Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Synchronisation: Basis-Mechanismen und -Ausdrucksmittel

- Threadsicherheit, kritische Abschnitte, Atomarität,
- Grundlegende Synchronisationsmittel:
  Mutex, Bedingungsvariable, volatile, join
- Java-Speichermodell

Kritische Abschnitte und Thread-sicherer Code

Threads und ihr Scheduling

- Für die Ausführung eines Threads ist die Existenz anderer Threads oft uninteressant: sie kann ignoriert werden
- Es gehört sogar den wesentlichen Bedingungen der Organisation des Codes in Threads, dass diese „für sich“ in der Regel arbeiten können.
- Das *Scheduling* – die exakte Art und Reihenfolge der Ausführung – kann dem *Scheduler* als ein – für die Logik der Ausführung – irrelevantes Detail überlassen werden

Kritische Abschnitte

- Manchmal kommen sich Threads jedoch in die Quere
- Das *Scheduling* ist dann für die Logik der Programmausführung nicht mehr egal es muss vom Programm(-ierer) beeinflusst / beschränkt werden
- Problematisch ist Programmcode der von einem Thread völlig korrekt ausgeführt werden kann, bei der gleichzeitigen / verschränkten Ausführung durch mehrere Threads aber eventuell unerwünschte Effekte hat.
- Dieser Code wird **kritischer Abschnitt** genannt.

Thread-sicher

Code, der kein kritischer Abschnitt ist, wird **Thread-sicher** genannt

Thread-Sicherheit

Eine korrekte Klasse ist Thread-sicher,

- **wenn sie auch dann noch korrekt ist,**
- **wenn ihr Code von beliebig vielen Threads**
  unter jedem denkbaren Scheduling
  mit jeder denkbaren Verschränkung
  ausgeführt wird.

„ihr Code“ ist dabei:

- **jede nicht-private statische Methode**
- **die Verwendung jedes nicht privaten statischen Datenfeldes**
- **jede nicht-private Methode eines bestimmten Objekts**
- **die Verwendung jedes nicht privaten Datenfeldes eines Objekts**

Eine korrekte Klasse ist nicht Thread-sicher,

wenn inkonsistente Zustände der Klasse / ihrer Objekte von außen beobachtet werden können.

Wechselseitiger Ausschluss

Eine nicht Thread-sichere Klasse kann durch wechselseitigen Ausschluss Thread-sicher gemacht werden

Wechselseitiger Ausschluss ist eine Beschränkung des *Schedulings:* Dem Sheduler wird die Freiheit, die Ausführung nach eigenen Kriterien beliebig zu zu gestalten, punktuell entzogen

Thread-Sicherheit

Beispiele

```
class Avarage_A {

  private var sum: Double = 0.0

  def avarage (x: Double, y: Double): Double = {
     sum = x+y;
     sum = sum / 2;
     sum
  }

}
```

Nicht thread-sicher:

Es sind eine Ausführungen durch mehrere Threads denkbar und möglich, die zu einem falschen Ergebnis führen.

Welche? Welche inkonsistenten Zustände werden beobachtbar?

```
class Avarage_B {

  def avarage (x: Double, y: Double): Double = {
     var sum = x+y;
     sum = sum / 2;
     sum
  }

}
```

Thread-sicher:

Es sind keine Ausführungen durch mehrere Threads denkbar und möglich, die zu einem falschen Ergebnis führen.

Warum?

Thread-Sicherheit

Beispiele

```
class Counter {

  private var counter: Int = 0;

  def inc: Int = {
    counter = counter + 1
    counter
  }

}
```

Thread-sicher: Erfüllt die Klasse ihre Spezifikation bei jeder denkbaren Ausführung durch mehrere Threads?

Thread-sicher in Bezug auf welche Spezifikation (Klasseninvariante) ?

Thread-Sicherheit

Beispiele

```
class Counter {

  private var counter: Int = 0;

  def inc: Int = {
    counter = counter + 1
    counter
  }

}
```

Nicht Thread-sicher, wenn die Spezifikation / Klasseninvariante lautet:

inc liefert stets die Zahl seiner Aufrufe

counter enthält die Zahl der Aufrufe von inc.

Warum – welche inkonsistenten Zustände werden sichtbar?

Ob eine Klasse threadsicher ist, kann nur in Bezug auf ihre Spezifikation beurteilt werden.

Thread-Sicherheit

Zustandslose Klassen sind immer Thread-sicher

(Darum wird der der funktionale Stil oft bevorzugt)

Methoden die nur auf den Stack zugreifen sind immer thread-sicher

(Klassen mit) Methoden die, auf Objekt- oder Klassen-Variablen zugreifen, müssen bewertet werden:

- **Erfolgt ein konkurrierender Zugriff?**
- **Wenn ja entstehen dabei kritische Abschnitte?**
- **Thread-Sicherheit ist nicht absolut, sondern relativ**
  - **zur Spezifikation der Methode**
  - **zur Atomarität der Aktionen**

Threadsichere Klassen / Methode vermeiden Wettbewerbssituationen

Wettbewerbssituationen entstehen durch ungewollte Verschränkungen

Verschränkungen sind im Rahmen der Atomarität von Aktionen möglich

Atomarität

Verschränkung

- Vermischte Ausführung von Teilaktionen durch unterschiedliche Threads

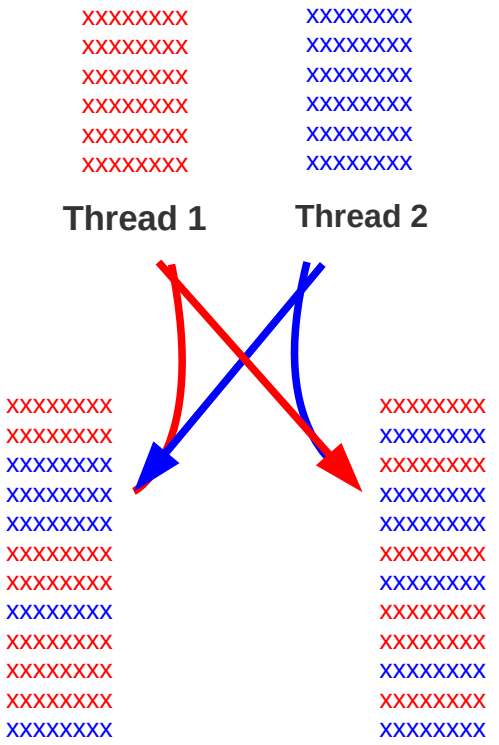
Atomare Aktion

- Eine atomare Aktion kann nicht durch eine andere Aktion unterbrochen oder durch eine andere gleichzeitig ablaufende Aktion in ihrem Verhalten beeinflusst werden
- Atomare Aktionen sind damit die „Untergrenze“ möglicher Verschränkungen
- Gegenseitiger Ausschluss = Erzwinge Atomarität, die nicht sowieso „vom System“ gewährleistet wird.

Atomare Aktionen der JVM

- Alle Zugriffe auf kleine Variablen sind atomar

  Kleine Variablen: int / boolean / Referenzen
- Alle anderen Zugriffe sich nicht atomar



Zwei mögliche verschränkte Ausführungen

Beipiel: Die Ausführung von

```
double b = 5.0;
```

ist nicht atomar: sie erfolgt u.U. in zwei Schritten ein anderer Thread könnte dazwischen gehen.

Nebenläufigkeit auf der JVM und in JVM-Sprachen

Mechnismen

Die JVM bietet einige grundlegende Mechanismen zur Nebenläufigkeit:

- **Threads:**
  Definieren, starten, stoppen
- **Synchronisation von Threads:**
  - **Mutexe, Bedingungsvariablen**
  - **Volatile Variablen**
  - **Join: Warten auf das Ende eines Threds**

Sprachmittel

Diese Mechanismen werden von JVM-basierten Sprachen (Java, Groovy, Clojure, Scala, ...) mehr oder weniger direkt unterstützt:

- **teils in der Sprache (Schlüsselworte, Sprachkonzepte),**
- **Teils als Bestandteil der API**

Monitor-Konzept: Basis-Inspiration der Nebenläufigkeit auf der JVM

Monitor-Konzept

Stand der SW-Technik in den 90ern (Zeit der Java-Definition)

Ideal für HW-nahe, systemnahe Anwendungen (Geplantes Einsatzgebiet von Java)

Entdeckt / definiert von Tony Hoare (Ende 1960er)

Basis-Idee: Kombiniere Datenabstraktionen (Objekte / Klassen) mit Synchronisation

Monitor = Klasse + Synchronisation

Kapsele

- Daten,
- ihre Zugriffsmethoden und
- deren Synchronisation

in einem Konstrukt (die „Daten“ sind für ihre korrekte Verwendung verantwortlich)

Bedingungssynchronisation und gegenseitiger Ausschluss

Monitore benötigen Systemunterstützung, üblich und in der JVM integriert sind:

- Mutexe für den gegenseitigen Ausschluss
- Bedingungsvariablen für die Bedingungssynchronisation

Synchronisation: JVM-Grundausstattung

Java verwendet zwei wichtige sehr wichtige Synchronisationsmittel: Mutexe und Bedingungsvariablen
Mit jedem Objekt ist jeweils ein Mutex und eine Bedingungsvariable verbunden,
die allerdings nur indirekt zugreifbar sind.

Mutex

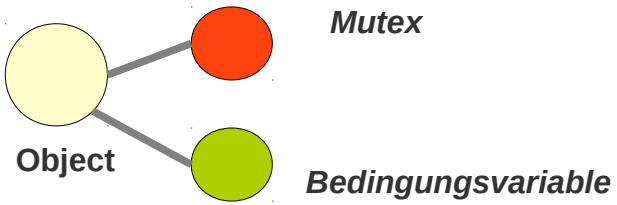
Mutexe dienen der Bedingungssynchronisation durch gegenseitigen Ausschluss.

synchronized nutzt den Mutex

Bedingungsvariable

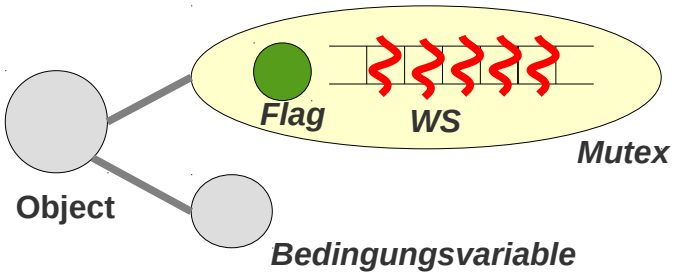
Bedinungsvariablen werden zur Bedingungs-synchronisation eingesetzt

wait / notify nutzen die Bedinungsvariable



Gegenseitiger Ausschluss: Vermeide Konflikte
Bedingungssynchronisation: Koordiniere Aktionen

JVM-Mechanismus: Mutex



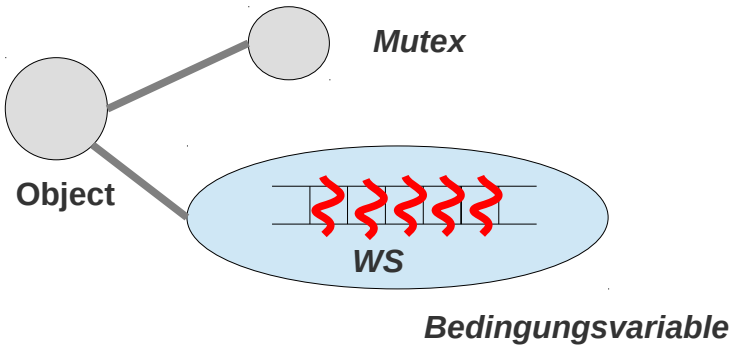
Ein Mutex besteht aus

- einem Flag (boolesche Variable): frei / nicht frei
- einer Warteschlange von Prozessen

Er bietet zwei Operationen

- lock (up, …)
  Atomare Aktion die bei freiem Mutex diesen nicht-frei setzt und bei nicht-freiem Mutex den aufrufenden Thread in der Warteschlange blockiert
- unlock (down, …)
  Atomare Aktion die den Mutex freigibt und einen der wartenden Threads reaktiviert, falls ein solcher existiert

JVM-Mechanismus: Bedingungsvariable



Ein Bedingungsvariable besteht aus

- einer **Warteschlange** von Prozessen

Sie bietet die Operationen

- **wait**
  Atomare Aktion die den aufrufenden Thread in der Warteschlange blockiert
- **notify** (signal, ...)
  Atomare Aktion die einen der Threads in der Warteschlange frei gibt
- **notifyAll** (broadcast, ...)
  Atomare Aktion die alle Threads in der Warteschlange frei gibt

Spurious notify:
Ein Thread kann auch ohne notify (spontan) reaktiviert werden!

Java-Sprachmittel: Gegenseitiger Ausschluss

Nebenläufigkeit wird in Java durch eine Mischung von Bibliotheksfunktionen und Sprachbestandteilen unterstützt.

Das Schlüsselwort **synchronized** (in Kombination mit **class**) sowie **volatile** gehört zu den Sprachbestandteilen

Die Bibliotheksfunktionalität steckt zunächst in der Klasse **Objekt**.

synchronized gibt es in Java in zwei Varianten

- **Synchronisierte Methoden**

  Beim Aufruf der Methode wird der Mutex des Objekts der Methode gelockt und beim Verlassen wird er wieder frei gegeben

  statische Methode: Mutex des Klassen-Objekts

  nicht-statische Methode: Mutex von this

- **Synchronisierte Code-Blöcke**

  Vor dem Betreten des Codeblocks wird der Mutex des angegeben Objekts gelockt, beim Verlassen wieder frei gegeben.

```
class C {

  synchronized static void m1() {
    . . .
  }

  void m2() {
     synchronized (getClass()) {
        . . .
     }
   }
}
```

Java: synchronisierte Methode und Codeblock mit Bezug auf den gleichen Mutex

Der gegenseitige Ausschluss basiert auf immer einem Mutex:

Nur Methoden oder Blöcke, die sich auf den Mutex des selben Objekts beziehen, operieren im gegenseitigen Ausschluss.

Java-Sprachmittel: Bedingungssynchronisation

Die Bibliotheksfunktionalität steckt zunächst in der Klasse **Object**.

Die relevanten Methoden der Klasse **Object** beziehen sich auf die Bedingungsvariable (Java-Sprech: „*this object's monitor*“) des Objekts.

Die Methoden sind:

- **void notify()**

  reakiviert einen wartenden Thread in der Warteschlange der Bedingungsvariable.

  Der aufrufende Thread muss im Besitz des Mutex' des Objekts sein.

  Der reaktivierte Thread ist nicht im Besitz des Mutex'.

- **void notifyAll()**

  Wie notify, nur werden alle in der Warteschlange reaktiviert.

- **void wait()**

  Der aufrufende Thread wird deaktiviert und in der Warteschlange der Bedingungsvariable abgelegt. Der aufrufende Thread muss im Besitz des Mutex' des Objekts sein.

- **void wait(long timeout)**

  Wie wait, nur mit Timerüberwachung. Nach Ablauf des Timers wird der Thread reaktiviert, er ist dann nicht im Besitz des Mutex.

- **void wait(long timeout, int nanos)**

  Wie vorherige Methode nur mit genauerer Zeitangabe.

Alle wait-Methoden können mit **interrupt** abgebrochen werden, sie werfen dann die InterruptedException, und setzen das Interrupt-Flag zurück.

Scala: Gegenseitiger Ausschluss

Scala nutzt die Mechanismen der JVM – mit Objekten assoziierte Mutexe und Bedingungsvariablen.

Die relevanten Methoden der Klasse Object sind in Scala verfügbar

Das Schlüsselwort synchronized steht in Scala nicht zur Verfügung: Alle Funktionalität zur Nebenläufigkeit ist damit API-basiert (kein Mix von Sprache und API wie in Java)

In Scala ist synchronized eine Methode der Klasse AnyRef:

```
final def synchronized[T0](arg0: ⇒ T0): T0
```

Ihr wird ein (nicht ausgewertetes) Argument übergeben, das einen Wert von beliebigem Typ T0 berechnet.

Meist ist das Argument ein Codeblock vom Typ Unit

Scala: Gegenseitiger Ausschluss

Beispiele (4 äquivalente Definitionen)

```
class Account(private var b:Int = 0) {

  def balance: Int = synchronized { b }

  def deposit(value: Int) : Unit = synchronized {
    b = b+value
  }

  def withdraw(value: Int) : Unit = synchronized {
    b = b-value
  }
}
```

```
class Account(private var b:Int = 0) {

  def balance: Int = synchronized[Int] { b }

  def deposit(value: Int) : Unit = synchronized[Unit] {
    b = b+value
  }

  def withdraw(value: Int) : Unit = synchronized[Unit] {
    b = b-value
  }
}
```

Der Scala-Compiler ergänzt die generischen Parameter

```
class Account(private var b:Int = 0) {

  def balance: Int =  b

  def deposit(value: Int) : Unit = synchronized {
    b = b+value
  }

  def withdraw(value: Int) : Unit = synchronized {
    b = b-value
  }
}
```

Der Lesezugriff auf eine Int-Variable muss nicht synchronisiert werden

```
class Account(private var b:Int = 0) {

  def balance: Int = this.synchronized { b }

  def deposit(value: Int) : Unit =
    this.synchronized {
      b = b+value
    }

  def withdraw(value: Int) : Unit =
    this.synchronized {
      b = b-value
    }
}
```

Das Objekt this wird wie üblich automatisch ergänzt

Scala: Bedingungssynchronisation

wait und notify / notifyAll sind Methoden der Klasse AnyRef:

```
final def notify(): Unit
          Wakes up a single thread that is waiting on the receiver object's monitor.
final def notifyAll(): Unit
          Wakes up all threads that are waiting on the receiver object's monitor.
final def synchronized[T0](arg0: ⇒ T0): T0
      def toString(): String
          Creates a String representation of this object.
final def wait(): Unit
final def wait(arg0: Long, arg1: Int): Unit
final def wait(arg0: Long): Unit
```

Auszug aus der Scala-Api-Doku. von AnyRef (~ Object in Java)

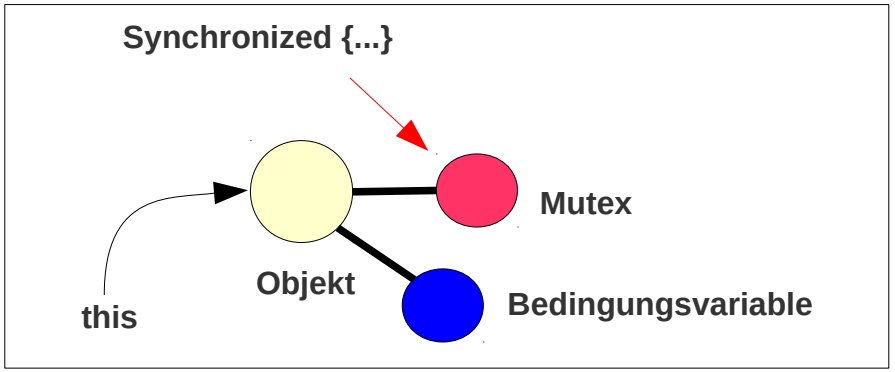
Die Methoden werden wie in Java verwendet

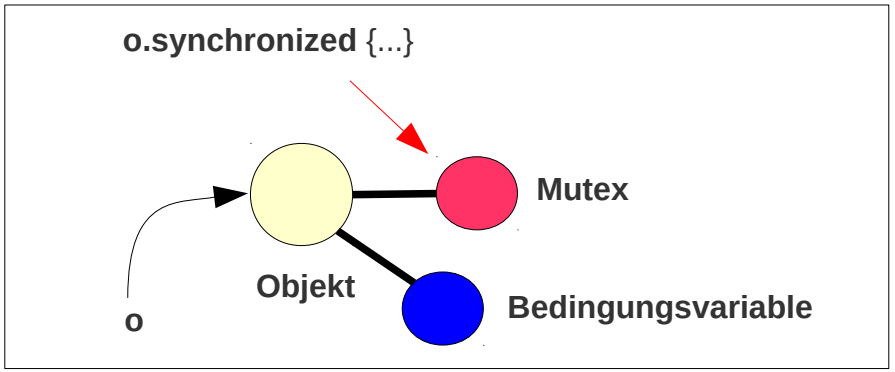
Zusammenspiel von Mutex und Bedingungsvariable

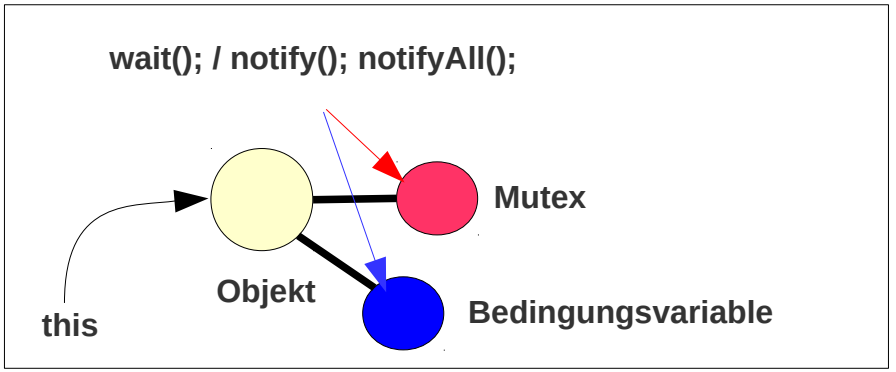
- Jedes synchronized bezieht sich auf einen Mutex
- Jedes Wait / notify bezieht sich
  - auf einen Mutex und
  - und eine Bedingungsvariable

  Dabei gilt:
  - Der aufrufende Thread muss im Besitz des Mutex sein
  - Die Bedingungsvariable muss die Bedingungsvariable des Objekts sein, zum dem der Mutex gehört

IllegalMonitorStateException wird geworfen, wenn bei der Aufrufer einer einer wait- / notify-Methode nicht im Besitz des richtigen Mutex ist.







JVM-Mechanismus: Volatile Variablen

Änderungen an einer Variablen müssen in anderen Threads nicht sofort bemerkt werden.
Volatile: Mache Änderungen über Thread-grenzen hinaus sofort sichtbar.

Variable mit Markierung volatile:

- **Einsatz**
  Minimalistischen Synchronisationsblock realisieren:
  - **Zugriffe sind atomar**
  - **Compiler darf Zugriffe nicht umordnen**
  - **Änderungen der Variablen durch einen anderen Thread werden bemerkt**

- **Mechanismus**
  Lese- und Schreib-Operationen werden im Hauptspeicher ausgeführt
  und nicht auf lokalen Caches oder CPU-Registern
  oder Caches werden bei jeder Änderung synchronisiert

  ***volatile* = volatil ~ veränderlich (und pass darum auf die Veränderungen auf!)**

Sprachmittel: Markierung einer Variablen als Volatile

Java: Schlüsselwort volatile:
Scala: Annotation @volatile

cancel wird von einem anderen Thread ausgeführt. Volatile garantiert dass diese Änderung in run bemerkt wird.

```
class DoSomething extends Thread {

  @volatile var canceled: Boolean = false

  def cancel() { canceled = true; }

  override def run(): Unit =
    while ( ! canceled ) {
      // do something
   }

}
```

Mechanismus: Volatile Variablen

volatile: ist nicht „transitiv“: Das worauf volatile Referenzen zeigen ist nicht volatil.

```
case class Person(var firstName: String, var name: String)

object Volatile_Main extends App {

 def thread(runCode: => Unit) : Thread =
    new Thread(new Runnable{
      override def run() : Unit = runCode
    }
  )

  @volatile var person: Person = null

  thread({
    while ( person == null) { Thread.sleep(10) }
    while ( person.firstName != "Karla") { Thread.sleep(10) }
    println(person)
  }) start

  thread({
    Thread.sleep(1000)
    person = Person("Karl", "Napp")
    Thread.sleep(1000)
    person.firstName = "Karla"
    person.name = "Kahl"
  }) start

}
```

volatile: Diese Änderung (Zuweisung an person) muss im ersten Thread sichtbar / beobachtbar sein.

Diese Änderungen (innerhalb von person) können aber müssen nicht im ersten Thread sichtbar / beobachtbar sein.

Mechanismus: Volatile Variablen

volatile: erzwingt die Sichtbarkeit vorheriger Operationen

```
@volatile var person: Person = null

thread({
  while ( person == null) { Thread.sleep(10) }
  while ( person.firstName != "Karla") { Thread.sleep(10) }
  println(person)
}) start

 thread({
  Thread.sleep(1000)
  person = Person("Karl", "Napp")
  Thread.sleep(1000)
  person.firstName = "Karla"
  person.name = "Kahl"
}) start


 thread({
  Thread.sleep(1000)

  val personTemp = Person("Karl", "Napp")

  Thread.sleep(1000)
  personTemp.firstName = "Karla"
  personTemp.name = "Kahl"

  person = personTemp
}) start
```

Alle Änderungen der Person vor der Zuweisung an die volatile Variable person müssen jetzt im zweiten Thread sichtbar sein.

Mechanismus: Join warte auf das Ende eines Threads

join: blockiert einen Thread bis ein anderer beendet wurde

Alle Änderungen die ein Thread vorgenommen hat sind nach einem join in anderen garantiert sichtbar.

```
object Join_Main extends App {

  def thread(runCode: => Unit) : Thread =
    new Thread(new Runnable{
      override def run() : Unit = runCode
    }
  )

  val t = thread({
    .....
    Thread.sleep(5000)
  })

  thread ({
    t.join
    .....
    println("Finally t has come to an end")
  }) start

  t start
}
```

Alle Änderungen (an globalen Variablen) hier

können hier beobachtet werden (auch wenn die Variablen nicht volatile sind).

http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4

JMM Java Memory Model: das Speichermodell der JVM

JMM: Abstraktion der realen Hardware in Bezug auf die Speicherung von Daten.

Das Speichermodell hat den Sinn eine exakte Definition der Wirkung von Lese- und Schreiboperationen zu geben

Dabei soll

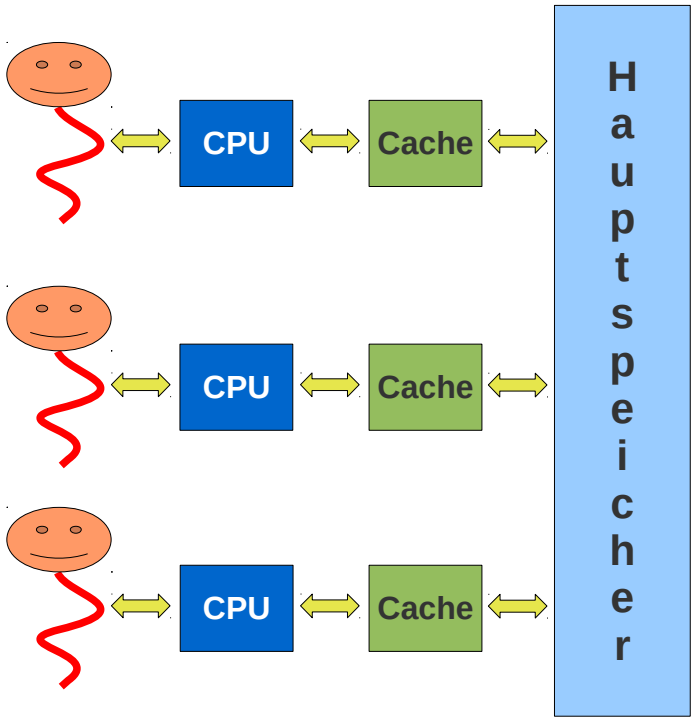
- **in Gegenwart von Caches und vielen Cores**
- **das Verhalten eines laufenden Programms so definiert werden, dass**
  - **effiziente Parallelverarbeitung und Caching möglich ist und trotzdem**
  - **ein Programm mit vielen Threads noch ein definiertes Verhalten hat, das nicht zu weit von einer naiven Semantik abweicht**

    Naive Semantik: Änderungen die ein Thread an globalen Dingen vornimmt, werden von allen Threads bemerkt, die diese Dinge beobachten

Prinzipien

Prinzip

- Das Speichermodell geht davon aus, dass jeder Threads die Daten in einem lokalen Cache speichert
- Datenänderungen erfolgen damit zunächst unabhängig voneinander
- Mindestens zu den Zeiten, die das JMM vorgibt, müssen die lokalen Kopien der Daten synchronisiert werden
- Die Implementierung der JVM hat dafür zu sorgen, dass auf der realen Hardware die Bedingungen erfüllt werden, ohne dabei unnötige und verlangsamende Synchronisationen vorzunehmen



Behandelte Phänomene

- Atomarität: *Welche Aktionen sind unteilbar*
- Sichtbarkeit: *Unter welchen Bedingungen ist eine von einem Thread verursachte Änderung für einen anderen beobachtbar*
- Reihenfolge: *In welcher Reihenfolge können Operationen anderer Threads beobachtet werden*

Atomarität

- Zugriffe auf Referenz-Variablen sind atomar
- Zugriffe auf Variablen mit primitivem Typ ist atomar – ausgenommen long / double
- Zugriff auf volatile Variablen ist atomar

Sichtbarkeit der Wirkung einer Aktion (*happens-before*-Relation)

- **Thread-lokal**: Jede Ursache zeigt sofort Wirkung. Eine Schreiboperation wird von jeder im gleichen Thread folgenden Leseoperation bemerkt.
  (Ansonsten können die Anweisungen beliebig umsortiert werden, Caches nicht aktualisiert werden, ...)
- **Lock** Alles, was vor einem Unlock geschrieben wurde, wird nach einer einem Lock folgenden Leseoperationen bemerkt
- **Volatile** Jede Schreiboperation wird von jeder folgenden Leseoperation bemerkt
- **Join** Jede Änderung des anderen Threads wird bemerkt
- **Final** Der Wert einer finalen Objektvariablen wird von jedem Thread bemerkt
- **Start** Jede Schreiboperation vor dem Start eines Threads wird von diesem bemerkt
- **Ende** Jede Schreiboperation eines Threads wird von anderen vor dessen Ende bemerkt
- **Interrupt** Ein Interrupt wird erst nach seiner Auslösung bemerkt

Sichtbarkeit der Wirkung einer Aktion (*happens-before*-Relation)

Generell:

Alle Aktionen in einem Thread dürfen beliebig reorganisiert und optimiert werden solange davon:

- **keine offensichtliche Ursache-Wirkungs-Beziehung im Thread selbst, oder**
- **eine der definierten Ursache-Wirkungs-Beziehung über Thread-Grenzen**

verletzt wird.

Zusammenfassung

Threads sollen möglichst unabhängig von einander agieren und als unabhängige Aktionen konzipiert werden.

In Wettbewerbssituationen muss gegenseitiger Ausschluss durch das Programm definiert werden. Das Kriterium ist die geforderte / notwendige Atomarität von Methoden.

Atomaritätsforderungen ergeben sich aus der Spezifikation einer Klasse / Methode

Atomare Aktionen des Systems sind die Basis des gegenseitigen Ausschlusses, diese müssen eventuell durch das Programm erweitert werden

Atomarität auf der JVM wird durch das JMM definiert.

Die Synchronistionsmittel der JVM leiten sich aus dem Monitorkonzept ab. Es sind

- **Mutexe für den gegenseitigen Ausschluss**
- **Bedingungsvariablen für die bedingungssynchronisation**
- **Dazu kommen noch volatible Variablen und die join-Operation**

Die Thread-übergreifende Kopplung von

- **Ursache – Wert schreiben**
- **und Wirkung – Wert lesen**

ist komplexer als naiv angenommen. Jeder Thread lebt zunächst in seiner Welt

Das JMM regelt auf welche Ursache-Wirkungs-Beziehungen man sich verlassen kann, indem ein Minimum an Synchonisationsaktionen der JVM / System-Hardware vorgeben wird.