

Universität Karlsruhe (TH)

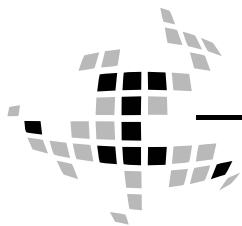
Forschungsuniversität · gegründet 1825

## Das Java-Speichermodell

Prof. Dr. Walter F. Tichy

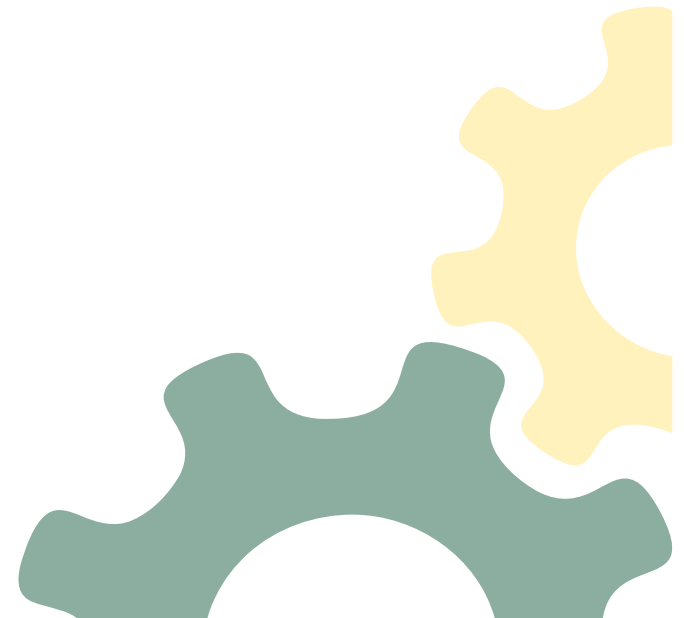
Dr. Victor Pankratius

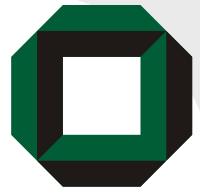
Ali Jannesari



Fakultät für **Informatik**

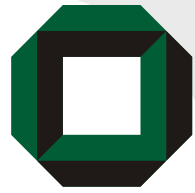
Lehrstuhl für Programmiersysteme





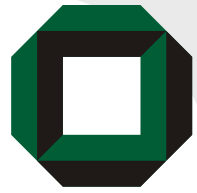
# Geschichte des Speichermodells

- Kapitel 17 der **Java-Sprachdefinition**
  - Lange unverändert (Java 1.1, 1.2, 1.3 bis 1.4).
  - **Von Anfang an umstritten**, da eine korrekte Umsetzung viele Optimierungen verbietet.
  - Die meisten Implementierungen von virtuellen Maschinen **halten sich daher nicht ganz genau an das Speichermodell**.
- **JSR-133 (Java Specification Request)**
  - Neufassung von Speichermodell und Faden Spezifikation
    - 29. Mai 2001: „Review Ballot“
    - 30. September 2004 „Final Release“
    - Teil von Java 1.5 „Tiger“
- [Links auf der letzten Folie](#)



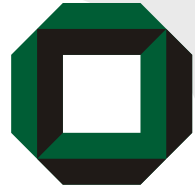
# Was ist ein Speichermodell?

- Regelt die Interaktionen von **Kontrollfäden** ...
  - ... mit dem **Hauptspeicher**, und
  - ... mit anderen **Kontrollfäden**.
- Beschreibt **Semantik paralleler Programme**
  - **Zentrale Frage**: Welchen Wert sieht eine Leseoperation von einer gemeinsamen Variable?
    - bei Schreiboperationen in anderen Kontrollfäden
      - bei „korrekter“ Synchronisation
      - bei Vorliegen von Wettlaufsituationen.
    - bei Duplikaten in Caches.



# Besonderheiten des Java Speichermodells

- Definierte Semantik auch für **falsch oder gar nicht synchronisierte** Programme.
  - Notwendig, um die **Sicherheitsgarantien** bei der Ausführung aufrecht zu erhalten.
  - Böartiger Code könnte **mutwillig** Wettlaufsituationen provozieren, um die **Sicherheitsmechanismen auszuhebeln**.



# Beispiel für Wettlaufsituation:

• Faden 1:

```
S="tmp/etc/passwd".substring(4);
```

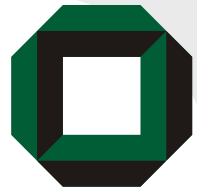
Faden 2:

```
File.delete(S);
```

Angenommen, Zeichenreihen wären nicht final und es gäbe keine Synchronisation zwischen Faden 1 und 2. Die substring-Operation ändert nur den Abstand vom Anfang der Zeichenreihe (hier von 0 auf 4).

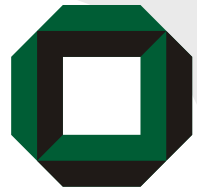
Nun könnte der parallel laufende Faden 2 vor der fertigen Konstruktion von S einen Verweis auf die unfertige Zeichenreihe bekommen, z.B. gerade noch mit Abstand 0 (Wettlaufsituation). S würde zur Überprüfung weitergegeben; Die delete-Operation wäre erlaubt. Nach der Überprüfung sei aber der Abstand zu 4 geändert. Jetzt würde die eigentliche delete-Operation starten, obwohl sie nun nicht mehr erlaubt ist.

Abhilfe: die beiden Fäden synchronisieren, um ihre Reihenfolge festzulegen. Ist hier aber nicht notwendig, da Zeichenreihen final sind, d.h. sie sind unveränderbar und Verweise auf sie werden erst erzeugt, wenn die Konstruktion fertig abgelaufen ist. Aber andere Typen sind nicht final und können daher bei unsynchronisiertem Zugriff unvorhersehbare Werte liefern.



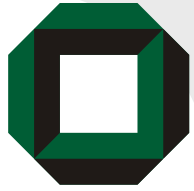
# Wofür braucht man ein Speichermodell?

- Erlaubt mögliche **Verhaltensweisen** eines parallelen Programms vorherzusagen.
  - Ziel: **Deterministische Ergebnisse trotz Nebenläufigkeit.**
  - Anforderung: zusätzliche „Hinweise“ (=Synchronisation) durch den Programmierer.
- Grundlage für die plattformunabhängige Ausführung paralleler Programme.
  - **Beschränkt VM-Implementierungen** bei **Abbildung des Java-Speichermodells** auf das Speichermodell der Ausführungsplattform.
  - **Erteilt Freiheiten** an **VM-Implementierungen** zu **möglichen Optimierungen/Code-Transformationen.**



# Ein intuitives Speichermodell: Sequentielle Konsistenz

- Eine Ausführungsreihenfolge ist **sequentiell konsistent**,
  - wenn alle Programmaktionen **total geordnet** sind,
  - und diese Ordnung konsistent mit der **Operationsreihenfolge** im Programm ist
- Das Ergebnis eines parallelen Programms kann bestimmt werden durch **beliebiges Ineinandermischen** der Operationen **aller seiner Kontrollfäden**.
- **Intuitiv**: Das ist die Art und Weise, wie man über die Ausführung paralleler Programme nachdenkt.
- **Eingeschränkt**: Verbietaet viele Optimierungen durch Übersetzer und Prozessor.



# Ausführungsreihenfolgen bei sequentieller Konsistenz

Situation:

gemeinsam genutzt:  $a == 0$ ,  $b == 0$

Faden 1:

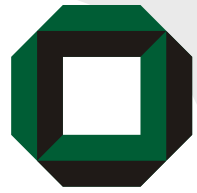
$r_1 = a$

$b = 1$

Faden 2:

$r_2 = b$

$a = 2$



# Ausführungsreihenfolgen bei sequentieller Konsistenz

a) Faden 1 zuerst:

gemeinsam genutzt:  $a == 0$ ,  $b == 0$

Faden 1:

$r_1 = a$

$b = 1$

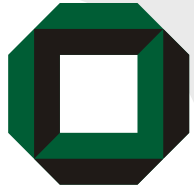
Faden 2:

$r_2 = b$

$a = 2$

Ergebnis:  $r_1 == 0$

$r_2 = 1$



# Ausführungsreihenfolgen bei sequentieller Konsistenz

b) Faden 2 zuerst:

gemeinsam genutzt:  $a == 0$ ,  $b == 0$

Faden 1:

Faden 2:

$$r_2 = b$$

$$a = 2$$

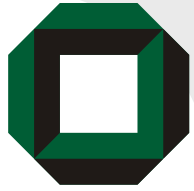
---

$$r_1 = a$$

$$b = 1$$

Ergebnis:  $r_1 == 2$

$$r_2 = 0$$



# Ausführungsreihenfolgen bei sequentieller Konsistenz

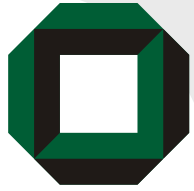
c) abwechselnd:

gemeinsam genutzt:  $a == 0$ ,  $b == 0$

Faden 1:

Faden 2:

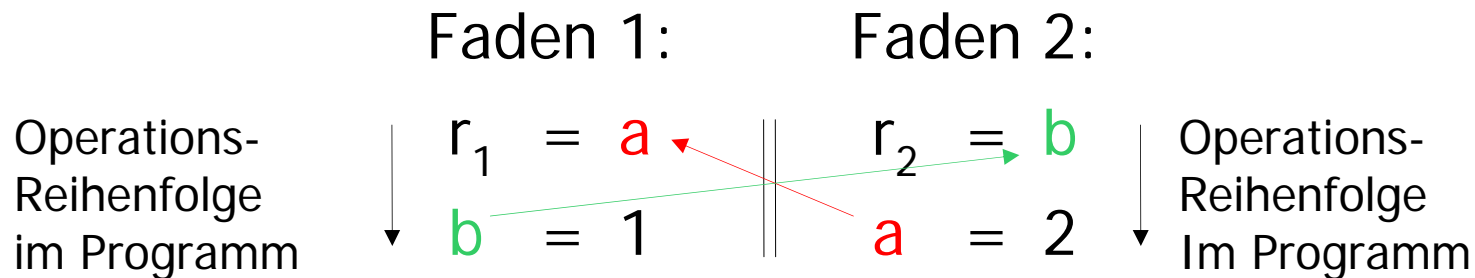
$r_1 = a$	$r_2 = b$
$b = 1$	$a = 2$
Ergebnis: $r_1 == 0$	$r_2 = 0$



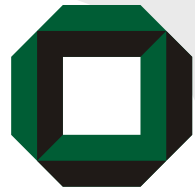
# Ausführungsreihenfolgen bei sequentieller Konsistenz

d) **unmöglich** bei sequentieller Konsistenz:

gemeinsam **genutzt**:  $a == 0, b == 0$



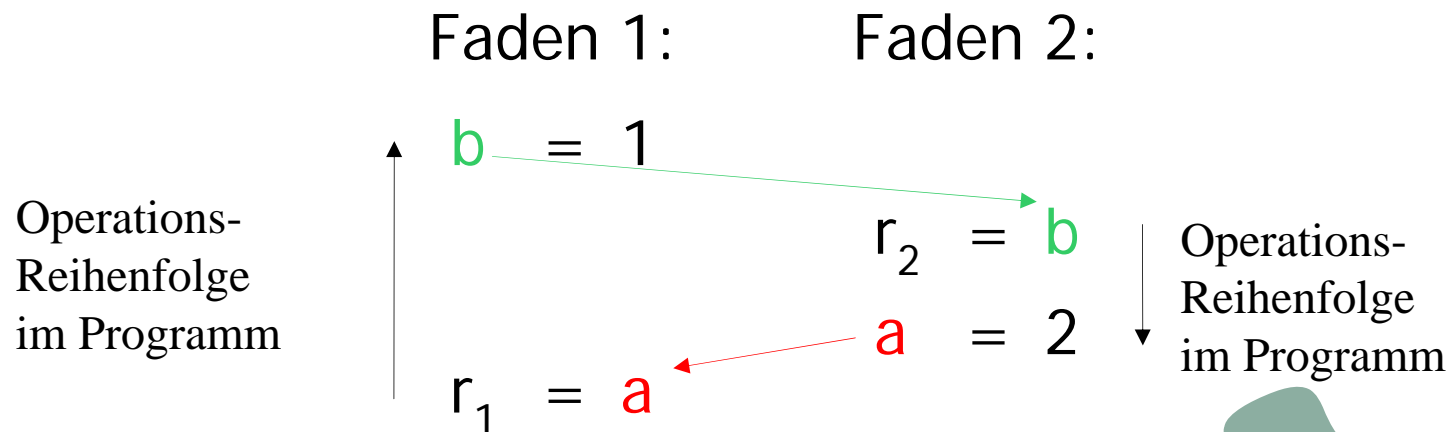
unmöglich:  $r_1 == 2$        $r_2 = 1$



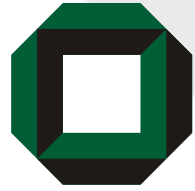
# Mögliche Reihenfolge im Java-Speichermodell

Im Java-Speichermodell ist sogar **diese Reihenfolge** möglich, da Umordnung durch Optimierung erlaubt ist, wenn (1) die Umordnung die Ausführung eines Fadens für sich genommen nicht beeinflusst und (2) keine Synchronisation mit einem anderen Faden vorkommt, die sequentielle Konsistenz erzwingen würde.

gemeinsam **genutzt**:  $a == 0$ ,  $b == 0$

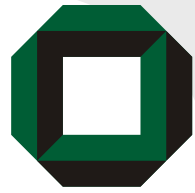


**möglich:**



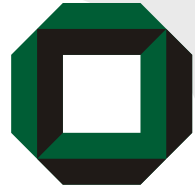
# Zugriffskonflikt

- Ein **Zugriff** ist entweder eine
  - Leseoperation, oder eine
  - Schreiboperation.
- **Gemeinsamer Zustand** ist
  - eine Instanzvariable,
  - eine statische Variable,
  - oder ein Feldelementauf die mehrere Kontrollfäden zugreifen können (auf der Halde).
- Zwei Zugriffe auf gemeinsamen Zustand **stehen in Konflikt**, wenn mindestens ein Zugriff eine **Schreiboperation** ist.
- **Aktionen** eines Fadens während seines Ablaufes sind Zugriffe und Synchronisationsaktionen (Sperrren/Freigeben von Monitoren, Zugriffe auf volatile-Variablen, Start eines Fadens)



# Die „Geschicht-Vorher-Beziehung“

- Jede **Aktion** in einem Kontrollfaden geschieht-vor jeder folgenden Aktion in **demselben Kontrollfaden**.
- Eine **Monitorfreigabe** geschieht-vor jeder folgenden Monitoranforderung **auf demselben Monitor**.
- Schreiben einer **volatile** Variable geschieht-vor jedem folgenden Lesezugriff **auf dieselbe Variable**.
- Ein Aufruf an **start()** an einem Thread-Objekt geschieht-vor jeder Aktion **im zugehörigen Kontrollfaden**.
- Alle Aktionen in einem Faden geschehen-vor einer erfolgreichen Rückkehr aus **join()** **an diesem Faden**.
- **Transitivität**: Wenn eine Aktion **a vor b** geschieht und **b vor c**, dann geschieht auch **a vor c**.

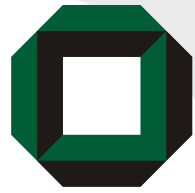


# Korrekte Synchronisierung

- Ein Programm mit zwei **in Konflikt stehenden Zugriffen**, die in keiner Geschicht-Vorher-Beziehung stehen, enthält eine mögliche **Wettlaufsituation**.

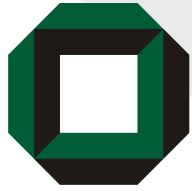
- Ein Programm ist nur dann **korrekt synchronisiert**, wenn es **keine Wettlaufsituationen** enthält.

- Alle möglichen Ausführungsergebnisse eines Programms **ohne Wettlaufsituationen** erscheinen **sequentiell konsistent**.



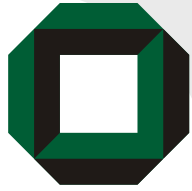
# Sichtbarkeit

- Eine Aktion in einem Kontrollfaden ist **sichtbar** für einen anderen Kontrollfaden, wenn das **Ergebnis** von dem anderen Kontrollfaden **beobachtet werden kann**.
- Damit eine Aktion  $a_1$  eines Kontrollfadens von einer Aktion  $a_2$  eines anderen Kontrollfadens **garantiert beobachtet** werden kann, muss  $a_1$  **vor**  $a_2$  passieren (d.h. in einer Geschicht-Vorher-Beziehung stehen).



# Beispiel: Sichtbarkeit

```
class Invisible {  
    boolean ok = true;  
  
    void work() {  
        int i = 0;  
        while (ok) { i++; }  
    }  
  
    void stop() {  
        ok = false;  
    }  
}
```



# Beispiel: Sichtbarkeit

```
class Invisible {  
    boolean ok = true;  
  
    void work() {  
        int i = 0;  
        while (ok) { i++; }  
    }  
  
    void stop() {  
        ok = false;  
    }  
}
```

$t_1$  muss nicht terminieren:

$t_0$ : write(ok, true)

geschieht vorher

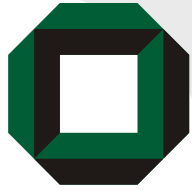
$t_1$ : read(ok)

$t_1$ : read(ok)

$t_1$ : read(ok)

Reihenfolge undefiniert

$t_2$ : write(ok, false)



# Beispiel: Sichtbarkeit

```
class Invisible {  
    volatile boolean ok = true;  
  
    void work() {  
        int i = 0;  
        while (ok) { i++; }  
    }  
  
    void stop() {  
        ok = false;  
    }  
}
```

repariert: volatile Variable

$t_0$ : write(ok, true)

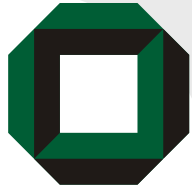
geschieht vorher

$t_1$ : read(ok)

$t_1$ : read(ok)

$t_1$ : read(ok)

$t_2$ : write(ok, false)



# Beispiel: Sichtbarkeit

```
class Invisible {  
    boolean ok = true;  
    void work() {  
        int i = 0;  
        while (true) {  
            synchronized(this) {if (!ok) break;}  
            i++;  
        }  
    }  
    synchronized void stop() {  
        ok = false;  
    }  
}
```

repariert: Synchronisation

$t_0$ : write(ok, true)

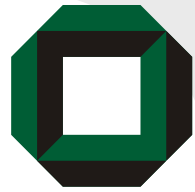
geschieht vorher

$t_1$ : read(ok)

$t_1$ : read(ok)

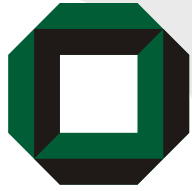
$t_1$ : read(ok)

$t_2$ : write(ok, false)

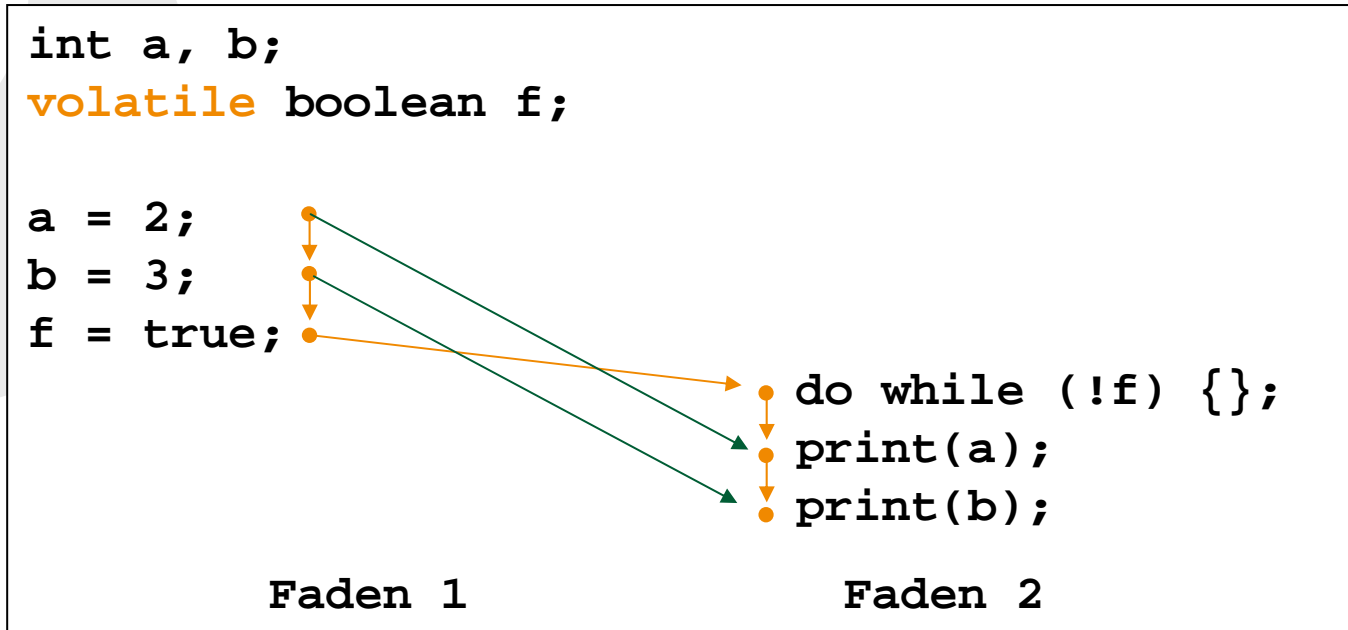


# volatile-Variablen und Felder

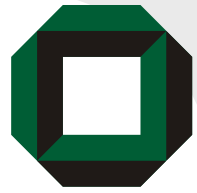
- Für eine **volatile** deklarierte, gemeinsam genutzte Variable (oder ein **volatile** deklariertes, gemeinsam genutztes Feld) gilt:
  - Die Änderung durch einen Schreibzugriff ist **sichtbar** in nachfolgenden Lesezugriffen anderer Fäden.
- Dies gilt, wie schon gezeigt, für gemeinsam genutzte Zustände möglicherweise nicht, es sei denn, die Schreiboperation im einen und die nachfolgende Leseoperation im anderen Faden werden in eine Geschicht-vorher Beziehung gesetzt!
- Bei Zugriffen auf als **volatile** deklarierten Zuständen ist die Geschicht-Vorher-Beziehung **automatisch** gegeben.
- Wegen der **Transitivität** hat der Zugriff auf einen als **volatile** deklarierten Zustand aber auch Auswirkungen auf **andere**, gemeinsam genutzte Zustände (s. nächste Folie).



# Volatile und Transitivität

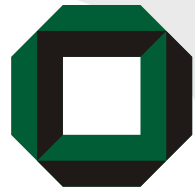


- Der schreibende Zugriff auf f in Faden 1 und der lesende Zugriff auf f in Faden 2 stehen in einer Geschicht-Vorher-Beziehung.
- Damit stehen **wegen der Transitivität** auch die jeweiligen Zugriffe auf a und b in Geschicht-Vorher-Beziehungen (grüne Pfeile).
- Anders formuliert: Wenn man eine Änderung an f bemerkt, kann man sicher sein, auch für a und b die aktuellen (also von Faden 1 geschriebenen) Werte zu lesen.



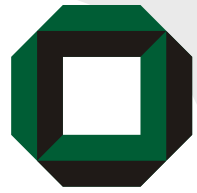
# Ohne korrekte Synchronisierung

- Problem: **Reihenfolge**
  - Die **Beobachtung** einer bestimmten Reihenfolge von Aktionen ist nur garantiert, wenn es eine **Geschicht-Vorher-Beziehung** zwischen Verursacher und Beobachter gibt.
  - Ansonsten sind **Umordnungen** erlaubt.
- Problem: **Atomizität**
  - Auch in einem Programm ohne Wettlaufsituationen kann **fehlende Atomizität** für unerwünschte Resultate verantwortlich sein
- Abhilfe: Synchronisation



# Unterschiede zu Java 1.0 - 1.4

- Sichtbarkeit bei Verwendung von mehreren Monitoren
  - **Schwächere** Garantien in Java 1.5:  
Vorher-Nachher-Beziehung entsteht nur durch Synchronisation an **demselben** Monitor.
- **Erweiterte** Garantien für volatile deklarierte Variablen
  - Reihenfolgebeziehung bei **gemischten Zugriffen** auf volatile und nicht-volatile Variablen.
  - Vor Java5 hatte volatile nur Auswirkungen auf die als volatile deklarierte Variable.
- Sicherheit bei der Objektinitialisierung
  - echte finale Variablen
- Rudimentäre Fairness
- Verhalten von wait()
  - Spontanes Erwachen ohne notify() ist erlaubt.



# Mehr Informationen über das Java Speichermodell

- Kapitel 17 von JLS, 3rd ed.  
<http://java.sun.com/docs/books/jls/index.html>
- <http://www.cs.umd.edu/users/pugh/java/memoryModel/jsr-133-faq.html>
- <http://www.jcp.org/en/jsr/detail?id=133>