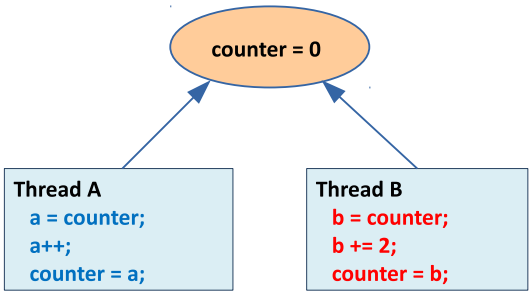

Abbildung 3-1: Beispiel einer Race Condition

Zugriffe verzahnt waren. Abbildung 3-2 zeigt beispielhaft drei mögliche Situationen.

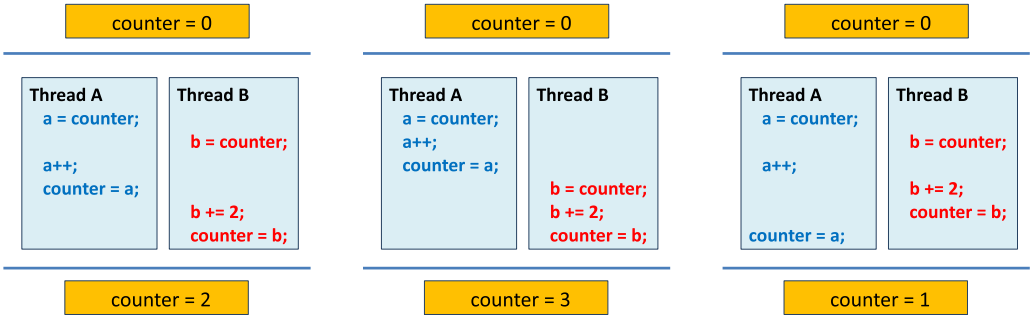

Abbildung 3-2: Drei mögliche Abläufe

Um Inkonsistenzen im Objektzustand vorzubeugen, müssen Zugriffe auf das Objekt *synchronisiert* werden. Ein Verfahren hierzu ist der *gegenseitige Ausschluss*. Solange ein Thread mit einem Objekt arbeitet, erhalten andere keinen Zugang dazu.

Bevor wir die in Java verwendeten, elementaren Konzepte zur Steuerung des gegenseitigen Ausschlusses betrachten, müssen wir noch diskutieren, welche Variablenarten überhaupt gefährdet sind. Dazu schauen wir uns zuerst das Speichermodell von Java (*Java Memory Model, JMM*) etwas genauer an.

3.2 Java-Speichermodell

Das Speichermodell einer Sprache regelt unter anderem die Interaktion von Threads mit dem Speicher. Hierbei wird festgelegt, wann und unter welchen Umständen von einem Thread geänderte Daten für andere sichtbar werden.

3.2.1 Stacks und Heap

Der Speicher wird innerhalb einer JVM in einen gemeinsamen Heap- und verschiedene Stack-Speicher unterteilt (vgl. Abb. 3-3). Jeder Thread hat einen eigenen Stack[1], der unter anderem lokale Variablen, Methodenparameter, Rückgabewerte und Verwaltungsinformationen für Methodenaufrufe beinhaltet. Alle Objekte in Java werden dagegen auf dem Heap abgelegt.

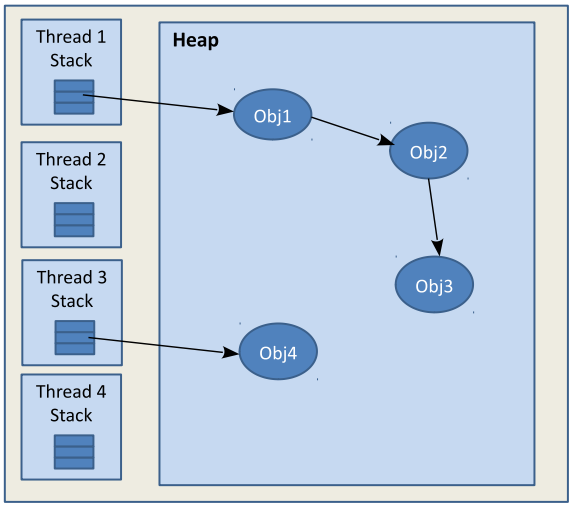

Abbildung 3-3: Speicherorganisation innerhalb der JVM

Im folgenden Codebeispiel kommen Variablen und Objekte vor, die in verschiedenen Arten von Speichern abgelegt werden.

```
class A
{
  public static void main(String[] args)
  {
    int a = 3;
    Integer b = new Integer();
  }
}
```

Auf dem Stack (des `main`-Threads) liegt die Referenzvariable `args`, die `int`-Variable `a` und die Referenzvariable `b`. All diese Variablen sind nur dem `main`-Thread zugänglich. Auf dem Heap befindet sich das erzeugte `Integer`-Objekt und das `String`-Array, in dem ggf. übergebene Aufrufparameter stehen[2]. In Abbildung 3-4 ist dies schematisch dargestellt.

[1] Wenn im Folgenden von Stack oder Heap gesprochen wird, ist immer der zugehörige Stack- bzw. Heap-Speicher gemeint.

[2] Auf dem Heap stehen außerdem diverse Objekte wie `System.out`, `System.in` usw., die beim Starten des Programms angelegt wurden.

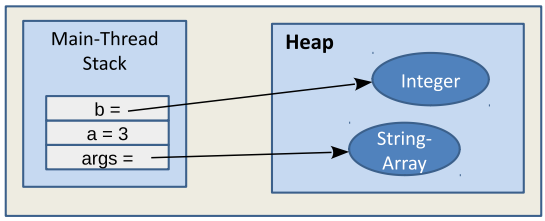

Abbildung 3-4: Daten auf dem Stack und Heap

Lokale Variablen gehören immer nur einem Thread. Zwei verschiedene Threads, die folgende Methode ausführen

```
public static void print(int i)
{
  int a = i*i;
  Integer b = new Integer(a);
  System.out.println(b);
}
```

haben dabei verschiedene, lokale Variablen `i`, `a` und `b` auf ihrem eigenen Stack (vgl. Abb. 3-5).

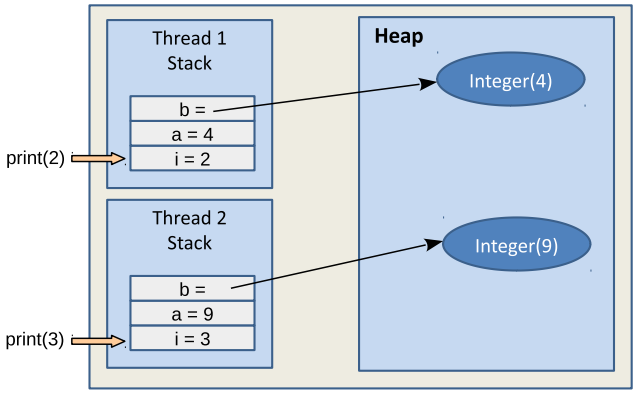

Abbildung 3-5: Lokale Daten zweier Threads

Lokale Variablen eines elementaren Datentyps[3] werden vollständig, d. h. mit ihrem Wert, auf dem Stack abgelegt. Lokale Referenztypen (zum Beispiel `String`, `Integer` usw.) liegen zwar auch auf dem Stack, die referenzierten Objekte aber auf dem Heap.

Besitzen nun zwei Threads die Referenz auf eine Instanz (vgl. Abb. 3-6), so können die Attribute der Instanz durch beide gelesen bzw. verändert werden.

[3] Die acht elementaren Datentypen in Java sind: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float` und `double`.

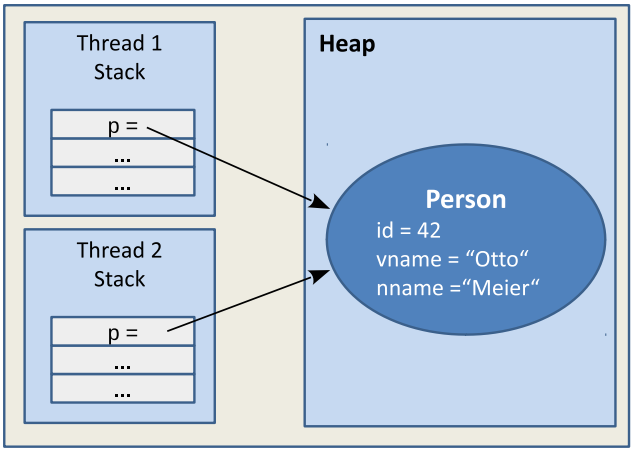

Abbildung 3-6: Gemeinsames Objekt auf dem Heap

3.2.2 Speicher auf der Hardwareebene

Das Modell mit Stacks und Heap entspricht der logischen Organisation des Speichers aus der Sicht der Programmiersprache. Auf der Hardwareebene liegen all diese Bereiche im RAM. Eine moderne Hardware hat üblicherweise eine andere Speicherarchitektur und -organisation [43]. Wie in Abbildung 3-7 zu sehen ist, besitzen die meisten CPUs einen Cache, eine Art schneller Speicher, der aus mehreren aufeinander aufbauenden Ebenen (Level) besteht. Tiefere Level sind dabei typischerweise sehr schnell, haben aber aus Kostengründen eine kleinere Größe. Aktuelle Prozessoren besitzen meist drei Cache-Level mit den Bezeichnungen L1, L2 und L3. Gängige Größen für L1 sind 4 bis 256 KB, für L2 64 bis 512 KB und für L3 2 bis 32 MB.

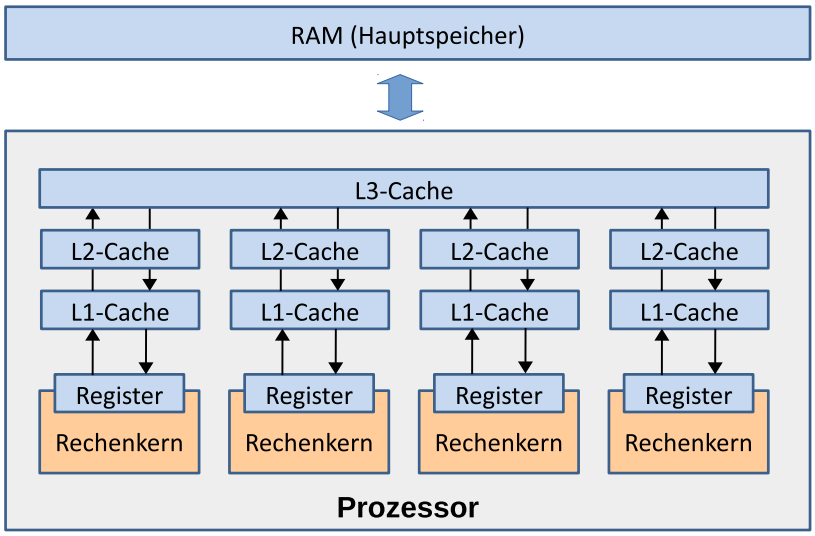

Abbildung 3-7: Moderne Prozessoren mit mehreren Kernen und zugehörigen Caches

Threads benutzen somit implizit einen Cache, in dem Teile des Speichers geladen und manipuliert werden. Objektdaten(-Attribute) können dann sowohl im Hauptspeicher als auch im Cache und teilweise in den Registern gehalten werden. Wir werden sehen, dass dies zu Problemen beim Umgang mit gemeinsamen Daten führen kann.

3.2.3 Probleme mit gemeinsam nutzbaren Daten

Sobald mehrere Threads auf gemeinsam benutzte Daten zugreifen, kann dies zu Problemen führen. Wenn Zugriffskonflikte bzw. Inkonsistenzen auftreten, können diese verschiedene Ursachen haben. Zum einen können Aktionen noch nicht abgeschlossen sein (*Atomarität*), zum anderen wurden Anweisungen umsortiert (*Umordnung*) oder Modifikationen sind noch nicht sichtbar (*Sichtbarkeit*).

Atomarität von Operationen

Eine Aktion wie `a++` besteht aus mehreren Schritten, die unterbrechbar sind, etwa:

```
reg <- a;       // Wert von a in Register laden
increment reg;
a <- reg;       // Wert zurückschreiben
```

Um die Konsistenz eines solchen kritischen Abschnitts zu gewährleisten, muss das Ganze atomar (unteilbar) ausgeführt werden. Dafür sind mehrere Mechanismen möglich.

Ein bekanntes Mittel, um Zugriffe auf Ressourcen zu koordinieren, ist *Locking*. Ein Lock entspricht einer Sperre, die ein Thread, bevor er in den kritischen Abschnitt eintritt, anfordern kann. Solange er sie besitzt, müssen alle anderen, die dieselbe Sperre haben möchten, warten. Erst wenn die Sperre explizit freigegeben wird, kann sie von genau einem der wartenden Threads erworben werden. Man spricht in dem Fall von wechselseitigem Ausschluss (*mutual exclusion* bzw. *Mutex*). Das Schlüsselwort `synchronized`, das wir unten genauer betrachten, realisiert ein Verfahren dafür.

Umordnung von Operationen

Die Java-Spezifikation erlaubt dem Compiler, aus Optimierungsgründen Anweisungen umzuordnen, solange die sogenannte sequenzielle Konsistenz (*sequential consistency*) gewährleistet ist (siehe auch weiter unten). Folgendes Programm verwendet zwei Klassenvariablen `A` und `B` sowie zwei lokale

Variablen `x` und `y`. Zu Beginn besitzen `A` und `B` den Wert null. Nebenläufig sollen nun folgende Aktionen durchgeführt werden:

```
// Thread 1            // Thread 2
x = A;                 y = B;
B = 1;                 A = 2;
```

Auf den ersten Blick scheinen die Endergebnisse `x == 2` und `y == 1` unmöglich zu sein. Da die Anweisungen für Thread 1 und Thread 2 für sich genommen keine Abhängigkeiten voneinander haben, darf der Compiler ihre Reihenfolgen vertauschen. Ein möglicher Ablauf wäre

```
// Thread 1            // Thread 2
B = 1;                 A = 2;
x = A;                 y = B;
```

was zu diesen anscheinend unmöglichen Endergebnissen führen kann.

Umordnungen können durch Einfügen sogenannter Speicherbarrieren (*memory barrier* bzw. *fences*) verhindert werden. Der Zugriff auf `volatile`-Variablen entspricht z. B. einer solchen Barriere. Alle Anweisungen, die im Code vor und nach dem Zugriff auf `volatile`-Variablen stehen, dürfen über diese Grenze nicht verschoben werden.

Sichtbarkeit von Änderungen

Wenn im obigen Beispiel zuerst Thread 1 seine Ausführung abarbeitet und danach erst Thread 2 beginnt, dann kann man auch ohne Umordnungseffekt `x == 0` und `y == 0` als Endergebnis erhalten. Es gibt in Java nämlich keine Garantie dafür, dass ein Thread sofort sehen kann, was ein anderer gemacht hat. Das liegt daran, dass jeder Thread aus Performance-Gründen so lange wie möglich seinen eigenen Cache verwendet.

Zu bemerken ist, dass dieser Effekt auf Maschinen mit nur einem Kern nicht auftritt, weil mit jedem Thread-Wechsel der Cache ausgetauscht wird. In einer Multicore-Umgebung ist die obige Anomalie durchaus möglich.

3.2.4 Sequenzielle Konsistenz

Die sequenzielle Konsistenz garantiert, dass ein Thread sehen kann, was die anderen vor ihm im Speicher geändert haben, und dass die Wirkung mit der Reihenfolge im Code übereinstimmt [33]. Aufgrund von Optimierungsmöglichkeiten wird in Java die strikte sequenzielle Konsistenz nicht unterstützt. Stattdessen wird eine schwächere Form garantiert (*relaxed memory consistency*): Ein Thread kann zum Beginn eines definierten Synchro-

nisationspunkts sehen, was andere vorher geändert haben, die denselben Synchronisationspunkt benutzen. Synchronisationspunkte sind:

- Zugriff auf `volatile`-Variablen: Das Schreiben einer `volatile`-Variablen bewirkt die Synchronisierung mit allen Threads, die zu einem späteren Zeitpunkt die Variable lesen.
- Lock-Objekte bzw. `synchronized`: Eine Lock-Freigabe synchronisiert alle durchgeführten Änderungen mit allen Threads, die danach den Lock erwerben.
- Starten eines Threads: Alle Aktionen vor dem Starten finden vor der ersten des neu gestarteten Threads statt.
- Die Initialisierung mit Defaultwerten (0, `false` oder `null`) aller Variablen sorgt für die Synchronisierung mit dem ersten Zugriff.
- Das Ende eines Threads bewirkt die Synchronisierung mit jeder Aktion eines auf dessen Ende wartenden Threads. Wenn z.B. ein `join`-Aufruf zurückkehrt, sieht der Aufrufer alle von dem Thread gemachten Änderungen.
- Wenn Thread T1 Thread T2 unterbricht, wird garantiert, dass alle Threads die Unterbrechung sehen. Ein Aufruf von `isInterrupted` liefert immer den aktuellen Unterbrechungsstatus.

Das Schreiben und Erneuern der Cache-lokalen Daten sind durch diese Regeln festgelegt.

Hinweis

Um das in der Sprache festgelegte Speichermodell umsetzen zu können, benötigt der Java-Compiler entsprechende Hardwareunterstützung. Je nach CPU sind verschiedene Realisierungen zu finden.

Zur Notation schreiben wir $a > b$, wenn die Aktion a vor b ausgeführt wird und der Effekt von a in b sichtbar ist, d.h., wenn a eine Speicheroperation ist, wird die Änderung in den Hauptspeicher geschrieben.

Doug Lea (siehe [35]) unterscheidet vier Typen von Speicherbarrieren (Synchronisationspunkte), mit denen man den Effekt des Java-Speichermodells erklären kann:

LoadLoad Für alle Ladeoperationen a vor und alle Ladeoperationen b nach der Barriere gilt: $a > b$.

StoreStore Für alle Schreiboperationen a vor und alle Schreiboperationen b nach der Barriere gilt: $a > b$.

LoadStore Für alle Ladeoperationen a vor und alle Schreiboperationen b nach der Barriere gilt: $a > b$.

StoreLoad Für alle Schreiboperationen a vor und Ladeoperationen b nach der Barriere gilt: $a > b$.

Die jeweilige Reihenfolge der Operationen vor oder nach einer Barriere ist dabei nicht festgelegt. Im Allgemeinen sind die Kombinationen dieser Typen möglich.

3.2.5 Thread-sichere Daten und unveränderliche Objekte

Jeder Thread greift während seiner Ausführung auf verschiedene Daten zu:

- Auf lokale Daten, die sich auf seinem eigenen Stack befinden. Diese müssen nicht geschützt werden.
- Auf globale Daten (im Form von Klassenobjekten bzw. -variablen). Wenn auf sie nur lesend zugegriffen wird, müssen keine gesonderten Maßnahmen getroffen werden. Werden sie aber von einem Thread geändert, müssen sie mit einem speziellen Mechanismus geschützt werden, um die Konsistenz der Daten zu gewährleisten.
- Auf gemeinsam benutzte Objekte, die auf dem Heap liegen. Objekte dieser Art werden von einem Thread angelegt und sind für alle anderen sichtbar, wenn sie deren Referenzen kennen. Wie globale Objekte müssen sie ggf. geschützt werden.

Der Zustand eines Objekts ist durch die Werte seiner Attribute gegeben. Temporäre Änderungen können das Objekt in einen inkonsistenten Zustand bringen, der nicht weiter schlimm ist, solange währenddessen kein weiterer Thread seine Daten liest.

Daten sind *Thread-sicher*, wenn sie von anderen Objekten immer in einem gültigen Zustand gesehen werden. Die Thread-Sicherheit wird in folgenden Fällen gewährleistet:

- Wenn Objekte nur von einem Thread verwendet werden. In einem Programmfluss gilt stets die sequenzielle Konsistenz. Der Compiler darf die Reihenfolge der unabhängigen Anweisungen mit der Garantie umordnen, dass jede Änderung einer Variablen für nachfolgende Schritte sichtbar ist.
- Wenn gemeinsame Objekte unveränderlich sind. Dabei sind einige Punkte zu beachten, die wir im Abschnitt 3.3 diskutieren.
- Wenn Thread-lokale Daten verwendet werden (siehe Abschnitt 3.6).
- Wenn Zugriffe auf veränderbare Objekte durch Sperren gesichert werden (siehe Kapitel 4). Unter Umständen ist der Einsatz des `volatile`-

Konzepts (siehe Abschnitt 3.4) oder *atomarer Operationen* ausreichend (siehe Kapitel 7).
- Wenn die Kommunikation zwischen Threads über ein sicheres Framework durchgeführt wird. Durch das Framework wird die Konsistenz des Objekts zum Beispiel durch Kopieren oder durch Schützen mit Sperren gewährleistet.

3.3 Unveränderbare Objekte

Ein Objekt ist unveränderbar (*immutable*), wenn sein Zustand nach der Erzeugung nicht mehr verändert werden kann. Ein richtiger Einsatz von unveränderbaren Objekten hat viele Vorteile:

- *Einfachheit:* Sie sind einfacher zu testen.
- *Thread-Sicherheit:* Sie können ohne Probleme von verschiedenen Threads benutzt werden.
- *Speichereffizienz:* Es werden zum Schutz der Objekte normalerweise keine Kopien benötigt.

Klassen für unveränderbare Objekte sind durch folgende Eigenschaften gekennzeichnet:

- Keine Methode darf die Objektdaten modifizieren.
- Alle Attribute sollen `final` sein. Sie erhalten nach der Konstruktion ihren festen Wert.
- Keine Überschreibung der Methoden in den Unterklassen. Die Klasse kann zum Beispiel `final` sein. Eine andere Möglichkeit besteht darin, Konstruktoren `private` zu deklarieren und (Klassen-)Fabrikmethoden (*factory method*) zu definieren, um unveränderbare Objekte zu liefern.
- Für den Fall, dass ein Attribut eine Referenz zu einem veränderbaren Objekt ist, muss Folgendes beachtet werden:
  - Keine Methode darf das Objekt verändern.
  - Das Objekt gehört ausschließlich der umschließenden Instanz. Das Objekt muss ggf. kopiert werden, wie im nachstehenden Codefragment zu sehen ist:

```
class Immutable
{
  private final char[] code;

  public Immutable(char[] code)
  {
    // Für den internen Gebrauch wird eine
    // Kopie angefertigt
```

```
    this.code = code.clone();
  }

  public char[] getCode()
  {
    // return code; wäre hier falsch!
    return code.clone();
  }
}
```

Hinweis

- Erst nachdem ein Konstruktor komplett durchlaufen ist, ist ein Objekt vollständig konstruiert. Da das Starten eines Threads einem Synchronisationspunkt entspricht, ist es sicher, wenn das unveränderbare Objekt nach dessen Konstruktion von einem anderen Thread benutzt wird.
- Die Bekanntmachung einer Referenz an andere Threads, deren zugehöriges Objekt erst später erzeugt wird, ist eine Fehlerquelle.

3.4 Volatile-Attribute

In Abschnitt 2.3.3 haben wir gesehen, dass eine Kennzeichnung mit `volatile` wichtig ist, wenn immer der aktuelle Wert benutzt werden soll. Das Java-Speichermodell sichert sogar noch mehr zu, nämlich:

- Das Schreiben und Lesen von `volatile`-Variablen sind unteilbar.
- Wenn Thread A eine `volatile`-Variable `v` modifiziert und Thread B sie danach liest, dann sind auch alle anderen von Thread A vor der Modifikation von `v` durchgeführten Änderungen für B sichtbar (*StoreLoad-Barriere*).
- Zugriffe auf `volatile`-Variablen dürfen nicht umgeordnet werden.
- Normale Anweisungen dürfen nicht mit Zugriffen auf `volatile`-Variablen vertauscht werden.

Betrachten wir das folgende Beispiel:

```
static int a  = 0;
static volatile int b = 0;


// Thread A
...
a = 3;
b = 2; // Änderung einer volatile Variablen

// Thread B
System.out.println(b + " " + a);
```

Es wird garantiert, dass auch alle durchgeführten Änderungen von Thread A, die vor der Zuweisung der `volatile`-Variablen `b` gemacht wurden, für Thread B sichtbar sind, wenn er auf `b` zugreift (*StoreLoad-Barriere*). In diesem Fall werden die Werte `2` und `3` ausgegeben[4].

Hinweis

Es ist zu beachten, dass für `volatile`-Referenzvariablen Folgendes gilt:

- Es wird garantiert, dass die Referenz immer aktuell und jede Änderung sichtbar ist.
- Es wird **nicht** garantiert, dass der »Inhalt« des referenzierten Objekts aktuell ist.

Praxistipp

In [28] werden folgende Regeln für den Einsatz von `volatile` empfohlen:

Regel 1 Der zu schreibende, neue Wert der Variablen ist unabhängig vom gegenwärtigen Wert (die Variable ist zustandslos).

Regel 2 Die Variable ist unabhängig von anderen.

[4] Bei `System.out.println(a + " " + b);` kann durchaus `0 0` ausgegeben werden, wenn Thread A die Zuweisung `b = 2;` noch nicht ausgeführt hat.

Typische Anwendungen sind einmalige Ereignisse (in Abschnitt 2.3.3 zum Stoppen eines Threads) oder die Veröffentlichung von Informationen (zum Beispiel Temperaturmessungen), die zustandslos sind.

Wie oben erwähnt wurde, entspricht der Zugriff auf eine `volatile`-Variable einem Punkt, bei dem der Cache mit dem Hauptspeicher synchronisiert wird. Dies ist eine relativ »teure« Angelegenheit. Zur Veröffentlichung von Aktualisierungen sind andere Lösungen (z.B. über das *Observer*-Muster) unter Umständen effizienter.

3.5 Final-Attribute

Neben den Sichtbarkeitsregeln für `volatile` gibt es auch Entsprechendes für mit `final` gekennzeichnete Attribute. Es gilt, dass sie entweder direkt bei der Deklaration oder im Konstruktor initialisiert werden können. Alle `final`-Attribute sind nach der Objekterzeugung komplett initialisiert und nicht mehr veränderbar. Dabei bezieht sich die »Unveränderbarkeit« nur auf das Attribut selbst. Handelt es sich bei dem Attribut um eine Referenz, so wird das `final` nicht auf den »Inhalt« des referenzierten Objekts ausgedehnt.

Sichtbarkeitsregeln für referenzierte Objekte

Werden über `final`-Attribute Objekte referenziert, so kann es beim falschen Gebrauch in einer Multithreaded-Umgebung zu unerwünschten Effekten kommen. Liest ein Thread zum ersten Mal ein `final`-Attribut, so werden auch alle über die Referenz erreichbaren Objekte, die sogenannte *transitive Hülle* gelesen und dauerhaft in den Cache übertragen. Es ist wichtig zu wissen, dass sich die transitive Hülle in die Tiefe über mehrere Ebenen des Objektgraphen erstrecken kann.

Solange alle beteiligten Objekte *immutable* sind, kann es zu keinem Seiteneffekt bei nebenläufigen Zugriffen kommen[5]. Ist das nicht der Fall, dann muss man aufpassen, da Veränderungen nicht unbedingt für andere Threads gleich sichtbar sind. Es sei denn, man benutzt hier die Mechanismen `volatile` oder `synchronized`[6].

[5] In dem Fall kann sich auch die Gestalt der transitiven Hülle nicht ändern.

[6] Das Schlüsselwort `synchronized` wird in Kapitel 4 besprochen.