Nebenläufige und verteilte Programme CS2301

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Synchronisation auf höherem Niveau

- JUC: Java-Util-Concurrent
- Kollektionen: synchronisiert / nebenläufig / mit Benachrichtigung
- Synchronisierer
- Atomare Variablen und Lock-freie Programme

JUC / *Java Util Concurrency*

Java 5: Einführung des package *java.util.concurrency* kurz JUC

Wesentliche Erweiterung der Concurrency-Features von Java

Ziel: Verbesserte Anwendungsentwicklung durch höhere (abstraktere) Hilfsmittel

Wesentliche Bestandteile:

- **Kollektionsklassen mit Synchronisation und / oder Nebenläufigkeit**
- **Atomare Klassen**
- **Synchronisierer**
- **Exceutor-Framework**

Given the difficulty of using wait and notify correctly, you should use the higher-level concurrency utilities instead.

J. Bloch, Effective Java 2nd edition

Übersicht Java-Kollektion

nicht threadsicher

- ArrayList
- LinkedList
- HashSet
- TreeSet
- HashMap
- TreeMap
- LinkedHashSet
- LinkedHashMap

Nicht in Multi-Threading-Kontext verwenden (ohne selbst für Synchronisation zu gesorgt zu haben).

synchronisiert threadsicher

- Vektor
- Stack
- Hashtable
- *synchronisierte Varianten nicht threadsicherer Kollektionen **Xyz**:* **Collections.synchronized*Xyz***

Alle Zugriffe erfolgen im gegenseitigen Ausschluss

Mehrere Threads können (eventuell) gleichzeitig zugreifen

nebenläufig und threadsicher

- ConcurrentHashMap
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- ConcurrentLinkedQueue

threadsicher mit Benachrichtigung

- ArrayBlockingQueue
- LinkedBlockingQueue
- SynchronousQueue
- PrioritiyQueue
- DelayQueue
- ConcurrentLinkedQueue

Operationen können blockieren bis ihre Ausführung möglich ist.

Alle Java-Kollektionen können auch in Scala verwendet werden.

Scala-Kollektionen: mutable / immutable

unveränderlich, *immutable*

unveränderliche Kollektionen sind **zustandslos** und damit **threadsicher**

veränderlich, *mutable*

- veränderliche Kollektionen sind problematisch, sowie und insbesondere, wenn von mehreren Threads auf sie zugegriffen wird.
- Wenn veränderliche Kollektionen in einem nebenläufigen Umfeld verwendet werden sollen: Verwende Kollektionen mit Synchronisationsunterstützung
- Scala bietet keine eigenen veränderlichen Klassen mit Synchronisationsunterstützung: Verwendet die Java- (JUC-) Klassen

Kollektionen mit Synchronisationsunterstützung

Synchronisierte Kollektionen

Synchronisierte Versionen von Kollektionstypen.

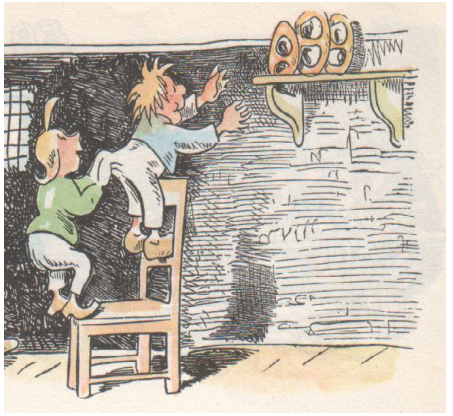
Funktion / Verwendung

- Threadsichere Varianten von Kollektionstypen
- Vorteile: Alle Zugriffe sind synchronisiert (atomar)
- Nachteil: Alle Zugriffe sind streng serialisiert

Bestandteile

- „alte“ (Java 1.4) synchronisierte Kollektionstypen
  - Vector
  - Hashtable
- Umschlag-Klassen
  - Erzeugt mit Collections.synchronized*XYZ*
  - z.B.

```
static <T> List<T> synchronizedList(List<T> l)
```



synchronisierter Kollektion:
Immer einer nach dem anderen !

Synchronisierte Kollektionen

Umschlag-Klassen / Beispiel

```
import java.util.Collections
import collection.JavaConverters

import scala.language.postfixOps

object SynchronizedCollections_Main extends App {

  val lst: collection.mutable.ListBuffer[Int] = collection.mutable.ListBuffer(1,2,3)

  val syncLst: java.util.List[Int] = Collections.synchronizedList(JavaConverters.bufferAsJavaList(lst))

  new Thread (() => {
    syncLst.add(5)
  }) start

  new Thread( () => {
    syncLst.add(6)
  }) start

  println(lst)
}
```

Der Zugriff auf auf die in* syncLst *eingebettete* lst *erfolgt im gegenseitigen Ausschluss – natürlich nur, wenn der Zugriff über* `syncLst` *erfolgt.

Kollektionen mit Synchronisationsunterstützung

Synchronisierte Kollektionen vergl. Java-API

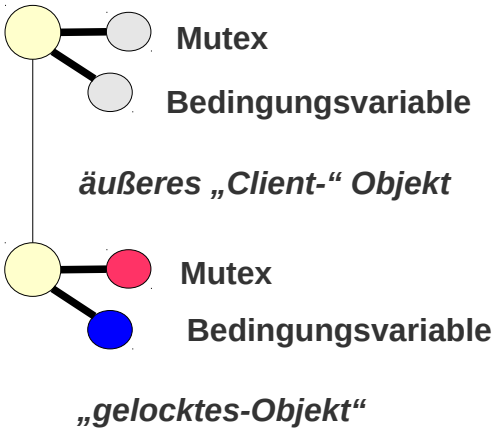
Übersicht Synchronisierte Varianten diverser Kollektionen erstellen

- **static <T> Collection<T> synchronizedCollection(Collection<T> c)**
  Returns a synchronized (thread-safe) collection backed by the specified collection.
- **static <T> List<T> synchronizedList(List<T> list)**
  Returns a synchronized (thread-safe) list backed by the specified list.
- **static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)**
  Returns a synchronized (thread-safe) map backed by the specified map.
- **static <T> Set<T> synchronizedSet(Set<T> s)**
  Returns a synchronized (thread-safe) set backed by the specified set.
- **static <K,V>SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)**
  Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
- **static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)**
  Returns a synchronized (thread-safe) sorted set backed by the specified sorted set

Kollektionen mit Synchronisationsunterstützung

Client-Side Locking

- Lock auf eine synchronisierte Kollektion setzen
- Damit: „Andocken“ an den für die interne Synchronisation verwendeten Mutex
- Anwendung:
  - Erweiterung der Funktionalität einer synchronisierten Kollektion
  - durch selbst definierte atomare Aktionen
- Beispiel:




```
object ClientSideLocking_Main extends App {

  // OK Lock auf den Mutex von syncLst
  def putIfAbsend_OK[T](syncLst: java.util.List[T], o: T): Unit =
    syncLst.synchronized {
      if (!syncLst.contains(o)) {
         syncLst.add(o);
      }
    }

  // Nicht OK: Lockt nicht den Mutex von syncLst
  def putIfAbsend_Bad[T](syncLst: java.util.List[T], o: T): Unit =
    synchronized {
      if (!syncLst.contains(o)) {
         syncLst.add(o);
      }
    }

}
```

Client-Side Locking und geschachtelte Monitore

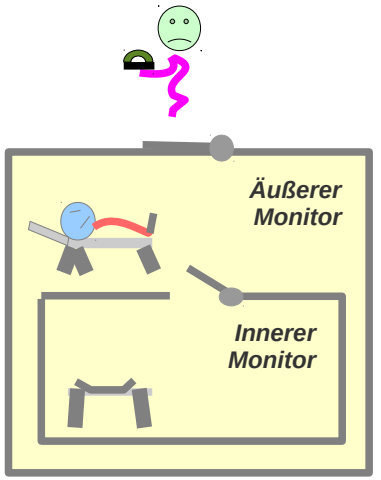
- Durch Client-Side-Locking kann die Monitor-Verschachtelung – und die damit verbundene Deadlock-Gefahr – vermieden werden
- Beispiel:

```
object MonitorAussen {
  private var belegt = false
  private var wert = 0

  def get: Int = synchronized {
      val w = MonitorInnen.get
      while (!belegt) wait();
      belegt = false;
      notifyAll();
      wert+w;
  }

  def put(w: Int): Unit = synchronized {
      while (belegt) wait();
      belegt = true;
      put(w);
      wert = w;
      notifyAll();
  }
}
```

MonitorInnen genauso wie MonitorAussen (siehe nächste Folie)

```
object ClientSideLockingNested_Main extends App {
  new Thread(() => {
    println(MonitorAussen.get)
  }) start

  Thread.sleep(1000)

  new Thread(() => {
    MonitorAussen.put(42)
  }) start

}
```

Deadlock



Kollektionen mit Synchronisationsunterstützung

Client-Side Locking und geschachtelte Monitore

– Durch Client-Side-Locking kann die Monitor-Verschachtelung – und die damit verbundene Deadlock-Gefahr – vermieden werden

```
object MonitorInnen {
  private var belegt = false
  private var wert = 0

  def get: Int = synchronized {
    while(!belegt) wait()
    belegt = false
    notifyAll();
    wert
  }
  def put(w: Int): Unit = synchronized {
    while(belegt) wait();
    belegt = true
    wert = w
    notifyAll();
  }
}
```

```
object ClientSideLockingNested_Main extends
  thread({
    println(MonitorAussen.get)
  }) start

  Thread.sleep(1000)

  thread({
    MonitorAussen.put(42)
  }) start

}
```

Korrigierte Version

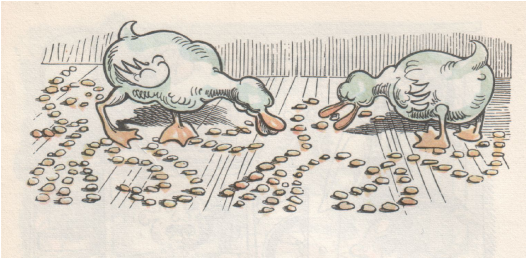
```
object MonitorAussen {
  private var belegt = false
  private var wert = 0

  def get: Int = MonitorInnen.synchronized {
      val w = MonitorInnen.get
      while (!belegt) MonitorInnen.wait();
      belegt = false;
      MonitorInnen.notifyAll();
      wert+w;
  }

  def put(w: Int): Unit = MonitorInnen.synchronized {
      while (belegt) MonitorInnen.wait();
      belegt = true;
      MonitorInnen.put(w);
      wert = w;
      MonitorInnen.notifyAll();
  }
}
```

OK: Kein Deadlock

Nebenläufige Kollektionen

Funktion / Verwendung nebenläufiger Kollektionen:

- Kollektionen mit sicherer gleichzeitiger Nutzung durch mehre Threads.
- Einfügen, Entnehmen, Iteration über die Kollektion erfolgt ohne gegenseitigen Ausschluss. (Im Gegensatz zu synchronisierten Kollektionen)
- Ziel: Effizienzsteigerung (Gleichzeitige Nutzung durch mehrere Threads)



nebenläufige Kollektion:
sicherer gleichzeitiger Zugriff !

Concurrent Map – Java

*java.util.concurrent.*ConcurrentMap

- Interface für ConcurrentHashMap, ConcurrentSkipListMap

*java.util.concurrent.*ConcurrentHashMap

- Threadsichere Operationen
- Erlaubt gleichzeitiges Lesen mehrerer Leser
- Erlaubt eine begrenzte Zahl von gleichzeitigen Schreibern (siehe concurrencyLevel: Parameter eines der Konstruktoren)
- Erlaubt begrenztes gleichzeitiges Lesen und Schreiben
- Implementierung durch Partitionierung:

  Die Tabelle wird in *concurrencyLevel* Partitionen aufgeteilt in denen sich jeweils ein Schreiber und beliebig viele Leser aufhalten können.
- Modifikationen gegenüber einer „normalen“ HashMap
  - Die size-Operation liefert eventuell veraltete Ergebnisse
  - Kann nicht mit *Client-side Locking* gesperrt werden
  - Atomare „zusammengefasste“ Operationen:
  - putIfAbsent(K key, V value) füge ein falls noch nicht vorhanden
  - remove(Object key, Object value) entferne falls vorhanden
  - replace(K key, V value) ersetze falls vorhanden
  - replace(K key, V oldValue, V newValue) ersetze falls der aktuelle Wert der vorgegebene ist

Concurrent Map – Scala-Variante

*scala.collection.concurrent.*Map

- Interface für nebenläufige Maps

*scala.collection.concurrent.*TrieMap

- Implementierung des Traits
  einzige konkrete Klasse in Scala die *scala.collection.concurrent.*Map implementiert
- Alternative zur java.util.concurrent.ConcurrentHashMap

```
import scala.collection.convert.decorateAsScala._

object ConcurrentMap_Main extends App {

  // a concurrent map from JUC
  val c_map_juc: collection.concurrent.Map[Int, String]  =
                new java.util.concurrent.ConcurrentHashMap[Int,String]().asScala

  // a concurrent map from scala
  val c_map_scala: collection.concurrent.Map[Int, String] =
                collection.concurrent.TrieMap[Int, String]()

}
```

Scala: TrieMap

Die JUC-Kollektionen sind in der Regel ausreichend

Sie wurden von Spezialisten über viele Jahre entwickelt, und

es besteht darum keinerlei Grund sie in Scala zu re-implementieren

Ausnahme: TrieMap

Eine nebenläufige Map-Implementierung die garantiert,

- dass während der Traversierung durch einen Thread die Map – für diesen Thread – nicht verändert wird.
- alle Modifikationen durch andere Threads erfolgen auf Kopien

Die Datenstruktur wird dabei nicht komplett kopiert, sondern nur soweit sie von den Änderungen betroffen ist

TrieMap ist damit vergleichbar mit und nutzt die gleichen Techniken wie funktionale Datenstrukturen (Datenstrukturen die sich (scheinbar) nicht verändern lassen, tatsächlich aber intern, aus Effizienzgründen, modifiziert werden)

Concurrent Map und *Client-side Locking*

Beispiel: kein *Client-side-Locking* bei ConcurrentHashMap

```
import collection.JavaConverters._

object EnhancedConcurrentMap {
  val mBase = new java.util.concurrent.ConcurrentHashMap[Int,String]()
  val m: collection.concurrent.Map[Int, String]  = mBase.asScala

  def myMethod: Unit = m.synchronized {
      println("start locking m --------");
      Thread.sleep(5000);
      println("stop  locking m ---------");
  }
  def put(k: Int, v: String) : Unit = m.put(k, v)
  def get(k: Int): Option[String] = m.get(k)
}
```

```
object ClientSideLockingConcurrentMap_Main extends App {

  import EnhancedConcurrentMap._

  new Thread( () => {
    myMethod
  }) start
  Thread.sleep(10)
  new Thread( () => {
     println("start accessing map");
     put(1, "one");   println("put 1");
     put(2, "two");   println("put 2");
     put(3, "three"); println("put 3");
     put(4, "four");  println("put 4");
     println(get(1));
     println(get(2));
     println(get(3));
     println(get(4));
     println("stop accessing map");
  }) start

}
```

```
start locking m --------
start accessing map
put 1
put 2
put 3
put 4
Some(one)
Some(two)
Some(three)
Some(four)
stop accessing map
stop  locking m ---------
```

Nebenläufige Kollektionen

CopyOnWriteArrayList / CopyOnWriteArraySet

- Threadsichere Operationen
- Nebenläufige Ausführung der Operationen: Erlaubt gleichzeitigen lesenden und/oder schreibenden Zugriff
- Iteratoren werfen nicht ConcurrentModificationException.
- Iteratoren beenden ihren Lauf auf dem Zustand der Liste mit dem sie begannen.
- Implementierung durch Kopieren

  Bei Modifikationen wird die Liste kopiert. Laufende Aktionen werden auf der originalen Version zu Ende geführt.
- CopyOnWriteArraySet

  Implementierung des Set-Interfaces mit einer CopyOnWriteArrayList
- Verwendung: Listen (Sets) mit seltenen Modifikationen und häufigen überlappenden Lesezugriffen, bei denen die Verwendung des aktuellen Stands nicht absolut notwendig ist. (z.B. Event Listening)

Weitere nebenläufige JUC-Kollektionen

- **Interfaces**
  - **ConcurrentNavigableMap**
- **Klassen**
  - **ConcurrentSkipListMap**
  - **ConcurrentHashMap**
  - **ConcurrentSkipListSet**

Siehe Java-API-Doku

Queue und Deque ohne Benachrichtigung

ConcurrentLinkedQueue

- Implementiert Queue mit Thread-sicherem nebenläufigen Zugriff
- ohne Benachrichtigung (d.h. ohne Bedingungssynchronisation)

ConcurrentLinkedDeque

- Implementiert Deque mit Thread-sicherem nebenläufigen Zugriff
- ohne Benachrichtigung (d.h. mit Bedingungssynchronisation)

TransferQueue

- Eine Queue die die synchrone Übergabe eines Wertes vom Produzenten an den Konsumenten ermöglicht.

Nebenläufige Kollektionen und Scala: Übersicht

Scala bietet kaum eigene Beiträge zu den nebenläufigen Kollektionen.

Scala

- Package scala.collection.concurrent
- Trait scala.collection.concurrent.Map
- Klasse scala.collection.concurrent.TrieMap

JUC

- Diverse nebenläufigen Kollektionen aus java.util.concurrent

Richtline

- Nutze die unveränderlichen Kollektionen von Scala. Diese sind per se nebenläufig und threadsicher.
- Sollten diese ein Effizienz-Problem erzeugen, erwäge den Einsatz nebenläufiger Kollektionen
- Nutze TrieMap wenn eine nebenläufige Map gebraucht wird und diese regelmäßig zu traversieren ist.
- Ansonsten: Verwende eine der nebenläufigen Klassen aus JUC

Kollektionen mit Benachrichtigung

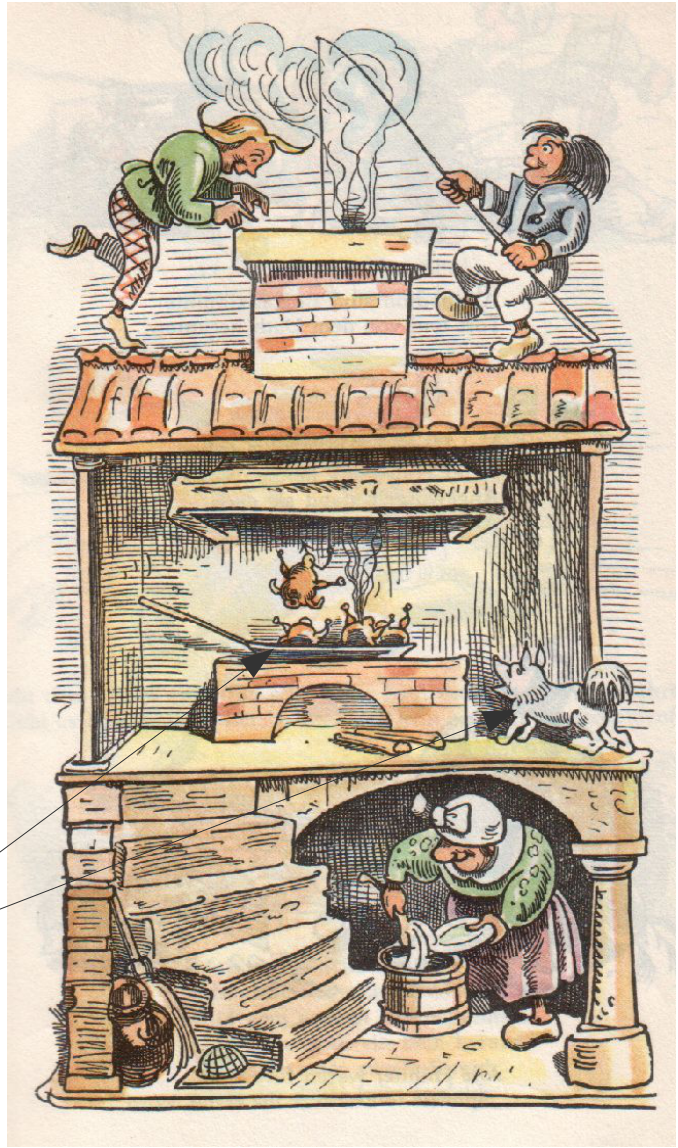
Kollektionen mit Benachrichtigung =

Kollektionen mit Bedingungssynchronisation

hauptsächlich handelt es sich um Queues (Warteschlangen)

Queue-Interfaces

- **Queue: java.util.Queue<E>**
- **Folge von Elementen, die auf Verarbeitung warten**
- **BlockingQueue**
  java.util.concurrent.BlockingQueue<E>
- **Folge von Elementen, die auf Verarbeitung warten, mit blockierenden Zugriffsoperation die auf die Verfügbarkeit von Daten / Ablageplätzen warten**



JUC-Kollektionen mit Benachrichtigung

Queue-Klassen aus JUC

- LinkedBlockingQueue
  - Implementierung als verkettete Liste – *FIFO-Verhalten*
  - Größe: Konstruktor / Integer.MAX_VALUE
- ArrayBlockingQueue
  - Implemtierung als Array fester Größe – *FIFO-Verhalten*
  - Größe: Konstruktor
- DelayQueue
  - Queue mit verzögerten Elementen (Elemente müssen eine bestimmte Zeit in der Queue verbringen) – FIFO-Verhalten auf den abgelaufenen
  - Größe: (logisch) unbeschränkt
- PriorityBlockingQueue
  - Queue mit vergleichbarer Elemete (Comparable): Kleinster zuerst
  - Größe: Konstruktor / (logisch) unbeschränkt
- SynchronousQueue
  - Rendezvous-Mechanismus für Threads
  - Größe : 0

JUC-Kollektionen mit Benachrichtigung

Deques

Warteschlangen die an beiden Enden in gleicher Art manipuliert werden können

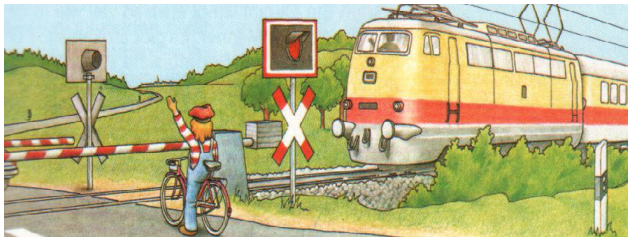
Interfaces

- Deque
- BlockingDeque

Klassen

- ArrayDeque
- ConcurrentLinkedDeque
- LinkedBlockingDeque
- Deque mit Benachrichtigung
- LinkedList

Synchronisierer



Synchronisierer: Objekte mit denen der Kontrollfluss von Threads gesteuert werden kann

System-Synchronisierer

- Jede Thread-Implementierung benötigt eine Grundausstattung an internen Synchronisiereren
- System-Synchronisierer von Java : Die Mutexe und Bedingungsvariablen, die zu jedem Objekt gehören

JUC Synchronisierer

- Mit Java 1.5 wurden als Bestandteil von java.util.concurrency weitere Synchronisierer-Klassen als Bestandteil der Java-API eingeführt.
- Damit können Anwendungen bessert strukturiert und auf höherem Abstraktionsniveau entwickelt werden

Synchronisierer

Konkrete Synchronisierer-Klassen

- **Lock** — expliziter Mutex / Leser-Schreiber-Sperre
- **Condition** — explizite Bedingungsvariablen
- **Queue / Deque** — Warteschlangen (verwendbar auch als Synchronisierer)
- **CyclicBarrier** — Barriere
- **CountDownLatch** — Schnapper
- **Semaphore** — Semaphor
- **Exchanger** — Begegnung von zwei Threads

AbstractQueuedSynchronizer

Basisklasse für weitere Synchronisierer

Beispiel: Semaphor-basierte Warteschlange

```
import java.util.concurrent.Semaphore

class SemBuffer[E](implicit  val me: Manifest[E]) {
  private val SIZE: Int  = 10;                        // Puffergroesse
  private val buf: Array[E] = new Array[E](SIZE)
  private var front: Int = 0;                              // Schreibposition
  private var rear: Int  = 0;                              // Leseposition

  // zwei Semaphore zur Bedingungssynchronisation
  private val full: Semaphore  = new Semaphore(0);  //zählt belegte Plaetze
  private val empty: Semaphore = new Semaphore(10); //zählt freie Plaetze

  // ein Semaphor zum gegenseitigen Ausschluss
  private val mutex: Semaphore  = new Semaphore(1);

  def put(x: E): Unit = {
    empty.acquire() // Achtung verschachtelte Mnitore! Deadlock-Gefahr
    mutex.acquire() // Erst empty prüfen, dann mutex locken!
      buf(front) = x; front = (front + 1) % SIZE;
    mutex.release()
    full.release()
  }

  def get(): E = {
    full.acquire()
    mutex.acquire()
      val v = buf(rear); rear = (rear + 1) % SIZE;
    mutex.release()
    empty.release()
    return v;
  }
}
```

Explizite Locks

ab Java 1.5 steht der Zugriff auf explizite Lock-Objekte zur Verfügung (statt nur Zugriff implizite Object-Mutexe via synchronized)

Vorteil flexiblere Verwendung möglich (Lock / Unlock in unterschiedlichen Methoden)

Interface *java.util.concurrent.locks* Interface Lock

Implementierung

- *java.util.concurrent.locks* Class ReentrantLock
- entpricht dem implizten Mutex (dieser ist ebenfalls reentrant)

Leser-Schreiber-Synchronisation

- *java.util.concurrent.locks* Interface ReadWriteLock
- *java.util.concurrent.locks* Class ReentrantReadWriteLock
- *java.util.concurrent.locks* Interface ReentrantReadWriteLock.ReadLock
- *java.util.concurrent.locks* Interface ReentrantReadWriteLock.WriteLock

Explizite Bedingungsvariablen

Condition

- ab Java 1.5 steht der Zugriff auf explizite Bedingungsvariablen zur Verfügung (statt nur implizit über wait / notify / notifyAll)
- Vorteil

  flexiblere Verwendung möglich (spezialisierte Warteschlangen möglich)

Interface

java.util.concurrent.locks **Interface Condition**

Implementierungen

- AbstractQueuedLongSynchronizer.ConditionObject
- AbstractQueuedSynchronizer.ConditionObject

Explizite Bedingungsvariablen

Beispiel Puffer

Zwei spezialisierte Thread-Warteschlangen:

- **WS 1: Produzenten die darauf warten, dass der Puffer nicht voll ist**
- **WS 2: Konsumenten die darauf warten, dass der Puffer nicht leer ist**

Realisiert mit zwei Bedingungsvariablen

- **explizite Bedingungsvariablen erlauben die Arbeit mit spezialisierten Warteschlangen**

Generell

für jede Bedingung, auf die gewartet wird, kann eine Warteschlange angelegt werden

- **Bedingung erfüllen ~> signal an die richtige Warteschlange**
- **unnötiges Aufwecken (signalAll/notifyAll) kann vermieden werden**
- **im Beispiel:**
  - **Condition notVoll : WS für Threads, die auf *nicht voll* warten (Produzenten)**
  - **Condition notLeer : WS für Threads, die auf *nicht leer* warten (Konsumenten)**

Explizite Mutexe und Bedingungsvariablen: Beispiel Puffer

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

class MutexCondBuffer[E](implicit  val me: Manifest[E]) {

  private val SIZE: Int  = 10;  // Puffergroesse
  private val buf: Array[E] = new Array[E](SIZE)
  private var count = 0;      //Belegte Pufferpositionen
  private var front: Int = 0;   // Schreibposition
  private var rear: Int  = 0;   // Leseposition

  // ein Mutex zum gegenseitigen Ausschluss
  private val mutex: Lock  = new ReentrantLock();

  // zwei Bedingungsvariablen Bedingungssynchronisation
  private val notfull: Condition  = mutex.newCondition();// hier warten Produzenten
  private val notempty:  Condition  = mutex.newCondition();// hier warten Konsumenten

  def put(x: E): Unit = {
    mutex.lock();
    while (!(count < SIZE)) notfull.await()
    buf(front) = x; front = (front + 1) % SIZE; count = count +1;
    notempty.signal()
    mutex.unlock();
  }

  def get(): E = {
    mutex.lock();
    while (! (count > 0)) notempty.await()
    val v = buf(rear); rear = (rear + 1) % SIZE; count = count-1;
    notfull.signal()
    mutex.unlock();
    v
  }
}
```

Atomare Variablen

Package *java.util.concurrent.atomic*

bietet einige Klassen mit

- **atomaren (nicht unterbrechbaren / verschränkungsfreien) Operationen**
- **Die intern ohne Betriebssystem- / JVM- Locks (Mutexe) implementiert sind**
  die Synchronisation erfolgt statt dessen auf der Hardware-Ebene
  (natürlich auch eventuell mit wartenden Threads, diese warten dann aktiv statt in Warteschlangen des Systems)
- **Vorteil:**
  - **oft effizienter**
    besser ein paar Noop-Befehle ausführen ist als ein Kontextwechsel
  - **Keine Deadlocks durch fehlerhafte Synchronisation**

Atomare Variablen

Beispiel

```
import java.util.concurrent.atomic.AtomicInteger

class MyAtomicInteger {
  private var counter: Int = 0;
  def incrementAndGet: Int = synchronized {
    counter = counter+1
    counter
  }
}

object Atomic_Main extends App {

  val myAtomicCounter = new MyAtomicInteger
  val jucAtomicCounter = new AtomicInteger()

  println(myAtomicCounter.incrementAndGet)
  println(jucAtomicCounter.incrementAndGet)
}
```

OK

Gut

Keine aufwendige Synchronisation, einfaches aktives Warten

Atomare Variablen

JUC: Atomare Klassen

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicIntegerFieldUpdater
- AtomicLong
- AtomicLongArray
- AtomicLongFieldUpdater
- AtomicMarkableReference
- AtomicReference
- AtomicReferenceArray
- AtomicReferenceFieldUpdater
- AtomicStampedReference

Siehe Java-API

Synchronisation – Resümee

Auf der JVM und in in allen JVM-Sprachen steht ein sehr reichhaltiges Repertoire an Synchronisations-Mitteln zur Verfügung.

Mit diesen können sowohl sehr systemnahe als auch anwendungsnahe Problemstellungen gelöst werden

Traditionelle ist Synchronisation eine Thema der Systemprorgammierung

Heute ist es gelegentlich (!) auch relevant in Anwendungen

- **Nutze stets möglichst „hohe Konstrukte“ z.B. JUC-Kollektionen, statt selbst gebastelten Lösungen**
- **Der Umgang mit *Mutexen, Bedingugsvariablen, Semaphoren, synchronized* etc. ist immer nur etwas für die, die wirklich wissen was sie tun!**