführung mit Verzögerung: schedule

Wenn eine Aufgabe nur ein einziges Mal, aber erst nach einer bestimmten Wartezeit ausgeführt werden soll, nutzen wir die Methode schedule(). Sie akzeptiert sowohl Runnable als auch Callable.

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

public class ScheduleBeispiel {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler = Executors
                .newScheduledThreadPool(1);

        System.out.println("Main: Aufgabe wird eingeplant...");

        // Ein Callable, das nach 3 Sekunden ein Ergebnis liefert
        Callable<String> verzögerteAufgabe = () -> {
            System.out.println("Worker: Führe Aufgabe aus!");
            return "Erfolgreich gelöscht";
        };

        // Aufgabe einplanen: (Aufgabe, Verzögerung, Zeiteinheit)
        ScheduledFuture<String> future = scheduler
                .schedule(verzögerteAufgabe, 3, TimeUnit.SECONDS);

        try {
            // Blockiert, bis die 3 Sekunden um sind und das Ergebnis
            // da ist
            String ergebnis = future.get();
            System.out
                    .println("Main: Ergebnis erhalten: " + ergebnis);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
```

```
        scheduler.shutdown();
    }
}
```

Das Besondere an der Klasse TimeUnit (ein Enum): Sie macht den Code extrem gut lesbar. Sie können anstatt unzähliger Millisekunden einfach TimeUnit.MINUTES, TimeUnit.HOURS oder TimeUnit.DAYS übergeben. Java rechnet das intern passend um.

9.4 Periodische Ausführung: Rate vs. Delay

Die wahre Stärke des Schedulers zeigt sich bei wiederkehrenden Aufgaben. Hier gibt es jedoch eine Stolperfalle, die in Prüfungen und im echten Entwickler-Alltag gerne übersehen wird: den Unterschied zwischen Fixed Rate und Fixed Delay.

9.4.1 scheduleAtFixedRate (Feste Taktung)

Bei dieser Methode definieren Sie einen starren Rhythmus (die Rate). Wenn Sie angeben "alle 10 Sekunden", dann startet die Aufgabe bei Sekunde 0, Sekunde 10, Sekunde 20 usw. – unabhängig davon, wie lange die Ausführung der Aufgabe selbst dauert.

Achtung: Was passiert, wenn die Aufgabe 12 Sekunden braucht, der Takt aber 10 Sekunden beträgt? Starten dann zwei Aufgaben parallel? Nein. Das Executor-Framework garantiert, dass dieselbe periodische Aufgabe niemals parallel zu sich selbst läuft. Die nächste Ausführung wird in die Warteschlange gestellt und startet sofort, nachdem die vorherige (zu langsame) Aufgabe fertig ist.

9.4.2 scheduleWithFixedDelay (Feste Pause)

Hier definieren Sie die Pause zwischen dem Ende der einen Ausführung und dem Start der nächsten Ausführung. Wenn Sie eine Pause von 10 Sekunden definieren und die Aufgabe 2 Sekunden dauert, sieht die Timeline so aus: Start bei 0 -> Ende bei 2 -> 10 Sekunden Pause -> Start bei 12 -> Ende bei 14 -> 10 Sekunden Pause -> Start bei 24.

Beispiel: Rate und Delay im Vergleich

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class PeriodischBeispiel {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler = Executors
                .newScheduledThreadPool(2);

        Runnable heartbeatTask = () -> {
            System.out.println("Heartbeat gesendet um: "
```

```
                    + System.currentTimeMillis() / 1000);
            try {
                Thread.sleep(2000); // Aufgabe dauert 2 Sekunden
            } catch (InterruptedException e) {
            }
        };

        // Variante A: Feste Taktung (Rate)
        // Parameter: Aufgabe, Startverzögerung, Periode, Zeiteinheit
        scheduler.scheduleAtFixedRate(heartbeatTask, 0, 5,
                TimeUnit.SECONDS);

        // Die Timeline für Variante A wäre (Startzeitpunkte): 0, 5,
        // 10, 15...
        // Die 2 Sekunden Laufzeit der Aufgabe fallen nicht ins
        // Gewicht.

        // ------------------------------------------------------------
--

        // Variante B: Feste Pause (Delay)
        // Parameter: Aufgabe, Startverzögerung, Pause nach Ende,
        // Zeiteinheit
        // scheduler.scheduleWithFixedDelay(heartbeatTask, 0, 5,
        // TimeUnit.SECONDS);

        // Die Timeline für Variante B wäre (Startzeitpunkte): 0, 7,
        // 14, 21...
        // (Start bei 0 + 2 Sek Dauer + 5 Sek Pause = nächster Start
        // bei 7)

        // Da periodische Aufgaben endlos laufen, lassen wir den
        // Main-Thread nach 20 Sekunden den Pool hart beenden.
        try {
            Thread.sleep(20000);
            System.out.println("Main: Beende Scheduler.");
            scheduler.shutdownNow();
        } catch (InterruptedException e) {
        }
    }
}
```

Faustregel:

- Nutzen Sie FixedRate, wenn absolute Pünktlichkeit entscheidend ist (z.B. ein Metronom in einer Musik-App oder das genaue Abgreifen von Sensordaten jede Sekunde).
- Nutzen Sie FixedDelay, wenn die Systemlast kontrolliert werden soll und die Aufgabe unvorhersehbar lange dauern kann (z.B. das Synchronisieren von Dateien mit einem Cloud-Server – Sie wollen erst 5 Minuten nach dem erfolgreichen Abschluss des letzten Syncs den nächsten starten).

9.5 Eine gefährliche Falle: Unbehandelte Exceptions

Es gibt ein kritisches Detail, das Sie bei periodischen Aufgaben zwingend beachten müssen: Wenn eine geplante Aufgabe eine unbehandelte Exception wirft (z.B. eine NullPointerException), bricht der Executor diese spezielle Aufgabe ab und plant sie NIE WIEDER ein! Der Thread-Pool stürzt zwar nicht ab (im Gegensatz zum alten Timer), aber Ihre periodische Aufgabe stirbt leise im Hintergrund. Es gibt keine Fehlermeldung auf der Konsole, es sei denn, Sie fangen das Future ab und rufen get() auf (was bei endlosen Aufgaben blockieren würde).

Best Practice: Umschließen Sie den gesamten Code in der run()-Methode einer periodischen Aufgabe mit einem schützenden try-catch(Exception e)-Block!

```
Runnable sichereAufgabe = () -> {
    try {
        // Hier passiert die fehleranfällige Geschäftslogik
        int ergebnis = 10 / 0; // Bäm! ArithmeticException
    } catch (Exception e) {
        // Fehler loggen! Die Exception wird hier verschluckt,
        // ABER der Scheduler plant die Aufgabe beim nächsten
        // Takt wieder ein!
        System.err
                .println("Fehler im Task: " + e.getMessage());
    }
};
```

Im folgenden Kapitel wechseln wir nun die Perspektive: Bisher ging es primär um Nebenläufigkeit (Concurrency) und das Verwalten von Aufgaben. Mit dem Fork/Join-Framework betreten wir nun die Welt der echten Parallelität (Parallelism), bei der es darum geht, die Rechenleistung von Multi-Core-Prozessoren für eine einzige, gewaltige Aufgabe maximal auszureizen.

10 Teile und Herrsche – Das ForkJoin-Framework

In den vorangegangenen Kapiteln haben wir Thread-Pools (ExecutorService) genutzt, um viele unabhängige Aufgaben (wie HTTP-Anfragen oder Datenbankabfragen) effizient abzuarbeiten. Solche Aufgaben sind oft "I/O-bound", das heißt, die Threads verbringen viel Zeit mit Warten auf externe Ressourcen.

Was aber, wenn wir eine extrem rechenintensive Aufgabe haben? Stellen Sie sich vor, wir müssen ein Array mit einer Milliarde Zahlen durchsuchen, ein riesiges Bild filtern oder eine komplexe Matrixmultiplikation durchführen. Solche Aufgaben sind "CPU-bound". Ein normaler Thread-Pool hilft uns hier nur bedingt weiter, wenn wir ihm die gesamte Arbeit als einen gewaltigen Task übergeben – dann rechnet nämlich nur ein einziger Kern, während die anderen Däumchen drehen.

Hierfür führte Java 7 das ForkJoin-Framework ein.

10.1 Das Konzept: Divide and Conquer (Teile und Herrsche)

Das Framework basiert auf dem berühmten algorithmischen Prinzip "Teile und Herrsche".

Anstatt eine riesige Aufgabe am Stück zu lösen, wenden wir folgenden rekursiven Algorithmus an:

1. Ist die Aufgabe klein genug, um sie direkt effizient zu berechnen?
    - Ja: Berechne das Ergebnis direkt (sequenziell).
    - Nein: Teile die Aufgabe in zwei (oder mehr) etwa gleich große Teilaufgaben auf (Fork).
2. Übergebe die Teilaufgaben an das Framework zur parallelen Berechnung.
3. Warte auf die Ergebnisse der Teilaufgaben und füge sie zum Gesamtergebnis zusammen (Join).

Durch dieses ständige Aufspalten entsteht ein Baum von immer kleiner werdenden Aufgaben, die sich perfekt auf alle verfügbaren Prozessorkerne verteilen lassen.

10.2 Die Magie unter der Haube: Work-Stealing

Sie könnten sich nun fragen: "Warum nutze ich dafür nicht einfach einen normalen FixedThreadPool?"

Wenn Sie eine Aufgabe in tausende kleine Teilaufgaben zerlegen und diese in die zentrale Warteschlange eines normalen Executors werfen, entsteht ein massiver

Flaschenhals: Alle Threads kämpfen (synchronisiert) um den Zugriff auf diese eine Schlange. Zudem warten übergeordnete Aufgaben auf die Ergebnisse ihrer Teilaufgaben und blockieren so die Threads im Pool.

Der ForkJoinPool löst dies durch einen genialen Work-Stealing-Algorithmus (Arbeitsdiebstahl):

- Jeder Thread im Pool hat seine eigene, lokale Warteschlange (eine sogenannte Deque – Double Ended Queue).
- Wenn ein Thread eine Aufgabe aufspaltet (fork), legt er die neuen Teilaufgaben oben auf seine eigene Schlange. Er arbeitet seine Schlange nach dem LIFO-Prinzip (Last-In, First-Out) ab. Die neuesten, kleinsten Aufgaben werden zuerst erledigt.
- Der Diebstahl: Wenn ein Thread seine eigene Schlange komplett abgearbeitet hat und arbeitslos wird, schaut er sich die Schlangen der anderen Threads an. Er "stiehlt" dann eine Aufgabe vom unteren Ende (FIFO) einer fremden Schlange. Das sind meistens die ältesten und größten Aufgaben, die sich noch weiter aufspalten lassen!

Dieses Design minimiert Konflikte zwischen den Threads extrem und sorgt für eine nahezu perfekte Auslastung aller CPU-Kerne.

10.3 Die Kernklassen des Frameworks

Um das Framework zu nutzen, benötigen wir drei Hauptkomponenten:

1. ForkJoinPool: Der spezialisierte Executor, der das Work-Stealing implementiert.
2. RecursiveAction: Eine Basisklasse für Aufgaben, die kein Ergebnis zurückliefern (z. B. Arrays sortieren, Bilder modifizieren). Ähnlich wie Runnable.
3. RecursiveTask<V>: Eine Basisklasse für Aufgaben, die ein Ergebnis vom Typ V zurückliefern (z. B. Summen bilden, Maxima finden). Ähnlich wie Callable.

10.4 Beispiel: Eine gigantische Summe berechnen

Schauen wir uns an, wie wir die Summe eines sehr großen Arrays von Zahlen berechnen. Wir erben von RecursiveTask<Long>, da wir ein Ergebnis (Long) erwarten.

```
import java.util.concurrent.ForkJoinPool;
```

```java
import java.util.concurrent.RecursiveTask;

// 1. Klasse ableiten von RecursiveTask (da wir einen Rückgabewert wollen)
class ArraySumTask extends RecursiveTask<Long> {
    // Ab wann ist eine Aufgabe "klein genug"? (Schwellenwert)
    private static final int THRESHOLD = 10_000;

    private long[] array;
    private int start;
    private int end;

    public ArraySumTask(long[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    // 2. Die compute()-Methode enthält die eigentliche Logik
    @Override
    protected Long compute() {
        int length = end - start;

        // Basisfall: Ist die Aufgabe klein genug? -> Direkt
        // berechnen!
        if (length <= THRESHOLD) {
            long sum = 0;
            for (int i = start; i < end; i++) {
                sum += array[i];
            }
            return sum;
        }

        // Rekursiver Fall: Aufgabe ist zu groß -> Teilen! (Fork)
        else {
            int middle = start + length / 2;

            // Linke und rechte Teilaufgabe erstellen
            ArraySumTask leftTask = new ArraySumTask(array, start,
                    middle);
            ArraySumTask rightTask = new ArraySumTask(array, middle,
                    end);

            // Beide Teilaufgaben zur parallelen Ausführung an den
            // Pool übergeben
            // invokeAll() ist eine praktische Hilfsmethode, die
            // implizit fork() aufruft und auf beide Ergebnisse
            // wartet.
            invokeAll(leftTask, rightTask);

            // Ergebnisse einsammeln (Join) und addieren
            long leftResult = leftTask.join();
            long rightResult = rightTask.join();

            return leftResult + rightResult;
        }
    }
}

public class ForkJoinBeispiel {
    public static void main(String[] args) {
        // Riesiges Array vorbereiten (z. B. 50 Millionen Zahlen)
        long[] numbers = new long[50_000_000];
```

```
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = i; // Einfachheitshalber mit Index füllen
        }

        System.out.println("Main: Erstelle ForkJoinPool...");
        // Nutzt standardmäßig so viele Threads, wie die CPU logische
        // Kerne hat
        ForkJoinPool pool = new ForkJoinPool();

        // Die Hauptaufgabe für das gesamte Array erstellen
        ArraySumTask mainTask = new ArraySumTask(numbers, 0,
                numbers.length);

        System.out.println("Main: Starte Berechnung...");
        long startTime = System.currentTimeMillis();

        // Aufgabe an den Pool übergeben und auf Ergebnis warten
        long totalSum = pool.invoke(mainTask);

        long endTime = System.currentTimeMillis();

        System.out.println("Gesamtsumme: " + totalSum);
        System.out.println("Dauer: " + (endTime - startTime) + " ms");

        pool.shutdown();
    }
}
```

10.5 Die Anatomie der compute()-Methode

Der Code innerhalb von compute() folgt immer exakt demselben Muster:

1. Prüfen: Ist die Arbeitsschicht kleiner als der THRESHOLD?
2. Sequenziell: Wenn ja, rechne es einfach direkt mit einer normalen for-Schleife aus. (Der Overhead, für 5 Zahlen extra Objekte zu erzeugen und Threads zu bemühen, wäre gigantisch!).
3. Aufteilen: Wenn nein, halbiere das Problem. Erzeuge leftTask und rightTask.
4. Fork & Join: Nutze invokeAll(left, right), um beide Teilaufgaben ins System zu werfen, und summiere die Ergebnisse mit join().[1]

Hinweis für Performance-Liebhaber: Der Schwellenwert (THRESHOLD) ist extrem wichtig. Ist er zu klein, erzeugen Sie Millionen von Task-Objekten und die Garbage Collection der JVM bricht zusammen. Ist er zu groß, wird die Aufgabe nicht oft genug

[1] invokeAll(leftTask, rightTask) ruft nicht für beide Aufgaben asynchron fork() auf. Stattdessen wirft es die erste Aufgabe in den Pool (fork()) und berechnet die zweite Aufgabe direkt im aktuellen Thread (compute()). Genau dieser Kniff (den aktuellen Thread sofort weiterarbeiten zu lassen, statt ihn auf zwei ausgelagerte Tasks warten zu lassen) macht das Fork/Join-Framework erst so unglaublich schnell und speichereffizient.

geteilt, um alle Kerne auszulasten. Meist liegt der Sweet-Spot bei Aufgaben, die wenige tausend Operationen umfassen.

10.6 Wann Sie Fork/Join NICHT nutzen sollten

Das Fork/Join-Framework ist ein Skalpell für sehr spezifische Probleme, kein Schweizer Taschenmesser für alles.

- Kein blockierendes I/O: Wenn Ihre Aufgaben Dateien lesen, auf Netzwerkanfragen warten oder Thread.sleep() aufrufen, ist Fork/Join die falsche Wahl. Die Threads im Pool würden blockieren, das Work-Stealing käme zum Erliegen und die Performance würde drastisch einbrechen. Für I/O-Aufgaben nutzen Sie klassische Thread-Pools oder (wie wir in Kapitel 13 sehen werden) Virtuelle Threads.
- Objekt-Overhead: Da jeder Knoten im Berechnungsbaum ein eigenes Objekt (ArraySumTask) erfordert, beansprucht das Framework viel Heap-Speicher.

Im folgenden Kapitel verknüpfen wir nun das komplexe Wissen über den ForkJoinPool aus dem vorherigen Kapitel mit der Eleganz der modernen, funktionalen Java-Programmierung.

11 Deklarative Datenparallelität – Parallele Streams

Im letzten Kapitel haben wir gesehen, wie mächtig das Fork/Join-Framework ist, wenn es darum geht, rechenintensive Aufgaben auf alle CPU-Kerne zu verteilen. Wir haben aber auch gesehen: Das Schreiben von eigenen RecursiveTask-Klassen erfordert viel Boilerplate-Code (Standardcode). Wir mussten die Datenstruktur manuell aufteilen, den Schwellenwert (THRESHOLD) festlegen und die Teilergebnisse händisch wieder zusammenfügen.

In der modernen Softwareentwicklung lautet das Ziel jedoch oft: Wir wollen uns auf die Fachlogik konzentrieren und die Infrastruktur dem Framework überlassen. Genau hier kommen Parallele Streams (eingeführt in Java 8) ins Spiel. Sie sind die komfortable, hochabstrahierte "Benutzeroberfläche" für den ForkJoinPool.

11.1 Der Paradigmenwechsel: Deklarativ statt Imperativ

Bevor wir Streams parallelisieren, müssen wir uns den Unterschied in der Denkweise klären:

- Imperative Programmierung (Das "Wie"): Bei einer klassischen for-Schleife beschreiben Sie dem Computer Schritt für Schritt, wie er über eine Liste iterieren soll (Index hochzählen, prüfen, ob das Ende erreicht ist).
- Deklarative Programmierung (Das "Was"): Mit der Stream-API beschreiben Sie nur noch, was das Endresultat sein soll (z. B. "Filtere alle ungeraden Zahlen heraus und summiere sie").

Diese deklarative Herangehensweise ist der Schlüssel zur Parallelisierung: Da Sie der Java Virtual Machine (JVM) nicht mehr exakt vorschreiben, in welcher Reihenfolge die Schleife durchlaufen werden muss, hat die JVM plötzlich die Freiheit, die Arbeit im Hintergrund aufzuteilen!

11.2 Wie aus sequenziell parallel wird

Die Erstellung eines parallelen Streams ist in Java verblüffend einfach. Es reicht der Austausch eines einzigen Methodenaufrufs.

Anstatt .stream() rufen Sie auf einer Collection einfach .parallelStream() auf. Alternativ können Sie einen bestehenden Stream mit der Methode .parallel() in einen parallelen Zustand versetzen.

```
List<String> worte = Arrays.asList("Apfel", "Banane", "Birne",
        "Erdbeere");

// Sequenziell: Ein einziger Thread verarbeitet Element für
```

```
// Element
worte.stream().map(String::toUpperCase)
        .forEach(System.out::println);

// Parallel: Mehrere Threads verarbeiten die Elemente
// gleichzeitig
worte.parallelStream().map(String::toUpperCase)
        .forEach(System.out::println);
```

Was passiert unter der Haube?

Sobald Sie .parallelStream() aufrufen, aktiviert Java automatisch drei Mechanismen:

1. Spliterator (Aufteiler): Die Datenquelle wird in kleine Chunks (Blöcke) zerlegt. Arrays und ArrayLists lassen sich hervorragend und billig exakt in der Mitte teilen.
2. Der unsichtbare ForkJoinPool: Die zerlegten Chunks werden als Aufgaben an den sogenannten ForkJoinPool.commonPool() übergeben. Dies ist ein globaler Thread-Pool, der in jeder JVM standardmäßig existiert und genau so viele Threads bereithält, wie Ihr Computer CPU-Kerne besitzt.[2]
3. Merging (Zusammenführen): Am Ende der Stream-Pipeline fügt Java die Teilergebnisse der einzelnen Threads automatisch wieder zusammen (z. B. bei einer .collect(Collectors.toList())-Operation).

11.3 Beispiel: Massendatenverarbeitung

Schauen wir uns ein Beispiel an, bei dem der Einsatz wirklich Sinn ergibt. Wir wollen in einer großen Menge von Zahlen überprüfen, wie viele davon Primzahlen sind. Das ist eine typische CPU-bound (rechenintensive) Aufgabe.

```
import java.util.stream.LongStream;

public class ParallelStreamBeispiel {

    // Eine rechenintensive Methode zur Primzahlprüfung
    public static boolean isPrime(long n) {
        if (n <= 1)
            return false;
        for (long i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0)
                return false;
        }
        return true;
```

[2] Der commonPool hält standardmäßig Anzahl der Kerne minus 1 (also Runtime.getRuntime().availableProcessors() - 1) Threads bereit. Das ist ein extrem wichtiges Design-Detail in Java: Die JVM rechnet den aufrufenden Thread (z. B. den Main-Thread, der .parallelStream() aufgerufen hat) mit ein! Der aufrufende Thread hilft selbst aktiv bei der Abarbeitung der Chunks mit. Zusammen mit den Threads aus dem Pool ergibt das dann die Anzahl der Kerne.

```
    }

    public static void main(String[] args) {
        long limit = 5_000_000L;

        System.out.println("Starte sequenzielle Berechnung...");
        long startSeq = System.currentTimeMillis();
        long countSeq = LongStream.rangeClosed(2, limit)
                .filter(ParallelStreamBeispiel::isPrime).count();
        long endSeq = System.currentTimeMillis();
        System.out.println("Gefunden: " + countSeq + " in "
                + (endSeq - startSeq) + " ms");

        System.out.println("\nStarte parallele Berechnung...");
        long startPar = System.currentTimeMillis();
        long countPar = LongStream.rangeClosed(2, limit).parallel()
                // Das magische Wort!
                .filter(ParallelStreamBeispiel::isPrime).count();
        long endPar = System.currentTimeMillis();
        System.out.println("Gefunden: " + countPar + " in "
                + (endPar - startPar) + " ms");
    }
}
```

Wenn Sie diesen Code auf einem modernen Multi-Core-Prozessor ausführen, werden Sie feststellen, dass die parallele Ausführung oft um den Faktor 3 bis 4 schneller ist (je nach Kern-Anzahl). Und das bei exakt einem Wort Code-Änderung!

11.4 Wann lohnt sich ein paralleler Stream? (Die Aufwand-Nutzen-Frage)

Wenn es so verlockend einfach ist, warum schreiben wir dann nicht einfach hinter jede Liste in unserem Programm ein .parallelStream()?

Die Antwort liegt im sogenannten Management-Overhead (Verwaltungsaufwand). Das Aufteilen von Daten in mehrere Häppchen, das Zuweisen dieser Häppchen an die verschiedenen Threads im Hintergrund und das abschließende, korrekte Zusammenkleben der Teilergebnisse kosten das System Zeit und Kraft.

Stellen Sie sich vor, Sie sind der Chefkoch und haben eine Aufgabe vor sich. Sie überlegen nun, ob Sie diese schnell selbst erledigen oder ob Sie Ihre Hilfsköche aus der Pause holen, ihnen die Aufgabe erklären, die Zutaten aufteilen und am Ende alles wieder auf einem großen Teller anrichten.

Ob sich dieser Organisationsaufwand lohnt, hängt immer von der Balance zwischen zwei Fragen ab: Wie viele Dinge müssen erledigt werden, und wie aufwendig ist jedes einzelne Ding?

Szenario 1: Sehr leichte Aufgaben (z. B. einfache Zahlen addieren oder Text in Großbuchstaben umwandeln)

Wenn Sie nur drei Karotten in Scheiben schneiden müssen, sind Sie alleine längst fertig, bevor Ihre Hilfsköche überhaupt ihre Schürzen umgebunden haben. Der Organisationsaufwand dauert hier viel länger als das eigentliche Schneiden. Ein paralleler Stream würde Ihr Programm in diesem Fall sogar ausbremsen. Damit sich das Aufteilen bei derart simplen Aufgaben überhaupt lohnt, muss der Berg an Arbeit gigantisch sein – wir sprechen hier von Millionen von Elementen (z. B. ein ganzer LKW voller Karotten).

Szenario 2: Sehr schwere Aufgaben (z. B. komplexe Bildanalysen, aufwendige Verschlüsselung oder langsame Datenbankabfragen)

Wenn die Aufgabe hingegen lautet: "Backe eine dreistöckige Hochzeitstorte mit Zuckerrosen", sieht die Welt ganz anders aus. Selbst wenn auf Ihrer Bestellliste nur kümmerliche zehn Torten stehen, lohnt es sich sofort, zehn Köche zu rufen. Das Delegieren und Erklären dauert vielleicht zwei Minuten, aber danach ist jeder Koch stundenlang beschäftigt. Die Zeit für die Organisation fällt hier absolut nicht mehr ins Gewicht.

Die einfache Faustregel lautet also: Ein paralleler Stream ist kein Wundermittel, das Code automatisch schneller macht. Er lohnt sich nur dann, wenn Sie entweder eine gewaltige Menge an Daten haben, oder wenn die Verarbeitung jedes einzelnen Elements so enorm zeitaufwendig ist, dass die anfängliche Organisation der Threads im Vergleich dazu unwichtig wird. Sind die Aufgaben klein und die Liste kurz, bleiben Sie beim normalen .stream().

11.5 Gefahren und Anti-Patterns

Neben dem Performance-Overhead gibt es zwei massive Stolperfallen bei parallelen Streams, die wir als Software-Ingenieure unbedingt vermeiden müssen.

11.5.1 Gefahr 1: Seiteneffekte (Side Effects)

Die Stream-API ist für funktionale, zustandslose Operationen gedacht. Wenn Sie innerhalb eines parallelen Streams auf Variablen zugreifen, die außerhalb des Streams liegen und diese verändern, provozieren Sie unweigerlich Datenkorruption (Race Conditions).

Schlechtes Beispiel (Anti-Pattern):

```
List<Integer> ergebnisse = new ArrayList<>();
IntStream.range(0, 1000).parallel().forEach(i -> {
```

```
            ergebnisse.add(i); // FATAL! ArrayList ist nicht
                               // thread-sicher!
        });
```

Lösung: Nutzen Sie immer die integrierten Reduktions-Methoden wie .collect(Collectors.toList()). Java kümmert sich dann intern sicher um das parallele Zusammenfügen.

11.5.2 Gefahr 2: Blockierende Operationen im commonPool

Wie bereits erwähnt, nutzen alle parallelen Streams in Ihrer gesamten JVM standardmäßig denselben ForkJoinPool.commonPool().

Wenn Sie nun in einem parallelen Stream Datenbankabfragen oder HTTP-Requests machen (I/O-Operationen), blockieren die Threads, während sie auf eine Antwort warten. Wenn Sie Pech haben, blockieren Sie damit den gesamten commonPool. Kein anderer paralleler Stream in Ihrer Anwendung kann dann noch arbeiten!

Merkregel: Parallele Streams sind für CPU-intensive Berechnungen im Arbeitsspeicher gedacht, nicht für I/O-Aufgaben.

Mit diesem Thema verlassen wir die Ebene der reinen Datenverarbeitung und betreten die moderne Architektur von asynchronen, ereignisgesteuerten Anwendungen.

12 Asynchrone und Reaktive Pipelines – CompletableFuture

In Kapitel 7 haben wir das Future-Interface kennengelernt. Es gab uns einen "Abholschein" für das Ergebnis einer Aufgabe, die in einem anderen Thread berechnet wurde. Wir haben jedoch ein massives Architekturproblem dieses klassischen Ansatzes verschwiegen, dem wir uns jetzt stellen müssen.

12.1 Das Dilemma des blockierenden Wartens

Stellen Sie sich vor, Sie programmieren einen Webshop. Wenn ein Kunde eine Bestellung aufgibt, müssen im Hintergrund drei Dinge passieren:

1. Die Kundendaten müssen aus der Datenbank geladen werden.
2. Basierend auf diesen Daten muss die Kreditkarte bei einem externen Zahlungsdienstleister belastet werden.
3. Danach muss eine Bestätigungs-E-Mail versendet werden.

Mit einem klassischen ExecutorService und Callable würden Sie das Laden der Daten starten und ein Future<Kundendaten> erhalten. Doch was dann? Um die Zahlung anzustoßen, brauchen Sie die Kundendaten. Sie müssen also future.get() aufrufen.

Und genau hier liegt das Problem: Die Methode get() blockiert! Der aufrufende Thread friert komplett ein und tut absolut nichts, während er auf die Datenbank wartet. In hochskalierenden Systemen ist ein blockierter Thread ein verschwendeter Thread.

12.2 Die Lösung: Reagieren statt Blockieren

Wir wollen nicht auf das Ergebnis warten. Wir wollen stattdessen definieren: "Wenn das Ergebnis irgendwann da ist, dann reiche es automatisch an die nächste Aufgabe weiter."

Dieses Paradigma nennt man reaktive Programmierung. Um dies in Java umzusetzen, wurde in Java 8 die Klasse CompletableFuture<V> eingeführt. Sie implementiert nicht nur das bekannte Future-Interface, sondern auch das Interface CompletionStage. Letzteres ist das eigentliche Herzstück: Es erlaubt uns, Pipelines aus asynchronen Aufgaben zusammenzubauen, bei denen ein Schritt nahtlos in den nächsten übergeht.

12.3 Der Startpunkt: supplyAsync und runAsync

Im Gegensatz zu klassischen Futures brauchen wir für CompletableFuture nicht zwingend einen eigenen ExecutorService zu erstellen (obwohl wir einen übergeben

können). Die Klasse bringt eigene Factory-Methoden mit, die Aufgaben standardmäßig im ForkJoinPool.commonPool() starten.

- runAsync(Runnable): Startet eine Aufgabe ohne Rückgabewert.
- supplyAsync(Supplier<V>): Startet eine Aufgabe, die einen Wert vom Typ V liefert.

```
import java.util.concurrent.CompletableFuture;

public class AsyncStartBeispiel {
    public static void main(String[] args) {
        System.out.println(
                "Main: Starte asynchrone Datenbankabfrage...");

        CompletableFuture<String> kundenDatenFuture = CompletableFuture
                .supplyAsync(() -> {
                    System.out.println("Worker: Lade Daten (Thread: "
                            + Thread.currentThread().getName() + ")");
                    simuliereWartezeit(2000); // Wartet 2 Sekunden
                    return "Kunde: Max Mustermann, ID: 4711";
                });

        System.out.println(
             "Abfrage läuft im Hintergrund. Main-Thread nicht blockiert!");

        // Verhindert, dass das Programm endet, bevor der Worker
        // fertig ist
        kundenDatenFuture.join();
    }

    private static void simuliereWartezeit(int ms) {
        try {
            Thread.sleep(ms);
        } catch (InterruptedException e) {
        }
    }
}
```

12.4 Pipelines bauen: Transformation und Konsum

Der wahre Zauber beginnt, wenn wir Methoden an unser CompletableFuture anhängen. Jede dieser Methoden liefert wiederum ein neues CompletableFuture zurück, wodurch eine Kette (Pipeline) entsteht.

Die wichtigsten Methoden zur Verkettung sind:

- thenApply(Function): Nimmt das Ergebnis des vorherigen Schritts, transformiert es und gibt ein neues Ergebnis weiter. (Vergleichbar mit map() bei Streams).

- thenAccept(Consumer): Nimmt das Ergebnis des vorherigen Schritts und "konsumiert" es (z. B. Ausgabe auf der Konsole). Es gibt keinen Wert an den nächsten Schritt weiter (Rückgabetyp Void).
- thenRun(Runnable): Führt einfach eine Aktion aus, wenn der vorherige Schritt fertig ist, ohne sich für dessen Ergebnis zu interessieren.

Lassen Sie uns das Webshop-Beispiel als Pipeline aufbauen:

```
CompletableFuture.supplyAsync(() -> {
    // Schritt 1: Lade ID (dauert 1 Sekunde)
    simuliereWartezeit(1000);
    return 4711;
}).thenApply(kundenId -> {
    // Schritt 2: Hole Namen zur ID
    System.out.println("Verarbeite ID: " + kundenId);
    return "Max Mustermann";
}).thenApply(name -> {
    // Schritt 3: Erstelle E-Mail-Text
    return "Hallo " + name + ", deine Bestellung ist da!";
}).thenAccept(emailText -> {
    // Schritt 4: Konsumiere das Ergebnis (E-Mail senden)
    System.out.println("Sende E-Mail: " + emailText);
});

System.out.println(
        "Pipeline wurde definiert. Alles läuft asynchron ab!");
```

Wichtig: Der Main-Thread rauscht innerhalb von Millisekunden durch diesen Code. Er definiert nur den Bauplan der Pipeline. Die tatsächliche Ausführung und Übergabe der Daten von Schritt zu Schritt übernimmt das Framework im Hintergrund.

12.5 Aufgaben kombinieren

Oft hängen Aufgaben nicht streng sequenziell voneinander ab, sondern wir müssen auf mehrere unabhängige asynchrone Ergebnisse warten, bevor wir weitermachen können.

Dafür gibt es thenCombine(). Es startet nicht beide Aufgaben, sondern verbindet zwei bereits laufende Futures, sobald beide abgeschlossen sind.

```
CompletableFuture<Double> preisBerechnung = CompletableFuture
        .supplyAsync(() -> {
            simuliereWartezeit(2000);
            return 89.99; // Preis in Euro
        });

CompletableFuture<Double> wechselkursAbfrage = CompletableFuture
        .supplyAsync(() -> {
            simuliereWartezeit(1500);
            return 1.08; // Kurs EUR zu USD
        });
```

```
// Kombiniere beide Futures
CompletableFuture<Double> preisInUsd = preisBerechnung
        .thenCombine(wechselkursAbfrage,
                (preis, kurs) -> preis * kurs
        // Diese BiFunction wird ausgeführt, wenn BEIDE fertig
        // sind
        );

preisInUsd.thenAccept(
        usd -> System.out.println("Preis in USD: " + usd));
```

In diesem Szenario laufen Preisberechnung und Kursabfrage parallel. Die Gesamtdauer ist nicht 2000 + 1500 = 3500 Millisekunden, sondern nur 2000 Millisekunden (die Dauer der längsten Aufgabe).

12.6 Fehlerbehandlung elegant gelöst

Erinnern Sie sich an den grausamen try-catch-Block rund um future.get() aus Kapitel 7? In einer asynchronen Pipeline funktioniert das nicht mehr, da der Fehler irgendwo im Hintergrund in einem anderen Thread auftritt.

CompletableFuture bietet Methoden, um Fehler direkt in die Pipeline einzubauen, ähnlich einem catch-Block. Die wichtigste Methode ist exceptionally().

```
CompletableFuture.supplyAsync(() -> {
    if (Math.random() > 0.5) {
        throw new RuntimeException(
                "Datenbank nicht erreichbar!");
    }
    return "Daten erfolgreich geladen.";
}).exceptionally(ex -> {
    // Wird NUR aufgerufen, wenn im vorherigen Schritt eine
    // Exception flog
    System.out.println(
            "Fehler aufgetreten: " + ex.getMessage());
    return "Standard-Ersatzdaten"; // Fallback-Wert
                                   // bereitstellen
}).thenAccept(daten -> {
    // Wird immer ausgeführt, entweder mit echten Daten oder
    // Ersatzdaten
    System.out.println("Verarbeite: " + daten);
});
```

Wenn die Datenbankverbindung fehlschlägt, fängt exceptionally() den Fehler ab, loggt ihn, liefert einen Standardwert ("Fallback") und die Pipeline kann normal weiterlaufen. Ein enorm robustes Design!

Im nächsten Kapitel folgt der krönende Abschluss unseres ersten großen Themenblocks. Es führt an den absoluten "State of the Art" der Java-Entwicklung

heran und zeigt, wie ein jahrelanges Architekturproblem durch ein geniales Update der Java Virtual Machine (JVM) gelöst wurde.

13 Der Paradigmenwechsel – Virtuelle Threads (Project Loom)

In unserer bisherigen Reise durch die nebenläufige Programmierung haben wir eine klare Evolution durchgemacht: Wir starteten mit direkten Betriebssystem-Threads (teuer und ressourcenhungrig). Um diese Ressourcen zu schonen, führten wir Thread-Pools ein (komplexer zu verwalten). Um wiederum Threads in Pools nicht durch Warten (blockierendes I/O) zu verschwenden, wichen wir auf asynchrone Pipelines wie CompletableFuture aus.

Asynchrone Programmierung ist extrem effizient, hat aber einen hohen Preis: Der Code ist schwerer zu lesen, extrem schwer zu debuggen (da der Stacktrace oft wertlos ist, wenn der Fehler in einem Hintergrund-Pool auftritt) und die traditionellen Kontrollstrukturen (if, for, try-catch) funktionieren über Thread-Grenzen hinweg nicht mehr intuitiv.

Mit Java 21 (ausgereift aus dem "Project Loom") stellte sich Java die Frage: Können wir den einfachen, blockierenden, sequenziellen Code von früher schreiben, aber die gigantische Skalierbarkeit von asynchronem Code erhalten? Die Antwort lautet: Ja, mit Virtuellen Threads.

13.1 Die Rückkehr zum M:N-Mapping

Erinnern wir uns an Kapitel 2: Ein klassischer Java-Thread (heute Plattform-Thread genannt) ist 1:1 an einen Betriebssystem-Thread (OS-Thread) gebunden. Da OS-Threads typischerweise 1 bis 2 MB Speicher für ihren Stack (Aufrufspeicher) reservieren, ist bei ein paar tausend Threads auf einem normalen Server Schluss (OutOfMemoryError).

Virtuelle Threads hingegen werden nicht vom Betriebssystem verwaltet, sondern exklusiv von der JVM. Sie sind leichtgewichtig (ihr Speicherbedarf wächst dynamisch und startet bei wenigen Bytes). Das bedeutet, Sie können problemlos Millionen von Virtuellen Threads gleichzeitig erzeugen!

Die JVM nutzt im Hintergrund weiterhin OS-Threads, aber deutlich weniger. Sie wendet ein M:N-Mapping an: Millionen von Virtuellen Threads (M) werden auf einen sehr kleinen Pool von OS-Threads (N) abgebildet. Diese zugrunde liegenden OS-Threads nennt man Carrier Threads (Träger-Threads).

13.2 Die Magie unter der Haube: Mounten und Unmounten

Wie schafft es die JVM, dass 1.000.000 Virtuelle Threads auf z. B. nur 8 Carrier Threads laufen? Das Geheimnis liegt im geschickten Umgang mit Blockaden (I/O-Operationen, Thread.sleep(), Datenbankabfragen).

Wenn ein klassischer Plattform-Thread eine Datenbankabfrage macht und auf die Antwort warten muss, schläft der zugrunde liegende OS-Thread ein. Der Kern der CPU bleibt im schlimmsten Fall ungenutzt.

Wenn ein Virtueller Thread auf eine Datenbankabfrage wartet, passiert Folgendes:

1. Unmount (Absteigen): Die JVM erkennt, dass der Virtuelle Thread blockieren wird. Sie nimmt den Virtuellen Thread sofort vom Carrier Thread herunter. Der aktuelle Zustand (die lokalen Variablen auf dem Stack) des Virtuellen Threads wird einfach im regulären Java-Heap-Speicher abgelegt.
2. Freigabe: Der Carrier Thread (der echte OS-Thread) ist nun frei! Er nimmt sich einfach sofort den nächsten wartenden Virtuellen Thread und führt diesen aus.
3. Mount (Aufsteigen): Sobald die Antwort aus der Datenbank da ist, weckt die JVM den Virtuellen Thread wieder auf, holt seinen Zustand aus dem Heap, setzt ihn auf irgendeinen gerade freien Carrier Thread und lässt ihn exakt dort weiterlaufen, wo er aufgehört hat.

Für Sie als Programmierer sieht es so aus, als würde der Thread blockieren. Unter der Haube blockiert die JVM jedoch niemals den kostspieligen OS-Thread.

13.3 Virtuelle Threads erstellen

Die Erstellung ist extrem einfach und integriert sich nahtlos in die bestehende Thread-API.

13.3.1 Variante A: Direkt über den Builder

Anstatt new Thread(...) verwenden wir nun die neuen Factory-Methoden der Klasse Thread:

```
public class VirtualThreadBeispiel {
    public static void main(String[] args)
            throws InterruptedException {
        System.out.println("Main-Thread startet...");

        // Startet einen einzelnen Virtuellen Thread
        Thread vThread = Thread.ofVirtual()
                .name("Mein-Virtueller-Thread").start(() -> {
                    System.out.println(
                            "Ich laufe in einem Virtuellen Thread!");
                    try {
```

```
                        // Dies blockiert KEINEN OS-Thread!
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                    }
                    System.out.println("Bin wieder aufgewacht!");
                });

        vThread.join(); // Auf das Ende warten funktioniert wie
                        // gewohnt
    }
}
```

13.3.2 Variante B: Der Executor ohne Pool

In Kapitel 8 haben wir gelernt, Threads in Pools wiederzuverwenden. Bei Virtuellen Threads ist Pooling ein Anti-Pattern! Sie sind so billig zu erzeugen und zu zerstören, dass wir für jede noch so kleine Aufgabe einfach einen komplett neuen Virtuellen Thread erstellen.

Dafür gibt es einen neuen ExecutorService:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.stream.IntStream;

public class VirtualExecutorBeispiel {
    public static void main(String[] args) {

        // Dieser Executor verwendet KEINEN Pool! Er erzeugt für jedes
        // submit() einen brandneuen Virtuellen Thread.
        try (ExecutorService executor = Executors
                .newVirtualThreadPerTaskExecutor()) {

            // Wir feuern problemlos 100.000 Aufgaben ab
            IntStream.range(0, 100_000).forEach(i -> {
                executor.submit(() -> {
                    try {
                        Thread.sleep(1000); // Simuliert I/O (z.B.
                                            // API-Aufruf)
                        // Beachten Sie: Alles ist in ca. 1 Sekunde
                        // fertig!
                    } catch (InterruptedException e) {
                    }
                });
            });

        } // Durch den try-with-resources Block wird executor.close()
          // implizit gerufen. Das wartet (wie awaitTermination) auf
          // alle Virtuellen Threads.

        System.out.println("Alle 100.000 Aufgaben erledigt.");
    }
}
```

Probieren Sie das einmal mit einem Plattform-Thread-Pool aus – Ihr Rechner wird in die Knie gehen. Hier läuft es butterweich durch!

13.4 Der richtige Einsatzzweck (I/O vs. CPU)

Virtuelle Threads sind kein Wundermittel, das Ihre Berechnungen plötzlich schneller macht. Es ist essenziell, dass Sie den Unterschied zum ForkJoin-Framework (Kapitel 10) verstehen.

- CPU-bound (Rechenintensiv): Wenn Sie eine Matrix berechnen oder Kryptografie betreiben, rechnet die CPU ununterbrochen. Der Virtuelle Thread kann nicht "unmounten", da er nie auf etwas warten muss. Er belegt den Carrier Thread dauerhaft. 1.000.000 Virtuelle Threads, die gleichzeitig rechnen wollen, würden das System genauso zum Stillstand bringen, als hätten Sie nur einen Kern. Für CPU-Aufgaben nutzen Sie weiterhin parallele Streams oder den ForkJoinPool!
- I/O-bound (Warte-intensiv): Web-Requests, Datei-Operationen, Datenbankabfragen. Hier warten Threads die meiste Zeit. Genau hier glänzen Virtuelle Threads.

Teil II:

Wenn Threads aufeinandertreffen

14 Die neue Denkweise – Interleaving, Koordination und geteilter Zustand

Herzlichen Glückwunsch! Sie haben den ersten großen Meilenstein dieses Buches erreicht. Sie wissen nun, wie Sie Aufgaben in Java nebenläufig oder parallel ausführen lassen können – sei es über klassische Threads, Thread-Pools, asynchrone Pipelines oder Virtuelle Threads.

Bisher glich unsere Programmierung jedoch einer gut organisierten Laufbahn, auf der jeder Athlet (Thread) stur in seiner eigenen, abgetrennten Spur rannte. Die Aufgaben waren völlig unabhängig voneinander. Doch in echten, komplexen Softwaresystemen ist das selten der Fall. Hier arbeiten Threads oft an denselben Daten, müssen Zwischenergebnisse austauschen oder aufeinander warten. Die Laufbahn wird zu einem chaotischen Marktplatz.

Dieses Kapitel gibt Ihnen einen umfassenden Überblick über die Konzepte, Notwendigkeiten und vor allem die Gefahren, die entstehen, wenn Threads beginnen, miteinander zu interagieren.

14.1 Der Abschied vom sequenziellen Denken

Der wohl größte Stolperstein für angehende Software-Ingenieure im Bereich der Nebenläufigkeit ist unsere eigene Intuition. Wir sind darauf trainiert, sequenziell zu denken: Zeile 1 wird ausgeführt, dann Zeile 2, dann Zeile 3. Wenn wir ein Programm zweimal mit denselben Eingaben starten, erwarten wir exakt denselben Ablauf und dasselbe Ergebnis (Determinismus).

In der nebenläufigen Programmierung verlieren wir diesen Determinismus.

Das Betriebssystem (bzw. die Java Virtual Machine) besitzt einen Scheduler (Planer). Dieser entscheidet völlig autonom, welcher Thread wann auf einem CPU-Kern rechnen darf, wann er pausiert wird und welcher Thread als Nächstes an der Reihe ist. Als Programmierer haben Sie darauf (fast) keinen Einfluss. Ihr Code könnte bei 1.000 Testläufen perfekt funktionieren und beim 1.001. Lauf beim Kunden plötzlich falsche Daten produzieren oder komplett einfrieren.

Um diese Unvorhersehbarkeit zu beherrschen, müssen wir ein völlig neues Konzept verstehen: das Interleaving.

14.2 Die Wurzel des Chaos: Interleaving (Verzahnung)

Der englische Begriff Interleaving bedeutet wörtlich übersetzt "Verzahnung" oder "Verschachtelung". Er beschreibt die Tatsache, dass die Befehle verschiedener Threads in einer völlig unvorhersehbaren Reihenfolge zeitlich ineinandergreifen.

Um das zu verstehen, müssen wir uns bewusst machen, dass eine simple Code-Zeile in Java für die CPU oft aus mehreren Einzelschritten besteht. Nehmen wir das klassische Beispiel einer Kontostandsänderung:

```
kontoStand = kontoStand + 10;
```

Für die CPU (und auf Ebene des Maschinen- oder Bytecodes) sind das drei völlig getrennte Operationen:

1. Lesen (Read): Lade den aktuellen Wert von kontoStand aus dem Hauptspeicher in ein lokales Register der CPU (z. B. den Wert 100).
2. Rechnen (Modify): Addiere 10 zu diesem Wert im Register (Ergebnis: 110).
3. Schreiben (Write): Schreibe den neuen Wert (110) aus dem Register zurück in den Hauptspeicher in die Variable kontoStand.

Was passiert nun, wenn Thread A und Thread B exakt im selben Moment versuchen, 10 Euro auf dasselbe Konto einzuzahlen? Bei einem glücklichen Ablauf (sequenziell) macht Thread A seine 3 Schritte (Kontostand ist 110), dann macht Thread B seine 3 Schritte (Kontostand ist 120). Alles ist gut.

Beim Interleaving könnte der Scheduler den Ablauf jedoch wie folgt verzahnen:

- Thread A liest den Wert 100.
- Der Scheduler pausiert Thread A und wechselt zu Thread B.
- Thread B liest den Wert (der im Speicher immer noch 100 ist!).
- Thread B addiert 10 (Ergebnis 110) und schreibt 110 in den Speicher.
- Der Scheduler wechselt zurück zu Thread A.
- Thread A macht da weiter, wo er aufgehört hat. Er hat noch seine "100" im lokalen Gedächtnis, addiert 10 und schreibt sein Ergebnis (110) in den Speicher. Er überschreibt damit einfach die Arbeit von Thread B!

Am Ende sind 110 Euro auf dem Konto, obwohl 120 Euro dort sein müssten. 10 Euro sind spurlos verschwunden. Dieses Phänomen nennt man ein Lost Update (Verlorene Aktualisierung), und es ist die direkte Folge von unkontrolliertem Interleaving.

14.3 Die drei Säulen der Thread-Interaktion

Um das Chaos des Interleavings zu bändigen und Threads sinnvoll zusammenarbeiten zu lassen, stützen wir uns auf drei zentrale Konzepte:

1. Synchronisation (Schutz)

Hierbei geht es um Sicherheit und Ausschluss. Wenn mehrere Threads auf denselben Speicherbereich (Shared State) zugreifen und mindestens einer davon die Daten verändert, müssen wir diesen Code-Bereich schützen.

- Ziel: Wir stellen sicher, dass bestimmte Code-Blöcke immer "atomar" (als unteilbare Einheit) ausgeführt werden. Solange ein Thread in diesem kritischen Bereich ist, müssen alle anderen Threads draußen warten.

2. Kommunikation (Datenaustausch)

Threads laufen oft auf unterschiedlichen CPU-Kernen. Jeder Kern hat seinen eigenen, rasend schnellen Zwischenspeicher (Hardware-Cache).

- Ziel: Bei der Kommunikation geht es um Sichtbarkeit (Visibility). Wir müssen der JVM mitteilen: "Wenn Thread A eine Variable ändert, stelle sicher, dass diese Änderung sofort aus dem lokalen CPU-Cache in den globalen Hauptspeicher geschrieben wird, damit Thread B den neuen Wert auch wirklich sehen kann!"

3. Koordination (Ablaufsteuerung)

Hier geht es nicht primär um den Schutz von Daten, sondern um das Timing und die Choreografie von Threads.

- Ziel: Mechanismen bereitzustellen, mit denen ein Thread effizient "schlafen" geschickt wird, bis ein Ereignis eintritt (z. B. "Warte, bis Daten in der Liste sind"), und wie andere Threads ihn aufwecken können ("Ich habe neue Daten eingefügt, du kannst jetzt weiterarbeiten!").

14.4 Wo lauern die Gefahren? Das Dilemma von Safety vs. Liveness

Wenn wir anfangen, unsere Programme mit Synchronisationsmechanismen (wie Sperren/Locks) auszustatten, bewegen wir uns auf einem schmalen Grat zwischen zwei Fehlerquellen.

Gefahr 1: Safety-Probleme (Die Daten sind kaputt)

Wenn wir zu wenig synchronisieren, schlägt das Interleaving erbarmungslos zu.

- Race Conditions (Wettlaufsituationen): Das Ergebnis des Programms hängt vom zufälligen Timing der Threads ab. Die Daten werden inkonsistent. Solche Fehler treten oft nur unter hoher Last auf und sind schwer zu debuggen (Heisenbugs).

Gefahr 2: Liveness-Probleme (Das Programm steht still)

Aus Angst vor kaputten Daten neigen Anfänger dazu, zu viel zu synchronisieren. Wenn wir überall dicke Sperren einbauen, laufen die Threads zwar sicher, aber nicht mehr parallel. Im schlimmsten Fall zerstören wir die "Lebendigkeit" (Liveness) des Programms komplett:

- Deadlock (Verklemmung): Thread A hat einen exklusiven Zugriff auf Ressource X und wartet auf Ressource Y. Thread B hat exklusiven Zugriff auf Ressource Y und wartet auf Ressource X. Beide warten bis in alle Ewigkeit. Das System ist tot.

14.5 Die Checkliste für Programmierer

Was bedeutet das nun für Ihren Alltag als Software-Entwickler? Wann müssen Sie im Code aufmerksam werden? Merken Sie sich ab jetzt diese goldene Regel:

Die Gefahr geht immer von "Shared Mutable State" aus (Geteilter, veränderbarer Zustand).

Stellen Sie sich bei neuen Klassen immer drei Fragen:

1. Ist der Zustand geteilt (Shared)? Greifen mehrere Threads auf dieselbe Instanz zu? Lokale Variablen auf dem Methoden-Stack sind immer sicher! Instanzvariablen sind potenziell gefährlich.
2. Ist der Zustand veränderbar (Mutable)? Wenn eine Variable final ist und nie wieder geändert werden kann, ist sie sicher.

3. Die Schnittmenge: Nur wenn ein Zustand von mehreren Threads gelesen und von mindestens einem Thread verändert wird, müssen Sie synchronisieren!

14.6 Ausblick: Unser Fahrplan für Teil II

Um die beschriebenen Gefahren zu meistern und robuste, nebenläufige Programme zu schreiben, werden wir uns in den kommenden Kapiteln systematisch durch das Arsenal der Java-API arbeiten – von den maschinennahen Basics bis hin zu modernen Frameworks:

- **Kapitel 15: Der klassische Schutzschild – Monitore und synchronized** Wir lösen das Race-Condition-Problem und lernen, wie das synchronized-Schlüsselwort und intrinsische Sperren in Java funktionieren.
- **Kapitel 16: Das Sichtbarkeitsproblem – Java Memory Model und volatile** Wir besiegen das CPU-Cache-Problem und verstehen das "Happens-Before"-Prinzip der JVM.
- **Kapitel 17: Klassische Thread-Kommunikation – wait() und notify()** Wir steuern den Ablauf von Threads über Bedingungsvariablen und lösen das klassische Erzeuger-Verbraucher-Problem.
- **Kapitel 18: Moderne und flexible Sperren – Das java.util.concurrent.locks-Paket** Wir überwinden die Grenzen von synchronized mit flexiblen Werkzeugen wie dem ReentrantLock und dem ReadWriteLock.
- **Kapitel 19: Thread-Choreografie – Barrieren, Zähler und Semaphoren** Wir lernen High-Level-Werkzeuge (CountDownLatch, CyclicBarrier) kennen, um ganze Gruppen von Threads zu koordinieren.
- **Kapitel 20: Thread-sichere Datenstrukturen – Concurrent Collections** Wir sehen uns an, warum eine normale ArrayList gefährlich ist und wie wir mit ConcurrentHashMap oder BlockingQueue Daten sicher und elegant teilen.
- **Kapitel 21: Höchstleistung ohne Sperren – Lock-Free Programming** Wir holen das Maximum an Performance heraus und lernen, wie atomare Variablen (AtomicInteger) völlig ohne blockierende Sperren funktionieren.
- **Kapitel 22: Wenn nichts mehr geht – Liveness-Probleme vermeiden** Zum Abschluss analysieren wir Architekturfehler wie Deadlocks im Detail und lernen Strategien, um sie beim Software-Design proaktiv zu verhindern.

15 Der klassische Schutzschild – Monitore und das synchronized-Schlüsselwort

Im letzten Kapitel haben wir gesehen, wie das Interleaving (die unvorhersehbare Verzahnung von Threads) dazu führen kann, dass Daten überschrieben werden oder verloren gehen. Das klassische Beispiel war das "Lost Update" bei einer einfachen Kontostandsänderung: kontoStand = kontoStand + 10;.

Da diese Anweisung aus mehreren maschinennahen Schritten besteht (Lesen, Rechnen, Schreiben), kann ein zweiter Thread genau dazwischen grätschen. Um dieses Chaos zu verhindern, müssen wir diese drei Schritte zu einer atomaren Operation zusammenfassen. Atomar bedeutet: Entweder wird die Operation ganz ausgeführt oder gar nicht, aber sie kann niemals mittendrin von einem anderen Thread unterbrochen oder beobachtet werden.

Das Werkzeug der Wahl in der klassischen Java-Programmierung hierfür ist das Schlüsselwort synchronized. Es implementiert das Prinzip des Mutual Exclusion (wechselseitiger Ausschluss, oft "Mutex" genannt).

15.1 Das Geheimnis der Java-Objekte: Der Monitor

Bevor wir Code schreiben, müssen wir verstehen, wie Java Sperren (Locks) technisch umsetzt. Die Erfinder von Java haben eine bemerkenswerte Designentscheidung getroffen: Jedes einzelne Objekt in Java besitzt automatisch ein eingebautes Schloss. Dieses unsichtbare Schloss wird als Monitor (oder auch Intrinsic Lock) bezeichnet.

Die Regel für diesen Monitor ist denkbar einfach:

1. Der Monitor hat genau einen Schlüssel.
2. Wenn ein Thread einen Code-Bereich betreten will, der durch dieses Objekt geschützt ist, muss er sich den Schlüssel schnappen.
3. Wenn er den Schlüssel hat, betritt er den Raum und schließt die Tür hinter sich ab. Er führt seinen Code aus.
4. Kommt ein anderer Thread an dieselbe Tür, findet er sie verschlossen vor. Er wird vom Betriebssystem in einen Wartezustand (Status BLOCKED) versetzt und verbraucht keine CPU-Zeit mehr. Er wartet einfach vor der Tür.
5. Sobald der erste Thread den Raum verlässt, gibt er den Schlüssel automatisch zurück. Die Tür geht auf, und ein anderer auf den Schlüssel wartender Thread darf eintreten.

15.2 Synchronisierte Methoden (Instanz-Ebene)

Die einfachste Art, diesen Monitor zu nutzen, ist das Voranstellen des Schlüsselworts synchronized in der Methodensignatur.

Schauen wir uns unsere fehlerhafte Bankkonto-Klasse an und reparieren sie:

```
public class Bankkonto {
    private int saldo = 0;

    // Durch "synchronized" wird die gesamte Methode zum kritischen
    // Bereich
    public synchronized void einzahlen(int betrag) {
        // Hier sind wir absolut sicher vor Interleaving!
        int aktuellerSaldo = this.saldo;
        aktuellerSaldo = aktuellerSaldo + betrag;
        this.saldo = aktuellerSaldo;
    }

    public synchronized int getSaldo() {
        return this.saldo;
    }
}
```

Was genau passiert hier? Wenn Sie eine Instanzmethode synchronisieren, verwendet Java implizit das aufgerufene Objekt selbst (also this) als Schloss. Wenn Thread A die Methode konto1.einzahlen(50) aufruft, schnappt er sich den Monitor von konto1. Wenn Thread B exakt im selben Moment konto1.einzahlen(20) aufrufen will, prallt er an der Tür ab und muss warten.

Ein extrem wichtiges Detail: Der Monitor sperrt nicht nur diese eine Methode, sondern das gesamte Objekt für andere synchronisierte Zugriffe! Wenn Thread A gerade Geld einzahlt, kann Thread B nicht einmal die Methode konto1.getSaldo() aufrufen. Auch diese ist synchronisiert und erfordert denselben Schlüssel (this).

15.3 Synchronisierte Blöcke (Feingranulare Sperren)

Eine ganze Methode zu sperren ist einfach, aber oft unklug. Sperren kosten Performance (da sie Parallelität verhindern). Der Bereich, in dem Threads nicht parallel arbeiten dürfen (die sogenannte Kritische Sektion), sollte immer so klein wie möglich gehalten werden.

Stellen Sie sich vor, Ihre Methode lädt zuerst aufwendig Daten aus dem Internet (was 2 Sekunden dauert) und aktualisiert erst dann den Kontostand. Es wäre fatal, die gesamte Methode zu synchronisieren, da dann kein anderer Thread parallel Daten laden dürfte.

Hier nutzen wir den synchronized-Block:

```
public class BessereBankkonto {
    private int saldo = 0;

    // Ein separates Objekt, das NUR als Schloss dient
    private final Object lockObject = new Object();

    public void einzahlen(int betrag) {
        // ... Code, der parallel laufen darf (z. B. Validierung des
        // Betrags) ...
        System.out.println(
                Thread.currentThread().getName() + " validiert...");

        // Ab hier beginnt der kritische Bereich!
        // Wir geben explizit an, WELCHES Objekt als Schloss dienen
        // soll.
        synchronized (this.lockObject) {
            this.saldo = this.saldo + betrag;
        } // Hier wird das Schloss automatisch wieder freigegeben

        // ... Weiterer Code, der wieder parallel laufen darf ...
    }

    public int getSaldo() {
        synchronized (this.lockObject) {
            return this.saldo;
        }
    }

}
```

Die Vorteile des Blocks:

1. Kürzere Wartezeiten: Die kritische Sektion wird minimiert.
2. Freie Wahl des Schlosses: Sie müssen nicht zwingend this verwenden. Sie können ein eigenes Object als dediziertes Schloss anlegen. Das ist Best Practice, da es verhindert, dass fremder Code von außen versehentlich (oder böswillig) auf Ihrer Instanz synchronized(konto) aufruft und so Ihr gesamtes System blockiert.

15.4 Statische Synchronisation (Klassen-Ebene)

Was passiert, wenn wir eine static-Methode synchronisieren? Statische Methoden gehören nicht zu einer konkreten Instanz (this existiert dort nicht), sondern zur Klasse selbst.

```
public class IdGenerator {
    private static int naechsteId = 1;

    public static synchronized int generiereId() {
        return naechsteId++;
    }
}
```

In diesem Fall nutzt Java das Klassen-Objekt als Monitor. Jede Klasse wird in der JVM durch ein Objekt vom Typ java.lang.Class repräsentiert (hier IdGenerator.class). Gäbe es in dieser Klasse auch normale, instanzbezogene synchronized-Methoden, würden diese sich nicht gegenseitig blockieren! Die statische Methode nutzt das Schloss von IdGenerator.class, die Instanzmethode nutzt das Schloss der jeweiligen Instanz (this). Das sind zwei völlig unterschiedliche Türen mit unterschiedlichen Schlüsseln.

15.5 Ein geniales Feature: Reentrancy (Wiedereintrittsfähigkeit)

Oft stellen sich Studierende an dieser Stelle folgende Frage: Was passiert, wenn eine synchronisierte Methode eine ANDERE synchronisierte Methode desselben Objekts aufruft? Sperrt sich der Thread dann selbst aus und das Programm bleibt für immer stehen (Deadlock)?

Die gute Nachricht lautet: Nein. Intrinsic Locks in Java sind "reentrant" (wiedereintrittsfähig).

```
public class ReentrancyBeispiel {
    public synchronized void methodeA() {
        System.out.println("In Methode A");
        methodeB(); // Ruft eine andere synchronisierte Methode auf!
    }

    public synchronized void methodeB() {
        System.out.println("In Methode B");
    }
}
```

Wenn ein Thread methodeA aufruft, erwirbt er den Monitor von this. Wenn er nun methodeB aufruft, prüft die JVM, welcher Thread den Schlüssel aktuell besitzt. Da es derselbe Thread ist, darf er passieren! Die JVM führt intern einfach einen Zähler. Bei Betreten von methodeA geht der Zähler auf 1, bei methodeB auf 2. Verlässt er methodeB, sinkt er auf 1. Erst wenn methodeA verlassen wird (Zähler auf 0), wird der Monitor wirklich für andere Threads freigegeben.

Fazit des Kapitels: Mit dem synchronized-Schlüsselwort haben wir unser wertvollstes Werkzeug kennengelernt, um Datenstrukturen vor dem "Lost Update" und anderen Race Conditions zu schützen. Es ist robust und einfach anzuwenden.

Doch wie in Kapitel 14 erwähnt, gibt es neben der Atomarität (dem Schutz) noch ein zweites großes Problem, wenn Threads auf denselben Speicher zugreifen: das Problem der Sichtbarkeit durch lokale CPU-Caches. Diesem Problem widmet sich das nächste Kapitel.

16 Das Sichtbarkeitsproblem – Das Java Memory Model und volatile

Im letzten Kapitel haben wir gelernt, wie wir mit synchronized verhindern können, dass zwei Threads sich beim gleichzeitigen Schreiben von Daten in die Quere kommen (Atomarität). Wir haben den Code quasi in einen Tresor gesperrt.

Doch was passiert, wenn wir gar keine komplexen Berechnungen wie saldo = saldo + 10 durchführen wollen? Was ist, wenn Thread A einfach nur einen Schalter (ein Flag) umlegt und Thread B diesen Schalter nur ablesen soll? Hier gibt es keinen "Lost Update", also brauchen wir doch auch keine Synchronisation, oder?

Die überraschende und für viele Anfänger frustrierende Antwort lautet: Doch, wir müssen eingreifen. Denn in der modernen Hardware-Architektur gibt es keine Garantie dafür, dass Thread B jemals sieht, was Thread A getan hat. Aus diesem Grund muss auch die getSaldo-Methode im vorherigen Kapitel synchronisiert werden. Willkommen beim Sichtbarkeitsproblem (Visibility Problem).

16.1 Die Hardware-Illusion: CPU-Caches

Um dieses Phänomen zu verstehen, müssen wir uns von der naiven Vorstellung verabschieden, dass alle Threads direkt auf denselben physikalischen Arbeitsspeicher (RAM) zugreifen.

Der Hauptspeicher (RAM) ist im Vergleich zu einer modernen CPU extrem langsam. Wenn eine CPU bei jedem Variablenzugriff auf den RAM warten müsste, würde sie die meiste Zeit untätig verbringen. Um dieses Nadelöhr zu umgehen, besitzen moderne Multi-Core-Prozessoren mehrere Schichten von rasend schnellen Zwischenspeichern: die CPU-Caches (L1, L2, L3).

Wenn ein Thread auf einem Kern gestartet wird, kopiert die CPU die benötigten Variablen aus dem langsamen Hauptspeicher in ihren eigenen, lokalen L1-Cache.

- Wenn der Thread die Variable liest, liest er sie aus seinem lokalen Cache.
- Wenn der Thread die Variable ändert, ändert er sie zunächst nur in seinem lokalen Cache.

Wann genau die Hardware entscheidet, diesen neuen Wert aus dem Cache zurück in den globalen Hauptspeicher (RAM) zu schreiben ("Flush"), ist völlig unvorhersehbar. Ebenso unvorhersehbar ist es, wann ein anderer CPU-Kern seinen eigenen Cache aktualisiert ("Refresh"), um Änderungen aus dem Hauptspeicher zu übernehmen.

16.2 Das Problem: Der Endlos-Schlaf

Schauen wir uns an, wie dieses Hardware-Verhalten unseren Java-Code ruinieren kann. Ein sehr typisches Szenario ist ein Hintergrund-Thread, der so lange laufen soll, bis ihn jemand von außen stoppt.

```
public class SichtbarkeitsProblem {
    // Ein einfaches Flag, das als Schalter dient
    private static boolean running = true;

    public static void main(String[] args)
            throws InterruptedException {
        // Hintergrund-Thread starten
        Thread worker = new Thread(() -> {
            System.out.println("Worker: Gestartet.");
            while (running) {
                // Tue nichts, drehe dich einfach im Kreis (Busy
                // Waiting). In der Realität würde hier Arbeit
                // verrichtet werden
            }
            System.out.println("Worker: Beendet.");
        });

        worker.start();

        // Der Main-Thread wartet kurz...
        Thread.sleep(1000);

        // ... und legt dann den Schalter um!
        System.out.println("Main: Versuche den Worker zu stoppen...");
        running = false;
        System.out.println(
                "Main: Schalter auf 'false' gesetzt. Main ist fertig.");
    }
}
```

Was passiert bei der Ausführung?

Wenn Sie diesen Code ausführen, werden Sie mit hoher Wahrscheinlichkeit Folgendes sehen:

1. "Worker: Gestartet."
2. Nach 1 Sekunde: "Main: Versuche den Worker zu stoppen..."
3. "Main: Schalter auf 'false' gesetzt. Main ist fertig."
4. ... und dann passiert nichts mehr. Das Programm beendet sich nicht. Die Ausgabe "Worker: Beendet." erscheint nie.

Warum?

Der Worker-Thread hat die Variable running (Wert: true) in seinen lokalen CPU-Cache geladen. Da er in der while-Schleife nichts anderes tut, optimiert die CPU (und der

Java-Compiler) den Code radikal: "Die Variable ändert sich in dieser Schleife nicht, also muss ich sie nicht ständig aus dem Hauptspeicher neu einlesen."

Der Main-Thread ändert die Variable auf false. Diese Änderung landet aber vielleicht nur im Cache des Main-Threads, oder der Worker-Thread weigert sich schlichtweg, seinen eigenen Cache zu aktualisieren. Der Worker ist "blind" für die Änderung.

16.3 Die Lösung: Das volatile-Schlüsselwort

Um dieses Problem zu lösen, müssen wir der Java Virtual Machine (JVM) und der Hardware den Befehl geben: "Optimiere diese Variable nicht weg! Wenn du sie liest, lies sie IMMER direkt aus dem Hauptspeicher. Wenn du sie schreibst, schreibe sie IMMER direkt in den Hauptspeicher."

In Java tun wir das, indem wir der Variablen das Schlüsselwort volatile (flüchtig) voranstellen.

```
// Mit 'volatile' garantieren wir die Sichtbarkeit über
// Thread-Grenzen hinweg
private static volatile boolean running = true;
```

Ändern Sie diese eine Zeile im obigen Code, und das Programm wird sich sofort und zuverlässig beenden.

16.4 Das Java Memory Model (JMM) und "Happens-Before"

Die Funktionsweise von volatile (und auch von synchronized) wird durch einen formalen Vertrag definiert: das Java Memory Model (JMM). Das JMM abstrahiert die komplexe Hardware (egal ob Intel, AMD oder ARM) und gibt uns Java-Entwicklern plattformunabhängige Garantien.

Das wichtigste Konzept des JMM ist die sogenannte Happens-Before-Beziehung (Passiert-Vorher). Es ist eine Garantie dafür, in welcher Reihenfolge Speicheroperationen für andere Threads sichtbar werden.

Zwei grundlegende Happens-Before-Regeln lauten:

1. Monitor-Lock-Regel: Das Verlassen eines synchronized-Blocks (unlock) "happens-before" dem nächsten Betreten (lock) desselben Monitors durch einen anderen Thread. Das bedeutet: synchronized löst das Sichtbarkeitsproblem automatisch mit! Wer den Raum betritt, sieht garantiert alles, was der Vorgänger hinterlassen hat.

2. Volatile-Variablen-Regel: Ein Schreibzugriff auf eine volatile-Variable "happens-before" jedem nachfolgenden Lesezugriff auf genau diese Variable.

Darüber hinaus verhindert volatile das sogenannte Instruction Reordering. Compiler und CPUs vertauschen oft die Reihenfolge von Befehlen, um sie schneller abzuarbeiten, solange das Ergebnis scheinbar gleich bleibt. Bei nebenläufigem Code kann dieses Vertauschen jedoch zu fatalen Fehlern führen. volatile zieht hier eine unsichtbare Grenze (Memory Barrier), über die hinweg Anweisungen nicht vertauscht werden dürfen.

16.5 Die goldene Regel: volatile vs. synchronized

Oft entsteht bei Studierenden nun der Eindruck: "Super, volatile ist viel kürzer und blockiert keine Threads wie synchronized. Dann nutze ich ab jetzt nur noch volatile!"

Das ist ein fataler Fehler! volatile und synchronized lösen zwei völlig unterschiedliche Probleme.

Eigenschaft	volatile	synchronized
Garantierte Sichtbarkeit (Visibility)?	Ja	Ja
Garantierte Atomarität (Atomicity)?	Nein!	Ja
Blockiert andere Threads (Locking)?	Nein	Ja

Ein volatile int zaehler; hilft Ihnen nicht beim Lost-Update-Problem aus Kapitel 12. Wenn zwei Threads gleichzeitig zaehler++ ausführen, gehen trotz volatile Zählungen verloren, denn zaehler++ besteht weiterhin aus drei separaten Schritten (Lesen, Addieren, Schreiben), die sich verzahnen können. volatile garantiert nur, dass der gelesene Wert aktuell ist, es verhindert aber nicht, dass beide Threads gleichzeitig lesen!

Wann nutzen wir also was?

- Nutzen Sie volatile, wenn eine Variable von mehreren Threads gelesen, aber nur von einem einzigen Thread geschrieben wird (wie unser running-Flag).
- Nutzen Sie synchronized (oder andere Sperren), wenn eine Variable von mehreren Threads gelesen und von mehreren Threads geschrieben (und auf Basis ihres alten Wertes verändert) wird.

17 Klassische Thread-Kommunikation – wait(), notify() und notifyAll()

In den vorangegangenen Kapiteln haben wir unser Programm gegen Datenverlust abgesichert. Mit synchronized haben wir exklusive Zugriffe erzwungen (Atomarität) und mit synchronized bzw. volatile sichergestellt, dass Änderungen für alle sofort sichtbar sind (Sichtbarkeit).

Bisherige Threads waren jedoch Egoisten. Sie haben sich den Monitor (den Schlüssel zum Objekt) geschnappt, ihre Arbeit erledigt und den Monitor wieder freigegeben. Oft reicht das aber nicht aus. Was ist, wenn ein Thread den Monitor zwar hat, aber feststellt, dass er seine Arbeit noch gar nicht ausführen kann, weil eine bestimmte Bedingung noch nicht erfüllt ist?

In diesem Kapitel widmen wir uns der Koordination: Wie legen wir Threads effizient schlafen und wie wecken sie sich gegenseitig auf?

17.1 Das Problem: Aktives Warten (Busy Waiting)

Stellen Sie sich vor, wir programmieren einen Postboten (Erzeuger) und einen Empfänger (Verbraucher). Beide teilen sich einen Briefkasten (Shared State). Der Empfänger möchte einen Brief lesen, aber der Briefkasten ist noch leer.

Ein naiver Ansatz wäre das sogenannte aktive Warten (Busy Waiting / Polling):

```
// Anti-Pattern: Das sollten Sie niemals tun!
public void aufBriefWarten() {
    while (briefkasten.istLeer()) {
        // Tue nichts, frage einfach in einer Endlosschleife
        // immer wieder nach!
    }
    briefkasten.lesen();
}
```

Dieser Ansatz ist katastrophal für die Systemressourcen. Der Thread des Empfängers läuft permanent auf 100 % CPU-Auslastung. Er verbrennt Strom und blockiert einen wertvollen Rechenkern, nur um immer wieder dieselbe Frage zu stellen. Das ist, als würden Sie sich alle zwei Sekunden in Ihr Auto setzen, zum Briefkasten fahren, hineinschauen und wieder zurückfahren.

Wir brauchen einen Mechanismus, der dem Empfänger sagt: "Geh schlafen, verbrauche keine CPU-Zeit mehr. Ich wecke dich auf, sobald ein Brief da ist!"

17.2 Der Warteraum des Monitors

Die Erfinder von Java haben die Lösung direkt in die Wurzel der Sprache eingebaut. Zur Erinnerung: Jedes Objekt in Java besitzt einen Monitor (ein eingebautes Schloss). Was wir bisher nicht erwähnt haben: Jeder Monitor besitzt zusätzlich einen eigenen Warteraum (Wait Set).

Die Methoden zur Steuerung dieses Warteraums finden sich daher nicht in der Klasse Thread, sondern in der Klasse java.lang.Object! Jedes Objekt erbt sie: wait, notify und notifyAll. Die wichtigste Grundregel: Sie dürfen wait(), notify() oder notifyAll() ausschließlich innerhalb eines synchronized-Blocks oder einer synchronized-Methode aufrufen, die genau jenes Objekt sperrt! Ein Thread kann nur den Schlüssel abgeben oder jemanden im Warteraum anstupsen, wenn er den Schlüssel aktuell auch besitzt. Andernfalls wirft Java sofort eine IllegalMonitorStateException.

17.2.1 wait()

Der aufrufende Thread gibt den Schlüssel (das Monitor-Lock) für das aufgerufene Objekt sofort frei, betritt den internen Warteraum des Objekts und wird vom Betriebssystem in den Schlafzustand (WAITING) versetzt. Er verbraucht ab sofort 0 % CPU-Leistung, bis er explizit wieder geweckt wird.

17.2.2 notify()

Weckt einen einzigen, vom Betriebssystem zufällig ausgewählten Thread auf, der im Warteraum des aufgerufenen Objekts schläft.

Der kritische Punkt: Aufgeweckt zu werden, bedeutet jedoch nicht, dass dieser Thread sofort weiterarbeitet! Der Thread erwacht zwar aus dem Schlaf (verlässt den Status WAITING), hat aber den Schlüssel für das Objekt noch nicht zurück. Er wechselt quasi in den Vorraum (Status BLOCKED). Und jetzt kommt das Wichtigste: Wenn der Thread, der notify() gerufen hat, den Schlüssel schließlich freigibt, bekommt unser frisch geweckter Thread diesen Schlüssel nicht automatisch überreicht. Er muss nun knallhart mit allen anderen Threads konkurrieren, die in genau diesem Moment ebenfalls versuchen, den Schlüssel für dieses Objekt zu ergattern (etwa völlig neue Threads, die gerade erst von außen bei einem synchronized-Block ankommen). Es gibt für geweckte Threads keine "Vorfahrtsregel" oder VIP-Behandlung! Wer den Schlüssel erbeutet, gewinnt; wer verliert, bleibt im Status BLOCKED.

17.2.3 notifyAll()

Weckt alle Threads auf, die im Warteraum des aufgerufenen Objekts schlafen.

Auch hier gilt das unerbittliche Prinzip des einzigen Schlüssels: Da es für das Objekt weiterhin nur einen einzigen Schlüssel gibt, können unmöglich alle geweckten Threads gleichzeitig weiterlaufen. Sie erwachen alle aus dem Schlafzustand und stürzen sich gemeinsam auf den Vorraum (BLOCKED). Dort konkurrieren sie nun alle erbittert untereinander – und ebenfalls mit möglichen neuen Threads von außen – um diesen einen Schlüssel. Nur ein einziger Thread wird gewinnen und seine Arbeit fortsetzen dürfen. Die anderen bleiben blockiert und müssen jedes Mal, wenn der Schlüssel wieder frei wird, erneut an dem Wettrennen teilnehmen.

17.3 Das Paradigma: Erzeuger und Verbraucher (Producer-Consumer)

Um diese drei Methoden in Aktion zu sehen, nutzen wir das berühmteste Koordinations-Muster der Informatik: das Erzeuger-Verbraucher-Problem.

Wir bauen einen Puffer (z.B. ein Lager).

- Producer: Möchte Items in das Lager legen. Wenn das Lager voll ist, muss er warten. Wenn er etwas hineinlegt, muss er den Consumer aufwecken.
- Consumer: Möchte Items aus dem Lager nehmen. Wenn das Lager leer ist, muss er warten. Wenn er etwas herausnimmt, muss er den Producer aufwecken, damit dieser wieder Platz zum Nachfüllen hat.

Die Implementierung des Puffers

Wir bauen eine einfache Klasse, die genau einen einzigen String (eine Nachricht) speichern kann.

```
public class NachrichtenPuffer {
    private String nachricht;
    // true, wenn der Puffer leer ist. false, wenn eine Nachricht
    // bereitliegt.
    private boolean istLeer = true;

    // Methode für den Producer (Schreiben)
    public synchronized void schreiben(String neueNachricht) {
        // 1. Prüfen, ob wir überhaupt arbeiten können
        while (!istLeer) {
            try {
                // Puffer ist voll! Wir müssen warten, bis der
                // Consumer liest.
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        // 2. Zustand ändern (Die eigentliche Arbeit)
        this.nachricht = neueNachricht;
        this.istLeer = false;
        System.out.println(
```

```
                "Producer hat geschrieben: " + neueNachricht);

        // 3. Andere informieren
        // Wecke alle Threads auf, die an diesem Monitor (this) warten
        notifyAll();
    }

    // Methode für den Consumer (Lesen)
    public synchronized String lesen() {
        // 1. Prüfen, ob wir überhaupt arbeiten können
        while (istLeer) {
            try {
                // Puffer ist leer! Wir müssen warten, bis der
                // Producer schreibt.
                wait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        // 2. Zustand ändern
        this.istLeer = true;
        System.out.println("Consumer hat gelesen: " + this.nachricht);

        // 3. Andere informieren
        // Wecke den Producer, damit er nachlegen kann
        notifyAll();

        return this.nachricht;
    }
}
```

Wenn Sie nun einen Thread starten, der in einer Schleife schreiben() aufruft, und einen zweiten, der lesen() aufruft, werden sich die beiden wie bei einem perfekten Ping-Pong-Spiel abwechseln, egal wie oft das Betriebssystem versucht, sie zu unterbrechen.

17.4 Die eiserne Regel: Das "Spurious Wakeup"

Schauen Sie sich den Code in der lesen()-Methode noch einmal ganz genau an. Dort steht eine while-Schleife, kein if-Block:

```
        // Richtig:
        while (istLeer) {
            wait();
        }

        // Falsch (Anti-Pattern!):
        if (istLeer) {
            wait();
        }
```

Warum ist das so essenziell? Warum muss der Thread, nachdem er von notifyAll() aufgeweckt wurde, die Bedingung istLeer noch einmal prüfen? Dafür gibt es zwei zwingende Gründe:

1. Konkurrenz nach dem Aufwachen: Wenn notifyAll() aufgerufen wird, wachen zwar alle wartenden Consumer auf, aber sie bekommen den Monitor nicht sofort! Sie müssen sich wieder um den Schlüssel streiten, genau wie jeder andere Thread auch. Es kann sein, dass Consumer A aufwacht, Consumer B ihn aber überholt, sich den Schlüssel schnappt und die Nachricht wegliest. Wenn Consumer A danach an die Reihe kommt und nicht noch einmal per while prüft, ob die Nachricht noch da ist, rennt er blind weiter und wirft vermutlich eine Exception (z. B. eine NullPointerException).
2. Spurious Wakeups (Falsches Erwachen): Aus extrem tief liegenden technischen Gründen in den Betriebssystem-Implementierungen (POSIX-Threads) erlaubt die Spezifikation, dass ein Thread aus einem wait() aufwacht, obwohl niemand notify() aufgerufen hat! Dies nennt man ein Spurious Wakeup. Wenn Sie ein if verwenden, würde der Thread in diesem Fall einfach weiterlaufen, obwohl die Daten noch gar nicht da sind. Die while-Schleife fängt diesen Systemfehler elegant ab.

Prüfungsrelevanter Merksatz: Ein wait() gehört immer in eine while-Schleife, die die Warte-Bedingung abprüft!

17.5 notify() vs. notifyAll(): Was ist besser?

Sie haben sich vielleicht gefragt, warum im Beispiel notifyAll() (wecke alle) und nicht notify() (wecke einen) verwendet wurde.

notify() ist minimal performanter, birgt aber eine massive Gefahr: das Lost Wakeup (Verlorenes Aufwachen). Stellen Sie sich vor, Sie haben zwei Producer und zwei Consumer. Ein Consumer liest und ruft notify() auf. Er möchte eigentlich einen Producer wecken. Der Zufall will es aber, dass der Scheduler den anderen Consumer weckt! Dieser zweite Consumer wacht auf, sieht, dass der Puffer leer ist, und legt sich wieder schlafen. Niemand hat den Producer geweckt. Das System verklemmt sich für immer in einem Deadlock.

Best Practice: Nutzen Sie in 99 % der Fälle immer notifyAll(). Es weckt alle auf. Die Threads, die nichts tun können, legen sich dank der while-Schleife ohnehin sofort wieder schlafen. Die Performance-Einbuße ist bei modernen Systemen meist vernachlässigbar, die gewonnene Sicherheit vor Deadlocks jedoch enorm.

18 Moderne und flexible Sperren – Das java.util.concurrent.locks-Paket

Mit dem synchronized-Schlüsselwort und den Methoden wait() und notifyAll() haben Sie im bisherigen Verlauf ein mächtiges, aber sehr starres Fundament kennengelernt. Diese Werkzeuge sind direkt in die Java Virtual Machine (JVM) integriert. Sie sind einfach zu verwenden, stoßen aber bei komplexen Architekturen schnell an ihre Grenzen.

Mit Java 5 wurde daher das Paket java.util.concurrent.locks eingeführt. Anstatt unsichtbare Monitore zu nutzen, arbeiten wir hier mit echten Java-Objekten, die Sperren (Locks) repräsentieren.

18.1 Die Grenzen von synchronized

Warum sollten wir uns überhaupt mit einer Alternative beschäftigen? Das klassische synchronized hat vier gravierende Nachteile:

1. Strenge Blockstruktur: Ein synchronized-Block muss immer in derselben Methode geöffnet und geschlossen werden. Sie können nicht in Methode A eine Sperre setzen und sie in Methode B wieder freigeben.
2. Endloses Warten (Kein Timeout): Wenn ein Thread an einem synchronized-Block ankommt und die Tür verschlossen ist, wartet er. Und er wartet. Wenn der Thread, der den Schlüssel hat, in eine Endlosschleife gerät, wartet unser Thread bis in alle Ewigkeit. Es gibt keine Möglichkeit zu sagen: "Warte maximal 3 Sekunden, und wenn die Tür dann nicht offen ist, mach etwas anderes."
3. Keine Unterbrechung: Ein Thread, der an einem synchronized-Block auf Einlass wartet, reagiert nicht auf Thread.interrupt(). Er lässt sich von außen nicht abbrechen.
4. Alles oder nichts: Sie können bei wait()/notifyAll() nicht gezielt steuern, wer aufgeweckt wird. Alle Threads teilen sich denselben Warteraum.

Das Interface Lock und seine Implementierungen lösen all diese Probleme auf elegante Weise.

18.2 Das ReentrantLock: Die objektorientierte Alternative

Die Standard-Implementierung des Lock-Interfaces ist das ReentrantLock. Der Name verrät es bereits: Genau wie synchronized ist auch diese Sperre reentrant

(wiedereintrittsfähig). Ein Thread, der die Sperre bereits hält, kann sie problemlos erneut anfordern, ohne sich selbst auszusperren.

So sieht die grundlegende Verwendung aus:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockBeispiel {
    // Das Lock-Objekt wird explizit instanziiert
    private final Lock schloss = new ReentrantLock();
    private int saldo = 0;

    public void einzahlen(int betrag) {
        schloss.lock(); // Sperre anfordern. Wenn sie belegt ist,
                        // wartet der Thread hier.
        try {
            // Kritischer Bereich
            this.saldo += betrag;
        } finally {
            // WICHTIG: Die Sperre MUSS im finally-Block freigegeben
            // werden!
            schloss.unlock();
        }
    }
}
```

Das eiserne Gesetz des ReentrantLock: Im Gegensatz zu synchronized gibt die JVM ein ReentrantLock nicht automatisch frei, wenn eine Exception auftritt! Wenn Sie das unlock() vergessen oder es wegen einer Exception übersprungen wird, bleibt die Sperre für immer geschlossen. Ihr System verklemmt sich unweigerlich. Ein Aufruf von lock() muss immer zwingend von einem try-finally-Block gefolgt werden, in dessen finally-Teil unlock() steht.

18.3 Fortgeschrittene Funktionen des ReentrantLock

Der Mehraufwand des try-finally-Blocks zahlt sich durch mächtige Zusatzfunktionen aus.

18.3.1 Polling: Ausprobieren mit tryLock()

Anstatt blind zu warten, kann ein Thread prüfen, ob die Sperre frei ist. Wenn nicht, geht er einfach weiter.

```
        if (schloss.tryLock()) {
            try {
                System.out.println(
                        "Ich habe die Sperre bekommen und arbeite nun.");
            } finally {
                schloss.unlock();
            }
        } else {
            System.out.println(
```

```
                    "Sperre belegt. Ich mache in der Zeit etwas anderes.");
        }
```

Es gibt auch eine überladene Version mit Timeout: schloss.tryLock(3, TimeUnit.SECONDS). Dies ist das ultimative Mittel, um festgefahrene Systeme (Deadlocks) zu verhindern.

18.3.2 Gerechtigkeit: Der Fairness-Parameter

Wenn mehrere Threads vor einem synchronized-Block warten und die Tür aufgeht, entscheidet der Scheduler der JVM völlig zufällig, wer als Nächstes eintreten darf. Ein Thread könnte theoretisch "verhungern" (Starvation). Beim ReentrantLock können Sie im Konstruktor "Fairness" aktivieren: new ReentrantLock(true).

- Fair: Der Thread, der am längsten wartet (First-In, First-Out), bekommt die Sperre garantiert als Nächstes.
- Hinweis: Fairness kostet Performance. Das ständige Umschalten und Beibehalten der Reihenfolge durch das Betriebssystem ist teuer. Nutzen Sie es nur, wenn absolutes Verhungern ein reales Problem darstellt.

18.4 Lese- und Schreib-Sperren: Das ReadWriteLock

Oft haben wir Datenstrukturen (wie einen Cache oder Konfigurationseinstellungen), die extrem oft gelesen, aber nur sehr selten verändert (geschrieben) werden.

Wenn wir ein normales ReentrantLock nutzen, blockieren sich lesende Threads gegenseitig. Das ist ineffizient! Wenn 100 Threads nur den Wert abfragen wollen, verändern sie nichts. Sie könnten das völlig gefahrlos exakt gleichzeitig tun. Gefahr droht nur, wenn jemand schreibt.

Hierfür gibt es das ReentrantReadWriteLock. Es verwaltet intern zwei getrennte Sperren.

Die Zugriffsregeln:

- Lese-Sperre (readLock): Mehrere Threads können diese Sperre gleichzeitig halten, solange niemand die Schreib-Sperre hält.
- Schreib-Sperre (writeLock): Nur ein einziger Thread darf diese Sperre halten. Wenn ein Thread schreiben will, müssen alle aktiven Leser erst fertig werden. Solange geschrieben wird, darf niemand lesen.

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
```

```
public class Cache {
    private final ReadWriteLock rwLock = new ReentrantReadWriteLock();
    private String daten = "Initialwert";

    public String lesen() {
        rwLock.readLock().lock(); // Viele Threads dürfen gleichzeitig
                                  // hier rein!
        try {
            return daten;
        } finally {
            rwLock.readLock().unlock();
        }
    }

    public void schreiben(String neuerWert) {
        rwLock.writeLock().lock(); // Exklusiver Zugriff! Alle anderen
                                   // warten.
        try {
            this.daten = neuerWert;
        } finally {
            rwLock.writeLock().unlock();
        }
    }
}
```

In lese-intensiven Anwendungen sorgt das ReadWriteLock für einen gigantischen Performance-Schub.

18.5 Gezieltes Wecken: Condition-Objekte

Erinnern Sie sich an das Producer-Consumer-Problem aus Kapitel 17? Wir hatten das Problem, dass notifyAll() alle wartenden Threads aufweckt. Wenn der Puffer voll ist und ein Consumer etwas herausnimmt, ruft er notifyAll() auf. Das weckt nicht nur die Producer auf (die wir eigentlich wecken wollen, damit sie nachlegen), sondern auch alle anderen Consumer (die sofort wieder einschlafen, weil der Puffer nach dem ersten Consumer schon wieder leer sein könnte). Das ist verschwendete CPU-Zeit.

Wir bräuchten eigentlich zwei getrennte Warteräume: Einen für die Producer ("Warte auf freien Platz") und einen für die Consumer ("Warte auf neue Daten").

Mit synchronized ist das unmöglich, da jedes Objekt nur exakt einen Monitor-Warteraum hat. Mit einem ReentrantLock können wir jedoch beliebig viele Condition-Objekte (Bedingungsvariablen) erzeugen!

Anstatt wait() rufen wir await() auf dem jeweiligen Condition-Objekt auf. Anstatt notify() rufen wir signal() (oder signalAll()) auf.

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BessererPuffer {
```

```
    private final Lock schloss = new ReentrantLock();

    // Zwei getrennte Warteräume erzeugen!
    private final Condition nichtVoll = schloss.newCondition();
    private final Condition nichtLeer = schloss.newCondition();

    private String daten;
    private boolean istLeer = true;

    public void schreiben(String wert) throws InterruptedException {
        schloss.lock();
        try {
            while (!istLeer) {
                nichtVoll.await(); // Producer wartet im Warteraum
                                   // "nichtVoll"
            }
            daten = wert;
            istLeer = false;

            // GEZIELTES Wecken: Wir wecken explizit nur die Consumer!
            nichtLeer.signal();
        } finally {
            schloss.unlock();
        }
    }

    public String lesen() throws InterruptedException {
        schloss.lock();
        try {
            while (istLeer) {
                nichtLeer.await(); // Consumer wartet im Warteraum
                                   // "nichtLeer"
            }
            String ergebnis = daten;
            istLeer = true;

            // GEZIELTES Wecken: Wir wecken explizit nur die Producer!
            nichtVoll.signal();

            return ergebnis;
        } finally {
            schloss.unlock();
        }
    }
}
```

Dieser Ansatz ist massiv effizienter, eleganter und verhindert das gefürchtete "Lost Wakeup"-Problem nahezu komplett, weshalb in modernem Java-Code (wie den eingebauten Collections) Condition-Objekte dem alten wait() und notify() vorgezogen werden.

18.6 Mehr als nur ein Türsteher: Der Aspekt der Sichtbarkeit

Wenn wir über Sperren (Locks) sprechen, haben wir meistens nur ein einziges Ziel vor Augen: Mutual Exclusion (wechselseitiger Ausschluss). Wir wollen verhindern, dass

zwei Threads gleichzeitig denselben kritischen Code-Bereich betreten und sich gegenseitig die Daten überschreiben.

Doch wir haben in den vorherigen Kapiteln gelernt, dass Datensicherheit in Java immer aus zwei Säulen besteht: dem Schutz vor zeitgleichem Zugriff und der garantierten Sichtbarkeit (Visibility) von Änderungen über CPU-Cache-Grenzen hinweg. Das synchronized-Schlüsselwort übernimmt bekanntermaßen freundlicherweise beide Aufgaben völlig automatisch. Wie aber verhält es sich mit den expliziten Lock-Klassen (wie dem ReentrantLock) aus dem java.util.concurrent.locks-Paket?

Die gute Nachricht lautet: Explizite Locks bieten exakt dieselben Speichersichtbarkeits-Garantien wie synchronized.

Das Java Memory Model (JMM) ist hier sehr streng und definiert für die Methoden des Lock-Interfaces eine glasklare sogenannte Happens-Before-Beziehung. Das bedeutet konkret:

- Beim Entsperren (unlock()): Wenn ein Thread seinen kritischen Bereich verlässt und lock.unlock() aufruft, zwingt das JMM die Hardware dazu, alle von diesem Thread gemachten Änderungen aus dem lokalen CPU-Cache in den gemeinsamen Hauptspeicher (RAM) zurückzuschreiben (zu "flushen"). Nichts bleibt im Verborgenen.
- Beim Sperren (lock()): Wenn kurz darauf ein anderer Thread für exakt dieselbe Sperre erfolgreich lock.lock() aufruft, wird sein eigener lokaler CPU-Cache invalidiert (als ungültig markiert). Er ist nun gezwungen, die Variablen frisch aus dem Hauptspeicher zu lesen.

Das Postboten-Prinzip

Man kann sich das Prinzip wie eine formelle Nachrichtenübergabe vorstellen: Der Aufruf von unlock() durch Thread A verpackt alle bisherigen Änderungen in einen Brief und wirft ihn in den Briefkasten. Der Aufruf von lock() durch Thread B leert diesen Briefkasten, bevor B mit der eigenen Arbeit beginnt. B sieht dadurch zwingend alles, was A vor dem unlock() getan hat.

Die goldene Regel bleibt bestehen

Dieser unsichtbare Datenabgleich funktioniert jedoch – genau wie bei den Monitoren von synchronized – nur unter einer unabdingbaren Voraussetzung: Beide Threads müssen exakt dasselbe Lock-Objekt verwenden! Wenn Thread A seine Daten ändert

und lock1.unlock() aufruft, Thread B aber beim Lesen seiner Daten lock2.lock() verwendet, gibt es keine Happens-Before-Beziehung zwischen den beiden. Die Threads benutzen unterschiedliche "Briefkästen", und Thread B liest möglicherweise völlig veraltete Daten aus seinem eigenen CPU-Cache. Wer Daten über Thread-Grenzen hinweg teilen will, muss sich zwingend auf denselben Türsteher einigen.

19 Thread-Choreografie – Barrieren, Zähler und Semaphoren

In den vergangenen Kapiteln haben wir gelernt, wie wir geteilte Datenstrukturen vor dem gleichzeitigen Zugriff schützen (synchronized, ReentrantLock) und wie wir Threads über Zustandsänderungen informieren (wait, notify, Condition). Das alles war oft sehr kleinteilig und erforderte viel Aufmerksamkeit (Stichwort: Lost Wakeup oder verpasste unlock()-Aufrufe).

In der Praxis haben wir jedoch oft Architektur-Anforderungen, die auf einer höheren Ebene liegen:

- "Der Haupt-Thread darf erst weiterarbeiten, wenn alle vier Hintergrund-Dienste erfolgreich gestartet sind."
- "Zehn Threads berechnen Teilergebnisse. Sie sollen sich an einem Punkt treffen, die Ergebnisse zusammenfügen und erst dann gemeinsam in die nächste Berechnungsrunde starten."
- "Wir haben nur 5 Datenbankverbindungen. Es dürfen maximal 5 Threads gleichzeitig auf die Datenbank zugreifen, der Rest muss warten."

Für genau diese "Choreografie" bietet das java.util.concurrent-Paket fertige, hochoptimierte Werkzeuge, die sogenannten Synchronizers. Sie ersparen uns das fehleranfällige Basteln mit eigenen Locks und Schleifen.

19.1 Der CountDownLatch: Ein Zähler, zwei Strategien

Der CountDownLatch (zu Deutsch etwa "Rückwärtszählende Sperre") aus dem Paket java.util.concurrent ist eines der simpelsten, aber gleichzeitig nützlichsten Werkzeuge zur Thread-Koordination. Im Kern ist er nichts anderes als ein Zähler, der bei seiner Erstellung auf einen bestimmten Startwert gesetzt wird.

Jeder Thread kann die Methode countDown() aufrufen, um den Zähler um eins zu verringern. Gleichzeitig können Threads die Methode await() aufrufen. Wer await() ruft, wird gnadenlos blockiert, bis der Zähler den magischen Wert 0 erreicht hat. Sobald die Null steht, öffnet sich die Schranke dauerhaft und alle wartenden Threads dürfen passieren.

Das Geniale am CountDownLatch ist, dass er in der Praxis für zwei völlig gegensätzliche Choreografien eingesetzt wird:

19.1.1 Szenario 1: Der Startschuss (N Threads warten auf ein 1 Signal)

In diesem Szenario wollen Sie verhindern, dass Ihre Arbeits-Threads zu früh loslegen. Stellen Sie sich einen 100-Meter-Sprint vor: Acht Läufer stehen in den Startblöcken, aber niemand darf loslaufen, bevor der Schiedsrichter die Startpistole abfeuert.

Hier initialisieren Sie den Latch mit dem Wert 1.

- Die N Worker-Threads (die Läufer): Werden gestartet, erledigen vielleicht noch winzige Vorbereitungen und rufen dann sofort latch.await() auf. Sie frieren ein und warten auf das Signal.
- Der Haupt-Thread (der Schiedsrichter): Erledigt in Ruhe alle nötigen Initialisierungen im System. Wenn alles bereit ist, ruft er ein einziges Mal latch.countDown() auf.
- Das Ergebnis: Der Zähler springt von 1 auf 0. Die Schranke öffnet sich und alle acht Läufer schießen im exakt selben Bruchteil einer Sekunde los. Dies ist perfekt für Lasttests, bei denen Sie eine absolute Gleichzeitigkeit von Anfragen erzwingen wollen.

```
import java.util.concurrent.CountDownLatch;

public class CountdownLatchSzenario1 {

    public static void main(String[] args) throws Exception {
        CountDownLatch startSchuss = new CountDownLatch(1);

        // N Threads starten und warten an der Startlinie
        for (int i = 0; i < 5; i++) {
            new Thread(() -> {
                try {
                    startSchuss.await(); // Warte auf das "Los!"
                    System.out.println("Thread rennt los!");
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }).start();
        }

        System.out.println("Schiedsrichter macht sich bereit...");
        Thread.sleep(2000);
        System.out.println("BAM! Los!");
        startSchuss.countDown(); // Zähler wird 0, alle 5 Threads
                                 // starten gleichzeitig!

    }
}
```

19.1.2 Szenario 2: Die Ziellinie (1 Thread wartet auf N Signale)

Das ist das genaue Spiegelbild zu Szenario 1 und der häufigere Anwendungsfall im Alltag. Hier haben Sie eine große Aufgabe in viele kleine Teilaufgaben zerlegt. Der Haupt-Thread kann erst weitermachen, wenn alle Teilaufgaben abgeschlossen sind.

Denken Sie an einen Reiseleiter (Haupt-Thread), der mit einer Gruppe von 5 Touristen (Worker-Threads) an einer Raststätte hält. Der Reiseleiter sagt: "Jeder macht, was er will, aber der Bus fährt erst weiter, wenn alle 5 wieder da sind."

Hier initialisieren Sie den Latch mit dem Wert N (in unserem Fall 5).

- Der Haupt-Thread (der Reiseleiter): Schickt die 5 Threads los und ruft dann sofort selbst latch.await() auf. Er legt sich schlafen, bis der Zähler auf 0 fällt.
- Die N Worker-Threads (die Touristen): Jeder macht seine Arbeit unabhängig von den anderen. Sobald ein Thread fertig ist, ruft er (am besten in einem finally-Block) latch.countDown() auf und beendet sich.
- Das Ergebnis: Der Zähler tickt langsam herunter (5, 4, 3, 2, 1...). Sobald der letzte Thread countDown() ruft, erreicht der Latch die 0. Der Reiseleiter-Thread wacht auf und der Bus fährt weiter (das Programm setzt seine Ausführung fort).

```
import java.util.concurrent.CountDownLatch;

public class CountdownLatchSzenario2 {

    public static void main(String[] args) throws Exception {
        int anzahlAufgaben = 5;
        CountDownLatch alleFertig = new CountDownLatch(
                anzahlAufgaben);

        // 5 Threads erledigen einen Teil der Arbeit
        for (int i = 0; i < anzahlAufgaben; i++) {
            new Thread(() -> {
                try {
                    System.out.println("Erledige Teilaufgabe...");
                    Thread.sleep((long) (Math.random() * 1000));
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                } finally {
                    alleFertig.countDown(); // "Ich bin fertig!"
                                            // (Zähler - 1)
                }
            }).start();
        }

        System.out.println(
                "Haupt-Thread wartet auf alle Ergebnisse...");
        alleFertig.await(); // Blockiert, bis 5x countDown() gerufen
                            // wurde
        System.out.println("Alle sind fertig, es geht weiter!");
    }
}
```

Eine wichtige Einschränkung: Der Name "Latch" (Schnappschloss) ist Programm. Einmal auf Null heruntergezählt, bleibt das Schloss für immer offen. Man kann einen CountDownLatch nicht zurücksetzen oder neu aufladen. Wer eine wiederverwendbare Barriere braucht, muss zum CyclicBarrier greifen

19.2 Der Treffpunkt: CyclicBarrier

Oft wird die CyclicBarrier (Zyklische Barriere) mit dem CountDownLatch verwechselt, doch sie verfolgt ein völlig anderes Paradigma.

Stellen Sie sich eine Wandergruppe vor: Sie vereinbaren, dass alle am ersten Aussichtspunkt aufeinander warten. Die Schnellen müssen dort warten, bis auch der Langsamste angekommen ist. Erst wenn alle da sind, geht die gesamte Gruppe gemeinsam weiter zum nächsten Aussichtspunkt.

Die Unterschiede zum Latch:

- Beim Latch (zumindest in Szenario 2) wartet ein Thread auf die Arbeit anderer. Bei der Barriere warten die Threads aufeinander.
- Ein Latch ist nach einmaliger Nutzung verbraucht. Eine CyclicBarrier ist zyklisch: Sobald die Schranke gefallen ist und alle Threads weitergelaufen sind, schließt sie sich wieder für die nächste Runde.
- Sie können der Barriere ein Runnable (eine Aktion) übergeben, das exakt einmal ausgeführt wird, sobald der letzte Thread ankommt, aber bevor die Schranke für alle geöffnet wird (perfekt zum Zusammenfügen von Teilergebnissen!).

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class BarrierBeispiel {
    public static void main(String[] args) {
        // Barriere für 3 Threads. Wenn alle da sind, wird das
        // Runnable ausgeführt.
        CyclicBarrier barriere = new CyclicBarrier(3, () -> {
            System.out.println(
                    "--- BARRIERE ERREICHT: Alle 3 sind da! " +
                    "Ergebnisse werden gemergt. ---");
        });

        for (int i = 1; i <= 3; i++) {
            final int threadNum = i;
            new Thread(() -> {
                try {
                    System.out.println("Thread " + threadNum
                            + " berechnet Phase 1...");
                    Thread.sleep((long) (Math.random() * 2000));
                    System.out.println("Thread " + threadNum
```

```
                        + " wartet an der Barriere.");

                // Thread blockiert hier und wartet auf die
                // anderen!
                barriere.await();

                System.out.println("Thread " + threadNum
                        + " berechnet Phase 2...");
            } catch (InterruptedException
                    | BrokenBarrierException e) {
                e.printStackTrace();
            }
        }).start();
    }
  }
}
```

19.3 Die Meisterklasse der Koordination: Der Phaser

Wir haben nun gesehen, wie wir Threads an Barrieren warten lassen können. Der CountDownLatch ist ein hervorragendes Werkzeug, hat aber einen entscheidenden Nachteil: Er ist ein Einweg-Ticket. Einmal auf null gezählt, ist er nutzlos. Die CyclicBarrier löst dieses Problem der Wiederverwendbarkeit, zwingt uns aber in ein starres Korsett: Wir müssen bei ihrer Erstellung exakt und unveränderlich festlegen, wie viele Threads teilnehmen (z. B. genau 5).

Was aber, wenn unsere Software dynamischer ist? Was, wenn wir einen mehrstufigen Prozess haben, bei dem in Phase 1 drei Threads arbeiten, in Phase 2 plötzlich fünf Threads benötigt werden und in Phase 3 nur noch zwei Threads übrig bleiben?

Für genau solche hochkomplexen, dynamischen Choreografien wurde in Java 7 der Phaser eingeführt. Er ist quasi das Schweizer Taschenmesser der Synchronisation: Er kann alles, was ein Latch oder eine Barrier kann – und noch viel mehr.

19.3.1 Die Metapher: Die mehrstufige Busrundreise

Stellen Sie sich eine mehrtägige Busrundreise vor. Die Reise ist in verschiedene Etappen (Phasen) unterteilt: Tag 1, Tag 2, Tag 3. Das Besondere an dieser Reise ist die absolute Flexibilität: An jedem Hotel (dem Ende einer Phase) können neue Touristen zusteigen oder alte aussteigen. Der Busfahrer (der Phaser) zählt einfach, wie viele Touristen aktuell angemeldet sind. Er fährt erst zur nächsten Etappe weiter, wenn alle aktuell angemeldeten Touristen im Bus sitzen.

Der Phaser teilt den Programmablauf in Phasen (beginnend bei Phase 0). Die beteiligten Threads werden als Parteien (Parties) bezeichnet.

Das Vokabular des Phasers ist enorm mächtig:

- register(): Ein neuer Thread meldet sich an. ("Ich steige ab jetzt in die Rundreise ein!") Der Zähler der erwarteten Parteien erhöht sich um 1.
- arriveAndAwaitAdvance(): Ein Thread ist mit seiner Arbeit für die aktuelle Phase fertig und blockiert, bis alle anderen angemeldeten Threads ebenfalls fertig sind. ("Ich sitze im Bus und warte, bis wir zur nächsten Etappe losfahren.")
- arriveAndDeregister(): Ein Thread ist fertig, meldet sich aber gleichzeitig für alle zukünftigen Phasen ab. ("Ich bin am Hotel angekommen, aber morgen fahre ich nach Hause.") Der Zähler der erwarteten Parteien sinkt für die nächste Phase um 1.
- arrive(): Ein Thread meldet Vollzug für die aktuelle Phase, wartet aber nicht auf die anderen, sondern rechnet direkt weiter. (Das entspricht dem countDown() beim Latch).

19.3.2 Der Phaser in Aktion

Schauen wir uns an, wie elegant der Phaser dynamische Thread-Gruppen über mehrere Phasen hinweg orchestriert:

```
import java.util.concurrent.Phaser;

public class PhaserBeispiel {

    public static void main(String[] args) {
        // Wir starten einen Phaser mit 0 angemeldeten Parteien
        Phaser phaser = new Phaser(0);

        // Wir simulieren 3 Arbeits-Threads
        for (int i = 1; i <= 3; i++) {
            // Jeder Thread meldet sich BEVOR er startet beim Phaser
            // an
            phaser.register();

            int threadNummer = i;
            new Thread(() -> {
                System.out.println("Thread " + threadNummer
                        + " startet Phase 0");

                if (threadNummer == 3) {
                    // Thread 3 hat nach Phase 0 keine Lust mehr und
                    // steigt aus
                    System.out.println(
                            "Thread 3 beendet Phase 0 und steigt aus.");
                    phaser.arriveAndDeregister();
                    return; // Thread beendet sich
                } else {
                    // Threads 1 und 2 warten aufeinander, um in Phase
                    // 1 überzugehen
                    phaser.arriveAndAwaitAdvance();
                }

                // --- Hier beginnt Phase 1 ---
```

```
                // Nur noch Thread 1 und 2 sind hier unterwegs
                System.out.println("Thread " + threadNummer
                        + " arbeitet in Phase 1");
                phaser.arriveAndAwaitAdvance(); // Warten auf das Ende
                                                // von Phase 1

            }).start();

        }
    }
}
```

Was passiert im Hintergrund?

1. Der Phaser startet in Phase 0 mit 3 registrierten Parteien.
2. Alle drei Threads erledigen ihre Arbeit für Phase 0.
3. Thread 3 ruft arriveAndDeregister() auf. Er signalisiert damit: "Ich bin für Phase 0 fertig, erwarte mich nicht für Phase 1!" Die Anzahl der erwarteten Parteien für die nächste Runde sinkt von 3 auf 2.
4. Sobald Thread 1 und 2 arriveAndAwaitAdvance() gerufen haben, ist Phase 0 offiziell abgeschlossen. Der Phaser erhöht intern seinen Phasen-Zähler auf 1 und lässt Thread 1 und 2 weiterlaufen.
5. In Phase 1 wartet der Phaser jetzt völlig automatisch nur noch auf 2 Parteien.

Wann nutze ich was?

Trotz seiner Macht sollten Sie den Phaser nicht blind überall einsetzen. Die Regel in der Praxis lautet:

- Brauchen Sie einen simplen, einmaligen Startschuss oder eine Ziellinie? Nehmen Sie den CountDownLatch.
- Haben Sie eine feste, unveränderliche Gruppe von Threads, die sich immer wieder an einem Punkt treffen muss? Nehmen Sie die CyclicBarrier.
- Ist Ihre Thread-Anzahl dynamisch, treten Threads während der Laufzeit bei oder treten aus, oder haben Sie einen komplexen Algorithmus, der in klar definierten Schritten/Generationen abläuft (z. B. genetische Algorithmen oder Simulationen)? Dann ist der Phaser Ihr bester Freund.

19.4 Der Türsteher: Semaphore

Manchmal geht es nicht darum, auf jemanden zu warten, sondern Ressourcen zu kontingentieren. Ein ReentrantLock ist binär: Entweder ist die Tür zu oder auf. Es lässt immer genau einen Thread herein.

Was ist aber, wenn unsere Hardware drei parallele Downloads verkraftet, aber nicht zehn? Hier kommt das Semaphore ins Spiel. Es funktioniert wie ein Türsteher vor einem exklusiven Club, in den maximal 3 Personen gleichzeitig dürfen.

Die Mechanik:

1. Sie erzeugen ein Semaphor mit einer bestimmten Anzahl an Genehmigungen (Permits), z.B. 3.
2. Bevor ein Thread die Ressource nutzt, muss er acquire() aufrufen. Gibt es freie Permits, nimmt er sich eins und läuft weiter. Sind alle vergeben, blockiert er und wartet in der Schlange.
3. Ist er fertig, ruft er release() auf und gibt sein Permit zurück an den Türsteher. Der nächste wartende Thread darf eintreten.

```
import java.util.concurrent.Semaphore;

public class SemaphoreBeispiel {
    public static void main(String[] args) {
        // Ein Club mit maximal 2 Plätzen
        Semaphore tuersteher = new Semaphore(2);

        // Wir lassen 5 Gäste (Threads) gleichzeitig loslaufen
        for (int i = 1; i <= 5; i++) {
            final int gastId = i;
            new Thread(() -> {
                try {
                    System.out.println(
                            "Gast " + gastId + " stellt sich an...");
                    tuersteher.acquire(); // Versuche, einen Platz zu
                                          // bekommen

                    System.out.println(
                            "--> Gast " + gastId + " ist im Club!");
                    Thread.sleep(2000); // Feiert für 2 Sekunden

                } catch (InterruptedException e) {
                } finally {
                    System.out.println("<-- Gast " + gastId
                            + " verlässt den Club.");
                    tuersteher.release(); // Platz wieder freigeben!
                }
            }).start();
        }
    }
}
```

Tipp aus der Praxis: Auch ein Semaphore kann man "fair" machen (new Semaphore(2, true)), sodass Threads strikt in der Reihenfolge ihres Eintreffens bedient werden.

19.5 Der direkte Tausch: Exchanger

Für einen sehr spezifischen Randfall gibt es noch den Exchanger. Er ist ein Rendezvous-Punkt für exakt zwei Threads.

Stellen Sie sich zwei Spione vor, die sich nachts auf einer Brücke treffen, um Aktenkoffer auszutauschen. Derjenige, der zuerst auf der Brücke ankommt, muss im Schatten warten. Erst wenn der zweite Spion eintrifft, werden die Koffer im exakt selben Moment getauscht, und beide gehen ihrer Wege.

In der Programmierung wird dies oft bei der Puffer-Rotation verwendet: Ein Thread füllt kontinuierlich einen Daten-Puffer (Erzeuger), ein anderer leert ihn (Verbraucher). Wenn der Puffer des Erzeugers voll und der des Verbrauchers leer ist, treffen sie sich am Exchanger und tauschen einfach die Puffer-Referenzen aus.

Beispiel: Der Datentausch

In diesem Beispiel tauschen zwei Threads einfache Textnachrichten aus. Um den "Warte-Effekt" (Rendezvous) zu demonstrieren, lassen wir den zweiten Thread absichtlich etwas später am Treffpunkt ankommen.

```
import java.util.concurrent.Exchanger;

public class ExchangerBeispiel {
    public static void main(String[] args) {
        // Der Exchanger wird typisiert. Hier tauschen wir Strings
        // aus.
        Exchanger<String> treffpunkt = new Exchanger<>();

        // Thread 1: Der schnelle Spion
        Thread spion1 = new Thread(() -> {
            String meinKoffer = "Geheime Dokumente von Spion 1";
            System.out.println(
                    "Spion 1: Bin am Treffpunkt. Warte auf Partner...");
            try {
                // exchange() blockiert, bis der andere Thread
                // ebenfalls exchange() aufruft!
                String erhaltenerKoffer = treffpunkt
                        .exchange(meinKoffer);
                System.out.println(
                        "Spion 1: Tausch erfolgreich! Habe erhalten: ["
                                + erhaltenerKoffer + "]");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        // Thread 2: Der langsame Spion
        Thread spion2 = new Thread(() -> {
            String meinKoffer = "Geldkoffer von Spion 2";
            try {
                // Simuliert eine Verspätung von 2 Sekunden
                Thread.sleep(2000);
                System.out.println(
```

```
                    "Spion 2: Puh, bin da. Lasst uns tauschen!");

                // Hier treffen sie aufeinander und der Tausch findet
                // statt
                String erhaltenerKoffer = treffpunkt
                        .exchange(meinKoffer);
                System.out.println(
                        "Spion 2: Tausch erfolgreich! Habe erhalten: ["
                                + erhaltenerKoffer + "]");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        spion1.start();
        spion2.start();
    }
}
```

Was passiert bei der Ausführung?

1. Spion 1 startet, ruft exchange() auf und blockiert sofort. Er wartet an der Brücke.
2. Zwei Sekunden lang passiert nichts.
3. Spion 2 wacht aus seinem sleep() auf, ruft ebenfalls exchange() auf.
4. In diesem Moment gibt der Exchanger dem Spion 1 den String von Spion 2 als Rückgabewert zurück – und umgekehrt. Beide Threads laufen anschließend mit den Daten des jeweils anderen weiter.

20 Thread-sichere Datenstrukturen – Concurrent Collections

In den vergangenen Kapiteln haben wir viel Zeit damit verbracht, einfache Variablen (int saldo) mit synchronized oder ReentrantLock vor dem gleichzeitigen Zugriff zu schützen.

In der Realität arbeiten wir jedoch selten mit einzelnen Integern. Meistens verwalten wir komplexe Datensammlungen: Listen von Benutzern, Maps von Konfigurationseinstellungen oder Warteschlangen von Aufgaben. Wenn wir hier die Standard-Klassen des Java Collections Framework (ArrayList, HashMap, HashSet) in einem Multi-Threading-Umfeld einsetzen, steuern wir unweigerlich auf eine Katastrophe zu.

Dieses Kapitel erklärt, warum das so ist und welche hochoptimierten Alternativen Java uns bietet.

20.1 Die Katastrophe: Warum ArrayList und HashMap explodieren

Normale Collections in Java sind nicht thread-sicher. Sie wurden auf maximale Performance im sequenziellen (Single-Threaded) Betrieb optimiert. Jeglicher Overhead für Synchronisation wurde weggelassen.

Was passiert, wenn zwei Threads gleichzeitig liste.add(element) auf einer normalen ArrayList aufrufen? Eine ArrayList basiert intern auf einem einfachen Array und einem Zähler (size). Das Hinzufügen besteht aus mehreren Schritten:

1. Prüfen, ob das interne Array noch Platz hat (ggf. ein größeres Array anlegen und Daten kopieren).
2. Das Element an den Index size schreiben.
3. Den Zähler size um 1 erhöhen.

Durch Interleaving (Kapitel 14) kann Folgendes passieren: Thread A und Thread B sehen beide size = 5. Beide schreiben ihr Element an Index 5. Das Element von Thread A wird von Thread B überschrieben (Lost Update). Danach erhöhen beide den Zähler. Die Liste meldet nun die Größe 7, obwohl auf Position 6 gar nichts steht (null). Die interne Datenstruktur ist korrupt.

Noch schlimmer ist es bei der HashMap. Wenn hier zwei Threads gleichzeitig neue Einträge hinzufügen und die Map zufällig genau in diesem Moment ihre internen Buckets vergrößern muss (Rehashing), können sich die internen verketteten Listen zu

einer Endlosschleife verheddern. Ruft dann ein Thread map.get(key) auf, friert die CPU bei 100 % Auslastung für immer ein!

Zusätzlich werfen diese Standard-Collections oft eine ConcurrentModificationException, wenn ein Thread über die Liste iteriert (z.B. mit einer for-each-Schleife) und ein anderer Thread in genau diesem Moment ein Element hinzufügt oder löscht.

20.2 Der traditionelle Schutzschild: Die synchronisierten Wrapper der Klasse Collections

Anstatt für alle Datenstrukturen eine eigene, Thread-sichere Variante zu programmieren, wählten die Java-Entwickler einen pragmatischen Weg: das Decorator-Pattern (Dekorierer-Muster).

Die Utility-Klasse java.util.Collections bietet eine Reihe von statischen Fabrikmethoden an. Diese Methoden nehmen eine unsichere Standard-Collection entgegen und stülpen einen "Schutzschild" (einen Wrapper) darüber. Dieser Wrapper implementiert exakt dasselbe Interface wie die Original-Collection, leitet aber jeden einzelnen Methodenaufruf durch einen synchronized-Block.

20.2.1 Kompletter Überblick: Die Synchronisations-Methoden der Klasse Collections

Die Klasse Collections stellt für jeden gängigen Collection-Typ in Java genau eine passende Wrapper-Methode zur Verfügung. Hier ist die vollständige Übersicht:

- Für einfache Listen und Sammlungen:
  - Collections.synchronizedCollection(Collection<T> c): Der allgemeinste Wrapper, gibt eine Thread-sichere Collection zurück.
  - Collections.synchronizedList(List<T> list): Der Klassiker für ArrayList oder LinkedList. Erhält die indexbasierten Zugriffe (get(int index)).
  - Collections.synchronizedSet(Set<T> s): Der Schutz für einfache Mengen wie das HashSet. Verhindert Duplikate Thread-sicher.
- Für sortierte und navigierbare Mengen:
  - Collections.synchronizedSortedSet(SortedSet<T> s): Für Sets, die eine feste Sortierung garantieren (wie das TreeSet).
  - Collections.synchronizedNavigableSet(NavigableSet<T> s): Die modernere Variante für Mengen, die gezielte Suchanfragen erlauben (z. B. "Finde das nächstgrößere Element"). (Eingeführt in Java 6).
- Für Schlüssel-Wert-Paare (Maps):

- Collections.synchronizedMap(Map<K,V> m): Der Standard-Schutz für die HashMap.
- Collections.synchronizedSortedMap(SortedMap<K,V> m): Für sortierte Maps wie die TreeMap.
- Collections.synchronizedNavigableMap(NavigableMap<K,V> m): Für navigierbare Maps mit komplexen Suchfunktionen.

20.2.2 Ein typisches Anwendungsbeispiel:

```
// 1. Die unsichere Liste erstellen
List<String> unsichereListe = new ArrayList<>();

// 2. Den Schutzschild überstreifen
List<String> sichereListe = Collections
        .synchronizedList(unsichereListe);

// Ab sofort greifen alle Threads NUR noch über 'sichereListe'
// zu!
sichereListe.add("Element 1"); // Ist jetzt Thread-sicher
```

20.2.3 Die große Falle: Warum Iterieren trotzdem abstürzt

Man könnte nun meinen, mit dem Aufruf von Collections.synchronizedList() seien alle Probleme gelöst. Die Methoden add(), remove() und size() sind nun durch das Monitor-Lock des Wrapper-Objekts geschützt.

Doch es gibt eine gewaltige, oft übersehene Schwachstelle: Das Durchlaufen der Liste (Iteration).

Wenn Sie eine Schleife über die Collection schreiben (egal ob mit einem expliziten Iterator oder der bequemen for-each-Schleife), sind das unter der Haube viele einzelne Methodenaufrufe (hasNext(), dann next(), dann wieder hasNext()). Der Wrapper schützt zwar jeden dieser Aufrufe einzeln, aber er schützt nicht die Iteration als Ganzes. Wenn Thread A gerade mitten in der Schleife steckt und Thread B in diesem Moment ein Element hinzufügt oder löscht, wirft die Liste eine ConcurrentModificationException.

Die goldene Regel lautet daher: Wer über eine durch Collections.synchronized... geschützte Collection iterieren will, muss den gesamten Schleifenblock manuell auf das Wrapper-Objekt synchronisieren!

```
List<String> sichereListe = Collections
        .synchronizedList(new ArrayList<>());

// ... Threads fügen wild Daten hinzu ...

// So NICHT (führt unweigerlich zum Absturz bei
// Nebenläufigkeit):
```

```
// for (String s : sichereListe) { System.out.println(s); }

// So ist es KORREKT:
synchronized (sichereListe) {
    for (String s : sichereListe) {
        System.out.println(s);
    }
}
```

20.2.4 Fazit: Der Flaschenhals der grobgranularen Sperren

Die Collections.synchronized...-Methoden sind unglaublich einfach anzuwenden und leisten für einfache Anwendungsfälle hervorragende Dienste.

Ihre Einfachheit ist jedoch gleichzeitig ihr größter Feind in Sachen Performance. Da sich alle Methoden denselben, einzigen Schlüssel (das Monitor-Lock) teilen, kann immer nur ein einziger Thread irgendetwas mit der Collection tun. Wenn Thread A gerade das erste Element liest, ist Thread B komplett blockiert, auch wenn er nur das letzte Element löschen wollte. Bei stark beanspruchten Systemen mit vielen Threads wird diese "grobgranulare" (coarse-grained) Synchronisation schnell zu einem massiven Flaschenhals.

20.3 Maßgeschneiderte Leistung: Die Concurrent-Klassen

Wie wir im vorherigen Abschnitt gesehen haben, ist der Schutzschild der Collections.synchronized...-Methoden extrem restriktiv. Das Prinzip des „einen, gigantischen Schlosses“ (Coarse-grained Locking) führt bei vielen Threads unweigerlich zu einem gewaltigen Stau. Wenn ein Thread liest, darf kein anderer schreiben. Wenn ein Thread schreibt, darf kein anderer lesen.

Mit der Einführung des Pakets java.util.concurrent (in Java 5) hat sich die Java-Welt von diesem plumpen Ansatz verabschiedet. Anstatt unsichere Datenstrukturen nachträglich in einen engen synchronized-Panzer zu zwängen, wurden völlig neue Datenstrukturen von Grund auf für die Nebenläufigkeit entworfen. Diese Klassen verwenden hochkomplexe, smarte Algorithmen unter der Haube: Sie nutzen feingranulare Sperren (Lock Striping), verzichten dank "Compare-And-Swap" (CAS) oft komplett auf blockierende Locks (Lock-free) oder kopieren Daten clever im Hintergrund.

Das Ergebnis: Mehrere Threads können gleichzeitig in diese Datenstrukturen schreiben und daraus lesen, ohne sich gegenseitig auszubremsen.

20.3.1 Der Überblick: Die Hochleistungs-Datenstrukturen

Wenn Sie in einem modernen Java-Programm nebenläufige Daten verwalten müssen, sollten Sie die alten Wrapper vergessen und stattdessen gezielt in diesen Werkzeugkasten greifen. Man kann diese Klassen grob nach ihrem Einsatzzweck unterteilen:

1. Die unangefochtene Königin der Maps

ConcurrentHashMap<K,V>: Dies ist der absolute Star unter den Concurrent-Klassen und der perfekte Ersatz für eine Collections.synchronizedMap(). Anstatt die gesamte Map zu sperren, unterteilt sie intern die Daten in viele kleine Segmente (Lock Striping). Lesezugriffe blockieren hier überhaupt nicht! Selbst Schreibzugriffe blockieren sich nur dann gegenseitig, wenn zwei Threads zufällig in exakt denselben internen "Daten-Eimer" (Bucket) schreiben wollen.

2. Sortierte Datenstrukturen (SkipLists)

Herkömmliche Bäume (wie die TreeMap oder das TreeSet) lassen sich extrem schwer ohne Komplett-Sperren parallelisieren, da beim Einfügen oft der halbe Baum neu ausbalanciert werden muss. Java nutzt stattdessen eine brillante Alternative namens SkipList (eine Art mehrspurige, verkettete Liste):

- ConcurrentSkipListMap<K,V>: Der Thread-sichere, hochperformante Ersatz für die TreeMap. Sie hält ihre Schlüssel-Wert-Paare stets sortiert und arbeitet intern nahezu vollständig lock-free.
- ConcurrentSkipListSet<T>: Das Gegenstück für Mengen. Der perfekte, sortierte Ersatz für ein Collections.synchronizedSortedSet().

3. Lock-Free Warteschlangen (Queues)

Wenn Sie eine Warteschlange benötigen, die von hunderten Threads gleichzeitig ohne Blockaden bombardiert werden kann, bietet Java asynchrone, auf CAS (Compare-And-Swap) basierende Listen an:

- ConcurrentLinkedQueue<T>: Eine unbegrenzte, absolut Thread-sichere Warteschlange nach dem FIFO-Prinzip (First-In-First-Out). Sie nutzt keine blockierenden Sperren, sondern smarte, atomare Referenz-Aktualisierungen.
- ConcurrentLinkedDeque<T>: Die Variante als "Double Ended Queue", bei der Elemente Thread-sicher sowohl vorne als auch hinten blitzschnell angefügt und entnommen werden können.

20.3.2 Das Ende der ConcurrentModificationException

Erinnern Sie sich an die große Falle der alten Collections.synchronized...-Wrapper aus dem letzten Abschnitt? Wer über sie iterieren wollte, musste mühsam einen manuellen synchronized-Block um die Schleife ziehen, sonst drohte der Absturz.

Bei den Concurrent-Klassen gehört dieses Problem der Vergangenheit an! Alle Iteratoren dieser neuen Klassen (wie der ConcurrentHashMap oder der CopyOnWriteArrayList) sind sogenannte Weakly Consistent Iterators (schwach konsistente Iteratoren).

Das bedeutet: Sie werfen niemals eine ConcurrentModificationException. Wenn Sie eine for-each-Schleife über eine ConcurrentHashMap starten und ein anderer Thread löscht oder fügt im selben Moment ein Element hinzu, stürzt Ihr Programm nicht ab. Der Iterator garantiert Ihnen, dass er den Zustand der Map exakt zu dem Zeitpunkt widerspiegelt, als die Schleife begann. Ob er Änderungen, die während der Schleife passieren, noch mitbekommt, ist dem Iterator überlassen – aber er wird definitiv nicht mit einem Fehler abbrechen. Sie benötigen für Iterationen über Concurrent-Klassen also keine externen Sperren mehr.

20.4 Für Lesefans: CopyOnWriteArrayList

Wenn Sie eine Liste haben, die sehr oft gelesen, aber nur extrem selten verändert wird (z. B. eine Liste von angemeldeten Listenern in einem Event-System), ist die CopyOnWriteArrayList die perfekte Wahl.

Die Mechanik: Jegliche Schreiboperation (add, set, remove) erzeugt unter der Haube eine komplett neue Kopie des internen Arrays. Leseoperationen arbeiten einfach auf der aktuellen Version des Arrays, völlig ohne Sperren.

Dadurch können Iteratoren niemals eine ConcurrentModificationException werfen. Wenn Sie eine for-each-Schleife starten, iteriert diese über den Zustand des Arrays zu genau dem Zeitpunkt, als die Schleife begann. Ändert ein anderer Thread die Liste in der Zwischenzeit, iterieren Sie ungestört über die alte Kopie weiter.

Vorsicht: Nutzen Sie diese Klasse niemals, wenn Sie ständig Daten hinzufügen. Das ständige Kopieren großer Arrays würde Ihren Speicher und Ihre CPU in Sekundenbruchteilen lahmlegen.

Auch für Mengen gibt es übrigens eine entsprechende Klasse: CopyOnWriteArraySet.

20.5 Die ultimative Lösung für Erzeuger & Verbraucher: BlockingQueue

Erinnern Sie sich an Kapitel 17? Dort haben wir das Producer-Consumer-Problem mühsam mit synchronized, wait(), notifyAll() und while-Schleifen selbst programmiert, um zu verhindern, dass Threads in leere Puffer greifen oder volle Puffer überlaufen lassen.

In der Praxis macht man das heute nie mehr selbst. Man nutzt eine BlockingQueue (blockierende Warteschlange). Sie kapselt die gesamte Logik der Warteräume (Conditions und Locks) in sich.

Es gibt verschiedene Implementierungen, z.B. die ArrayBlockingQueue (feste Größe) oder die LinkedBlockingQueue (optional unbegrenzt).

Die zwei magischen Methoden sind:

- put(Element): Fügt ein Element hinzu. Wenn die Schlange voll ist, blockiert der aufrufende Thread automatisch, bis wieder Platz ist.
- take(): Holt ein Element heraus. Wenn die Schlange leer ist, blockiert der aufrufende Thread automatisch, bis neue Daten da sind.

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockingQueueBeispiel {
    public static void main(String[] args) {
        // Ein Puffer mit Platz für maximal 3 Nachrichten
        BlockingQueue<String> puffer = new ArrayBlockingQueue<>(3);

        // Der Producer-Thread
        new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    System.out.println(
                            "Producer produziert: Nachricht " + i);
                    // Blockiert automatisch, wenn der Puffer voll
                    // ist!
                    puffer.put("Nachricht " + i);
                    Thread.sleep(500); // Produziert etwas schneller
                }
            } catch (InterruptedException e) {
            }
        }).start();

        // Der Consumer-Thread
        new Thread(() -> {
            try {
                while (true) { // Läuft endlos
                    // Blockiert automatisch, wenn der Puffer leer
                    // ist!
                    String nachricht = puffer.take();
                    System.out.println(
                            "Consumer verarbeitet: " + nachricht);
```

```
                    Thread.sleep(1500); // Verarbeitet etwas langsamer
                }
            } catch (InterruptedException e) {
            }
        }).start();
    }
}
```

Wenn Sie diesen Code laufen lassen, sehen Sie, wie der Producer erst 3 Nachrichten in die Queue schiebt und dann automatisch schlafen geht, bis der langsamere Consumer das erste Element mit take() entnimmt und Platz schafft. Keine einzige Zeile synchronized oder wait in unserem eigenen Code!

20.6 Spezialisten für besondere Aufgaben: Weitere BlockingQueue-Klassen

Bisher haben wir uns auf die ArrayBlockingQueue und die LinkedBlockingQueue konzentriert. Diese beiden sind die unangefochtenen Arbeitstiere für das klassische Producer-Consumer-Pattern, wenn es einfach nur darum geht, Elemente streng nach der Reihenfolge ihres Eintreffens (First-In-First-Out, FIFO) abzuarbeiten.

Doch nicht jedes Problem in der Softwareentwicklung lässt sich mit einer einfachen "Wer zuerst kommt, mahlt zuerst"-Schlange lösen. Für speziellere Anforderungen bietet das java.util.concurrent-Paket vier weitere, hochinteressante BlockingQueue-Implementierungen an:

20.6.1 Der VIP-Bereich: PriorityBlockingQueue

Stellen Sie sich die Notaufnahme eines Krankenhauses vor. Auch hier gibt es eine Warteschlange, aber sie funktioniert nicht nach dem FIFO-Prinzip. Ein Patient mit einem Herzinfarkt wird sofort behandelt, auch wenn jemand mit einem verstauchten Knöchel schon zwei Stunden länger wartet.

Genau so arbeitet die PriorityBlockingQueue. Sie ignoriert die Reihenfolge des Einfügens komplett. Stattdessen sortiert sie ihre Elemente intern nach ihrer Priorität. Damit das funktioniert, müssen die Objekte, die Sie in diese Queue legen, das Interface Comparable implementieren (oder Sie übergeben der Queue bei der Erstellung einen eigenen Comparator). Wenn ein Consumer take() aufruft, bekommt er immer automatisch das Element mit der höchsten Priorität (bzw. dem kleinsten Wert gemäß der Sortierung) ausgehändigt. Hinweis: Diese Queue ist im Gegensatz zur ArrayBlockingQueue "unbounded" (theoretisch unbegrenzt). Die Methode put() wird hier also niemals blockieren, weil die Schlange voll ist – höchstens geht Ihnen der Arbeitsspeicher aus!

20.6.2 Das Zeitschloss: DelayQueue

Manchmal wollen wir Aufgaben zwar sofort in eine Warteschlange legen, sie sollen aber erst in der Zukunft bearbeitet werden dürfen. Denken Sie an ein System, das Erinnerungs-E-Mails verschicken soll, oder an einen Cache, bei dem alte Einträge nach 10 Minuten ablaufen sollen.

Hier schlägt die Stunde der DelayQueue. Sie ist quasi eine Warteschlange mit eingebauten Zeitschlössern. Elemente, die Sie hier einfügen, müssen das Interface Delayed implementieren. Sie geben jedem Element bei der Erstellung mit auf den Weg, wie lange es "gesperrt" bleiben soll. Der Clou: Wenn ein Consumer take() aufruft, schaut die Queue nach, ob die Verzögerungszeit (Delay) bei irgendeinem Element bereits abgelaufen ist. Ist das nicht der Fall, blockiert der Consumer so lange, bis das erste Zeitschloss aufspringt. Die DelayQueue ist intern übrigens eine PriorityQueue, die die Elemente so sortiert, dass das Element, dessen Zeit als Erstes abläuft, ganz vorne steht.

20.6.3 Die direkte Übergabe (ohne Puffer): SynchronousQueue

Was passiert, wenn man eine Warteschlange mit der Kapazität von exakt Null baut? Man erhält die faszinierende SynchronousQueue.

Diese Queue hat keinen internen Speicher. Sie speichert keine Elemente zwischen, nicht mal ein einziges. Sie ist eher ein Treffpunkt für direkte Übergaben – wie bei einem Staffellauf, wo der Läufer den Stab direkt in die Hand des nächsten Läufers drücken muss. Wenn ein Producer put() aufruft, blockiert er sofort auf unbestimmte Zeit – und zwar so lange, bis in exakt diesem Moment ein Consumer auf der anderen Seite take() aufruft und das Element direkt entgegennimmt. Umgekehrt blockiert ein Consumer bei take(), bis ein Producer auftaucht, der ihm etwas in die Hand drückt. Die SynchronousQueue wird extrem gerne in Thread-Pools (Executors.newCachedThreadPool()) eingesetzt, bei denen ankommende Aufgaben nicht erst in einer Liste gepuffert, sondern sofort einem freien Thread zugewiesen werden sollen.

20.6.4 Der garantierte Empfang: LinkedTransferQueue

Eingeführt mit Java 7, ist die LinkedTransferQueue eine Art "Best-of" der bisherigen Queues. Sie kombiniert die Eigenschaften einer normalen Queue (sie kann Elemente puffern) mit der direkten Übergabe der SynchronousQueue.

Ihre Superkraft ist die Methode transfer(E e).

- Bei einem normalen put() wirft der Producer sein Element einfach in die Warteschlange, dreht sich um und geht seiner Wege (er vertraut darauf, dass es irgendwann bearbeitet wird).
- Bei transfer() hingegen legt der Producer das Element in die Schlange und blockiert dann so lange, bis ein Consumer dieses spezifische Element mit take() entnommen hat. Das ist extrem nützlich für Systeme, bei denen der Producer eine Garantie oder eine Art "Empfangsbestätigung" braucht, dass seine Nachricht nicht nur im Postausgang liegt, sondern tatsächlich beim Empfänger angekommen ist. Sie bietet außerdem Methoden wie tryTransfer(), um zu sagen: "Wenn gerade ein Consumer wartet, gib es ihm sofort. Wenn nicht, behalte es erst gar nicht, ich mache es selbst."

20.7 Fazit: Das richtige Werkzeug für geteilte Datenstrukturen

Wenn es in der nebenläufigen Programmierung darum geht, Daten in Listen, Mengen oder Schlüssel-Wert-Paaren über Thread-Grenzen hinweg zu verwalten, stehen Sie vor einer wichtigen architektonischen Entscheidung. Sie könnten versuchen, Standard-Collections wie die ArrayList oder HashMap mit eigenen synchronized-Blöcken abzusichern. Oder Sie greifen auf die älteren Wrapper-Methoden wie Collections.synchronizedMap() zurück.

Die wichtigste Lektion dieses Kapitels lautet jedoch: Verzichten Sie im modernen Java auf beides!

Eine performante, blockierungsfreie und absolut fehlerresistente Datenstruktur von Grund auf neu abzusichern, ist extrem fehleranfällig und führt schnell zu versteckten Race Conditions. Die alten Collections.synchronized...-Wrapper wiederum sind zwar sicher, drosseln aber durch ihr grobgranulares Locking (ein einziges Schloss für die gesamte Datenstruktur) die Performance Ihrer Anwendung massiv. Wenn ein Thread liest, darf kein anderer schreiben – ein klassischer Flaschenhals.

Wann immer Sie Datenstrukturen über Thread-Grenzen hinweg teilen müssen, sollte Ihr erster und einziger Blick in das Paket java.util.concurrent fallen.

Klassen wie die ConcurrentHashMap, die CopyOnWriteArrayList oder die vielfältige Familie der BlockingQueue-Implementierungen sind das absolute Mittel der Wahl. Sie sind nicht einfach nur "irgendwie Thread-sicher", sondern wurden von ausgewiesenen Koryphäen der Nebenläufigkeit entworfen, gnadenlos getestet und auf absolute Hochleistung getrimmt. Unter der Haube nutzen diese Klassen brillante, hochkomplexe Algorithmen wie Lock-Striping (feingranulare Sperren) und Compare-

And-Swap (Lock-free-Mechanismen). Diese Techniken ermöglichen es unzähligen Threads, gleichzeitig und ohne nennenswerte Blockaden auf die Daten zuzugreifen. Es ist in der Praxis nahezu unmöglich, diese Performance und Fehlerresistenz durch eigenen Code auch nur ansatzweise zu schlagen.

21 Höchstleistung ohne Sperren – Lock-Free Programming und atomare Variablen

In fast allen bisherigen Kapiteln des zweiten Teils haben wir ein pessimistisches Weltbild vertreten: Wir gingen davon aus, dass andere Threads uns ganz sicher in die Quere kommen, wenn wir auf geteilte Daten zugreifen. Unsere Lösung war immer, die Tür abzuschließen (synchronized, ReentrantLock), unsere Arbeit zu erledigen und die Tür wieder aufzusperren.

Dieses sogenannte Locking (Sperren) hat jedoch gravierende Nachteile:

1. Performance-Verlust (Context Switches): Wenn ein Thread vor einer verschlossenen Tür steht, wird er vom Betriebssystem "schlafen" gelegt (blockiert) und später wieder aufgeweckt. Dieses Umschalten (Context Switch) kostet die CPU massiv Zeit.
2. Gefahr von Deadlocks: Wo es Sperren gibt, gibt es Verklemmungen.
3. Prioritätsinversion: Ein unwichtiger Hintergrund-Thread hält eine Sperre, ein extrem wichtiger System-Thread muss untätig warten.

Was wäre, wenn wir ein optimistisches Weltbild einnehmen? "Ich versuche einfach, meine Änderung durchzuführen. Meistens funkt mir niemand dazwischen. Falls doch, merke ich das und probiere es einfach sofort noch einmal, ohne jemals schlafen zu gehen!"

Dieses Konzept nennt man Lock-Free Programming (Sperrfreie Programmierung).

21.1 Die Hardware-Magie: Compare-And-Swap (CAS)

Lock-Free Programming wäre in Software allein unmöglich. Wir brauchen die Hilfe der Hardware. Moderne Prozessoren (egal ob Intel, AMD oder Apple Silicon) bieten spezielle Maschinenbefehle an, die extrem komplexe Operationen in einem einzigen, unteilbaren (atomaren) Taktzyklus ausführen.

Der berühmteste und wichtigste dieser Befehle heißt Compare-And-Swap (CAS) – Vergleiche und Tausche.

Eine CAS-Operation nimmt immer drei Parameter entgegen:

1. Speicheradresse: Wo liegt die Variable, die ich ändern will?
2. Erwarteter alter Wert: Was denke ich, was aktuell dort steht?
3. Neuer Wert: Was möchte ich stattdessen dorthin schreiben?

Die CPU führt nun atomar (ohne dass ein anderer Kern dazwischenfunken kann) Folgendes aus: "Schaue an die Speicheradresse. Steht dort noch mein erwarteter alter Wert? Wenn JA, überschreibe ihn mit dem neuen Wert und melde 'Erfolg'. Wenn NEIN (weil ein anderer Thread schneller war), tue gar nichts und melde 'Fehlschlag'."

21.2 Die Endlosschleife des Optimisten

Wie nutzen wir diesen CAS-Befehl nun, um z.B. einen Zähler sicher zu erhöhen? Wir bauen eine kleine, rasend schnelle Schleife (oft Spin-Loop genannt).

Das Muster sieht immer so aus:

1. Lese den aktuellen Wert der Variablen.
2. Berechne den neuen Wert (z.B. aktueller Wert + 1).
3. Führe CAS aus: "Ersetze den aktuellen Wert durch den neuen Wert, aber nur, wenn der aktuelle Wert in der Zwischenzeit nicht von jemand anderem verändert wurde!"
4. War CAS erfolgreich? Super, wir sind fertig.
5. Ist CAS fehlgeschlagen? Mist, ein anderer Thread war schneller. Gehe zurück zu Schritt 1 und versuche es erneut (mit dem nun aktuelleren Wert).

Da diese Schleife die Threads niemals auf Betriebssystem-Ebene blockiert (sie rechnen einfach sofort weiter), ist sie bei leichter bis mittlerer Konkurrenz um ein Vielfaches schneller als ein synchronized-Block.

21.3 Das Paket java.util.concurrent.atomic

In Java müssen Sie glücklicherweise keine Hardware-Befehle manuell aufrufen. Die Java-Entwickler haben CAS-Operationen tief in der JVM (via nativer Aufrufe) verankert und bieten uns das Paket java.util.concurrent.atomic an.

Die wichtigsten Klassen sind:

- AtomicInteger und AtomicLong für Zahlen.
- AtomicBoolean für Wahrheitswerte (Flags).
- AtomicReference<V> für beliebige Objektreferenzen.

21.3.1 Beispiel: Der sperrfreie Zähler

Erinnern Sie sich an unser Kontostands-Problem aus Kapitel 12? So lösen Profis das Problem heute – völlig ohne synchronized oder ReentrantLock:

```
import java.util.concurrent.atomic.AtomicInteger;

public class AtomaresKonto {
    // Unser Zähler kapselt den Wert und die CAS-Logik
    private final AtomicInteger saldo = new AtomicInteger(0);

    public void einzahlen(int betrag) {
        // Diese Methode kapselt die gesamte CAS-Schleife für uns!
        // Sie liest, addiert und swappt absolut atomar und
        // thread-sicher.
        saldo.addAndGet(betrag);
    }

    public int getSaldo() {
        return saldo.get();
    }
}
```

Die Klasse AtomicInteger bietet uns viele Bequemlichkeits-Methoden wie incrementAndGet() (entspricht ++i) oder getAndIncrement() (entspricht i++).

21.3.2 Unter der Haube von AtomicInteger

Um zu verstehen, wie wertvoll diese Klassen sind, schauen wir uns an, wie wir eine eigene Operation mit der grundlegenden Methode compareAndSet (dem direkten Java-Äquivalent zum CAS-Befehl) programmieren würden. Angenommen, wir wollen den Saldo nur verdoppeln, aber es gibt dafür keine fertige Methode in AtomicInteger.

```
    public void verdoppeln() {
        int aktuell;
        int neu;

        do {
            aktuell = saldo.get(); // 1. Lesen
            neu = aktuell * 2; // 2. Berechnen

            // 3. CAS ausführen!
            // Liefert true, wenn der Wert erfolgreich getauscht
            // wurde. Liefert false, wenn ein anderer Thread schneller
            // war.
        } while (!saldo.compareAndSet(aktuell, neu));
        // 4. Bei false: Wiederholen!
    }
```

21.4 Eine kleine Schwäche: Das ABA-Problem

Lock-Free Programming ist genial, hat aber einen klassischen theoretischen Stolperstein, den Sie als Informatiker kennen müssen: das ABA-Problem.

Stellen Sie sich vor, Thread 1 liest den Wert "A". Bevor er seinen neuen Wert berechnet und den CAS-Befehl absetzt, funkt Thread 2 dazwischen. Thread 2 ändert den Wert

von "A" auf "B". Danach kommt Thread 3 und ändert den Wert von "B" wieder zurück auf "A".

Nun führt Thread 1 seinen CAS-Befehl aus: "Ist der Wert noch 'A'?". Die Hardware schaut nach, sieht ein "A" und meldet Erfolg! Thread 1 denkt, es hätte sich nichts verändert. In Wirklichkeit wurde das Objekt dazwischen aber zweimal ausgetauscht. Bei einfachen Zahlen (Kontostand) ist das meist egal. Bei komplexen Datenstrukturen (wie verketteten Listen, bei denen "A" eine Speicheradresse repräsentiert) kann dies jedoch zu Speicherzugriffsfehlern führen, da der Knoten "A" in der Zwischenzeit vielleicht recycelt wurde.

Lösung in Java: Für solche extremen Spezialfälle bietet Java die Klasse AtomicStampedReference. Sie speichert neben der eigentlichen Referenz noch einen "Stempel" (einen Zähler). Bei jeder Änderung wird der Stempel hochgezählt (A1 -> B2 -> A3). Der CAS-Befehl vergleicht nun Referenz und Stempel, wodurch das ABA-Problem sicher erkannt wird.

Hier ist die erweiterte Fassung Ihres Abschnitts, die die AtomicMarkedReference ergänzt und ein anschauliches Code-Beispiel für den Einsatz der Stempel-Logik enthält:

21.5 Der theoretische Stolperstein: Das ABA-Problem

Lock-Free Programming ist genial, hat aber einen klassischen theoretischen Stolperstein, den Sie als Informatiker kennen müssen: das ABA-Problem.

Stellen Sie sich vor, Thread 1 liest den Wert „A“. Bevor er seinen neuen Wert berechnet und den CAS-Befehl absetzt, funkt Thread 2 dazwischen. Thread 2 ändert den Wert von „A“ auf „B“. Danach kommt Thread 3 und ändert den Wert von „B“ wieder zurück auf „A“.

Nun führt Thread 1 seinen CAS-Befehl aus: „Ist der Wert noch 'A'?“. Die Hardware schaut nach, sieht ein „A“ und meldet Erfolg! Thread 1 denkt, es hätte sich nichts verändert. In Wirklichkeit wurde das Objekt dazwischen aber zweimal ausgetauscht. Bei einfachen Zahlen (wie einem Kontostand) ist das oft egal. Bei komplexen Datenstrukturen (wie verketteten Listen, bei denen „A“ eine Speicheradresse repräsentiert) kann dies jedoch zu katastrophalen Fehlern führen, da der Knoten „A“ in der Zwischenzeit vielleicht recycelt oder intern umstrukturiert wurde.

21.5.1 Die Lösung: Zeitstempel und Markierungen

Für solche Spezialfälle bietet Java zwei spezialisierte Klassen an, die neben der Referenz eine zusätzliche Information atomar mit verwalten:

- AtomicStampedReference: Sie speichert neben der Referenz einen „Stempel“ (einen int-Zähler). Bei jeder Änderung wird dieser Stempel manuell hochgezählt (A, Version 1 → B, Version 2 → A, Version 3). Der CAS-Befehl vergleicht nun sowohl die Referenz als auch den Stempel. Da der Stempel bei der Rückkehr zu „A“ nun 3 statt 1 ist, schlägt der CAS-Befehl bei Thread 1 korrekterweise fehl.
- AtomicMarkedReference: Diese Klasse funktioniert ähnlich, nutzt aber statt eines Zählers nur ein einfaches boolean-Flag („markiert“ oder „nicht markiert“). Dies ist nützlich, wenn es nicht auf die Anzahl der Änderungen ankommt, sondern man lediglich markieren möchte, dass ein Knoten logisch bereits gelöscht oder veraltet ist, während er physisch noch in der Struktur existiert.

21.5.2 Ein Beispiel mit AtomicStampedReference

Schauen wir uns an, wie man das ABA-Problem mit einem Stempel technisch unterbindet:

```
// Wir starten mit dem Objekt "A" und dem Stempel (Version) 1
String initialRef = "A";
int initialStamp = 1;
AtomicStampedReference<String> asr = new AtomicStampedReference<>(
        initialRef, initialStamp);

// --- Thread 1 bereitet seinen Zug vor ---
int stampBefore = asr.getStamp(); // liest 1
String refBefore = asr.getReference(); // liest "A"

// --- Thread 2 simuliert das ABA-Problem (A -> B -> A) ---
asr.compareAndSet("A", "B", asr.getStamp(),
        asr.getStamp() + 1); // A1 -> B2
asr.compareAndSet("B", "A", asr.getStamp(),
        asr.getStamp() + 1); // B2 -> A3

// --- Thread 1 versucht nun seinen CAS-Befehl ---
// Er will von "A" auf "C" wechseln, aber nur wenn Version
// noch 1 ist!
boolean success = asr.compareAndSet(refBefore, "C",
        stampBefore, stampBefore + 1);

System.out.println("Update erfolgreich? " + success);
// Ausgabe: false (da die Referenz zwar "A" ist, aber der
// Stempel 3 statt 1)
```

Fazit für die Praxis: In 99 % aller Fälle werden Sie mit AtomicInteger oder AtomicReference auskommen. Wenn Sie jedoch beginnen, eigene hochoptimierte,

verkettete Datenstrukturen (wie lock-free Stacks oder Queues) zu bauen, sind die Stamped- oder Marked-Varianten Ihre Lebensversicherung gegen das tückische ABA-Szenario.

21.6 Das Hochlast-Wunder: Adder und Accumulator

Ist AtomicLong also immer die perfekte Lösung? Nicht ganz. Wenn Sie ein System haben, bei dem hunderte Threads gleichzeitig versuchen, denselben AtomicLong hochzuzählen (z. B. ein globaler Klick-Zähler auf einer gigantischen Website), schlagen die internen CAS-Operationen andauernd fehl. Die Threads hängen in ihren Spin-Loops (Wiederholungsschleifen) fest, blockieren die CPU und die Performance sinkt drastisch.

Für genau dieses Szenario (extrem hohe Schreibkonkurrenz) führte Java 8 eine neue Familie von Klassen ein, die intern auf einer hochoptimierten Architektur (bekannt als Striped64) basieren.

Anstatt alle Threads auf exakt denselben Speicherbereich loszulassen, legen diese Klassen intern ein Array von Variablen (Zellen) an. Wenn viele Threads gleichzeitig schreiben wollen, weicht die Klasse aus und gibt verschiedenen Threads quasi eigene, private kleine Zähler, die sie ohne Konkurrenz aktualisieren können. Erst wenn Sie am Ende das Gesamtergebnis abfragen, rechnet Java alle privaten Zähler zusammen. Ein absoluter Meilenstein für Hochlast-Systeme!

Diese Familie teilt sich in zwei Kategorien auf: die Spezialisten (Adder) und die Generalisten (Accumulator).

21.6.1 Die Spezialisten: LongAdder und DoubleAdder

Diese beiden Klassen können exakt eine einzige Sache, diese dafür aber extrem gut: Addieren und Subtrahieren. Wenn Sie in Ihrem Programm einfach nur Ereignisse zählen wollen – etwa die Anzahl der verarbeiteten HTTP-Requests oder die Menge an gesendeten Bytes –, dann sind LongAdder oder DoubleAdder das perfekte Werkzeug.

Sie rufen einfach increment(), decrement() oder add(wert) auf. Wenn Sie den endgültigen Zählerstand benötigen, rufen Sie sum() (oder intValue()) auf, woraufhin die Klasse alle internen Zellen addiert und das finale Ergebnis liefert.

21.6.2 Die Generalisten: LongAccumulator und DoubleAccumulator

Was aber, wenn Sie bei massiver Nebenläufigkeit gar keine Summe bilden wollen? Was, wenn Sie den absoluten Höchstwert (Maximum) aus Millionen von Messwerten

finden wollen? Oder das Minimum? Hier müssen die Adder passen, da sie fest auf das Plus-Zeichen verdrahtet sind.

Für diese Fälle gibt es den LongAccumulator und den DoubleAccumulator. Anstatt die Rechenoperation fest einzubauen, verlangen diese Klassen bei ihrer Erstellung zwei Dinge von Ihnen:

- Einen Startwert (Identity).
- Eine eigene Rechenregel in Form einer Lambda-Expression (z. B. einen LongBinaryOperator), die exakt definiert, wie ein neu übergebener Wert mit dem bisherigen Zwischenergebnis verrechnet werden soll.

Ein Beispiel: Wir suchen den absoluten Höchstwert

```
// 1. Parameter: Die Rechenregel (Nimm immer den größeren der
// beiden Werte)
// 2. Parameter: Der Startwert (Wir fangen bei 0 an)
LongAccumulator maxSucher = new LongAccumulator(Math::max, 0);

// Hunderte Threads werfen nun wild Zahlen in den Accumulator:
maxSucher.accumulate(10);
maxSucher.accumulate(55);
maxSucher.accumulate(5);

// Am Ende rufen wir get() auf, um alle internen Zellen
// anhand unserer Regel (Math::max) zusammenzufassen.
System.out.println("Der Höchstwert war: " + maxSucher.get());
// Ausgabe: 55
```

Sie rufen bei diesen Klassen nicht mehr add() auf, sondern accumulate(). Das System nimmt den neuen Wert, wendet Ihre Lambda-Funktion (hier Math::max) lokal und konfliktfrei an und speichert das neue Ergebnis.

21.6.3 Die Faustregel für die Praxis

Die Entscheidung ist denkbar einfach:

- Geht es in Ihrem Code nur darum, Zähler hoch- oder runterzuzählen? Greifen Sie blind zum LongAdder (oder DoubleAdder), da er etwas speicherschonender und die API noch simpler ist.
- Benötigen Sie jedoch eine andere mathematische Verknüpfung (Maximum, Minimum, Multiplikation), ist der LongAccumulator (oder DoubleAccumulator) Ihr maßgeschneidertes Werkzeug für Hochlast-Szenarien.

21.7 Speicherschlank optimieren: Die AtomicFieldUpdater

Klassen wie AtomicInteger sind fantastisch für blockierungsfreie Zähler. Da sie jedoch vollwertige Objekte sind, erzeugen sie durch ihren Objekt-Header einen gewissen

Speicher-Overhead. Wenn Sie eine riesige Datenstruktur – etwa einen Graphen mit zehn Millionen Knoten – entwerfen und jeder Knoten einen eigenen Zähler benötigt, würde die Verwendung von zehn Millionen AtomicInteger-Objekten enormen Arbeitsspeicher verschwenden.

Für exakt solche Extremfälle bietet Java die AtomicFieldUpdater (z. B. AtomicIntegerFieldUpdater). Sie ermöglichen atomare Operationen (wie incrementAndGet oder compareAndSet) direkt auf primitiven Variablen, ohne dass für jeden Zähler ein neues Objekt instanziiert werden muss.

Der Trick: Sie deklarieren in Ihrer Klasse lediglich eine primitive volatile-Variable. Den Updater selbst erzeugen Sie (via Reflection) nur ein einziges Mal als statische Konstante für alle Instanzen dieser Klasse:

```
import java.util.concurrent.atomic.AtomicIntegerFieldUpdater;

public class Datenknoten {

    // Zwingend volatile, aber kein teures Objekt!
    volatile int zugriffsZaehler = 0;

    // Ein einziger statischer Updater für ALLE Knoten-Instanzen
    private static final AtomicIntegerFieldUpdater<Datenknoten> UPDATER =
      AtomicIntegerFieldUpdater.newUpdater(Datenknoten.class,
                                           "zugriffsZaehler");

    public void inkrementiereZugriff() {
        // Wir übergeben dem Updater das aktuelle Objekt (this)
        UPDATER.incrementAndGet(this);
    }
}
```

Die Spielregeln: Damit das funktioniert, muss die Zielvariable zwingend mit volatile deklariert sein. Zudem benötigt der Updater, da er textbasiert ("zugriffsZaehler") über Reflection arbeitet, die entsprechenden Sichtbarkeitsrechte auf das Feld.

Fazit für die Praxis: Die AtomicFieldUpdater sind ein absolutes Profi-Werkzeug zur Mikro-Optimierung. Sie werden tief im Inneren des JDKs (etwa in der ConcurrentHashMap) massenhaft genutzt, um Bytes zu sparen. Im normalen Entwickleralltag, wo es selten auf jedes Kilobyte ankommt, sollten Sie aus Gründen der Lesbarkeit und Refactoring-Sicherheit weiterhin beruhigt zum klassischen AtomicInteger greifen.

22 Wenn nichts mehr geht – Liveness-Probleme analysieren und vermeiden

In Kapitel 12 haben wir die zwei großen Gegenspieler der nebenläufigen Programmierung kennengelernt: Safety (Sicherheit – "Nichts Falsches darf passieren") und Liveness (Lebendigkeit – "Irgendetwas Gutes muss irgendwann passieren").

Um die Safety zu gewährleisten, haben wir in den vergangenen Kapiteln fleißig Sperren (synchronized, ReentrantLock) verteilt. Doch je mehr Sperren Sie in einem System verbauen, desto höher ist das Risiko, die Liveness zu zerstören. Wenn Threads sich gegenseitig blockieren oder Ressourcen unfair verteilt werden, friert Ihr Programm ein oder reagiert extrem langsam.

In diesem Kapitel sezieren wir die drei großen Liveness-Probleme: Deadlock, Livelock und Starvation. Wir lernen, wie sie entstehen, wie man sie im Code erkennt und – am wichtigsten – wie man sie als Software-Architekt proaktiv vermeidet.

22.1 Der absolute Stillstand: Deadlock (Verklemmung)

Der Deadlock ist der bekannteste und am meisten gefürchtete Fehler in der Nebenläufigkeit. Ein Deadlock tritt auf, wenn zwei oder mehr Threads ewig aufeinander warten, weil jeder eine Sperre hält, die der andere benötigt, um weiterzumachen.

Die klassische Anatomie eines Deadlocks: Stellen Sie sich vor, wir programmieren ein Überweisungssystem. Um Geld von Konto A auf Konto B zu überweisen, muss der Thread beide Konten sperren, damit in der Zwischenzeit niemand anderes die Salden manipuliert.

```
public class Bank {
    public void ueberweisen(Konto von, Konto nach, int betrag) {
        // 1. Sperre das Quell-Konto
        synchronized (von) {
            System.out.println(Thread.currentThread().getName()
                    + " hat Sperre für " + von.getName());

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            } // Simuliert etwas Rechenzeit

            // 2. Sperre das Ziel-Konto
            synchronized (nach) {
                System.out.println(Thread.currentThread().getName()
                        + " hat Sperre für " + nach.getName());
                von.abheben(betrag);
                nach.einzahlen(betrag);
```

```
                }
            }
        }
    }
```

Auf den ersten Blick sieht dieser Code vollkommen sicher aus. Die Daten sind geschützt. Doch was passiert, wenn Kunde 1 (Thread 1) Geld von Konto A auf Konto B überweist, und Kunde 2 (Thread 2) im exakt selben Moment Geld von Konto B auf Konto A überweist?

Das fatale Interleaving (Der Ablauf):

1. Thread 1 ruft ueberweisen(A, B, 100) auf. Er schnappt sich den Monitor von A.
2. Thread 2 ruft ueberweisen(B, A, 50) auf. Er schnappt sich den Monitor von B.
3. Thread 1 erreicht den inneren Block und will B sperren. Die Tür ist zu. Thread 1 blockiert und wartet.
4. Thread 2 erreicht den inneren Block und will A sperren. Die Tür ist zu. Thread 2 blockiert und wartet.

Beide Threads schlafen ein. Da ein Thread, der auf ein synchronized wartet, seine bereits gehaltenen Sperren nicht aufgibt, werden A und B niemals wieder freigegeben. Das Programm ist tot.

22.2 Strategien zur Deadlock-Vermeidung

Wie verhindern wir dieses Desaster? In der Informatik gibt es mehrere bewährte Strategien.

22.2.1 Strategie 1: Feste Sperr-Reihenfolge (Lock Ordering)

Ein Deadlock kann mathematisch bewiesen nur dann entstehen, wenn es einen Zirkelschluss (Kreis) in den Wartebedingungen gibt. Brechen wir diesen Kreis auf, ist ein Deadlock unmöglich.

Die Regel lautet: Alle Threads müssen die benötigten Sperren immer in exakt derselben globalen Reihenfolge anfordern.

Für unser Bank-Beispiel bedeutet das: Wir sortieren die Konten einfach nach ihrer (hoffentlich eindeutigen) ID oder Kontonummer. Wir sperren immer zuerst das Konto mit der kleineren ID, unabhängig davon, ob es das Quell- oder Zielkonto ist!

```
public void sichereUeberweisung(Konto konto1, Konto konto2,
        int betrag) {
    Konto erstesLock = (konto1.getId() < konto2.getId()) ? konto1
```

```
                : konto2;
        Konto zweitesLock = (konto1.getId() < konto2.getId()) ? konto2
                : konto1;

        synchronized (erstesLock) {
            synchronized (zweitesLock) {
                // Führe die Überweisung durch (Richtung beachten!)
                // ...
            }
        }
    }
```

Wenn nun Thread 1 von A nach B und Thread 2 von B nach A überweist, versuchen beide Threads zwingend, zuerst Konto A zu sperren. Wer zuerst kommt, gewinnt. Der andere wartet ganz brav VOR der ersten Tür und blockiert noch gar nichts. Deadlock abgewendet!

22.2.2 Strategie 2: Timeouts mit tryLock()

Wie in Kapitel 16 gelernt, ist das endlose Warten von synchronized gefährlich. Wenn wir stattdessen ein ReentrantLock verwenden, können wir mit tryLock(timeout) eine Obergrenze festlegen.

Wenn ein Thread nach 3 Sekunden die zweite Sperre nicht bekommt, bricht er den Vorgang ab, gibt die erste Sperre wieder frei (sehr wichtig!), wartet eine zufällige kurze Zeitspanne und versucht die gesamte Transaktion einfach von vorn.

22.3 Viel Lärm um nichts: Livelock

Ein Livelock ist der fiese Cousin des Deadlocks. Die Threads sind hier nicht vom Betriebssystem blockiert. Sie laufen weiter, sie ändern ihre Zustände, aber das System als Ganzes macht keinen einzigen Schritt Fortschritt.

Die Metapher: Stellen Sie sich zwei höfliche Menschen vor, die sich auf einem engen Flur entgegenkommen.

1. Person A weicht nach links aus.
2. Person B weicht im selben Moment nach rechts aus (aus ihrer Sicht also auch nach links).
3. Sie stehen sich wieder im Weg!
4. Beide entschuldigen sich, Person A weicht nach rechts aus, Person B nach links.
5. Sie stehen sich schon wieder im Weg!

Beide sind extrem aktiv (CPU-Auslastung liegt bei 100 %), aber niemand kommt an dem anderen vorbei.

In der Software entsteht ein Livelock oft genau dann, wenn man versucht, Deadlocks schlecht zu lösen (z.B. wenn zwei Threads ständig tryLock() aufrufen, fehlschlagen, beide ihre ersten Locks aufgeben, beide sofort wieder das erste Lock greifen, wieder beim zweiten fehlschlagen und so weiter).

Die Lösung: Führen Sie Zufälligkeit (Randomization) ein! Genau wie in der Netzwerktechnik (beim CSMA/CD-Verfahren von Ethernet), wo Computer nach einer Kollision eine zufällige Zeit lang warten, bevor sie neu senden. Wenn Thread A 10 Millisekunden wartet, Thread B aber zufällig 40 Millisekunden, ist der fatale Rhythmus gebrochen und einer kommt durch.

22.4 Das vergessene Kind: Starvation (Verhungern)

Starvation tritt auf, wenn ein Thread zwar theoretisch arbeiten könnte, aber praktisch niemals die Ressourcen (CPU-Zeit oder Sperren) zugeteilt bekommt, weil andere Threads sich immer vordrängeln. Das System läuft zwar, aber ein bestimmter Teil der Anwendung "verhungert".

Ursachen für Starvation:

1. Falsche Thread-Prioritäten: In Java können Sie Threads Prioritäten von 1 (MIN_PRIORITY) bis 10 (MAX_PRIORITY) geben. Wenn Sie einen Thread auf 1 setzen und ständig neue Threads mit Priorität 10 starten, wird das Betriebssystem dem armen 1er-Thread vielleicht niemals Rechenzeit gönnen. Best Practice: Finger weg von Thread-Prioritäten in Java! Überlassen Sie das dem OS-Scheduler.
2. Unfaire Locks: Bei einem normalen synchronized-Block oder einem Standard-ReentrantLock gibt es keine Garantie, in welcher Reihenfolge wartende Threads bedient werden. Ein neu ankommender Thread kann einem Thread, der schon seit einer Minute wartet, die Sperre einfach vor der Nase wegschnappen.
3. Endlosschleifen von Lesern: Bei einem ReadWriteLock (Kapitel 18) kann ein Schreib-Thread verhungern, wenn das System permanent von tausenden Lese-Threads bombardiert wird und das Lese-Lock niemals für auch nur eine Millisekunde komplett frei wird.

Die Lösung: Nutzen Sie "faire" Sperren (new ReentrantLock(true)), wenn Sie bemerken, dass einzelne Threads chronisch benachteiligt werden. Bedenken Sie aber den in Kapitel 18 erwähnten Performance-Overhead.

Teil III:

Praxis, Performance, Perspektiven – Nebenläufigkeit im realen Einsatz

23 Praxis

Nachdem wir in den Teilen I und II das komplette architektonische und sprachliche Fundament der parallelen Programmierung in Java gelegt haben, betreten wir nun die Werkstatt. In der realen Softwareentwicklung reicht es nicht aus, Code zu schreiben, der "meistens" funktioniert. Er muss testbar, messbar, wartbar und zukunftssicher sein. Dieser dritte Teil des Buches widmet sich den handwerklichen Fähigkeiten, den Best Practices und dem Blick über den Tellerrand.

Kapitel 24: Die Jagd nach dem Heisenbug – Testen und Debuggen

Nebenläufigen Code zu testen, ist eine der größten Herausforderungen der Softwareentwicklung. Dieses Kapitel zeigt, wie man den fehlenden Determinismus bändigt.

- Die Heisenbug-Falle: Warum Fehler beim Debuggen (durch Breakpoints) oft verschwinden (weil Breakpoints das Timing/Interleaving verändern) und wie man stattdessen clever loggt (inklusive Thread-Namen).
- Testen mit Bordmitteln: Wie man CountDownLatch und CyclicBarrier in JUnit-Tests einsetzt, um bestimmte Interleaving-Szenarien und Race Conditions künstlich zu provozieren.
- Moderne Test-Frameworks: Einführung in Bibliotheken wie Awaitility (um elegant auf asynchrone Zustandsänderungen in Tests zu warten, statt Thread.sleep() zu nutzen) und jcstress (das Java Concurrency Stress Testing Tool der JDK-Entwickler, um das Java Memory Model auf die Probe zu stellen).
- Werkzeuge der Profis: Fortgeschrittene Nutzung von Thread Dumps (jstack) und Profilern (VisualVM, Java Flight Recorder) zur Laufzeitanalyse von Deadlocks und Flaschenhälsen.

Kapitel 25: Performance messen, aber richtig – Microbenchmarking

"Mein paralleler Stream ist schneller!" – Eine oft getätigte Aussage von Anfängern, die meistens auf falschen Messmethoden basiert.

- Die Lügen von System.currentTimeMillis(): Warum einfache Stoppuhren in Java wertlos sind (Garbage Collection Pausen, Hintergrund-Prozesse).
- Der JIT-Compiler (Just-In-Time): Wie die JVM zur Laufzeit Code optimiert, toten Code entfernt (Dead Code Elimination) und warum unaufgewärmter Code (Warmup) zu völlig falschen Messergebnissen führt.

- Der Standard: JMH (Java Microbenchmark Harness): Wie man professionelle Microbenchmarks schreibt. Wir zeigen, wie man JMH aufsetzt, um den echten Durchsatz von synchronisierten vs. lock-free Ansätzen wissenschaftlich fundiert zu messen.

Kapitel 26: Aus dem Nähkästchen – Best Practices, Nice-to-Haves und No-Gos

Eine kompakte Sammlung von Mustern und Anti-Mustern, die jeden Code Review überstehen.

- Die absolute Best Practice: Immutability (Unveränderlichkeit): Warum Klassen, deren Zustand sich nach der Erstellung nicht mehr ändert (z.B. durch record in Java oder konsequentes final), der ultimative und eleganteste Schutz vor Race Conditions sind (wo nichts geändert wird, muss nichts synchronisiert werden).
- Geheime Waffen: Um teures Locking komplett zu vermeiden, geben ThreadLocals jedem Thread eine eigene Instanz eines Objektes, wodurch Datenzugriffe physisch isoliert bleiben. Als moderne, sicherere Ergänzung ermöglichen Scoped Values das effiziente Teilen unveränderlicher Daten innerhalb einer streng begrenzten Aufrufhierarchie, ohne die Speicherrisiken klassischer Thread-Lokalvariablen.
- Anti-Patterns (No-Gos): Warum Methoden wie Thread.stop(), suspend() und resume() extrem gefährlich sind (und in modernen Java-Versionen endgültig entfernt werden).
- Das berüchtigte "Double-Checked Locking" Anti-Pattern (und warum es vor Java 5 kaputt war).
- Die Gefahr von "Thread Leaks" in selbstgebauten Thread-Pools.

Kapitel 27: Über den Tellerrand – Alternative Paradigmen (Nice-to-Have)

Java entwickelt sich rasant weiter, aber auch andere Konzepte haben die nebenläufige Welt nachhaltig geprägt. Ein Blick auf Architekturen, die über die klassische Thread-Programmierung hinausgehen:

- Das Actor-Modell (z. B. Akka-Framework): Anstatt sich mit Locks um geteilten Speicher zu streiten, verfolgt dieses Modell den "Share Nothing"-Ansatz. Ein System besteht aus isolierten Einheiten (Acotrs), die ausschließlich über das Senden und Empfangen von asynchronen, unveränderlichen Nachrichten

kommunizieren. Dies macht Systeme extrem skalierbar und fehlertolerant, da ein Absturz eines Actors den Rest des Systems unberührt lässt.

- Reactive Streams (RxJava, Project Reactor): Diese Philosophie dreht das klassische Daten-Holen (Pull) in ein Daten-Schicken (Push) um. Das Herzstück ist das Konzept des Backpressure (Gegendruck): Ein langsamer Verbraucher kann dem schnellen Erzeuger signalisieren, sein Tempo zu drosseln, um nicht mit Daten überflutet zu werden. Um diese reaktiven Konzepte universell nutzbar zu machen, wurde mit der Flow-API (seit Java 9) ein standardisierter Satz an Interfaces direkt in das JDK integriert, der als gemeinsamer Nenner für die Interoperabilität verschiedener Bibliotheken dient.

Kapitel 28: Die physikalischen Grenzen und Javas Zukunft

Zum Abschluss erden wir die Erwartungen und schauen, wohin die Reise der JVM geht.

- Amdahlsches Gesetz (Amdahl's Law): Die bittere mathematische Realität. Warum 100 Kerne Ihr Programm nicht zwingend 100-mal schneller machen (der Flaschenhals des sequenziellen Code-Anteils).
- Die "Memory Wall": Warum das eigentliche Problem oft nicht mehr die CPU-Geschwindigkeit ist, sondern der Transport der Daten vom RAM zur CPU.
- Was bringt die Zukunft?
    - Project Valhalla (Value Objects): Wie Java das Speicherlayout (Memory Layout) revolutionieren will, um Cache-Misses zu minimieren und Parallelausführung extrem zu beschleunigen.
    - Die Vector API: Echte Datenparallelität auf Hardware-Ebene, indem sie Java-Code direkt in hochoptimierte SIMD-Befehle (Single Instruction, Multiple Data) der CPU übersetzt, um mehrere Datenpunkte in einem einzigen Taktzyklus gleichzeitig zu berechnen.
    - Project Loom (Strukturierte Nebenläufigkeit / Structured Concurrency): Mit Structured Concurrency führt Java ein Modell ein, das zusammengehörige parallele Aufgaben in einer klaren Hierarchie bündelt, sodass Unteraufgaben automatisch sauber beendet oder abgebrochen werden, sobald der übergeordnete Task fertig ist oder einen Fehler wirft.

24 Die Jagd nach dem Heisenbug – Testen und Debuggen

Sie haben nun ein ganzes Arsenal an Werkzeugen kennengelernt, um nebenläufigen Code zu schreiben. Doch in der professionellen Softwareentwicklung gilt: Code, der nicht automatisiert getestet werden kann, ist defekter Code.

Das Testen und Debuggen von Multi-Threading-Anwendungen stellt uns jedoch vor ein gewaltiges Problem: In Kapitel 12 haben wir gelernt, dass der Betriebssystem-Scheduler die Verzahnung (das Interleaving) der Threads bei jedem Programmstart völlig zufällig wählt. Wir haben den Determinismus verloren. Wie schreibt man also einen Unit-Test (der eigentlich immer exakt dasselbe tun soll) für ein System, das sich bei jedem Durchlauf minimal anders verhält?

24.1 Die Heisenbug-Falle und warum der Debugger lügt

Wenn in einem normalen, sequenziellen Programm ein Fehler auftritt, setzen Sie einen Breakpoint (Haltepunkt) in Ihrer IDE, starten den Debugger und gehen den Code Schritt für Schritt durch, bis Sie den Fehler finden.

Wenn Sie das bei nebenläufigem Code versuchen, werden Sie ein bizarres Phänomen erleben: Sobald Sie den Debugger anwerfen, verschwindet der Fehler! Starten Sie das Programm danach wieder normal, stürzt es wieder ab.

Dieser Effekt wird in der Informatik scherzhaft Heisenbug genannt (in Anlehnung an die Heisenbergsche Unschärferelation aus der Physik: Die bloße Beobachtung eines Systems verändert dessen Zustand).

Warum passiert das? Ein Bug in nebenläufigem Code (wie eine Race Condition) tritt meistens nur bei einem ganz spezifischen, unglücklichen Interleaving auf – z. B. wenn Thread A genau eine Millisekunde vor Thread B an einer Variablen ankommt. Setzen Sie nun einen Breakpoint, hält die IDE den ankommenden Thread an. Die anderen Threads laufen im Hintergrund aber oft weiter (oder werden alle abrupt pausiert). Sie verändern das natürliche Timing der Threads drastisch. Das unglückliche Interleaving tritt nicht mehr auf, der Fehler versteckt sich.

Die Lösung: Logging statt Debugging. In der nebenläufigen Welt greifen Profis daher seltener zum Debugger und öfter zu gut durchdachtem Logging. Praxis-Tipp: Konfigurieren Sie Ihren Logger (z.B. Logback[3] oder Log4j[4]) immer so, dass er den

[3] https://github.com/qos-ch/logback
[4] https://github.com/apache/logging-log4j2

Namen des aktuellen Threads mit ausgibt! Nur so können Sie im Nachhinein den wahren Ablauf rekonstruieren.

```
        // Schlecht: Man weiß nicht, wer das geschrieben hat
        System.out.println("Wert geändert auf: " + wert);

        // Gut: Thread-Name ist zwingend erforderlich für die Analyse!
        System.out.println("[" + Thread.currentThread().getName()
                + "] Wert geändert auf: " + wert);
```

24.2 Testen mit Bordmitteln: CountDownLatch in JUnit

Schauen wir uns an, wie wir einen echten Test schreiben. Angenommen, wir wollen testen, ob eine bestimmte Klasse wirklich thread-sicher ist. Wir müssen also künstlich extremen Stress erzeugen, indem wir hunderte Threads gleichzeitig auf die Klasse loslassen.

Ein typischer Anfängerfehler in JUnit sieht so aus:

```
    @Test
    public void falscherTest() {
        MeinZaehler zaehler = new MeinZaehler();

        // Wir starten 100 Threads im Hintergrund
        for (int i = 0; i < 100; i++) {
            new Thread(() -> zaehler.erhoehen()).start();
        }

        // FATAL: Der Main-Thread des JUnit-Tests läuft sofort weiter!
        // Die Assertion wird geprüft, während die Threads im
        // Hintergrund noch starten.
        // Der Test wird fast immer fehlschlagen, weil das Ergebnis
        // noch nicht 100 ist.
        assertEquals(100, zaehler.getWert());
    }
```

Um das zu reparieren, nutzen wir unser Wissen aus Kapitel 19: Wir choreografieren den Test mit einem CountDownLatch!

```
import org.junit.jupiter.api.Test;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class ConcurrencyTest {

    @Test
    public void richtigerTest() throws InterruptedException {
        MeinZaehler zaehler = new MeinZaehler();
        int anzahlThreads = 100;

        // 1. Latch erstellen, das auf 100 Signale wartet
        CountDownLatch alleFertig = new CountDownLatch(anzahlThreads);
```

```
        ExecutorService pool = Executors.newFixedThreadPool(10);

        for (int i = 0; i < anzahlThreads; i++) {
            pool.submit(() -> {
                try {
                    zaehler.erhoehen();
                } finally {
                    // 2. Jeder Thread meldet sich nach getaner Arbeit
                    // ab!
                    alleFertig.countDown();
                }
            });
        }

        // 3. Der JUnit-Test blockiert hier und wartet auf alle
        // Worker-Threads!
        alleFertig.await();
        pool.shutdown();

        // 4. Erst jetzt dürfen wir das Ergebnis sicher prüfen
        assertEquals(100, zaehler.getWert(),
                "Zähler ist nicht thread-sicher!");
    }
}
```

Tipp: Führen Sie solche Tests nicht nur einmal aus. Da Race Conditions vom Zufall abhängen, sollten Concurrency-Tests in einer CI/CD-Pipeline (z.B. Jenkins, GitHub Actions) oft mit der Annotation @RepeatedTest(100) versehen werden, um sie hundertmal hintereinander abzufeuern.

24.3 Elegantes Testen: Das Awaitility-Framework

Das manuelle Verwalten von CountDownLatch-Objekten in Testklassen bläht den Code schnell auf. In der Industrie hat sich daher für asynchrone Tests eine externe Open-Source-Bibliothek als Quasi-Standard etabliert: Awaitility.[5]

Awaitility erlaubt es Ihnen, Wartebedingungen deklarativ (als fließenden Text) aufzuschreiben, ohne eigene Locks oder unzuverlässige Thread.sleep()-Aufrufe zu nutzen.

```
    import static org.awaitility.Awaitility.await;
    import static java.util.concurrent.TimeUnit.SECONDS;

    @Test
    public void testMitAwaitility() {
        NachrichtenSystem system = new NachrichtenSystem();

        // Startet einen asynchronen Prozess
        system.verarbeiteNachrichtenImHintergrund();

        // Awaitility prüft nun z.B. alle 100ms, ob die Bedingung wahr
        // wird. Nach spätestens 5 Sekunden wirft es eine TimeoutException
```

[5] https://github.com/awaitility/awaitility

```
        // und der Test schlägt fehl.
        await().atMost(5, SECONDS).until(() ->
                                    system.istVerarbeitungFertig());

        // Wenn wir hier ankommen, war die Bedingung rechtzeitig erfüllt
        assertEquals(10, system.getVerarbeiteteNachrichten());
    }
```

24.4 Die Wahrheit ans Licht bringen: jcstress

Wie wir gesehen haben, ist Nebenläufigkeit tückisch. Ein Programm kann auf Ihrem Laptop tausendmal korrekt laufen und erst unter Hochlast auf einem 64-Kern-Server im Rechenzentrum plötzlich subtile Datenfehler (wie das ABA-Problem oder Sichtbarkeitsprobleme) zeigen. Herkömmliche Unit-Tests (wie JUnit) finden solche Fehler fast nie, da sie die nötige „Race Condition“ nicht erzwingen können.

Hier kommt jcstress[6] (*Java Concurrency Stress Tests*) ins Spiel. Es ist das offizielle Werkzeug der OpenJDK-Entwickler, um die Korrektheit des Java-Speichermodells zu prüfen.

Wie funktioniert es? Anstatt einen Test nur einmal auszuführen, lässt jcstress Ihren Code in Millionen von Iterationen auf allen verfügbaren CPU-Kernen gleichzeitig laufen. Dabei werden die Threads so geschickt gegeneinander verzögert und „gestresst“, dass selbst extrem seltene Hardware-Effekte provoziert werden.

Das Ergebnis: Nach dem Testlauf erhalten Sie eine detaillierte Tabelle. Darin steht nicht nur „Erfolg“ oder „Fehler“, sondern eine Auflistung aller Zustände, die jcstress beobachtet hat – auch solche, die laut Java-Spezifikation eigentlich unmöglich sein sollten.

Fazit: Wenn Sie eine eigene, hochperformante Concurrent-Datenstruktur entwickeln, ist jcstress das einzige Werkzeug, das Ihnen die Gewissheit gibt, dass Ihr Code nicht nur „meistens“, sondern unter allen physikalischen Bedingungen korrekt funktioniert. Es ist der „Lügendetektor“ für Java-Concurrency.

24.5 Werkzeuge der Profis: Dem Stillstand auf der Spur

Selbst der beste Code kann in der Produktion hängen bleiben. Wenn Ihr System plötzlich nicht mehr reagiert oder die CPU-Last ohne erkennbaren Grund auf 100 % schießt, helfen klassische Logausgaben oft nicht weiter. Jetzt schlägt die Stunde der Diagnose-Werkzeuge, mit denen Sie direkt in das "Gehirn" der laufenden JVM schauen können.

[6] https://github.com/openjdk/jcstress

24.5.1 Der Röntgenblick: Thread Dumps mit jstack

Ein Thread Dump ist eine Momentaufnahme aller aktiven Threads zu einem exakten Zeitpunkt. Das JDK liefert hierfür das Kommandozeilen-Tool jstack mit. Es zeigt Ihnen für jeden Thread den aktuellen Status (RUNNABLE, WAITING, BLOCKED) und den exakten Stacktrace (wo im Code er sich gerade befindet).

jstack ist so intelligent, dass es zyklische Abhängigkeiten ("Thread A wartet auf Lock 1, den Thread B hält...") automatisch erkennt und am Ende des Dumps prominent als "Found 1 deadlock" markiert.

24.5.2 Live-Überwachung mit VisualVM

Wenn Sie eine grafische Oberfläche bevorzugen, ist VisualVM (oder das modernere JVisualVM) das Werkzeug der Wahl. Es bietet eine Echtzeit-Ansicht der Thread-Aktivitäten. Sie sehen in bunten Zeitstrahlen, welcher Thread gerade arbeitet (grün), welcher schläft (blau) und welcher durch einen Monitor blockiert ist (rot). Dies ist ideal, um "Livelocks" oder Threads aufzuspüren, die sich gegenseitig die Rechenzeit stehlen.

24.5.3 Die Blackbox: Java Flight Recorder (JFR)

Während ein Thread Dump nur ein flüchtiges Foto ist, arbeitet der Java Flight Recorder wie eine Dashcam im Auto: Er zeichnet im Hintergrund fortlaufend alle wichtigen Ereignisse der JVM auf. Das Besondere daran ist die Effizienz. Da der JFR tief in den Kern der Java-Maschine integriert ist, verbraucht er kaum spürbare Rechenleistung (meist unter 1 %), sodass er sogar auf echten Produktionsservern permanent mitlaufen kann.

- Die Analyse mit JDK Mission Control: Wenn Ihr System abstürzt oder plötzlich langsam wird, „spulen“ Sie die Aufzeichnung einfach zurück. Mit dem Analyse-Tool JDK Mission Control sehen Sie in einer Zeitachse genau, was zum kritischen Zeitpunkt passiert ist.
- Flaschenhälse finden: Der JFR protokolliert nicht nur, *dass* ein Thread gewartet hat, sondern auch *wie lange* und *auf welches spezifische Objekt*. So entlarven Sie sofort „heiße Locks“ – also Synchronisationspunkte, an denen sich Ihre Threads gegenseitig ausbremsen, ohne dass es zu einem harten Deadlock kommt.

Fazit für die Praxis: Verlassen Sie sich bei Nebenläufigkeitsproblemen niemals auf Vermutungen. Ein kurzer jstack oder eine JFR-Aufnahme liefert Ihnen in Sekunden die Beweise, für die Sie mit manuellem Debugging Tage bräuchten.

25 Performance messen, aber richtig – Microbenchmarking

"Ich habe .parallelStream() hinzugefügt, aber mein Programm ist jetzt doppelt so langsam! Warum?" Diesen Satz hören Informatik-Dozenten und Senior-Entwickler regelmäßig. Nebenläufigkeit hat ihren Preis (Overhead für Thread-Erstellung, Synchronisation, Context Switches). Ob ein paralleler Ansatz wirklich schneller ist als ein sequenzieller, hängt von vielen Faktoren ab (Hardware, Datenmenge, Art der Aufgabe). Wir müssen es also messen.

Doch das Messen von Code-Ausführungszeiten (Microbenchmarking) in Java ist ein Minenfeld. Wer hier naiv herangeht, misst nicht die Leistung seines Codes, sondern zufällige Hintergrundprozesse der Java Virtual Machine (JVM).

25.1 Die Stoppuhr-Lüge: System.currentTimeMillis()

Der intuitive, aber völlig falsche Ansatz, den fast jeder Anfänger wählt, sieht so aus:

```
// ANTI-PATTERN: So misst man keine Performance in Java!
long start = System.currentTimeMillis();
// oder System.nanoTime()

meineAufwendigeParalleleMethode();

long ende = System.currentTimeMillis();
System.out.println("Dauer: " + (ende - start) + " ms");
```

Wenn Sie diesen Code fünfmal hintereinander ausführen, werden Sie vermutlich fünf völlig unterschiedliche Ergebnisse erhalten (z.B. 400ms, 150ms, 120ms, 110ms, 115ms). Warum schwankt das so extrem und warum ist der erste Durchlauf fast immer der langsamste?

Dafür gibt es drei Hauptschuldige:

1. Betriebssystem-Rauschen (OS Noise): Ihr Betriebssystem führt im Hintergrund hunderte Prozesse aus. Wenn während Ihrer Messung zufällig der Virenscanner anspringt, verliert Ihr Thread CPU-Zeit und Ihre Messung ist verfälscht.
2. Garbage Collection (GC): Die JVM räumt im Hintergrund nicht mehr benötigte Objekte auf. Springt der Garbage Collector genau zwischen start und ende an, friert er Ihre Anwendung möglicherweise für ein paar Millisekunden ein ("Stop-the-World"-Pause). Sie messen dann die Aufräumarbeit der JVM, nicht Ihren Algorithmus.

3. Der JIT-Compiler: Dies ist der wichtigste und komplexeste Faktor, dem wir einen eigenen Abschnitt widmen müssen.

25.2 Die Streiche des JIT-Compilers

Java ist (historisch gesehen) eine interpretierte Sprache. Der Java-Code wird in Bytecode übersetzt, den die JVM zur Laufzeit Zeile für Zeile liest und ausführt. Das ist relativ langsam.

Moderne JVMs besitzen jedoch einen Just-In-Time (JIT) Compiler. Dieser beobachtet das Programm während es läuft. Wenn er feststellt, dass eine bestimmte Methode sehr oft aufgerufen wird (ein sogenannter "Hot Spot"), stoppt er kurz, übersetzt diesen Bytecode in hochoptimierten, maschinenspezifischen C-Code (Maschinencode) und tauscht ihn im Hintergrund aus.

Das bedeutet für unsere Stoppuhr-Messung:

- Warmup: Beim ersten Aufruf (start bis ende) messen Sie den langsamen, interpretierten Bytecode plus die Zeit, die der JIT-Compiler braucht, um den Code zu analysieren und zu optimieren! Erst nach tausenden Durchläufen (der sogenannten Warmup-Phase) hat der Code seine maximale Geschwindigkeit erreicht.
- Dead Code Elimination (Toter Code): Der JIT-Compiler ist extrem schlau. Wenn Sie eine parallele Berechnung durchführen, das Ergebnis aber nirgendwo in einer Variable speichern oder ausdrucken, erkennt der Compiler das. Er denkt sich: "Dieses Ergebnis wird nie benutzt, also lösche ich einfach die gesamte Berechnung!" Ihre Stoppuhr misst dann 0 Millisekunden, weil der JIT-Compiler Ihren Code buchstäblich wegoptimiert hat.

25.3 Die professionelle Lösung: JMH (Java Microbenchmark Harness)

Um diese Hardware- und JVM-Effekte auszugleichen, haben die Entwickler der Java Virtual Machine selbst ein Werkzeug geschrieben: JMH[7]. Es ist der absolute Industriestandard für Microbenchmarks (das Messen sehr kleiner Code-Ausschnitte) in Java.

JMH kümmert sich vollautomatisch um:

- Mehrere Warmup-Iterationen, bevor die echte Messung beginnt.

[7] https://github.com/openjdk/jmh

- Mehrere Mess-Iterationen, um einen statistisch sauberen Durchschnitt zu bilden.
- Das Verhindern von Dead Code Elimination.
- Das Isolieren der Messung in separaten JVM-Prozessen (Forks), um störende Einflüsse früherer Durchläufe auszuschließen.

25.4 Ein echter JMH-Benchmark im Code

Um JMH zu nutzen, fügen wir es meistens über ein Build-Tool wie Maven oder Gradle als Abhängigkeit (Dependency) zu unserem Projekt hinzu. Danach schreiben wir eine Testklasse, die stark an JUnit erinnert, aber andere Annotationen verwendet.

Messen wir doch einmal objektiv, ob ein paralleler Stream (Kapitel 11) beim Summieren von Zahlen wirklich schneller ist als eine normale for-Schleife.

```
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;
import java.util.concurrent.TimeUnit;
import java.util.stream.LongStream;

// Konfiguration des Benchmarks (kann auch über die Kommandozeile
// gesteuert werden)
@BenchmarkMode(Mode.AverageTime) // Wir wollen die durchschnittliche
                                 // Zeit pro Aufruf messen
@OutputTimeUnit(TimeUnit.MILLISECONDS) // in Millisekunden
@Warmup(iterations = 3, time = 1) // 3 Aufwärmrunden à 1 Sekunde
@Measurement(iterations = 5, time = 1) // 5 echte Messrunden à 1
                                       // Sekunde
@Fork(1) // Starte den Benchmark in einer komplett neuen, frischen JVM
@State(Scope.Thread) // Definiert den Zustand für unsere Testdaten
public class StreamBenchmark {

    private long limit = 10_000_000L;

    // --- Benchmark 1: Die klassische Schleife ---
    @Benchmark
    public void sequenzielleSchleife(Blackhole blackhole) {
        long summe = 0;
        for (long i = 1; i <= limit; i++) {
            summe += i;
        }
        // Blackhole verhindert die Dead Code Elimination!
        // Wir zwingen die JVM, das Ergebnis zu "konsumieren".
        blackhole.consume(summe);
    }

    // --- Benchmark 2: Der parallele Stream ---
    @Benchmark
    public void parallelerStream(Blackhole blackhole) {
        long summe = LongStream.rangeClosed(1, limit).parallel()
                .sum();
        blackhole.consume(summe);
    }
}
```

Die Magie des Blackhole (Schwarzes Loch)

Beachten Sie den Parameter Blackhole. Dieses Objekt wird von JMH bereitgestellt. Wenn Sie Ihr berechnetes Ergebnis in blackhole.consume() werfen, simulieren Sie für den JIT-Compiler, dass der Wert essenziell wichtig ist und weiterverwendet wird. Der Compiler darf die Schleife also nicht wegoptimieren, selbst wenn wir den Wert danach nie wieder ansehen.

25.5 Messergebnisse richtig lesen

Wenn Sie JMH starten, rattert das Programm einige Zeit über die Kommandozeile und führt die Warmups durch. Am Ende erhalten Sie eine strukturierte Tabelle, die so aussehen könnte:

```
Benchmark                              Mode  Cnt   Score   Error  Units

StreamBenchmark.sequenzielleSchleife   avgt    5   3,214 ± 0,105  ms/op

StreamBenchmark.parallelerStream       avgt    5   0,952 ± 0,081  ms/op
```

Wie interpretiert man das?

- Mode (avgt): Wir messen die durchschnittliche Zeit (Average Time).
- Score: Der parallele Stream brauchte im Durchschnitt 0,952 Millisekunden, die sequenzielle Schleife 3,214 Millisekunden. Bei 10 Millionen einfachen Rechenoperationen auf einem Multi-Core-Rechner hat sich die Parallelisierung hier also tatsächlich gelohnt!
- Error (±): Die statistische Fehlerquote (Konfidenzintervall). Da die Werte (3,2 ± 0,1 vs. 0,9 ± 0,08) weit auseinanderliegen, ist unser Ergebnis statistisch signifikant und nicht nur Rauschen.

Hätten wir das Limit in unserem Code jedoch auf nur 100 gesetzt, hätte JMH uns erbarmungslos gezeigt, dass der parallele Stream plötzlich massiv langsamer ist als die for-Schleife, weil der Verwaltungs-Overhead der Threads die eigentliche Rechenzeit übersteigt.

Fazit:

Trauen Sie im Bereich der Nebenläufigkeit niemals Ihrem Bauchgefühl und niemals System.currentTimeMillis(). Wenn Sie Architektur-Entscheidungen (wie die Einführung komplexer Thread-Pools oder Fork/Join-Konstrukte) treffen, untermauern Sie diese stets mit sauberen JMH-Benchmarks.

26 Aus dem Nähkästchen – Best Practices, Nice-to-Haves und No-Gos

Sie kennen nun die Syntax, die Frameworks und die Theorie hinter der Nebenläufigkeit in Java. Wenn Sie morgen in ein echtes Software-Projekt einsteigen, reicht es jedoch nicht aus, einfach überall synchronized hinzuschreiben, wo es knallt. Ein guter Software-Architekt zeichnet sich nicht dadurch aus, dass er die kompliziertesten Sperr-Mechanismen baut, sondern dadurch, dass er sie durch smartes Design gar nicht erst benötigt.

In diesem Kapitel fassen wir die wichtigsten Muster (Patterns) und Anti-Muster (No-Gos) zusammen, die Ihnen in jedem professionellen Code-Review begegnen werden.

26.1 Die absolute Best Practice: Immutability (Unveränderlichkeit)

Erinnern Sie sich an die goldene Regel aus Kapitel 12? Gefahr droht nur bei Shared Mutable State (geteiltem, veränderbarem Zustand).

Bisher haben wir immer versucht, das Problem zu lösen, indem wir den Zugriff durch Sperren koordiniert haben. Der viel elegantere Weg ist jedoch, das Problem an der Wurzel zu packen: Wir entfernen einfach das "Mutable" (veränderbar).

Ein Objekt ist immutable (unveränderlich), wenn sein Zustand nach der Konstruktion niemals wieder geändert werden kann.

- Wenn sich ein Objekt nicht ändern kann, kann es auch keinen "Lost Update" geben.
- Wenn sich ein Objekt nicht ändern kann, gibt es kein Sichtbarkeitsproblem (Sichtbar ist immer nur der finale Zustand der Erstellung).
- Konsequenz: Immutable Objekte sind von Natur aus zu 100 % thread-sicher! Sie können ein solches Objekt bedenkenlos an 10.000 Threads gleichzeitig übergeben, ohne ein einziges synchronized oder ReentrantLock zu verwenden.

Wie baut man das in Java?

1. Machen Sie alle Felder private und final.
2. Bieten Sie keine Setter-Methoden an.
3. Wenn eine Methode den Zustand "ändern" soll, darf sie nicht das aktuelle Objekt manipulieren, sondern muss ein komplett neues Objekt mit den neuen Werten zurückgeben (genau so funktioniert die Klasse java.lang.String).

Seit Java 14 geht das noch viel einfacher mit Records:

```
// Ein Record ist standardmäßig komplett immutable und damit thread-sicher!
public record Punkt(int x, int y) {

    // Wenn wir den Punkt verschieben wollen, ändern wir nicht x und
    // y, sondern geben einen neuen Punkt zurück.
    public Punkt verschiebe(int deltaX, int deltaY) {
        return new Punkt(this.x + deltaX, this.y + deltaY);
    }
}
```

Tipp für die Praxis: Versuchen Sie immer, Datenstrukturen, die zwischen Threads hin- und hergereicht werden (z. B. Konfigurationen, Events, DTOs), als Immutable Objects oder Records zu designen. Es eliminiert 90 % aller potenziellen Nebenläufigkeits-Bugs.

26.2 Die Geheimwaffe: ThreadLocal (Share Nothing)

Wenn wir den Zustand nicht unveränderlich machen können, gibt es noch einen zweiten Trick: Wir teilen ihn einfach nicht (Share Nothing).

Stellen Sie sich vor, Sie haben ein Hilfsobjekt, das sehr teuer in der Erstellung ist, weshalb Sie nicht für jede Aktion ein neues new aufrufen wollen. Gleichzeitig ist dieses Objekt aber nicht thread-sicher. Das berühmteste Beispiel hierfür in der älteren Java-API ist der SimpleDateFormat zur Datumsformatierung. Wenn zwei Threads denselben SimpleDateFormat nutzen, zerreißen sie sich intern gegenseitig die Kalender-Daten.

Die Lösung ist die Klasse ThreadLocal<T>.

Sie wirkt wie eine normale Variable, aber unter der Haube fungiert sie als eine Art unsichtbare Map, die den aktuellen Thread als Schlüssel nutzt. Jeder Thread, der get() oder set() aufruft, interagiert nur mit seiner eigenen, privaten Kopie dieses Objekts!

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class DatumsFormatierer {

    // Jeder Thread bekommt seinen GANZ EIGENEN SimpleDateFormat!
    private static final ThreadLocal<SimpleDateFormat> formatierer
        = ThreadLocal
            .withInitial(() -> new SimpleDateFormat("yyyy-MM-dd"));

    public static String formatiere(Date datum) {
        // formatierer.get() liefert die Instanz, die exakt diesem
        // Thread gehört. Kein synchronized nötig, da kein anderer
        // Thread diese Instanz jemals sieht!
        return formatierer.get().format(datum);
    }
}
```

26.3 Der Nachfolger für die Loom-Ära: Scoped Values

Im vorherigen Abschnitt haben wir ThreadLocal als clevere Lösung kennengelernt, um Kontextinformationen (wie eine Transaktions-ID oder einen angemeldeten Benutzer) unsichtbar durch tiefe Aufrufhierarchien zu reichen. Lange Zeit war das die absolute Best Practice in der Java-Welt. Doch mit dem Paradigmenwechsel durch Project Loom (siehe Kapitel 13) und der Einführung von Millionen Virtueller Threads ist ThreadLocal plötzlich zu einer tickenden Zeitbombe für Ihren Arbeitsspeicher geworden.

26.3.1 Das Problem mit ThreadLocal bei Millionen Threads

Klassische Plattform-Threads waren teuer und wurden in kleinen Pools (z. B. 200 Threads in einem Webserver) wiederverwendet. Ein ThreadLocal pro Thread fiel speichertechnisch kaum ins Gewicht.

Virtuelle Threads hingegen sind so billig, dass wir für jeden noch so kleinen Request, ja sogar für winzige Unteraufgaben, einen eigenen Thread starten. Wenn wir nun ThreadLocal bzw. InheritableThreadLocal nutzen, um Daten an Kind-Threads weiterzugeben, muss die JVM für jeden einzelnen dieser hunderttausenden Virtuellen Threads eine tiefe Kopie der Datenstruktur im Speicher anlegen. Der Overhead explodiert.

Zudem leidet ThreadLocal unter zwei grundlegenden Designschwächen:

1. Veränderbarkeit (Mutability): Jeder Code, der Zugriff auf die Variable hat, kann den Wert über die set()-Methode überschreiben. Bei komplexen Datenflüssen verliert man schnell den Überblick, wer den Kontext wann manipuliert hat.
2. Unbegrenzte Lebensdauer: Ein Wert bleibt im ThreadLocal, bis jemand explizit remove() aufruft. Vergisst man dies, entsteht ein schleichendes Memory Leak.

26.3.2 Die moderne Lösung: java.lang.ScopedValue

Um dieses Problem zu lösen, hat Java mit den Scoped Values ein von Grund auf neues, leichtgewichtiges Konstrukt eingeführt, das exakt auf die Bedürfnisse von Virtuellen Threads zugeschnitten ist.

Ein ScopedValue funktioniert ähnlich wie ein ThreadLocal, unterliegt aber strengen, sicherheitsstiftenden Regeln:

- Unveränderlich (Immutable): Einmal gesetzt, kann der Wert nicht mehr geändert, sondern nur gelesen werden.

- Zeitlich begrenzt (Scoped): Der Wert existiert nur für die Dauer eines ganz bestimmten Code-Blocks. Sobald dieser Block verlassen wird, räumt die JVM den Wert automatisch auf. Ein remove() ist weder nötig noch möglich. Memory Leaks sind damit per Design ausgeschlossen!

26.3.3 Scoped Values in der Praxis

Schauen wir uns an, wie elegant sich dieses Konzept im Code ausdrücken lässt. Zuerst definieren wir den ScopedValue – typischerweise als public static final Konstante, genau wie früher beim ThreadLocal.

```
public class SecurityContext {
    // Erstellen eines leeren Scoped Values für den aktuellen Benutzer
    public static final ScopedValue<String> LOGGED_IN_USER = ScopedValue
            .newInstance();
}
```

Wenn nun eine Anfrage (Request) in unserem System ankommt, binden wir den authentifizierten Benutzer an diesen Scope und führen innerhalb dieses Scopes unsere Geschäftslogik aus:

```
public class WebServer {
    public void handleRequest(Request req) {
        String user = authenticate(req);

        // Wir binden den 'user' und führen DANN die Methode aus.
        // Die Bindung gilt NUR für die Dauer der run-Methode!
        ScopedValue.where(SecurityContext.LOGGED_IN_USER, user)
                .run(() -> prozessiereBestellung());

        // Sobald wir hier ankommen, ist der Scope automatisch beendet
        // und der Wert restlos aus dem Speicher entfernt.
    }

    private void prozessiereBestellung() {
        // ... tiefe Aufrufhierarchie ...
        verschickeBestellbestaetigung();
    }
}
```

Tief unten in unserer Architektur, in einer Methode, die die Benutzer-Information eigentlich gar nicht als Parameter übergeben bekommen hat, können wir den Wert nun sicher abrufen:

```
public class EmailService {
    public void verschickeBestellbestaetigung() {
        // Prüfen, ob wir uns aktuell in einem gebundenen Scope
        // befinden
        if (SecurityContext.LOGGED_IN_USER.isBound()) {

            // Den Wert sicher abholen
```

```
            String currentUser = SecurityContext.LOGGED_IN_USER.get();
            System.out.println("Sende E-Mail an: " + currentUser);

        } else {
            System.out.println(
                    "Fehler: Kein Benutzer-Kontext gefunden!");
        }
    }
}
```

26.3.4 Fazit für Ihre Architektur

Merken Sie sich für moderne Java-Anwendungen eine einfache Faustregel: Sobald Sie anfangen, massiv mit Virtuellen Threads zu arbeiten, hat ThreadLocal in Ihrem Code ausgedient. Nutzen Sie stattdessen ScopedValue. Es macht Ihren Code nicht nur speichereffizienter, sondern zwingt Sie durch seine Unveränderlichkeit und den strikten Gültigkeitsbereich automatisch zu einer saubereren und fehlerresistenteren Architektur.

26.4 Der Friedhof der Thread-Methoden: stop(), suspend(), resume()

Wenn Sie in Ihrer IDE meinThread. eintippen, werden Ihnen eventuell Methoden wie stop(), suspend() (pausieren) und resume() (fortsetzen) vorgeschlagen. Sie sind jedoch alle durchgestrichen (als deprecated markiert) und in den neuesten Java-Versionen werfen sie sogar sofort eine UnsupportedOperationException bzw. wurden inzwischen sogar ganz aus der Klasse Thread entfernt.

Warum dürfen wir einen Thread nicht einfach mit stop() abschießen? Stellen Sie sich einen Thread wie einen Chirurgen vor, der gerade mitten in einer Operation am offenen Herzen (an einer komplexen Datenstruktur innerhalb eines synchronized-Blocks) ist. Wenn Sie stop() aufrufen, schießt die JVM den Chirurgen sofort ab. Das Schlimmste daran: Er lässt sofort alle Monitore (Schlüssel) fallen. Die Tür zum Operationssaal schwingt auf. Der nächste Thread betritt den Raum und findet einen Patienten (die Datenstruktur) in einem völlig zerstörten, halb-geänderten, inkonsistenten Zustand vor.

Sie dürfen Threads niemals hart von außen abbrechen. Die Lösung: Sie müssen Threads kooperativ beenden. Setzen Sie ein volatile boolean running = false Flag (wie in Kapitel 6) oder nutzen Sie thread.interrupt(), um den Thread aufzuwecken und ihn höflich zu bitten, seine Arbeit selbst sauber zu beenden und den "Operationssaal" ordentlich zu hinterlassen.

26.5 Das berüchtigte Anti-Pattern: Double-Checked Locking

Dies ist ein Stück Code, das Ihnen in Vorstellungsgesprächen für Senior-Positionen fast garantiert begegnen wird. Es geht um das Singleton-Pattern (eine Klasse, von der es nur exakt eine Instanz geben darf) und Lazy Initialization (das Objekt erst dann erzeugen, wenn es wirklich das erste Mal gebraucht wird).

Der naive Ansatz ist nicht thread-sicher:

```
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton(); // Race Condition! Mehrere
                                        // Threads könnten das tun.
        }
        return instance;
    }
}
```

Um das zu beheben, könnten wir die ganze Methode synchronized machen. Das kostet aber bei jedem späteren Aufruf Performance, obwohl die Instanz längst existiert. Clevere Entwickler erfanden in den 90er Jahren das Double-Checked Locking:

```
// ACHTUNG: Dieser Code ist (ohne volatile) in Java KAPUTT!
public class BrokenSingleton {
    private static BrokenSingleton instance;

    public static BrokenSingleton getInstance() {
        if (instance == null) { // 1. Check (ohne Lock für gute
                                // Performance)
            synchronized (BrokenSingleton.class) {
                if (instance == null) { // 2. Check (mit Lock, falls
                                        // jemand schneller war)
                    instance = new BrokenSingleton();
                }
            }
        }
        return instance;
    }
}
```

Warum ist das ein No-Go? Wegen des Instruction Reorderings (Befehlsumordnung) des Compilers (Kapitel 14)! Der Befehl instance = new Singleton() besteht intern aus drei Schritten: (1) Speicher reservieren, (2) Konstruktor ausführen, (3) Referenz auf den Speicher der Variable instance zuweisen. Die JVM darf Schritt 2 und 3 vertauschen! Wenn Thread A den Speicher reserviert und die Referenz zuweist, bevor der Konstruktor fertig ist, sieht Thread B beim "1. Check", dass instance nicht mehr

null ist. Thread B greift sich das Objekt und versucht damit zu arbeiten, greift aber auf ein halb-fertiges, leeres Objekt zu und stürzt ab!

Die Lösung: Entweder Sie fügen das Schlüsselwort volatile vor die Variable instance ein (das verbietet das Vertauschen der Schritte). Oder noch viel besser: Nutzen Sie einfach Enums für Singletons oder überlassen Sie Dependency Injection Frameworks (wie Spring) die Erstellung!

26.6 Unsichtbare Gefahr: Thread Leaks in Server-Umgebungen

Ein letztes No-Go betrifft das Lebenszyklus-Management. Wenn Sie einen eigenen ExecutorService (Thread-Pool) erstellen und diesen am Ende nicht mit shutdown() beenden (Kapitel 8), laufen die Threads im Hintergrund weiter.

Wenn dieser Code in einer Methode eines Webservers (z. B. Tomcat) aufgerufen wird und bei jeder Web-Anfrage (Request) ein neuer, ungeschlossener Pool erzeugt wird, haben Sie ein sogenanntes Thread Leak. Ihr Server erzeugt hunderte neue Threads pro Minute, räumt sie nie auf und stürzt nach wenigen Stunden mit einem OutOfMemoryError: unable to create new native thread unwiderruflich ab. Faustregel: Wer einen Thread oder Pool startet, ist auch verantwortlich, ihn wieder zu schließen (ideal im finally-Block oder via Try-With-Resources, wie bei Virtuellen Threads gesehen).

Fazit: Mit diesen Best Practices heben Sie sich deutlich von Anfängern ab. Sie wissen, dass Unveränderlichkeit der beste Schutz ist, dass man Threads nicht abschießt und warum klug aussehende Optimierungen wie Double-Checked Locking oft trügerisch sind.

27 Über den Tellerrand – Alternative Paradigmen

Bis zu diesem Punkt in unserem Buch haben wir uns fast ausschließlich in einem bestimmten Paradigma bewegt: dem Shared-State Concurrency Model (Nebenläufigkeit mit geteiltem Zustand).

Unsere Grundannahme war immer: Es gibt Daten im Speicher (z. B. ein Bankkonto oder eine Liste), und mehrere Threads wollen gleichzeitig darauf zugreifen. Wir haben dieses Problem gelöst, indem wir den Zugriff durch Sperren (synchronized, ReentrantLock) oder durch clevere Hardware-Tricks (AtomicInteger) koordiniert haben.

Wenn Sie jedoch Systeme bauen, die nicht nur auf einem einzigen Server, sondern auf einem Cluster von hundert Servern laufen (verteilte Systeme), stößt dieses Modell an seine absoluten Grenzen. Sie können kein synchronized-Lock über drei Rechenzentren hinweg setzen. Um dieses Skalierungsproblem zu lösen, hat die Industrie alternative Paradigmen entwickelt, die völlig ohne klassische Sperren auskommen.

27.1 Das Actor-Modell: Nachrichten statt Sperren

Das Actor-Modell (Akteur-Modell) stellt die bisherige Welt auf den Kopf. Die Grundregel lautet hier: Share Nothing! (Teile absolut nichts!)

Anstatt Daten in die Mitte zu legen und Threads darum kämpfen zu lassen, kapseln wir den Zustand vollständig in unabhängigen Einheiten, den sogenannten Actors (Akteuren).

Die Prinzipien eines Actors:

1. Privater Zustand: Ein Actor besitzt seinen eigenen Zustand (z.B. den Kontostand). Kein anderer Thread, kein anderes Objekt und kein anderer Actor kann jemals direkt auf diesen Zustand zugreifen oder ihn lesen.
2. Die Mailbox: Jeder Actor hat einen eigenen "Briefkasten" (eine Warteschlange).
3. Nachrichtenbasierte Kommunikation (Message Passing): Wenn Actor A möchte, dass Actor B seinen Kontostand ändert, darf er keine Methode auf B aufrufen. Er muss eine asynchrone Nachricht ("Überweise 100 Euro") in die Mailbox von B werfen.
4. Sequenzielle Verarbeitung: Der Actor B arbeitet seine Mailbox streng sequenziell ab – eine Nachricht nach der anderen.

Warum ist das so genial? Da ein Actor immer nur genau eine Nachricht zur selben Zeit verarbeitet, gibt es innerhalb des Actors niemals Interleaving, Race Conditions oder Lost Updates! Sie können den Code innerhalb des Actors völlig sequenziell (ohne ein einziges synchronized) schreiben, obwohl das Gesamtsystem aus Millionen von Actors besteht, die hochgradig parallel auf hunderten Servern laufen.

Praxisbezug: Dieses Modell wurde ursprünglich durch die Programmiersprache Erlang (die z.B. WhatsApp antreibt) berühmt. In der Java-Welt ist das Framework Akka[8] die bekannteste und mächtigste Implementierung dieses Paradigmas.

27.2 Reaktive Programmierung und die Java Flow-API

Wenn wir über moderne, hochgradig skalierbare Systeme sprechen, stoßen wir unweigerlich auf das Konzept der Reaktiven Programmierung (Reactive Streams). Anstatt Daten aktiv abzufragen und dabei Threads zu blockieren (das sogenannte Pull-Prinzip), dreht die reaktive Programmierung den Spieß um: Das System reagiert auf Datenströme, die kontinuierlich "gepusht" werden (Push-Prinzip) – vergleichbar mit einem Fließband in einer Fabrik, das sich nur dann bewegt, wenn auch wirklich Bauteile ankommen.

Lange Zeit war dieses Paradigma eine Domäne externer Bibliotheken wie RxJava[9], Akka oder Project Reactor[10]. Doch weil dieses Konzept für die Verarbeitung massiver, asynchroner Datenströme so wichtig wurde, hat man sich entschieden, einen offiziellen Standard direkt ins JDK aufzunehmen.

27.2.1 Der JDK-Standard: java.util.concurrent.Flow

Seit Java 9 ist die Spezifikation der Reactive Streams nativ in Java integriert – und zwar in Form der Klasse java.util.concurrent.Flow. Diese Klasse ist im Grunde ein Container, der vier essenzielle Schnittstellen (Interfaces) definiert, die das Zusammenspiel von reaktiven Datenströmen standardisieren:

1. Flow.Publisher<T> (Der Produzent): Das ist die Quelle unseres Datenstroms. Ein Publisher produziert Elemente vom Typ T und stellt sie bereit. Er besitzt nur eine einzige Methode: subscribe(), mit der sich Interessenten anmelden können.
2. Flow.Subscriber<T> (Der Konsument): Das Gegenstück zum Publisher. Der Subscriber empfängt die Elemente. Er definiert Methoden, um auf neue Daten

[8] https://github.com/akka/akka-core
[9] https://github.com/ReactiveX/RxJava
[10] https://github.com/reactor/reactor-core

(onNext), Fehler (onError) oder das erfolgreiche Ende des Datenstroms (onComplete) zu reagieren.

3. Flow.Subscription (Die Verbindung): Das ist das eigentliche Geheimnis der Flow-API. Wenn sich ein Subscriber bei einem Publisher anmeldet, erhält er als Antwort ein Subscription-Objekt. Über dieses Objekt kommunizieren die beiden miteinander.
4. Flow.Processor<T, R> (Der Vermittler): Ein Processor ist Publisher und Subscriber in einem. Er nimmt Daten vom Typ T entgegen, transformiert sie (z. B. filtert oder mappt er sie) und reicht sie als Typ R an den nächsten Subscriber weiter.

27.2.2 Das Kernkonzept: Backpressure (Rückstau)

Warum reicht nicht einfach das klassische Observer-Pattern, das wir seit Jahrzehnten kennen? Die Antwort lautet Backpressure (zu Deutsch: Gegendruck oder Rückstau-Kontrolle).

Stellen Sie sich vor, ein Publisher liest Sensordaten ein und feuert 100.000 Ereignisse pro Sekunde ab. Der Subscriber, der diese Daten in eine Datenbank schreiben soll, schafft aber nur 1.000 pro Sekunde. Ohne Backpressure würde der Subscriber gnadenlos überflutet, sein Arbeitsspeicher würde volllaufen und die Anwendung mit einem OutOfMemoryError abstürzen.

Die Flow.Subscription löst dieses Problem elegant. Der Subscriber ruft auf der Subscription die Methode request(long n) auf. Er sagt dem Publisher damit explizit: "Sende mir maximal n Elemente. Danach warte, bis ich explizit neue anfordere." Der Konsument diktiert also das Tempo. Das System atmet.

27.2.3 Die Flow-API in der Praxis

Es ist wichtig zu verstehen, dass die Flow-API im JDK in erster Linie eine Spezifikation (Interfaces) ist, um Interoperabilität zu schaffen. Das JDK selbst liefert (mit Ausnahme des sehr nützlichen java.util.concurrent.SubmissionPublisher) keine gigantische Bibliothek an fertigen reaktiven Operatoren mit.

Der große Vorteil ist jedoch: Wenn Sie heute externe reaktive Frameworks nutzen, implementieren diese alle unter der Haube die Java Flow-API. Sie können also einen Publisher aus Spring WebFlux nahtlos an einen Subscriber aus RxJava übergeben. Die Flow-API ist die standardisierte Brücke, die alle reaktiven Frameworks in der Java-Welt miteinander verbindet.

27.2.4 Code-Beispiel

In diesem Beispiel nutzen wir die Standard-Implementierung SubmissionPublisher, um Zahlen zu senden, und implementieren einen eigenen Subscriber, der diese verarbeitet.

```
import java.util.concurrent.Flow;
import java.util.concurrent.SubmissionPublisher;

public class FlowApiDemo {

    public static void main(String[] args)
            throws InterruptedException {
        // 1. Der Publisher: Erzeugt Daten und schickt sie an
        // Subscriber
        try (SubmissionPublisher<Integer> publisher =
               new SubmissionPublisher<>()) {

            // 2. Der Subscriber: Definiert, was mit den Daten
            // passiert
            Flow.Subscriber<Integer> subscriber = new Flow.Subscriber<>() {
                private Flow.Subscription subscription;

                @Override
                public void onSubscribe(
                        Flow.Subscription subscription) {
                    this.subscription = subscription;
                    // Backpressure: Wir fordern das erste Element an
                    subscription.request(1);
                }

                @Override
                public void onNext(Integer item) {
                    System.out.println("Empfangen: " + item);
                    // Nach Verarbeitung fordern wir das nächste
                    // Element an
                    subscription.request(1);
                }

                @Override
                public void onError(Throwable throwable) {
                    throwable.printStackTrace();
                }

                @Override
                public void onComplete() {
                    System.out
                            .println("Fertig! Alle Daten empfangen.");
                }
            };

            // 3. Subscriber beim Publisher registrieren
            publisher.subscribe(subscriber);

            // 4. Daten senden
            System.out.println("Sende Daten...");
            publisher.submit(10);
            publisher.submit(20);
            publisher.submit(30);
```

```
            // Kurz warten, da die Verarbeitung asynchron in anderen
            // Threads läuft
            Thread.sleep(1000);
        }
    }
}
```

Die wichtigsten Punkte im Detail:

1. onSubscribe: Dies ist der "Handschlag". Der Subscriber erhält ein Subscription-Objekt. Wichtig: Ohne den Aufruf von subscription.request(n) passiert gar nichts!
2. request(n) (Backpressure): Hier steuert der Subscriber den Datenfluss. Er sagt dem Publisher: "Ich bin bereit für genau *n* weitere Elemente." Das verhindert, dass ein langsamer Consumer von einem schnellen Producer überrannt wird.
3. SubmissionPublisher: Eine praktische Standard-Implementierung des Publisher-Interfaces, die bereits einen Thread-Pool nutzt, um die Daten asynchron auszuliefern.
4. Asynchronität: Beachten Sie, dass onNext in einem anderen Thread aufgerufen wird als publisher.submit. Deshalb ist das Thread.sleep am Ende des Beispiels nötig, damit das Hauptprogramm nicht beendet wird, bevor die Daten verarbeitet wurden.

27.3 Wann wählt man welches Paradigma?

Um Ihnen eine Orientierung zu geben, fassen wir die Architekturentscheidungen kurz zusammen:

- Klassische Java-Nebenläufigkeit (Threads, Locks, Concurrent Collections): Perfekt für Anwendungen auf einem einzelnen Server, bei denen Sie maximale Kontrolle über die Hardware und extreme Performance bei niedrigster Latenz (Verzögerung) benötigen.
- Virtuelle Threads (Project Loom): Der neue Standard für klassische "Request-Response"-Architekturen (wie Standard-Webserver oder Datenbankabfragen). Sie ermöglichen massiven Durchsatz blockierender Operationen bei extrem simplem Code.
- Actor-Modell (Akka): Die beste Wahl für stark verteilte Systeme (über viele Server hinweg), bei denen einzelne Entitäten (z.B. "Ein Warenkorb", "Ein Spieler im Multiplayer-Game") komplexen internen Zustand haben, der geschützt werden muss.

- Reactive Streams (RxJava / Reactor): Ideal für stark datengetriebene Anwendungen, bei denen Ereignisse asynchron durch komplexe Pipelines fließen (z.B. Echtzeit-Datenverarbeitung, Aktienkurse, IoT-Sensordaten) und Backpressure zwingend erforderlich ist.

Fazit: Als moderner Software-Ingenieur müssen Sie nicht nur wissen, wie man einen Hammer (synchronized) bedient, sondern auch, dass es für bestimmte Aufgaben vielleicht besser ist, einen Schraubendreher (Actors) zu verwenden.

28 Die physikalischen Grenzen und Javas Zukunft

Herzlichen Glückwunsch! Sie haben eine lange, komplexe und faszinierende Reise hinter sich. Von den maschinennahen Grundlagen der Betriebssystem-Threads über feingranulare Locks und blockierungsfreie Algorithmen bis hin zu hochgradig skalierbaren, virtuellen Threads und reaktiven Architekturen.

Zum Abschluss dieses Buches müssen wir uns jedoch einer unbequemen Wahrheit stellen: Egal wie perfekt unser Code designt ist, wir unterliegen den Gesetzen der Physik. Mehr CPU-Kerne bedeuten nicht automatisch unendlich viel mehr Leistung.

28.1 Die bittere Logik: Warum mehr Kerne nicht immer mehr Tempo bedeuten

Eine der häufigsten Fehlannahmen in der Informatik lautet: „Wenn mein Programm auf einem Kern 100 Sekunden braucht, dann dauert es auf einem 100-Kern-Prozessor nur noch 1 Sekunde."

In der Realität ist das fast nie der Fall. Der Grund dafür ist simpel: Jedes Programm besteht aus Aufgaben, die sich wunderbar aufteilen lassen, und Aufgaben, die zwingend nacheinander (sequenziell) ablaufen müssen. Bevor Sie zum Beispiel 100 Webseiten gleichzeitig herunterladen können, müssen Sie erst einmal die Liste der Adressen aus einer Datei lesen – und das kann nur ein einziger Thread zurzeit tun.

28.1.1 Die Analogie: Das Hausbau-Dilemma

Stellen Sie sich vor, Sie bauen ein Haus. Viele Aufgaben lassen sich perfekt parallelisieren: Wenn Sie 10 Maurer statt einem einsetzen, steht die Wand fast zehnmal so schnell.

Doch es gibt Arbeiten, die sich nicht beschleunigen lassen, egal wie viel Personal Sie haben:

- Das Fundament muss zwei Wochen trocknen, bevor darauf gemauert werden kann.
- Der Bauantrag muss erst genehmigt werden, bevor der erste Bagger rollt.

Selbst wenn Sie 1.000 Arbeiter auf die Baustelle schicken, wird das Haus niemals in einer Stunde fertig sein, weil das Fundament trotzdem seine Zeit zum Trocknen braucht. Diese festen, unteilbaren Wartezeiten sind der „sequenzielle Anteil".

28.1.2 Die schockierende Konsequenz

Der Erfolg Ihrer Parallelisierung wird nicht durch die Menge der Prozessoren bestimmt, sondern durch die Größe Ihres sequenziellen Flaschenhalses.

Nehmen wir an, Ihr Programm ist extrem gut optimiert: 95 % der Arbeit können perfekt auf beliebig viele Kerne verteilt werden. Nur winzige 5 % des Codes müssen starr nacheinander ablaufen (z. B. das Initialisieren der Datenbankverbindung am Start).

Was passiert nun, wenn Sie theoretisch unendlich viele Prozessorkerne zur Verfügung hätten? Die 95 % der Arbeit würden in fast Nullzeit erledigt sein. Aber die 5 % des sequenziellen Codes bleiben hartnäckig bestehen. Sie müssen immer noch nacheinander abgearbeitet werden.

Das bedeutet: Selbst mit einem Supercomputer, der eine Million Kerne besitzt, wird Ihr Programm niemals schneller sein als das Zwanzigfache der ursprünglichen Zeit. Warum genau das Zwanzigfache? Weil die 5 % Restzeit den Deckel bilden (100 % Gesamtzeit geteilt durch 5 % Unteilbarkeit = 20).

Die Lehre für die Praxis: Echte Performance-Steigerung bedeutet oft nicht, noch mehr Threads auf ein Problem zu werfen. Es bedeutet stattdessen, den „unbeweglichen“ Teil des Programms – den sequenziellen Flaschenhals – so klein wie möglich zu machen. Wer den unteilbaren Anteil von 5 % auf 2 % drückt, erreicht mehr als derjenige, der die Anzahl der Prozessoren verdoppelt.

28.2 Die Speichermauer (The Memory Wall)

Ein oft unterschätztes physikalisches Problem bei der Parallelisierung ist die sogenannte Memory Wall. Während die Rechengeschwindigkeit von CPUs über Jahrzehnte hinweg rasant gestiegen ist, hinkt die Zugriffsgeschwindigkeit auf den Hauptspeicher (RAM) weit hinterher.

Stellen Sie sich eine moderne CPU wie einen Formel-1-Motor vor, der 300 km/h fahren könnte. Der Hauptspeicher ist jedoch ein alter Tankwagen, der den Kraftstoff nur mit 30 km/h liefert. Das Ergebnis: Der Rennwagen steht die meiste Zeit an der Box und wartet auf Benzin. In der Informatik nennen wir das einen CPU-Stall (ein Stocken).

28.2.1 Das "Verstreute-Daten-Problem" in Java

In Java schlägt dieses Problem besonders hart zu, weil das objektorientierte Design oft gegen die Arbeitsweise der CPU-Caches arbeitet.

Stellen Sie sich vor, Sie haben ein Array mit 1.000 Punkt-Objekten (Point[]). In einer idealen Welt für die CPU lägen alle X- und Y-Koordinaten dieser Punkte wie Perlen auf einer Schnur direkt hintereinander im Speicher. Die CPU könnte sie in einem Rutsch in den Cache laden (Pre-fetching).

In Java sieht die Realität jedoch so aus:

1. Im Array selbst liegen nur Referenzen (quasi die Hausnummern/Adressen) der Objekte.
2. Die eigentlichen Point-Objekte liegen völlig verstreut irgendwo im riesigen Hauptspeicher (dem Heap).

28.2.2 Das "Pointer-Chasing" (Zeiger-Jagd)

Wenn ein paralleler Stream nun über dieses Array iteriert, passiert Folgendes:

- Die CPU liest die erste Adresse aus dem Array.
- Sie muss den "Blick abwenden" und im fernen RAM nachschauen, was an dieser Adresse steht.
- Während sie auf das Objekt wartet, ist der CPU-Cache nutzlos, da die Daten des nächsten Punktes wahrscheinlich ganz woanders liegen.

Das nennt man Pointer-Chasing. Die CPU springt kreuz und quer durch den Arbeitsspeicher. Da Caches am effizientesten arbeiten, wenn Daten lokal (direkt nebeneinander) liegen, kommt es ständig zu sogenannten Cache Misses.

Die bittere Konsequenz für die Parallelisierung: Sie können noch so viele Threads auf ein Problem werfen – wenn alle Threads gleichzeitig versuchen, über den schmalen Pfad zum RAM auf verstreute Daten zuzugreifen, verpufft der Geschwindigkeitsvorteil. Die Threads stehen sich gegenseitig im Weg, nicht weil sie nicht rechnen können, sondern weil die "Speichermauer" den Datenfluss begrenzt.

28.3 Javas Antwort auf die Zukunft

Die Architekten der Java Virtual Machine sind sich dieser Grenzen bewusst und arbeiten in großen, mehrjährigen Projekten an revolutionären Lösungen.

28.3.1 Project Valhalla: Value Objects

Um die Speichermauer einzureißen und das zermürbende Pointer-Chasing zu beenden, arbeitet Java an einer der tiefgreifendsten architektonischen Neuerungen seiner Geschichte: Project Valhalla.

Das Kernstück dieses Projekts sind die sogenannten Value Objects (Werte-Objekte). Das Entwickler-Mantra von Valhalla lautet: "Codes like a class, works like an int" (Liest sich wie eine Klasse, arbeitet wie ein primitiver Datentyp).

Bisher zwingt uns Java bei jedem noch so kleinen Objekt einen immensen unsichtbaren Overhead auf. Jedes Objekt besitzt zwingend einen Header für die Garbage Collection, eine eindeutige Speicher-Identität und einen eigenen Monitor (Lock) für eventuelle synchronized-Zugriffe. Bei Value Objects wird dieser Ballast komplett über Bord geworfen. Sie deklarieren bewusst, dass sie keine eigene Identität benötigen und unveränderlich (immutable) sind. Da sie keine Identität haben, brauchen sie auch keinen Monitor-Lock – sie sind von Natur aus absolut thread-sicher.

Der entscheidende Durchbruch für unsere parallelen Programme ist jedoch das sogenannte Memory Flattening (Speicherabflachung).

Erinnern wir uns an das Point[]-Array aus Abschnitt 28.2. Wenn wir die Klasse Point in Zukunft als Value Object definieren, ändert die Java Virtual Machine radikal, wie dieses Array im Speicher abgelegt wird:

Das Array enthält nun keine Referenzen mehr. Stattdessen nimmt die JVM die reinen Nutzdaten der Objekte (die X- und Y-Koordinaten) und legt sie als flachen, durchgehenden Block direkt hintereinander in den Speicher. Aus dem verstreuten Referenz-Chaos wird auf Hardware-Ebene genau das, was sich die CPU wünscht: echte Perlen auf einer Schnur (X, Y, X, Y, X, Y...).

Der Effekt auf die Hardware und Parallelität: Dadurch entfällt das zuvor beschriebene Pointer-Chasing komplett. Wenn ein Thread nun über das Array iteriert und den ersten Punkt liest, zieht der Hardware-Prefetcher der CPU automatisch die direkt angrenzenden Koordinaten der nächsten Punkte als komplette "Cache Line" in den rasend schnellen L1-Cache. Es gibt keine Cache Misses mehr und der CPU-Cache wird wieder zu einer echten Waffe.

Für datenintensive, parallele Algorithmen ist dieser Wegfall des Speicher-Flaschenhalses ein gigantischer Gewinn. Die Threads müssen nicht mehr am Nadelöhr des RAMs aufeinander warten, sondern werden ununterbrochen mit Daten gefüttert. Die Formel-1-Motoren können endlich ihre volle Leistung auf die Straße bringen.

28.3.2 Datenparallelität im Kleinen: Die Vector API (SIMD)

In den vorherigen Kapiteln haben wir uns intensiv damit beschäftigt, wie wir Aufgaben auf mehrere Threads – und damit auf mehrere CPU-Kerne – verteilen. Das ist die klassische Parallelität im Großen. Doch es gibt noch eine weitere, oft übersehene Ebene der Leistungssteigerung: tief unten auf der Hardware-Ebene innerhalb eines einzigen CPU-Kerns.

Moderne Prozessoren beherrschen eine Technik namens SIMD (Single Instruction, Multiple Data). Das bedeutet: Anstatt in einem Taktzyklus nur zwei einzelne Zahlen zu addieren, kann eine CPU mit speziellen, breiten Vektor-Registern (wie AVX-256 oder AVX-512) gleich vier, acht oder gar sechzehn Zahlenpaare absolut gleichzeitig verarbeiten – mit einem einzigen Maschinenbefehl.

Um bei unserer Küchenmetapher zu bleiben: Wenn Multithreading bedeutet, dass wir vier Köche einstellen, um vier Karotten zu schneiden, dann ist SIMD ein einzelner Koch, der ein spezielles, extrem breites Messer besitzt, mit dem er vier Karotten mit nur einem einzigen, mächtigen Schlag simultan zerkleinert.

Lange Zeit war diese hardwarenahe Optimierung in Java eine Art "Blackbox", die höchstens vom Just-in-Time-Compiler (JIT) unter der Haube automatisch – aber oft unvorhersehbar – angewendet wurde (Auto-Vectorization). Mit der Vector API (als Teil von Project Panama) ändert sich das: Java gibt Entwicklern nun eine explizite Schnittstelle an die Hand, um diese Vektor-Hardware gezielt und plattformunabhängig anzusteuern.

Wenn Sie beispielsweise zwei gigantische Arrays mathematisch miteinander verrechnen müssen (ein typischer Fall bei KI-Anwendungen, Matrizen-Multiplikation oder der Bildverarbeitung), schreiben Sie keine klassische for-Schleife mehr. Stattdessen nutzen Sie die Vector API, um sogenannte Lanes zu definieren und ganze Blöcke von Daten auf einen Schlag in der Hardware zu berechnen. Das Ergebnis ist ein gigantischer Performance-Schub, ganz ohne den Overhead, den die Verwaltung zusätzlicher Threads verursachen würde.

Dass sich die Vector API aktuell noch in der Entwicklungs- bzw. Inkubator-Phase befindet, zeigt eindrucksvoll, wie eng die großen JDK-Erneuerungen miteinander verwoben sind: Die Java-Architekten warten für die finale Fertigstellung der Vector API auf den Abschluss von Project Valhalla (siehe vorheriger Abschnitt).

Warum? SIMD-Operationen sind so unglaublich schnell, dass der kleinste Speicher-Overhead sie sofort wieder ausbremsen würde. Solange Java-Arrays noch aus Objekten

bestehen können, die auf dem Heap verstreut sind und teuer entpackt werden müssen (Boxing/Unboxing), läuft die Vector API mit angezogener Handbremse. Erst wenn Valhalla die speichereffizienten Value Classes liefert, wird die Vector API ihr volles, atemberaubendes Potenzial für die Datenparallelität entfalten können.

28.3.3 Project Loom: Strukturierte Nebenläufigkeit (Structured Concurrency)

Wir haben die bahnbrechenden Virtuellen Threads bereits in Kapitel 13 kennengelernt. Sie erlauben es uns, Blockaden zu ignorieren und völlig problemlos Millionen von Threads gleichzeitig zu starten. Doch das gigantische Projekt *Loom* ist damit noch nicht abgeschlossen. Denn wenn man plötzlich Millionen von Threads starten kann, steht man vor einem massiven organisatorischen Problem: Wie behält man die Kontrolle über dieses Heer an Aufgaben?

In der traditionellen Java-Welt (vor Project Loom) war Nebenläufigkeit weitgehend unstrukturiert. Wenn ein Thread (der „Vater“) eine Teilaufgabe an einen ExecutorService übergab, wurde ein neuer Thread (das „Kind“) gestartet. Vater und Kind waren ab diesem Moment völlig unabhängig voneinander unterwegs (Fire-and-Forget). Das führte in der Praxis zu massiven Problemen:

1. Thread Leaks (Verwaiste Threads): Wenn der Vater-Thread durch eine Exception abstürzte oder abgebrochen wurde, lief der Kind-Thread im Hintergrund unbemerkt weiter. Er verschwendete CPU-Zeit, blockierte Datenbankverbindungen und erzeugte sogenannte „Geister-Threads“, die den Server langsam in die Knie zwangen.
2. Fehlerbehandlung: Wenn der Kind-Thread fehlschlug, war es extrem schwer, den Vater-Thread rechtzeitig darüber zu informieren, um die Gesamtaufgabe geordnet abzubrechen.

Um bei unserer Küchenmetapher zu bleiben: Stellen Sie sich einen Chefkoch (Vater-Thread) vor, der drei Sous-Chefs (Kind-Threads) anweist, jeweils die Vorspeise, das Hauptgericht und das Dessert für einen VIP-Gast zuzubereiten. Nach fünf Minuten verbrennt der erste Sous-Chef die Vorspeise irreparabel. Der VIP-Gast verlässt wutentbrannt das Restaurant. In der unstrukturierten Nebenläufigkeit kochen die anderen beiden Sous-Chefs ahnungslos weiter, verbrauchen teure Zutaten und blockieren die Herde, obwohl das Menü längst hinfällig ist.

Die Lösung: Ein Block, ein gemeinsames Schicksal

Genau dieses Chaos beseitigt die Strukturierte Nebenläufigkeit (Structured Concurrency), die Java mit der neuen Klasse StructuredTaskScope einführt. Die Grundidee ist von der normalen, sequenziellen Programmierung entlehnt: Der Lebenszyklus eines Threads wird strikt an den Code-Block (den lexikalischen Gültigkeitsbereich) gebunden, in dem er gestartet wurde.

Wenn Thread A nun drei Unter-Threads startet, werden diese logisch zu einer einzigen Einheit gebündelt. Der Code-Block kann erst verlassen werden, wenn alle Unter-Aufgaben abgeschlossen sind. Es gilt das Prinzip „Alle für einen, einer für alle“ – ähnlich wie bei einer Datenbanktransaktion:

- Short-Circuiting bei Fehlern: Wenn auch nur ein einziger Unter-Thread mit einer Exception abstürzt, greift der StructuredTaskScope sofort ein. Er sendet automatisch ein Abbruchsignal (Interrupt) an alle noch laufenden Geschwister-Threads. Die nutzlose Arbeit wird sofort gestoppt (die anderen Sous-Chefs legen die Messer nieder).
- Automatisches Aufräumen: Selbst wenn der Vater-Thread von außen abgebrochen wird, propagiert sich dieser Abbruch automatisch kaskadenartig nach unten an alle Unter-Threads.

Es gibt keine herumgeisternden, verwaisten Hintergrund-Threads mehr. Der Code wird wieder so lesbar, nachvollziehbar und fehlerresistent wie ganz normaler sequenzieller Code. Zusammen mit den Virtuellen Threads vollendet die Strukturierte Nebenläufigkeit damit Javas Vision für die moderne, hochskalierbare und sichere Softwarearchitektur der Zukunft.

Teil IV:

Die Choreografie in der Praxis –

Das Großküchen-Projekt

29 Überblick über das Großküchen-Projekt

Sie haben es weit gebracht! In den ersten drei Teilen dieses Buches haben wir uns intensiv mit den theoretischen Fundamenten, den abstrakten Konzepten und den gefährlichen Fallstricken der nebenläufigen Programmierung in Java beschäftigt. Sie kennen nun die Unterschiede zwischen Prozessen und Threads, verstehen die Mechanismen von Monitoren und Sperren und haben die modernen High-Level-APIs des java.util.concurrent-Pakets kennengelernt.

Doch graue Theorie allein macht noch keinen Meister der Nebenläufigkeit. Die wahre Herausforderung – und die wahre Kunst – besteht darin, diese einzelnen Puzzleteile zu einer funktionierenden, performanten und sicheren Gesamtarchitektur zusammenzufügen.

Genau das werden wir in diesem vierten und letzten Teil des Buches tun.

29.1 Die Rückkehr in die Großküche

Erinnern Sie sich an die Metapher aus dem Vorwort dieses Buches? Wir haben die CPU mit einer Großküche verglichen, in der die Rechenkerne die Arbeitsstationen und die Threads unsere Köche sind.

In den folgenden sieben Kapiteln werden wir diese Metapher in echten, lauffähigen Java-Code übersetzen. Wir eröffnen gemeinsam die Pizzeria „Concurrency“. Wir werden eine vollständige Restaurantsimulation programmieren – angefangen bei der Bestellannahme über die Zubereitung bis hin zur Auslieferung und Abrechnung.

Wir werden das System nicht von Beginn an perfekt bauen. Stattdessen wählen wir einen iterativen Ansatz. Wir starten mit einer naiven, sequenziellen Lösung, wie sie ein Anfänger schreiben würde, und beobachten, wie sie unter Last zusammenbricht oder ineffizient arbeitet. Anschließend werden wir das System Schritt für Schritt refaktorieren und mit den Werkzeugen aus den Theoriekapiteln optimieren, bis wir am Ende eine hochmoderne, cloud-native Architektur vor uns haben.

29.2 Die 7 Iterationen auf unserem Weg zur Perfektion

Dieser Praxisteil ist in sieben Ausbaustufen (Iterationen) unterteilt. Jedes der folgenden Kapitel widmet sich einer Iteration und löst ein spezifisches Problem der Nebenläufigkeit:

- **Kapitel 30 (Iteration 1): Die Pizzeria eröffnet (Sequenziell & Erste Threads)** Wir starten mit einem einzigen Koch (dem Main-Thread), der völlig überlastet ist. Wir lernen, wie wir erste Hilfsköche (Runnable und Thread)

einstellen, um Pizzen parallel zu backen, und wie wir auf ihren Feierabend warten (join()).

- **Kapitel 31 (Iteration 2): Schichtplan und Kellner (Thread-Pools & Futures)** Das ständige Einstellen und Feuern von Köchen verbraucht zu viele Ressourcen. Wir strukturieren um: Wir führen festangestellte Köche in einem Thread-Pool (ExecutorService) ein und geben unseren Kellnern asynchrone Abholscheine (Callable und Future) für die fertigen Gerichte.
- **Kapitel 32 (Iteration 3): Chaos in der Speisekammer (Geteilter Zustand & Locks)** Sobald mehrere Köche auf dieselben begrenzten Zutaten zugreifen, entsteht ein gefährlicher Shared State. Wir provozieren absichtlich eine Race Condition in unserem Inventar und retten die Speisekammer anschließend durch synchronized-Monitore und performante Lese-/Schreibsperren (ReentrantReadWriteLock).
- **Kapitel 33 (Iteration 4): Die Ticket-Schiene (Producer-Consumer & Concurrent Collections)** Um die Kellner und Köche zu entkoppeln, hängen wir eine Bestell-Schiene auf. Wir implementieren das professionelle Producer-Consumer-Pattern mittels einer BlockingQueue und optimieren unsere Registrierkasse mit einer sperrfreien (lock-free) atomaren Variablen (AtomicInteger).
- **Kapitel 34 (Iteration 5): Meisterhafte Koordination (Barrieren & Semaphoren)** Ein VIP-Tisch verlangt, dass alle vier Gänge exakt gleichzeitig serviert werden, während in der Küche ein Ofen ausfällt. Wir zähmen das Timing unserer Threads millimetergenau mit einer CyclicBarrier und regeln den Zugriff auf die knappen Öfen mit einem Semaphore.
- **Kapitel 35 (Iteration 6): Die landesweite Kette (Project Loom & CompletableFuture)** Unsere Pizzeria geht online und muss plötzlich 50.000 Bestellungen gleichzeitig verarbeiten. Normale Threads crashen den Server. Wir revolutionieren unsere Architektur durch Virtuelle Threads (Project Loom) und bauen ein asynchrones, nicht-blockierendes Fließband mit CompletableFuture.
- **Kapitel 36 (Iteration 7): Gesundheitsamt und Sterne-Bewertung (Testing & Benchmarking)** Zum großen Finale müssen wir beweisen, dass unser asynchrones System fehlerfrei und schnell ist. Wir lernen, wie man den Unwägbarkeiten von Threads beim Testing (Heisenbugs) mit dem Awaitility-Framework begegnet und wie wir die Performance unserer Kasse mit dem JMH (Java Microbenchmark Harness) wissenschaftlich exakt messen.

29.3 Bereit für die Praxis?

Dieses Projekt ist so konzipiert, dass Sie den Code auf Ihrem eigenen Rechner mittippen und ausführen können. Es wird Ihnen helfen, das berüchtigte "Aha"-Erlebnis zu haben, wenn Sie sehen, wie die theoretischen Puzzleteile nahtlos ineinandergreifen.

Binden Sie sich die Schürze um, schalten Sie die IDE ein – lassen Sie uns kochen! Blättern Sie um zu Kapitel 30 und lassen Sie uns die Türen der Pizzeria "Concurrency" öffnen.

30 Iteration 1 – Die Pizzeria eröffnet (Sequenziell & Erste Threads)

Willkommen in der Pizzeria „Concurrency“! In diesem und den folgenden Kapiteln werden wir all das, was Sie bisher in der Theorie über Nebenläufigkeit gelernt haben, in die Praxis umsetzen. Wir bauen gemeinsam ein lauffähiges System auf, beginnend bei einem sehr naiven Ansatz, den wir iterativ zu einer hochskalierbaren, fehlerresistenten Architektur weiterentwickeln.

Denken Sie an unsere Metapher aus der Einleitung: Die CPU ist unsere Großküche. Bisher stand dort nur ein einziger Koch – der Main-Thread. Schauen wir uns an, was passiert, wenn dieser Koch ein Restaurant ganz alleine betreiben muss.

30.1 Der Flaschenhals: Die sequenzielle Pizzeria

Es ist Eröffnungstag. Drei Kunden betreten den Laden und bestellen kurz hintereinander eine Pizza Margherita, eine Salami und eine Funghi. Unser Chefkoch (der main-Thread) nimmt die Bestellungen entgegen und fängt an zu arbeiten.

```
public class PizzeriaSequenziell {

    public static void main(String[] args) {
        System.out.println(
                "Chef (Main-Thread): Pizzeria ist geöffnet!");

        long startZeit = System.currentTimeMillis();

        backePizza("Margherita");
        backePizza("Salami");
        backePizza("Funghi");

        long endZeit = System.currentTimeMillis();
        System.out.println("Chef: Alle Pizzen fertig nach "
                + (endZeit - startZeit) + " ms. Wir schließen!");
    }

    static void backePizza(String name) {
        System.out.println("Chef backt " + name + "...");
        try {
            // Wir simulieren die Backzeit von 2 Sekunden
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("Chef: " + name + " ist fertig!");
    }
}
```

Die Ausgabe dieses Programms ist ernüchternd:

```
Chef (Main-Thread): Pizzeria ist geöffnet!
Chef backt Margherita...
Chef: Margherita ist fertig!
```

```
Chef backt Salami...
Chef: Salami ist fertig!
Chef backt Funghi...
Chef: Funghi ist fertig!
Chef: Alle Pizzen fertig nach 6030 ms. Wir schließen!
```

Die Zubereitung dauert über 6 Sekunden. Der arme Kunde, der die Pizza Funghi bestellt hat, musste warten, bis die Margherita und die Salami nacheinander gebacken wurden, obwohl noch drei weitere Öfen in der Küche leer stehen (unsere restlichen CPU-Kerne)! Dies ist die klassische Einschränkung deterministischer, sequenzieller Programme: Sie skalieren nicht mit der verfügbaren Hardware.

30.2 Wir stellen Hilfsköche ein: Threads und Runnables

Um unsere Lieferzeiten zu verbessern, müssen wir Aufgaben parallelisieren. Wenn wir drei freie Öfen haben, sollten auch drei Pizzen gleichzeitig gebacken werden. Wir lagern die Aufgabe des Pizzabackens (die *Task*) in eine eigene Klasse aus, die das Interface Runnable implementiert.

```
class PizzaTask implements Runnable {
    private final String pizzaName;

    public PizzaTask(String pizzaName) {
        this.pizzaName = pizzaName;
    }

    @Override
    public void run() {
        String kochName = Thread.currentThread().getName();
        System.out.println(
                kochName + " beginnt mit " + pizzaName + "...");

        try {
            // Simulierte Backzeit
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            System.out.println(
                    kochName + " wurde beim Backen unterbrochen!");
            Thread.currentThread().interrupt();
        }

        System.out.println(
                kochName + " meldet: " + pizzaName + " ist fertig!");
    }
}
```

Nun modifizieren wir unsere Pizzeria. Der Chefkoch (main-Thread) backt nicht mehr selbst. Er delegiert die Arbeit, indem er neue Threads ("Hilfsköche") erschafft, ihnen die Arbeitsanweisung (Runnable) in die Hand drückt und sie mit start() an die Arbeit schickt.

```
public class PizzeriaConcurrent {
```

```
    public static void main(String[] args)
            throws InterruptedException {
        System.out.println(
                "Chef (Main-Thread): Pizzeria ist geöffnet!");
        long startZeit = System.currentTimeMillis();

        // Köche einstellen und Aufgaben zuweisen
        Thread koch1 = new Thread(new PizzaTask("Margherita"),
                "Koch-1");
        Thread koch2 = new Thread(new PizzaTask("Salami"), "Koch-2");
        Thread koch3 = new Thread(new PizzaTask("Funghi"), "Koch-3");

        // Die Arbeit beginnt! (Nebenläufigkeit startet hier)
        koch1.start();
        koch2.start();
        koch3.start();

        // Der Chef muss warten, bis alle fertig sind, bevor er
        // abschließt
        koch1.join();
        koch2.join();
        koch3.join();

        long endZeit = System.currentTimeMillis();
        System.out.println("Chef: Alle Pizzen fertig nach "
                + (endZeit - startZeit) + " ms. Wir schließen!");
    }
}
```

Die neue Ausgabe:

```
Chef (Main-Thread): Pizzeria ist geöffnet!
Koch-2 beginnt mit Salami...
Koch-1 beginnt mit Margherita...
Koch-3 beginnt mit Funghi...
Koch-1 meldet: Margherita ist fertig!
Koch-3 meldet: Funghi ist fertig!
Koch-2 meldet: Salami ist fertig!
Chef: Alle Pizzen fertig nach 2045 ms. Wir schließen!
```

30.3 Was wir erreicht haben (und was nicht)

Wir haben die Gesamtzeit von über 6 Sekunden auf knapp 2 Sekunden reduziert. Die Aufgaben wurden wirklich parallel abgearbeitet. Einige wichtige Beobachtungen:

1. Unvorhersehbarkeit: Achten Sie auf die Ausgabe! Obwohl wir koch1 (Margherita) zuerst gestartet haben, hat Koch-2 (Salami) zuerst losgelegt. Und Koch-1 war vor Koch-3 fertig. Wie in Kapitel 4 besprochen, haben wir keinen Einfluss auf den *Thread-Scheduler* des Betriebssystems. Die Ausführungsreihenfolge ist nicht mehr deterministisch.

2. Die Bedeutung von join(): Hätten wir am Ende nicht koch1.join() und die anderen aufgerufen, wäre der Main-Thread direkt weitergelaufen und hätte die Pizzeria geschlossen (Programm beendet), *während* die Hilfsköche noch

gebacken hätten! join() ist unsere Synchronisations-Barriere für den Feierabend.

Der Ausblick auf die nächste Iteration: Unser System ist jetzt nebenläufig, aber es ist noch weit von einer guten Architektur entfernt. Für jede einzelne bestellte Pizza erstellen wir ein neues Thread-Objekt und lassen es danach sterben. In der echten Welt stellt ein Restaurantbesitzer nicht für jede Pizza einen neuen Koch ein und feuert ihn direkt danach wieder – das Erzeugen von Threads ist teuer!

Im nächsten Kapitel (Iteration 2) werden wir dieses Problem lösen, indem wir feste Posten vergeben und die Köche in einem Thread-Pool organisieren. Zudem brauchen unsere Kellner eine Möglichkeit zu erfahren, wann genau eine bestimmte Pizza abholbereit ist – hierfür werden wir Callable und Future einführen.

31 Iteration 2 – Schichtplan und Kellner (Thread-Pools & Futures)

In der ersten Iteration haben wir unsere Pizzeria erfolgreich parallelisiert. Mehrere Pizzen wurden gleichzeitig gebacken, die Wartezeiten für unsere Kunden sanken drastisch. Dennoch würde ein echter Restaurantbesitzer bei unserer aktuellen Personalpolitik die Hände über dem Kopf zusammenschlagen.

Erinnern Sie sich an unseren Code: new Thread(new PizzaTask(...)).start(). Für jede einzelne Pizza, die bestellt wurde, haben wir einen komplett neuen Koch (Thread) eingestellt, ihn die Pizza backen lassen und ihn direkt danach gefeuert (der Thread wurde beendet). In Kapitel 7 haben wir gelernt: Das Erzeugen nativer Betriebssystem-Threads ist ein teurer, zeitintensiver Prozess. Wenn an einem Samstagabend 500 Pizzen bestellt werden, bricht unser System unter der Last der ständigen Thread-Erzeugung zusammen.

31.1 Feste Posten: Der Thread-Pool

Anstatt ständig neues Personal anzuheuern, strukturieren wir unsere Küche um. Wir stellen drei festangestellte Köche ein, die die ganze Schicht über an ihren Posten bleiben. Wenn eine Bestellung reinkommt, übernimmt ein freier Koch. Wenn alle Köche beschäftigt sind, kommt die Bestellung in eine Warteschlange, bis ein Koch wieder Kapazitäten hat.

Genau dieses Konzept wird in Java durch das Executor-Framework (Kapitel 7) abgebildet. Wir nutzen einen FixedThreadPool mit vier Threads.

31.2 Abholscheine für die Kellner: Callable und Future

Wir haben noch ein zweites Problem: Unser bisheriges PizzaTask implementierte das Interface Runnable. Dessen run()-Methode ist void – sie gibt nichts zurück. Unser Chefkoch wusste am Ende nur, dass *irgendjemand* fertig war. Er hatte die fertige Pizza aber nicht in der Hand, um sie dem Kellner zu übergeben.

Wir wechseln daher zum Interface Callable<V> (Kapitel 8). Es erlaubt uns, eine Aufgabe zu definieren, die ein konkretes Ergebnis zurückliefert – in unserem Fall ein Objekt der neuen Klasse Pizza.

Wenn wir eine Callable-Aufgabe an unseren Thread-Pool übergeben, erhalten wir sofort ein Future<Pizza>-Objekt zurück. Dieses Future ist wie der Abholschein oder der vibrierende Pager, den Sie in manchen Restaurants nach der Bestellung

bekommen. Der Kellner kann sofort weiterarbeiten (andere Tische bedienen) und später mit future.get() die fertige Pizza einfordern.

31.3 Die Umsetzung im Code

Zuerst definieren wir unsere simple Pizza-Klasse und unsere neue Aufgabe als Callable:

```
import java.util.concurrent.Callable;

// Unsere Rückgabe-Klasse
class Pizza {
    private final String name;

    public Pizza(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

// Der Task, der nun ein Ergebnis (die Pizza) zurückliefert
class PizzaBestellung implements Callable<Pizza> {
    private final String pizzaName;

    public PizzaBestellung(String pizzaName) {
        this.pizzaName = pizzaName;
    }

    @Override
    public Pizza call() throws Exception {
        String kochName = Thread.currentThread().getName();
        System.out.println(kochName + ": Beginne mit " + pizzaName);

        // Simulierte Backzeit von 2 Sekunden
        Thread.sleep(2000);

        System.out.println(kochName + ": " + pizzaName
                + " ist fertig gebacken!");
        return new Pizza(pizzaName); // Die fertige Pizza wird
                                     // zurückgegeben
    }
}
```

Nun modernisieren wir unser Restaurant. Der main-Thread übernimmt die Rolle des Oberkellners, der Bestellungen aufnimmt, sie an die Küche (den Pool) weiterleitet und die Abholscheine (Futures) sammelt.

Java

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
```

```
public class PizzeriaThreadPool {

    public static void main(String[] args) {
        System.out.println(
                "Oberkellner: Pizzeria öffnet mit 3 Köchen!");

        // Unser Schichtplan: Genau 3 festangestellte Köche (Threads)
        ExecutorService kuechenPersonal = Executors
                .newFixedThreadPool(3);

        // Hier speichern wir die Abholscheine der Kellner
        List<Future<Pizza>> abholscheine = new ArrayList<>();

        String[] bestellungen = { "Margherita", "Salami", "Funghi",
                "Hawaii", "Tonno" };

        // 1. Bestellungen annehmen und in die Küche rufen
        for (String wunsch : bestellungen) {
            System.out.println("Oberkellner: Gebe Bestellung für "
                    + wunsch + " auf.");
            // submit() übergibt die Aufgabe und liefert SOFORT das
            // Future zurück
            Future<Pizza> schein = kuechenPersonal
                    .submit(new PizzaBestellung(wunsch));
            abholscheine.add(schein);
        }

        // 2. Warten und Servieren
        System.out.println(
                "Oberkellner: Alle Bestellungen sind in der Küche. " +
                "Ich warte auf die Fertigstellung...");

        try {
            for (Future<Pizza> schein : abholscheine) {
                // get() blockiert den Oberkellner, bis GENAU DIESE
                // Pizza fertig ist
                Pizza fertigePizza = schein.get();
                System.out.println("Oberkellner: Serviere "
                        + fertigePizza.getName() + " an den Tisch.");
            }
        } catch (InterruptedException | ExecutionException e) {
            System.err.println(
                    "Fehler in der Küche: " + e.getMessage());
        }

        // 3. Feierabend machen (Pool herunterfahren)
        System.out.println(
                "Oberkellner: Alle serviert. Wir schließen!");
        kuechenPersonal.shutdown();
    }
}
```

Die Ausgabe des Programms offenbart die Magie des Pools:

```
Oberkellner: Pizzeria öffnet mit 3 Köchen!
Oberkellner: Gebe Bestellung für Margherita auf.
Oberkellner: Gebe Bestellung für Salami auf.
Oberkellner: Gebe Bestellung für Funghi auf.
Oberkellner: Gebe Bestellung für Hawaii auf.
Oberkellner: Gebe Bestellung für Tonno auf.
```

```
Oberkellner: Alle Bestellungen sind in der Küche. Ich warte auf die
Fertigstellung...
pool-1-thread-2: Beginne mit Salami
pool-1-thread-1: Beginne mit Margherita
pool-1-thread-3: Beginne mit Funghi
pool-1-thread-1: Margherita ist fertig gebacken!
Oberkellner: Serviere Margherita an den Tisch.
pool-1-thread-2: Salami ist fertig gebacken!
pool-1-thread-1: Beginne mit Hawaii
pool-1-thread-3: Funghi ist fertig gebacken!
Oberkellner: Serviere Salami an den Tisch.
Oberkellner: Serviere Funghi an den Tisch.
pool-1-thread-2: Beginne mit Tonno
pool-1-thread-1: Hawaii ist fertig gebacken!
Oberkellner: Serviere Hawaii an den Tisch.
pool-1-thread-2: Tonno ist fertig gebacken!
Oberkellner: Serviere Tonno an den Tisch.
Oberkellner: Alle serviert. Wir schließen!
```

31.4 Analyse: Was wir aus Iteration 2 lernen

Betrachten Sie die Ausgabe sehr genau. Obwohl wir fünf Pizzen bestellt haben, tauchen nur die Threads pool-1-thread-1 bis pool-1-thread-3 auf. Als die ersten drei Pizzen in den Ofen geschoben wurden, waren alle drei Köche beschäftigt. Die Bestellungen für "Hawaii" und "Tonno" blieben in der unsichtbaren Warteschlange (Work-Queue) des ExecutorService liegen. Sobald pool-1-thread-1 mit der Margherita fertig war, griff er sich sofort selbstständig die Hawaii-Bestellung aus der Schlange. Ressourcen-Recycling par excellence!

Zudem blockiert schein.get() unseren main-Thread (den Oberkellner) elegant genau so lange, bis das jeweilige Ergebnis asynchron berechnet wurde, ohne dass wir uns noch mit rudimentärem Thread.join() herumschlagen müssen.

Der Ausblick auf die nächste Iteration: Unsere Architektur ist nun effizient. Doch bisher arbeitet jeder Koch völlig isoliert an seiner eigenen Pizza. Was passiert, wenn alle Köche gleichzeitig auf eine gemeinsame Ressource zugreifen müssen? Im nächsten Kapitel (Iteration 3) führen wir eine Speisekammer ein. Wir werden erleben, wie es zu chaotischen Fehlbeständen (Race Conditions) kommt, wenn zwei Köche gleichzeitig nach der letzten Dose Tomatenmark greifen, und wie wir dieses Chaos mit *Locks* und synchronized bändigen.

32 Iteration 3 – Chaos in der Speisekammer (Geteilter Zustand & Locks)

Unsere Pizzeria läuft dank des Thread-Pools und der asynchronen Abholscheine (Futures) aus der letzten Iteration nun äußerst effizient. Doch bisher gab es ein unrealistisches Detail in unserer Simulation: Wir haben so getan, als hätten wir unendlich viele Zutaten.

Jedes echte Restaurant besitzt ein Inventar – eine Speisekammer. Und genau hier, wenn mehrere Threads (unsere Köche) anfangen, auf dieselben Daten (die Zutaten) zuzugreifen und diese zu verändern, betreten wir die gefährlichste Zone der Nebenläufigkeit: den geteilten Zustand (Shared State).

32.1 Die Speisekammer ohne Türsteher: Die Race Condition

Führen wir eine Klasse Speisekammer ein, die unseren Vorrat an Tomaten verwaltet. Jeder Koch muss, bevor er eine Pizza backen kann, prüfen, ob noch genug Tomaten da sind. Wenn ja, entnimmt er eine Tomate.

```
class Speisekammer {
    private int tomaten = 2; // Wir haben nur noch 2 Tomaten für die
                             // Schicht!

    public boolean nimmTomate(String kochName) {
        if (tomaten > 0) {
            System.out.println(kochName + " sieht, dass noch "
                    + tomaten + " Tomate(n) da sind.");

            // Eine winzige Pause, simuliert den Weg vom Regal zur
            // Arbeitsfläche
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
            }

            tomaten--; // Tomate entnehmen
            System.out.println(kochName
                    + " hat eine Tomate genommen. Rest: " + tomaten);
            return true;
        } else {
            System.out.println(
                    kochName + " meldet: Keine Tomaten mehr da!");
            return false;
        }
    }
}
```

Wir lassen nun drei unserer Köche (Threads) aus dem Pool *gleichzeitig* eine Pizza Margherita zubereiten. Alle drei benötigen eine Tomate, aber wir haben nur zwei im Regal.

Die katastrophale Ausgabe:

```
pool-1-thread-1 sieht, dass noch 2 Tomate(n) da sind.
pool-1-thread-2 sieht, dass noch 2 Tomate(n) da sind.
pool-1-thread-3 sieht, dass noch 2 Tomate(n) da sind.
pool-1-thread-1 hat eine Tomate genommen. Rest: 1
pool-1-thread-3 hat eine Tomate genommen. Rest: 0
pool-1-thread-2 hat eine Tomate genommen. Rest: -1
```

Wir haben -1 Tomaten! Was ist passiert? Dies ist ein klassisches Verzahnungsproblem (Interleaving), das wir im theoretischen Teil besprochen haben. Alle drei Köche haben *gleichzeitig* in die Speisekammer geschaut (die if-Bedingung geprüft). Für alle drei war die Bedingung tomaten > 0 wahr (da der Wert noch 2 war). Dann wurden sie vom Betriebssystem-Scheduler durch das kleine Thread.sleep() kurz pausiert. Als sie nacheinander weiterarbeiteten, zogen alle drei blindlings den Wert ab.

32.2 Der klassische Schutz: Das synchronized-Schlüsselwort

Wie wir in Kapitel 15 gelernt haben, müssen wir den kritischen Bereich absichern. Die Prüfung des Bestands und die tatsächliche Entnahme müssen zu einer unteilbaren (atomaren) Operation verschmelzen. Wir nutzen den eingebauten Monitor des Speisekammer-Objekts, indem wir die Methode mit dem Schlüsselwort synchronized versehen.

```
class SpeisekammerSynchronized {
    private int tomaten = 2;

    // Nur ein Thread darf diese Methode zur gleichen Zeit ausführen!
    public synchronized boolean nimmTomate(String kochName) {
        if (tomaten > 0) {
            System.out.println(kochName + " sieht, dass noch "
                    + tomaten + " Tomate(n) da sind.");
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
            }
            tomaten--;
            System.out.println(kochName
                    + " hat eine Tomate genommen. Rest: " + tomaten);
            return true;
        }
        System.out.println(
                kochName + " meldet: Keine Tomaten mehr da!");
        return false;
    }
}
```

Die Ausgabe ist nun absolut fehlerfrei:

```
pool-1-thread-1 sieht, dass noch 2 Tomate(n) da sind.
pool-1-thread-1 hat eine Tomate genommen. Rest: 1
pool-1-thread-3 sieht, dass noch 1 Tomate(n) da sind.
pool-1-thread-3 hat eine Tomate genommen. Rest: 0
pool-1-thread-2 meldet: Keine Tomaten mehr da!
```

Thread-1 schließt die Tür der Speisekammer ab, prüft den Bestand und entnimmt die Tomate in aller Ruhe. Erst wenn er die Methode verlässt (und den Monitor freigibt), darf Thread-3 eintreten. Thread-2 muss als letztes eintreten und stellt dann völlig korrekt fest, dass die Vorräte erschöpft sind.

32.3 Die Hochleistungs-Optimierung: Das ReentrantReadWriteLock

Unser synchronized-Schloss funktioniert einwandfrei. Doch in einer echten Pizzeria betreten die Köche die Speisekammer oft nur, um den Bestand auf dem Klemmbrett abzulesen, ohne etwas herauszunehmen.

Wenn 10 Köche gleichzeitig nur *nachschauen* wollen, blockieren sie sich mit unserem groben synchronized völlig unnötig gegenseitig – ein massiver Performance-Flaschenhals.

Wie wir in Kapitel 18 gelernt haben, ist das ReentrantReadWriteLock hierfür die perfekte, maßgeschneiderte Lösung. Es erlaubt es uns, den harmlosen Lesezugriff vom kritischen Schreibzugriff zu trennen.

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

class SpeisekammerOptimiert {
    private int tomaten = 2;
    private final ReadWriteLock rwLock = new ReentrantReadWriteLock();

    // Wird sehr oft aufgerufen - blockiert sich NICHT gegenseitig!
    public int getTomatenBestand() {
        rwLock.readLock().lock(); // Viele Threads dürfen gleichzeitig
                                  // hier rein!
        try {
            return tomaten;
        } finally {
            rwLock.readLock().unlock();
        }
    }

    // Wird seltener aufgerufen - blockiert ALLE anderen
    public boolean nimmTomate(String kochName) {
        rwLock.writeLock().lock(); // Exklusiver Zugriff! Alle anderen
                                   // warten.
        try {
            if (tomaten > 0) {
                tomaten--;
                return true;
            }
            return false;
        } finally {
            // WICHTIG: Die Sperre MUSS im finally-Block freigegeben
            // werden!
            rwLock.writeLock().unlock();
        }
    }
}
```

Durch den Einsatz des ReadWriteLock haben wir unsere Speisekammer nicht nur threadsicher gemacht, sondern sie auch für extrem hohe parallele Lasten optimiert. Die Köche können den Bestand nun rasend schnell abfragen, da mehrere Threads die Lese-Sperre gleichzeitig halten dürfen, solange niemand die Schreib-Sperre hält. Und denken Sie immer an das eiserne Gesetz: Ein Aufruf von lock() muss immer zwingend von einem try-finally-Block gefolgt werden, in dessen finally-Teil unlock() steht.

Der Ausblick auf die nächste Iteration:

Unsere Köche arbeiten nun absolut fehlerfrei mit gemeinsamen Ressourcen. Aber die Kommunikation zwischen den Kellnern und der Küche ist noch immer zu eng gekoppelt. Der Oberkellner übergibt jede Bestellung direkt an den Pool. In einer echten Großküche gibt es eine Ticket-Schiene, an die die Kellner die Bons hängen und von der sich die Köche bei Bedarf die nächste Arbeit nehmen.

Im kommenden Kapitel (Iteration 4) werden wir dieses System professionell entkoppeln, indem wir das *Producer-Consumer-Pattern* mithilfe einer hochperformanten BlockingQueue einführen und die Einnahmen unserer Kasse blitzschnell und völlig ohne Sperren über atomare Variablen verwalten!

33 Iteration 4 – Die Ticket-Schiene (Producer-Consumer & Concurrent Collections)

In den bisherigen Iterationen haben wir unsere Pizzeria immer weiter optimiert. Wir haben einen effizienten Thread-Pool für die Köche eingerichtet und unsere Speisekammer mit Locks vor Race Conditions geschützt. Doch wenn wir genau hinsehen, gibt es immer noch ein organisatorisches Problem zwischen dem Service (den Kellnern) und der Küche (den Köchen).

Bisher ist der Kellner direkt zur Küche gelaufen und hat dem Chefkoch (dem Thread-Pool) die Bestellung persönlich in die Hand gedrückt (executor.submit()). Was aber, wenn die Küche völlig überlastet ist? Der Kellner muss warten und kann in dieser Zeit keine neuen Gäste bedienen. Die beiden Systeme – Bestellannahme und Zubereitung – sind zu stark aneinander gekoppelt.

33.1 Die Entkopplung: Das Producer-Consumer-Muster

In einer echten, gut organisierten Großküche gibt es eine Ticket-Schiene (Bon-Leiste).

- Der Kellner schreibt die Bestellung auf einen Bon und klemmt ihn an die Schiene. Danach geht er sofort zurück zu den Gästen. Er ist der Producer (Erzeuger).
- Die Köche schauen auf die Schiene. Sobald ein Koch frei ist, reißt er den nächsten Bon ab und beginnt zu kochen. Die Köche sind die Consumer (Verbraucher).

Dieses Prinzip nennt man das *Producer-Consumer-Pattern* (Kapitel 20). Es entkoppelt Produzenten und Konsumenten durch einen dazwischenliegenden Puffer.

33.2 Die Magie der BlockingQueue

In Java liefert uns das Paket java.util.concurrent die perfekte Datenstruktur für unsere Ticket-Schiene: Die BlockingQueue. Wir verwenden hier speziell eine ArrayBlockingQueue mit einer festen Kapazität (z.B. Platz für 10 Bons).

Das Geniale an der BlockingQueue ist ihre eingebaute Thread-Sicherheit und ihre blockierenden Eigenschaften:

- Wenn die Schiene voll ist (10 Bons hängen dort), blockiert die Methode put(). Der Kellner muss kurz warten, bis ein Koch einen Bon wegnimmt.

- Wenn die Schiene leer ist, blockiert die Methode take(). Der Koch legt sich schlafen (verbraucht keine CPU-Zeit), bis ein Kellner einen neuen Bon aufhängt.

Wir müssen keine eigenen wait()- oder notify()-Aufrufe mehr programmieren – die Queue übernimmt die gesamte komplexe Choreografie für uns!

33.3 Die Kasse ohne Schloss: AtomicInteger

Neben der Ticket-Schiene führen wir noch eine Kasse ein, um den Tagesumsatz zu berechnen. Jedes Mal, wenn eine Pizza fertig ist, werfen wir 10 Euro in die Kasse. Wir könnten die Kasse nun wieder mit synchronized absichern (wie unsere Speisekammer in Iteration 3). Da wir hier aber nur einen einfachen Integer-Wert hochzählen, nutzen wir eine wesentlich schnellere, *lock-free* (sperrfreie) Variante: Den AtomicInteger (Kapitel 21). Er nutzt auf Hardware-Ebene spezielle CAS-Operationen (Compare-And-Swap), um den Wert threadsicher und ohne teure Monitore zu erhöhen.

33.4 Die Umsetzung im Code

Bauen wir unser neues, entkoppeltes System auf:

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.atomic.AtomicInteger;

public class PizzeriaTicketSchiene {

    // Unsere Ticket-Schiene bietet Platz für maximal 5 Bestellungen
    // gleichzeitig
    private static final BlockingQueue<String> TICKET_SCHIENE =
        new ArrayBlockingQueue<>(5);

    // Unsere threadsichere Kasse (Lock-Free)
    private static final AtomicInteger KASSE_IN_EURO =
        new AtomicInteger(0);

    public static void main(String[] args) {
        System.out.println(
                "Pizzeria öffnet. Ticket-Schiene ist bereit!");

        // Wir stellen 2 Köche (Consumer) ein
        Thread koch1 = new Thread(new Koch(), "Koch-Luigi");
        Thread koch2 = new Thread(new Koch(), "Koch-Mario");
        koch1.start();
        koch2.start();

        // Wir stellen 1 Kellner (Producer) ein
        Thread kellner = new Thread(new Kellner(), "Kellner-Peter");
        kellner.start();
    }

    // --- Der Producer ---
    static class Kellner implements Runnable {
        @Override
```

```
        public void run() {
            String[] bestellungen = { "Margherita", "Salami",
                    "Funghi", "Hawaii", "Tonno", "Spezial" };
            try {
                for (String bestellung : bestellungen) {
                    System.out
                            .println(Thread.currentThread().getName()
                                    + ": Hänge '" + bestellung
                                    + "' an die Schiene.");

                    // put() blockiert, falls die Schiene voll ist!
                    TICKET_SCHIENE.put(bestellung);

                    Thread.sleep(500); // Kellner braucht kurz, um zum
                                       // nächsten Tisch zu gehen
                }
                // Ein spezielles Ticket signalisiert Feierabend
                // (Poison Pill)
                TICKET_SCHIENE.put("FEIERABEND");
                TICKET_SCHIENE.put("FEIERABEND");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    // --- Der Consumer ---
    static class Koch implements Runnable {
        @Override
        public void run() {
            try {
                while (true) {
                    // take() blockiert, falls die Schiene leer ist!
                    String bestellung = TICKET_SCHIENE.take();

                    if (bestellung.equals("FEIERABEND")) {
                        System.out.println(
                                Thread.currentThread().getName()
                                        + ": Macht Feierabend!");
                        break; // Beendet die Endlosschleife
                    }

                    System.out.println(Thread.currentThread()
                            .getName() + ": Nehme '" + bestellung
                            + "' von der Schiene und backe...");
                    Thread.sleep(2000); // Simulierte Backzeit

                    // Pizza ist fertig, Geld in die Kasse
                    // (threadsicher ohne lock!)
                    int neuerUmsatz = KASSE_IN_EURO.addAndGet(10);
                    System.out
                            .println(Thread.currentThread().getName()
                                    + ": '" + bestellung
                                    + "' fertig! Umsatz jetzt: "
                                    + neuerUmsatz + "€");
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

Die Ausgabe zeigt das flüssige Ineinandergreifen (Pipelining):

Plaintext

```
Pizzeria öffnet. Ticket-Schiene ist bereit!
Kellner-Peter: Hänge 'Margherita' an die Schiene.
Koch-Luigi: Nehme 'Margherita' von der Schiene und backe...
Kellner-Peter: Hänge 'Salami' an die Schiene.
Koch-Mario: Nehme 'Salami' von der Schiene und backe...
Kellner-Peter: Hänge 'Funghi' an die Schiene.
Kellner-Peter: Hänge 'Hawaii' an die Schiene.
Kellner-Peter: Hänge 'Tonno' an die Schiene.
Koch-Luigi: 'Margherita' fertig! Umsatz jetzt: 10€
Koch-Luigi: Nehme 'Funghi' von der Schiene und backe...
Koch-Mario: 'Salami' fertig! Umsatz jetzt: 20€
Koch-Mario: Nehme 'Hawaii' von der Schiene und backe...
Kellner-Peter: Hänge 'Spezial' an die Schiene.
...
```

33.5 Analyse: Die Kraft der Concurrent Collections

Beobachten Sie, was passiert ist: Der Kellner rattert seine Bestellungen herunter und hängt sie an die Schiene (put), vollkommen unabhängig davon, wie lange die Köche zum Backen brauchen. Als Funghi, Hawaii und Tonno aufgehängt wurden, waren beide Köche noch beschäftigt. Die Bestellungen ruhten sicher in der BlockingQueue.

Auch haben wir ein klassisches Muster für das Beenden von Endlosschleifen in Consumer-Threads angewandt: Die *Poison Pill* (Giftpille). Anstatt die Threads hart abzuschießen, legt der Kellner am Ende des Tages für jeden Koch ein spezielles "FEIERABEND"-Ticket auf die Schiene. Sobald ein Koch dieses zieht, verlässt er geordnet die Küche.

Mit der BlockingQueue und dem AtomicInteger haben wir zum ersten Mal High-Level-Werkzeuge aus java.util.concurrent genutzt. Diese Klassen sind von den Java-Architekten hochgradig auf Performance optimiert worden und extrem resistent gegen Deadlocks und Race Conditions. Wann immer Sie in der Praxis Listen, Queues oder Maps über mehrere Threads hinweg teilen wollen: Greifen Sie niemals zu den Standard-Klassen (wie ArrayList oder HashMap), sondern nutzen Sie die thread-sicheren Varianten (wie CopyOnWriteArrayList oder ConcurrentHashMap).

Der Ausblick auf die nächste Iteration: Unsere Standard-Bestellungen laufen nun perfekt durch die Pipeline. Doch an Tisch 4 sitzt ein VIP-Gast. Er hat ein 3-Gänge-Menü bestellt und besteht darauf, dass alle Teller exakt *gleichzeitig* serviert werden! Zudem haben wir plötzlich nur noch 3 Backöfen, um die sich unsere Köche streiten müssen. Im nächsten Kapitel (Iteration 5) werden wir die Königsdisziplin der Thread-

Koordination kennenlernen: Wir zähmen unsere Köche mit einer CyclicBarrier und regeln den Zugriff auf die knappen Backöfen souverän mit einem Semaphore.

34 Iteration 5 – Meisterhafte Koordination (Barrieren & Semaphoren)

Unsere Ticket-Schiene aus dem letzten Kapitel läuft reibungslos. Das Producer-Consumer-Pattern hat den Service und die Küche wunderbar entkoppelt. Das Tagesgeschäft floriert.

Doch an diesem Abend betritt ein berüchtigter Restaurantkritiker unsere Pizzeria „Concurrency“ und nimmt an Tisch 4 (dem VIP-Tisch) Platz. Er hat ein exquisites 4-Gänge-Überraschungsmenü bestellt. Seine eiserne Bedingung: Alle vier Gerichte müssen exakt zur selben Sekunde heiß auf dem Tisch stehen. Gleichzeitig gibt es ein unvorhergesehenes technisches Problem in der Küche: Einer unserer vier teuren Steinöfen ist ausgefallen. Wir haben zwar vier festangestellte Köche im Pool, aber nur noch drei funktionierende Backöfen.

Wir stehen vor zwei völlig neuen Herausforderungen der Nebenläufigkeit:

1. Ressourcenbegrenzung: Wie verhindern wir, dass vier Köche versuchen, sich in drei Öfen zu drängen?
2. Gleichzeitigkeit (Rendezvous): Wie zwingen wir die schnellen Köche, mit ihren fertigen Tellern zu warten, bis auch der letzte Koch seinen Gang beendet hat?

34.1 Der Türsteher für den Ofen: Das Semaphore

Bisher kannten wir für den Zugriffsschutz nur das synchronized-Schloss oder das ReentrantLock. Diese arbeiten binär: *Ein* Thread darf rein, alle anderen müssen warten (Mutual Exclusion, Mutex). Bei unseren Backöfen greift dieses Konzept zu kurz. Wir wollen nicht nur einen Koch an die Öfen lassen, sondern exakt *drei* gleichzeitig. Der vierte Koch soll warten.

Die Lösung aus dem java.util.concurrent-Paket ist das Semaphore (Kapitel 19). Stellen Sie sich das Semaphore wie einen Korb mit Schlüsseln vor. Wir initialisieren unser Semaphore(3) mit drei Schlüsseln. Bevor ein Koch anfangen kann zu backen, muss er sich einen Schlüssel aus dem Korb nehmen (acquire()). Sind alle drei Schlüssel weg, blockiert acquire() den vierten Koch automatisch so lange, bis einer der ersten drei Köche fertig ist und seinen Schlüssel zurück in den Korb wirft (release()).

34.2 Der Sammelpunkt: Die CyclicBarrier

Unser zweites Problem ist das gleichzeitige Servieren. Wenn Koch Luigi mit der Vorspeise nach 2 Minuten fertig ist, darf er sie nicht sofort zum VIP-Tisch bringen,

sonst wird sie kalt, während Koch Mario noch am Hauptgang arbeitet. Luigi muss an der Durchreiche warten.

Hierfür nutzen wir die CyclicBarrier. Sie ist wie eine unsichtbare Schranke in der Küche. Wir definieren beim Erstellen, auf wie viele Threads (Parteien) diese Schranke warten soll – in unserem Fall 4 Köche. Sobald ein Koch mit seinem Gericht fertig ist, ruft er an der Barriere await() auf. Dieser Aufruf blockiert seinen Thread. Er steht nun buchstäblich an der Durchreiche und wartet. Erst wenn der *vierte* Koch await() aufruft, fällt die Schranke! Alle vier Threads werden exakt im selben Moment aufgeweckt und können gemeinsam zum Tisch marschieren.

Als Bonus erlaubt uns die CyclicBarrier, eine Aktion (ein Runnable) zu definieren, die automatisch ausgeführt wird, *genau in dem Moment, in dem die Schranke fällt*, aber *bevor* die Threads weiterlaufen. Das ist der perfekte Moment für den Oberkellner, laut „Service!“ zu rufen.

34.3 Die Umsetzung im Code

Bauen wir diese meisterhafte Choreografie auf:

```
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Semaphore;

public class PizzeriaKoordination {

    // 3 Öfen verfügbar
    private static final Semaphore OFEN_SEMAPHORE = new Semaphore(3);

    // Schranke für 4 Gerichte. Wenn alle 4 da sind, ruft der
    // Oberkellner das Kommando.
    private static final CyclicBarrier VIP_TISCH_BARRIERE =
        new CyclicBarrier(
            4, () -> {
                System.out.println(
                        "\n* OBERKELLNER: Alle 4 Gerichte sind heiß " +
                        " und bereit! Servieren! *\n");
            });

    public static void main(String[] args) {
        System.out.println(
                "Oberkellner: Bestellung für den VIP-Tisch ist da. " +
                "4 Köche, an die Arbeit!");

        // 4 Köche bekommen jeweils ein unterschiedliches Gericht mit
        // unterschiedlichen Zeiten
        Thread k1 = new Thread(new VipGericht("Antipasti", 1000),
                "Koch-1");
        Thread k2 = new Thread(new VipGericht("Pasta Mista", 3000),
                "Koch-2");
        Thread k3 = new Thread(new VipGericht("Pizza Trüffel", 2000),
                "Koch-3");
        Thread k4 = new Thread(new VipGericht("Tiramisu", 1500),
                "Koch-4");
```

```
        k1.start();
        k2.start();
        k3.start();
        k4.start();
    }

    static class VipGericht implements Runnable {
        private final String gericht;
        private final int zubereitungsZeit;

        public VipGericht(String gericht, int zubereitungsZeit) {
            this.gericht = gericht;
            this.zubereitungsZeit = zubereitungsZeit;
        }

        @Override
        public void run() {
            String koch = Thread.currentThread().getName();
            try {
                System.out.println(koch + " bereitet die Zutaten für "
                        + gericht
                        + " vor und wartet auf einen Ofen...");

                // 1. SEMAPHORE: Auf einen freien Ofen warten
                OFEN_SEMAPHORE.acquire();
                try {
                    System.out.println(
                            koch + " hat einen Ofen ergattert! Backt "
                                    + gericht + "...");
                    Thread.sleep(zubereitungsZeit); // Unterschiedliche
                                                    // Backzeiten
                } finally {
                    // WICHTIG: Den Ofen (Schlüssel) im finally-Block
                    // IMMER wieder freigeben!
                    System.out.println(koch
                        + " ist fertig. Räumt den Ofen für den Nächsten.");
                    OFEN_SEMAPHORE.release();
                }

                // 2. BARRIERE: Mit dem fertigen Teller an der
                // Schranke warten
                System.out.println(koch + " wartet mit dem fertigen '"
                        + gericht + "' an der Durchreiche.");
                VIP_TISCH_BARRIERE.await();

                // Diese Zeile wird von allen 4 Köchen absolut
                // gleichzeitig ausgeführt!
                System.out.println(koch + " serviert " + gericht
                        + " an den VIP-Tisch!");

            } catch (Exception e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

Die Ausgabe zeigt das faszinierende Zusammenspiel:

```
Oberkellner: Bestellung für den VIP-Tisch ist da. 4 Köche, an die Arbeit!
Koch-1 bereitet die Zutaten für Antipasti vor und wartet auf einen Ofen...
Koch-2 bereitet die Zutaten für Pasta Mista vor und wartet auf einen Ofen...
Koch-3 bereitet die Zutaten für Pizza Trüffel vor und wartet auf einen Ofen...
Koch-4 bereitet die Zutaten für Tiramisu vor und wartet auf einen Ofen...
Koch-1 hat einen Ofen ergattert! Backt Antipasti...
Koch-2 hat einen Ofen ergattert! Backt Pasta Mista...
Koch-3 hat einen Ofen ergattert! Backt Pizza Trüffel...
Koch-1 ist fertig. Räumt den Ofen für den Nächsten.
Koch-1 wartet mit dem fertigen 'Antipasti' an der Durchreiche.
Koch-4 hat einen Ofen ergattert! Backt Tiramisu...
Koch-3 ist fertig. Räumt den Ofen für den Nächsten.
Koch-3 wartet mit dem fertigen 'Pizza Trüffel' an der Durchreiche.
Koch-4 ist fertig. Räumt den Ofen für den Nächsten.
Koch-4 wartet mit dem fertigen 'Tiramisu' an der Durchreiche.
Koch-2 ist fertig. Räumt den Ofen für den Nächsten.
Koch-2 wartet mit dem fertigen 'Pasta Mista' an der Durchreiche.

* OBERKELLNER: Alle 4 Gerichte sind heiß und bereit! Servieren! *

Koch-2 serviert Pasta Mista an den VIP-Tisch!
Koch-3 serviert Pizza Trüffel an den VIP-Tisch!
Koch-4 serviert Tiramisu an den VIP-Tisch!
Koch-1 serviert Antipasti an den VIP-Tisch!
```

34.4 Analyse der Choreografie

Studieren Sie die Ausgabe präzise. Zunächst wollen alle vier Köche an den Ofen. Aber das Semaphore lässt nur Koch-1, Koch-2 und Koch-3 hindurch. Koch-4 (Tiramisu) muss zuschauen und blockiert bei acquire(). Erst als Koch-1 mit den schnellen Antipasti fertig ist und release() aufruft, wird Koch-4 freigeschaltet und bekommt den Ofen.

Währenddessen stauen sich die fertigen Köche (Koch-1, Koch-3 und Koch-4) am await() der CyclicBarrier. Sie tun absolut nichts, sie warten nur darauf, dass der langsame Koch-2 mit der Pasta Mista endlich fertig wird. Als dieser schließlich als Letzter await() aufruft, löst die Barriere das Kommando des Oberkellners aus und schlagartig brechen alle vier Threads aus ihrer Blockade aus, um das Essen gemeinsam zum Tisch zu tragen.

Der Kritiker ist begeistert. Die Synchronisation war fehlerfrei.

Der Ausblick auf die nächste Iteration: Unsere Pizzeria in dieser Stadt ist nun das effizienteste und bestorganisierte Restaurant überhaupt. Doch der Erfolg zieht Kreise. Der Chef beschließt, aus der Pizzeria eine landesweite Kette mit Online-Lieferdienst zu machen. Plötzlich haben wir nicht mehr 50 Bestellungen am Abend, sondern 50.000 gleichzeitige Verbindungen. Unsere klassischen Thread-Pools mit Betriebssystem-Threads stoßen nun an die gnadenlose Speichermauer (die sogenannte *C10K-Problematik*). Im nächsten Kapitel (Iteration 6) werden wir die ultimative Modernisierung durchführen: Wir werfen die alten Threads über Bord, stellen unsere

Architektur auf leichtgewichtige Virtuelle Threads (Project Loom) um und verketten unsere Prozesse zu extrem reaktiven, asynchronen Pipelines mithilfe von CompletableFuture!

35 Iteration 6 – Die landesweite Kette (Project Loom & CompletableFuture)

Unsere Pizzeria „Concurrency“ in der Kleinstadt läuft perfekt. Das hat sich herumgesprochen. Das Management hat beschlossen, das Konzept auszurollen: Wir werden zu einer landesweiten Franchise-Kette mit einer zentralen Online-Bestellplattform.

Gestern Abend ging die Plattform live. Um 19:00 Uhr passierte es: Statt 50 Bestellungen am Abend hatten wir plötzlich 50.000 gleichzeitige Zugriffe auf unseren Server. Unser System ist nach wenigen Sekunden mit einem katastrophalen OutOfMemoryError abgestürzt. Was ist passiert?

35.1 Die Speichermauer: Das C10K-Problem

Erinnern Sie sich an Iteration 2? Dort haben wir das Problem gelöst, dass das ständige Erstellen von Threads zu teuer ist, indem wir einen Thread-Pool (FixedThreadPool) mit 4 Köchen eingeführt haben.

Für unsere landesweite Kette reicht ein Pool mit 4 Threads natürlich nicht. Um 50.000 Kunden gleichzeitig zu bedienen, ohne dass 49.996 von ihnen in der Warteschlange hängen, bräuchten wir theoretisch 50.000 Threads in unserem Pool.

Doch hier schlägt die Physik zu: Jeder herkömmliche Java-Thread ist ein direkter Wrapper (eine Hülle) um einen echten Betriebssystem-Thread. Jeder dieser OS-Threads belegt allein für seinen Call-Stack etwa 1 bis 2 Megabyte im Arbeitsspeicher. 50.000 Threads würden also auf einen Schlag ca. 50 bis 100 Gigabyte RAM verschlingen! Der Server kollabiert. Dies nennt man das C10K-Problem (Das Problem der 10.000 gleichzeitigen Verbindungen).

35.2 Die Rettung: Virtuelle Threads (Project Loom)

In einer Großküche verhält sich ein OS-Thread wie ein festangestellter Sternekoch: Er ist extrem teuer im Unterhalt, braucht viel Platz und Sie können sich nur wenige davon leisten. Wenn dieser Sternekoch nun darauf wartet, dass der Teig im Ofen aufgeht (eine blockierende I/O-Operation, wie unser Thread.sleep()), kostet er Sie weiterhin ein Vermögen, obwohl er gerade nichts tut.

Mit Project Loom (eingeführt in Java 21) hat Java die Virtuellen Threads erschaffen. Virtuelle Threads sind wie gigantische Scharen von flinken, billigen Hilfskräften. Sie werden nicht mehr vom Betriebssystem verwaltet, sondern direkt von der Java Virtual

Machine (JVM). Ein Virtueller Thread belegt nur wenige Byte im Speicher. Sie können problemlos Millionen (!) davon gleichzeitig starten.

Der magische Trick: Wenn ein Virtueller Thread auf etwas warten muss (z.B. den Ofen oder eine Datenbankabfrage), wird er von der JVM sofort "eingefroren" und vom physischen Prozessorkern abgekoppelt. Der Kern ist sofort frei für den nächsten Virtuellen Thread. Sobald die Wartezeit vorbei ist, taut die JVM den Thread wieder auf.

35.3 Das Fließband: CompletableFuture

Da wir nun zigtausende Bestellungen gleichzeitig annehmen können, brauchen wir eine bessere Organisation für den Ablauf: Bestellen → Backen → Ausliefern → Bezahlen.

Bisher hat unser Main-Thread (der Oberkellner) auf das Ergebnis gewartet (future.get()). Doch Blockieren ist in reaktiven Systemen verboten.

Wir führen das CompletableFuture (Kapitel 12) ein. Es erlaubt uns, eine asynchrone Pipeline (ein Fließband) aufzubauen. Wir deklarieren lediglich: *"Wenn Aufgabe A fertig ist, gib das Ergebnis automatisch an Aufgabe B weiter, und danach an Aufgabe C"*. Der Main-Thread muss nicht mehr warten, das System steuert sich selbst.

35.4 Die Umsetzung im Code

Schauen wir uns an, wie drastisch sich unser Code durch Virtuelle Threads und funktionale Pipelines modernisiert:

Java

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class PizzeriaCloudLoom {

    public static void main(String[] args) {
        System.out.println("Cloud-Zentrale: System fährt hoch...");

        // Die Revolution: Ein Executor, der für JEDE Aufgabe einen
        // neuen Virtuellen Thread startet!
        // Kein Pool-Limit mehr. Wir können Millionen hiervon starten.
        try (ExecutorService loomExecutor = Executors
                .newVirtualThreadPerTaskExecutor()) {

            long startZeit = System.currentTimeMillis();

            // Wir simulieren 10.000 gleichzeitige Bestellungen!
            CompletableFuture<?>[] alleBestellungen =
                new CompletableFuture[10000];
```

```
            for (int i = 0; i < 10000; i++) {
                final int bestellNummer = i;

                // Wir bauen unser asynchrones Fließband auf:
                alleBestellungen[i] = CompletableFuture
                        // 1. Asynchron bestellen (Start der Kette im
                        // Virtuellen Thread)
                        .supplyAsync(() -> bestellungAufnehmen(
                                bestellNummer), loomExecutor)
                        // 2. Sobald bestellt, asynchron backen
                        .thenApplyAsync(pizza -> pizzaBacken(pizza),
                                loomExecutor)
                        // 3. Sobald gebacken, ausliefern
                        // (konsumieren)
                        .thenAcceptAsync(
                                fertigePizza -> pizzaAusliefern(
                                        fertigePizza),
                                loomExecutor);
            }

            // Wir warten (nur für dieses Beispiel), bis alle 10.000
            // Pipelines durchgelaufen sind
            CompletableFuture.allOf(alleBestellungen).join();

            long endZeit = System.currentTimeMillis();
            System.out.println(
                 "Cloud-Zentrale: Alle 10.000 Bestellungen verarbeitet in "
                        + (endZeit - startZeit) + " ms!");

        } // Durch 'try-with-resources' wird der Executor hier sicher
          // geschlossen (Strukturierte Nebenläufigkeit!)
    }

    // --- Unsere Pipeline-Stationen ---

    static String bestellungAufnehmen(int id) {
        // Die Ausgabe kommentieren wir aus, sonst explodiert unsere
        // Konsole bei 10.000 Zeilen
        // System.out.println("Aufnahme: Bestellung #" + id + " auf "
        // + Thread.currentThread());
        return "Pizza-" + id;
    }

    static String pizzaBacken(String pizza) {
        try {
            // Dies blockiert den Virtuellen Thread, ABER NICHT den
            // echten OS-Thread!
            // Der OS-Thread übernimmt in diesen 100ms hunderte andere
            // Pizzen.
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
        return pizza + " [gebacken]";
    }

    static void pizzaAusliefern(String pizza) {
        // System.out.println("Auslieferung: " + pizza +
        // " erfolgreich zugestellt!");
    }
}
```

Die bemerkenswerte Ausgabe:

```
Cloud-Zentrale: System fährt hoch...
Cloud-Zentrale: Alle 10.000 Bestellungen verarbeitet in 425 ms!
```

35.5 Analyse: Die Magie von Loom und funktionaler Ketten

Lassen Sie sich dieses Ergebnis auf der Zunge zergehen: Wir haben 10.000 Pizzen gebacken. Jede Pizza hat eine simulierte Backzeit (Thread.sleep()) von 100 Millisekunden. Wenn wir diese sequenziell gebacken hätten, hätte das über 16 Minuten gedauert.

Unser System hat alle 10.000 Pizzen in weniger als einer halben Sekunde verarbeitet!

Und das völlig ohne den gefürchteten OutOfMemoryError. Durch Executors.newVirtualThreadPerTaskExecutor() haben wir der JVM gestattet, für jede der 10.000 Aufgaben einen separaten Virtuellen Thread zu starten. Als diese Threads beim Backen (sleep) pausieren mussten, wurden sie quasi schwerelos in den Hintergrund geschoben. Das Betriebssystem hat davon überhaupt nichts mitbekommen.

Gleichzeitig haben wir mit den .thenApplyAsync()-Ketten des CompletableFuture die gefürchtete Callback-Hölle vermieden. Unser Code liest sich wie ein sauberes Rezept: Nimm Bestellung → dann backe → dann liefere aus.

Zudem nutzen wir hier im Ansatz schon die im Buch abschließend behandelte Strukturierte Nebenläufigkeit (Structured Concurrency). Durch den try-with-resources-Block (try (ExecutorService …)) stellen wir sicher, dass der Main-Thread diesen Block niemals verlässt, bevor nicht alle darin gestarteten Unter-Threads geordnet beendet (oder bei Fehlern sauber abgebrochen) wurden. Keine verwaisten Threads mehr im Hintergrund!

Der Ausblick auf das Finale:

Unser System ist jetzt eine state-of-the-art Cloud-Architektur. Es skaliert grenzenlos und arbeitet völlig asynchron. Doch bevor wir es endgültig an die Aktionäre der Franchise-Kette übergeben, müssen wir beweisen, dass unser Code auch unter Extremlasten keine versteckten Bugs enthält und die Performance hält, was sie verspricht.

Im alles entscheidenden Kapitel 36 (Iteration 7) rufen wir das Gesundheitsamt zur Code-Inspektion! Wir lernen, wie man diese komplexen asynchronen Konstrukte mit Awaitility professionell testet und wie wir unsere lock-free Kasse aus Iteration 4 mit einem JMH Microbenchmark wissenschaftlich exakt vermessen.

36 Iteration 7 – Gesundheitsamt und Sterne-Bewertung (Testing & Benchmarking)

Unsere landesweite Franchise-Kette läuft dank Projekt Loom und CompletableFuture blitzschnell und reibungslos. Das System skaliert hervorragend. Doch bevor wir an die Börse gehen, stehen zwei entscheidende Prüfungen an:

1. Das Gesundheitsamt (Qualitätssicherung) verlangt den Beweis, dass unser asynchroner Code unter allen Umständen fehlerfrei arbeitet (Testing).
2. Die Sterne-Tester (Performance-Analysten) wollen wissenschaftlich exakte Messwerte sehen, die beweisen, dass unsere teuren Optimierungen (wie lock-freie Kassen) wirklich schneller sind (Benchmarking).

Die Herausforderung: Wie wir im Vorwort dieses Buches gelernt haben, ist Nebenläufigkeit nicht deterministisch. Ein klassischer Unittest funktioniert hier nicht ohne Weiteres.

36.1 Das Problem asynchroner Tests (Heisenbugs)

Stellen Sie sich vor, wir wollen unsere Ticket-Schiene (die BlockingQueue aus Iteration 4) mit klassischem JUnit testen. Der Kellner (Thread A) hängt eine Bestellung auf, der Koch (Thread B) nimmt sie ab und erhöht den Umsatz in der Kasse.

Ein naiver Test sähe so aus:

```
@Test
void testeTicketSchiene() {
    Kellner kellner = new Kellner();
    new Thread(kellner).start(); // Kellner fängt an zu arbeiten

    // FALSCH! Der Main-Thread prüft die Kasse sofort,
    // noch bevor der Koch überhaupt angefangen hat zu backen!
    assertEquals(10, KASSE_IN_EURO.get());
}
```

Dieser Test wird fast immer fehlschlagen. Der main-Thread des Tests rast zum assertEquals, während der Kellner-Thread vielleicht noch gar nicht vom Betriebssystem gestartet wurde. Fügen wir nun aus Verzweiflung ein Thread.sleep(2000) vor das assertEquals ein, wird der Test extrem langsam. Und was, wenn der Koch mal 2,1 Sekunden braucht? Dann schlägt der Test sporadisch fehl (sogenannte *Flaky Tests*). Ändern wir den Code, um den Fehler zu suchen, verschwindet er plötzlich – das ist der gefürchtete *Heisenbug* (Kapitel 24).

36.2 Das Gesundheitsamt: Professionelles Testing mit Awaitility

Um asynchronen Code professionell zu testen, nutzen wir das Open-Source-Framework Awaitility. Es folgt einem simplen Prinzip: Anstatt stur eine bestimmte Zeit zu warten, *pollt* (fragt) Awaitility in regelmäßigen Abständen nach, ob eine Bedingung erfüllt ist, und bricht spätestens nach einem definierten Timeout ab.

So sieht unser Test für die Pizzeria mit Awaitility aus:

```
import org.junit.jupiter.api.Test;
import static org.awaitility.Awaitility.await;
import static java.util.concurrent.TimeUnit.SECONDS;
import static org.junit.jupiter.api.Assertions.assertEquals;

class PizzeriaTest {

    @Test
    void testeBestellungBisZurKasse() {
        // 1. Arrange: System vorbereiten
        PizzeriaTicketSchiene.starteSchicht();

        // 2. Act: Kellner simuliert EINE Bestellung
        PizzeriaTicketSchiene.TICKET_SCHIENE.add("Margherita");

        // 3. Assert: Warten auf das asynchrone Ergebnis (Die Magie
        // von Awaitility)
        await().atMost(5, SECONDS) // Maximal 5 Sekunden warten
                                   // (Timeout)
                .until(() -> PizzeriaTicketSchiene.KASSE_IN_EURO
                        .get() == 10);

        // Wenn wir hier ankommen, war die Bedingung innerhalb der 5
        // Sekunden erfüllt!
        assertEquals(10, PizzeriaTicketSchiene.KASSE_IN_EURO.get());
    }
}
```

Awaitility prüft nun z.B. alle 100 Millisekunden den Wert der Kasse. Sobald der Koch fertig ist (vielleicht nach exakt 200ms), meldet Awaitility sofort Erfolg und der Test geht weiter. Keine verschwendete Wartezeit, keine Flaky Tests. Das Gesundheitsamt ist hochzufrieden!

36.3 Die Sterne-Bewertung: Benchmarking mit JMH

In Iteration 3 und 4 haben wir behauptet, dass ein AtomicInteger (Lock-Free) oder ein ReentrantReadWriteLock viel schneller ist als ein massives synchronized-Schloss. Aber in der Informatik gilt: Wer nicht misst, der rät.

Java-Code zu messen ist jedoch extrem schwierig. Die Java Virtual Machine (JVM) optimiert Code zur Laufzeit (Just-In-Time Compilation), räumt den Speicher auf (Garbage Collection) und entfernt sogar toten Code, der nichts bewirkt. Ein einfaches

System.currentTimeMillis() vor und nach einer Schleife liefert völlig nutzlose Fantasiezahlen.

Hier betritt das JMH (Java Microbenchmark Harness) die Bühne. Es ist das Standardwerkzeug der JDK-Entwickler, um Code wissenschaftlich fundiert zu messen. JMH wärmt die JVM auf (Warmup-Phasen), isoliert die Tests und verhindert, dass der JIT-Compiler unseren Test-Code wegoptimiert.

Wir lassen nun unsere alte synchronized-Kasse gegen unsere moderne AtomicInteger-Kasse antreten. Dazu hämmern 10 Threads gleichzeitig auf beide Kassen ein.

```
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

// Wir messen die durchschnittliche Zeit pro Aufruf (in Nanosekunden)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
@Threads(10) // 10 Threads greifen GLEICHZEITIG zu (Hohe *Contention*)
public class KassenBenchmark {

    // --- Variante A: Die langsame Kasse ---
    private int syncKasse = 0;

    @Benchmark
    public synchronized int messeSynchronizedKasse() {
        syncKasse += 10;
        return syncKasse;
    }

    // --- Variante B: Die Lock-Free Kasse ---
    private final AtomicInteger atomicKasse = new AtomicInteger(0);

    @Benchmark
    public int messeAtomicKasse() {
        return atomicKasse.addAndGet(10);
    }
}
```

Das vernichtende Ergebnis des Benchmarks: Nachdem JMH 5 Minuten lang Millionen von Aufrufen durchgeführt hat, spuckt es das Ergebnis aus:

```
Benchmark                              Mode  Cnt     Score    Error  Units
KassenBenchmark.messeAtomicKasse       avgt   10    12,453 ±  0.812  ns/op
KassenBenchmark.messeSynchronizedKasse avgt 10   145,102 ± 12.431  ns/op
```

Der Beweis ist erbracht: Die lock-freie AtomicInteger-Kasse benötigt im Durchschnitt nur 12 Nanosekunden pro Aufruf, während die synchronized-Kasse mit 145 Nanosekunden über 10-mal so langsam ist! Das ständige Sperren und Entsperren des Monitors durch das Betriebssystem bremst unsere 10 Threads massiv aus. Die Sterne-Tester überreichen uns feierlich die Höchstwertung für unsere Architektur.

36.4 Fazit des Großküchen-Projekts

Wir haben es geschafft. Was als chaotische Pizzeria mit einem einzelnen überforderten Koch begann, haben wir iterativ in eine hochskalierbare, fehlerresistente und reaktive Cloud-Architektur transformiert. Sie haben gesehen, wie Threads erstellt und in Pools verwaltet werden. Sie haben Race Conditions erlebt und sie mit Locks und atomaren Variablen bezwungen. Sie haben mit Barrieren und Queues komplexe Choreografien erschaffen und schließlich die Grenzen der Hardware mit Project Loom durchbrochen.

Ein letztes Wort: Wie wir im Vorwort dieses Buches angekündigt haben, haben Sie die tröstliche, berechenbare sequenzielle Welt hinter sich gelassen. Die Programmierung nebenläufiger Systeme gilt oft als die Königsdisziplin der Softwareentwicklung. Sie haben mit diesem Buch das theoretische und praktische Rüstzeug erhalten, um diese Herausforderung zu meistern. Nutzen Sie Ihre neuen Fähigkeiten mit Bedacht. Schreiben Sie Tests, messen Sie mit Benchmarks und vor allem: Bauen Sie Software, auf die Sie stolz sein können!