### Das Schlüsselwort volatile

Threads arbeiten zwar alle innerhalb des Prozesses, aber das Betriebssystem kann natürlich den Prozess auf mehrere Prozessorkerne verteilen und so kann es vorkommen, dass zwei Threads eines Prozesses auf verschiedenen Kernen laufen.

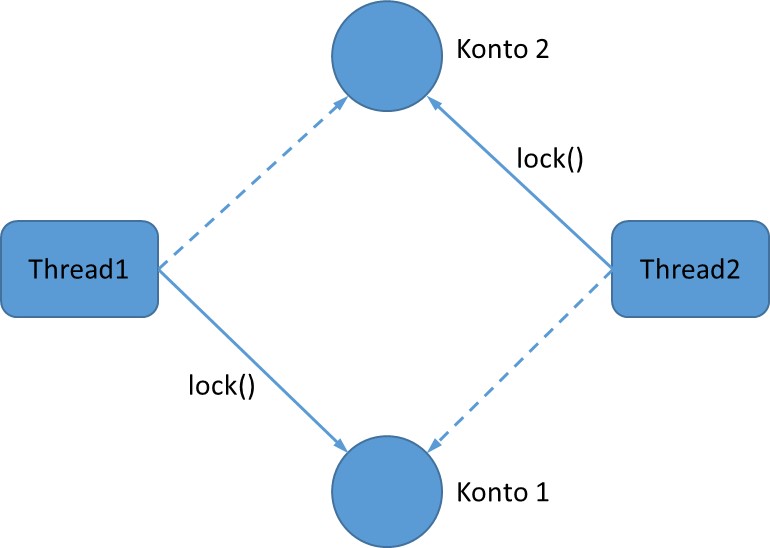
Da Variable, die der Thread braucht, diesem als Kopie zur Verfügung gestellt werden, kann es passieren, dass eine scheinbar gemeinsam genutzte Variable in Wahrheit bei beiden Threads nur jeweils Kopien sind. Um das zu verhindern, können wir den vorgesehenen Wert mit dem Schlüsselwort

**private volatile int wert = 3;**

versehen.

Alle Threads, welche die Variable **wert** verwenden, greifen dann auf die gleiche Stelle im Speicher zu. Da es die Variable nur einmal gibt, können nun auch keine Inkonsistenzen mehr auftreten.

### Dann gibt es noch Deadlocks und - was ist ein Livelock?

Betrachten wir die folgende Situation: Ein Thread1 hat gerade Konto 1 blockiert, um von dort eine Überweisung auf Konto 2 vorzunehmen. Währenddessen hat ein anderer Thread Konto 2 blockiert, um von dort eine Überweisung auf Konto 1 vorzunehmen. Thread 1 wartet nun auf die Freigabe von Konto 2, um einen Betrag von Konto 1 auf Konto 2 zu machen, während Thread 2 ebenfalls wartet, nur umgekehrt. Die beiden Threads befinden sich in einem Threadlock, den sie ohne fremde Hilfe nicht auflösen können.

Die Situation, dass ein Thread darauf wartet, dass eine Ressource freigegeben wird, nennen wir Deadlock.

Beim Livelock ein Thread dauerhaft beschäftigt und wird niemals fertig.

Wie können wir aber nun unseren beiden Threads helfen? Denn dieses Problem tritt ja schon bei einfachen Variablen auf: wenn beide Threads auf die gleich einfache Variable zugreifen wollen, z. B. die Variable i inkrementieren wollen? Um die Variable zu inkrementieren, muss Thread1 zuerst den Wert aus dem Speicher auslesen und in den Akku geladen (der Speicherplatz kann nur speichern, nicht rechnen).

Nehmen wir beispielsweise an, i hätte den Wert 5. Thread1 holt sich den Wert der Variablen i, um sie auf 6 zu erhöhen. Gleichzeitig holt sich Thread2 den Wert, um ihn auf 6 zu inkrementieren. Thread1 schreibt den Wert 6 eine Kleinigkeit früher zurück und dann erst der Thread2. Stellen wir uns nun noch vor, dass es sich wie in der Grafik um ein Konto, nämlich um unsres handeln würde und die 2 Threads Personen wären, die uns jeweils 100€ überweisen wollten, hätten wir anschließend nur 100€ mehr und nicht 200€.

Um das zu verhindern, müssen wir die Operation mit dem Schlüsselwort **synchronize** schützen. In Java können wir mit **synchronize** Methoden als auch Codeblöcke schützen.

Eine Methode, welche ein Thread sicheres Inkrement durchführt sieht folgendermaßen aus.

**public synchronized void increment(){**

**i++;**

**}**

**public void increment(){**

**synchronized(this){**

**i++;**

**}**

**}**

Möchte man nur einen Bereich, also einen Codeblock schützen, muss man das zugehörige Objekt mit angeben (siehe Block in der Methode).

Der Code mit einem solchen Schutz kann nur durch einen einzigen Thread gleichzeitig, also insbesondere nicht parallel, ausgeführt werden.

Damit verlieren wir aber etwas Gleichzeitigkeit, wenn wir **synchronize** zu häufig einsetzen.

### Synchronisierung von Threads

Kehren wir zu unserem Zähler vom Anfang zurück. Mit dem Erlernten können wir nun erkennen, dass die Zählschleife und die Oberfläche in verschiedenen Threads laufen.

Ändert sich das, wenn wir das Panel in die Klasse mit dem Zähler lokal einbauen? Wenn wir es ausprobieren, sehen wir – nein! Es liegt also irgendwo beim Panel selbst, an dem Frame oder an der Grafik. So genau müssen wir das gar nicht wissen.

Dennoch möchten wir jetzt erreichen, dass jeder einzelne Zähler auf der Oberfläche zu sehen ist. Wir müssen also den Zählthread dazu bringen, dass er der GUI Zeit lässt, den Wert zu zeichnen. Der eine Thread soll also auf den anderen warten. Hier stehen uns mehrere Möglichkeiten zur Verfügung:

### wait und notify

Mit wait und notify können wir zwei Threads aufeinander warten lassen. Der Thread, der gerade nicht arbeiten kann, ruft wait auf und wartet auf die Ressource. Hat er sie, bearbeitet er sie und ruft anschließend notify auf, damit andere wartende Threads mitbekommen, dass eine Ressource freigegeben wurde.

### sleep

Es ist manchmal schwierig, an andere Threads des Systems zu kommen, um dort notify aufzurufen. Oft reicht es aber, den eigenen Thread einfach für eine bestimmte Zeit in den inaktiven Zustand zu versetzen, um beispielsweise dem Thread, der die GUI aktualisiert Gelegenheit zu geben, genau dies zu tun.

### Zusammenfassung:

Den Zugriff auf gemeinsame Daten zweier oder mehrerer Threads schützen wir mit synchronized. Mit wait, sleep, notify sorgen wir dafür, dass andere Threads Rechenzeit bekommen (wir wissen ja nicht, ob das Betriebssystem die Threads echt parallelisiert oder ob sie nur parallelisiert werden könnten (und dann in Wahrheit nacheinander abgearbeitet werden, allerdings in unbekannter Reihenfolge).