

# Parallele Programmierung



## 8. Lock-Objekte und Semaphore

# Überblick

- Lock-Interface
- ReentrantLock
- ReadWriteLock
- StampedLock
- Semaphore



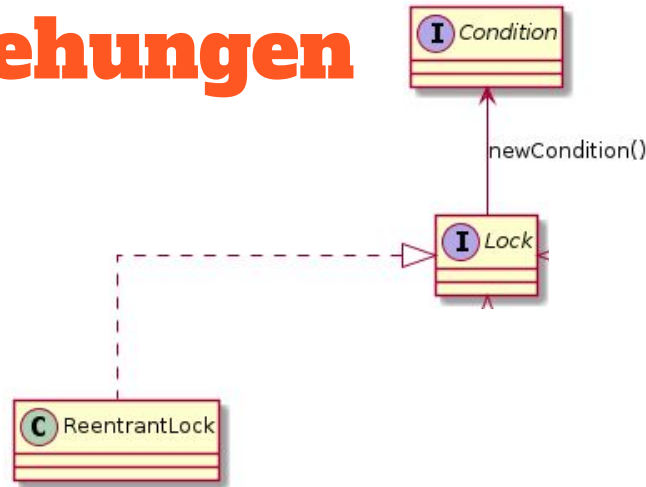
# ReentrantLock

# Nachteile von `synchronized`

- durch `synchronized` blockierter Thread kann nicht mit `interrupt()` unterbrochen werden. `interrupt()` wirkt sich erst nach der Blockierung aus (pp.08.01.SynchInterrupt).
- kein Timeout
- keine Regel für Reihenfolge der Zuteilung an mehrere wartende Tasks
- nur Blockstruktur (Eintritt und Austritt in unterschiedlichen Methoden nicht möglich)

# Lock-Klassen und ihre Beziehungen

java.util.concurrent.locks



# Nachteile von `synchronized`

- durch `synchronized` blockierter Thread kann nicht mit `interrupt()` unterbrochen werden. `interrupt()` wirkt sich erst nach der Blockierung aus (pp.08.01.SynchInterrupt).
- kein Timeout
- keine Regel für Reihenfolge der Zuteilung an mehrere wartende Tasks
- nur Blockstruktur (Eintritt und Austritt in unterschiedlichen Methoden nicht möglich)

# Nachteile von synchronized

- durch synchronized blockierter Thread kann nicht mit `interrupt()` unterbrochen werden. `interrupt()` wirkt sich erst nach der Blockierung aus (pp.08.01.SynchInterrupt).
- kein Timeout
- keine Regel für Reihenfolge der Zuteilung an mehrere wartende Tasks
- nur Blockstruktur (Eintritt und Austritt in unterschiedlichen Methoden nicht möglich)
- **Vorteile von ReentrantLock**
  - **Unterbrechbarkeit**
  - **Timeout**
  - **Fairness**
  - **“Non-Blockstruktur”**

# Lock-Interface

```
public interface Lock {  
    public void lock();  
    public void lockInterruptibly()  
        throws InterruptedException;  
    public boolean tryLock();  
    public boolean tryLock(long time, TimeUnit unit)  
        throws InterruptedException;  
    public void unlock();  
    public Condition newCondition();  
}
```

ein so blockierter Thread wird durch interrupt() unterbrochen.

liefert false, falls Lock-Instanz belegt ist, sonst lock()




# Benutzungsmuster

## Lock-Interface

Folie mit  
Anmerkungen

```
Lock lock = new ReentrantLock();  
Lock lock = new ReentrantLock(fairness);
```

boolean: als nächstes wird  
der am längsten wartende  
Thread entblockt



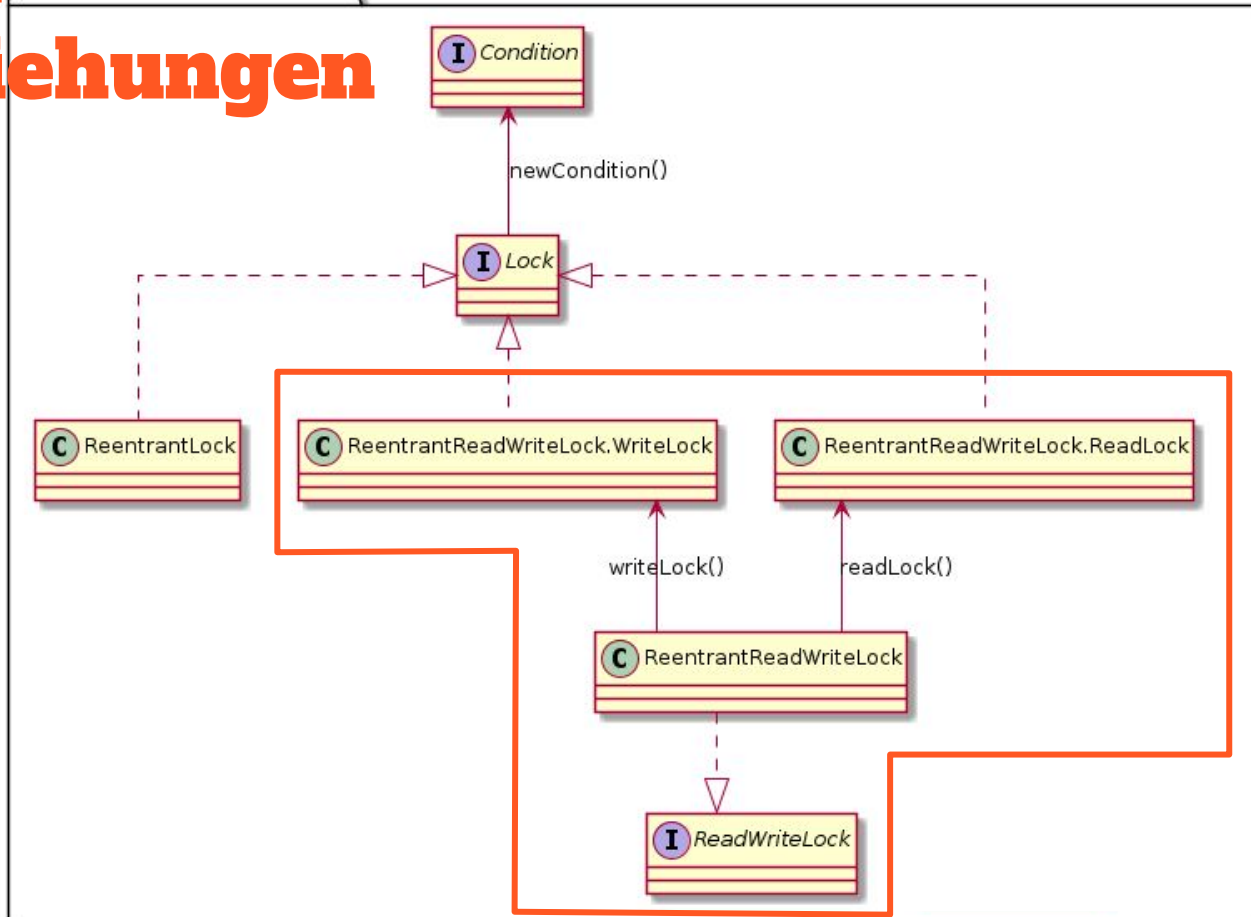
```
lock.lock();  
try {  
    //...  
} finally {  
    lock.unlock();  
}
```

```
if(lock.tryLock()) {  
    try {  
        //...  
    } finally {  
        lock.unlock();  
    }  
}
```

# ReadWriteLock

# Lock-Klassen und ihre Beziehungen

java.util.concurrent.locks



# ReadWriteLock

- warum sollten nicht mehrere Threads gleichzeitig lesend auf eine Variable zugreifen dürfen, wenn ein gleichzeitiges Schreiben ausgeschlossen ist?
- Ein Lock, der ReadLock und WriteLock besitzt
  - ReadLock: mehrere dürfen in den kritischen Abschnitt, wenn der dazugehörige WriteLock nicht geschlossen ist.
  - WriteLock: nur einer darf in den kritischen Abschnitt und auch nur, wenn der ReadLock nicht gerade benutzt wird.
- Die Vergabe der Locks erfolgt in der Reihenfolge, wie sie angefordert wurden.

# ReadWriteLock

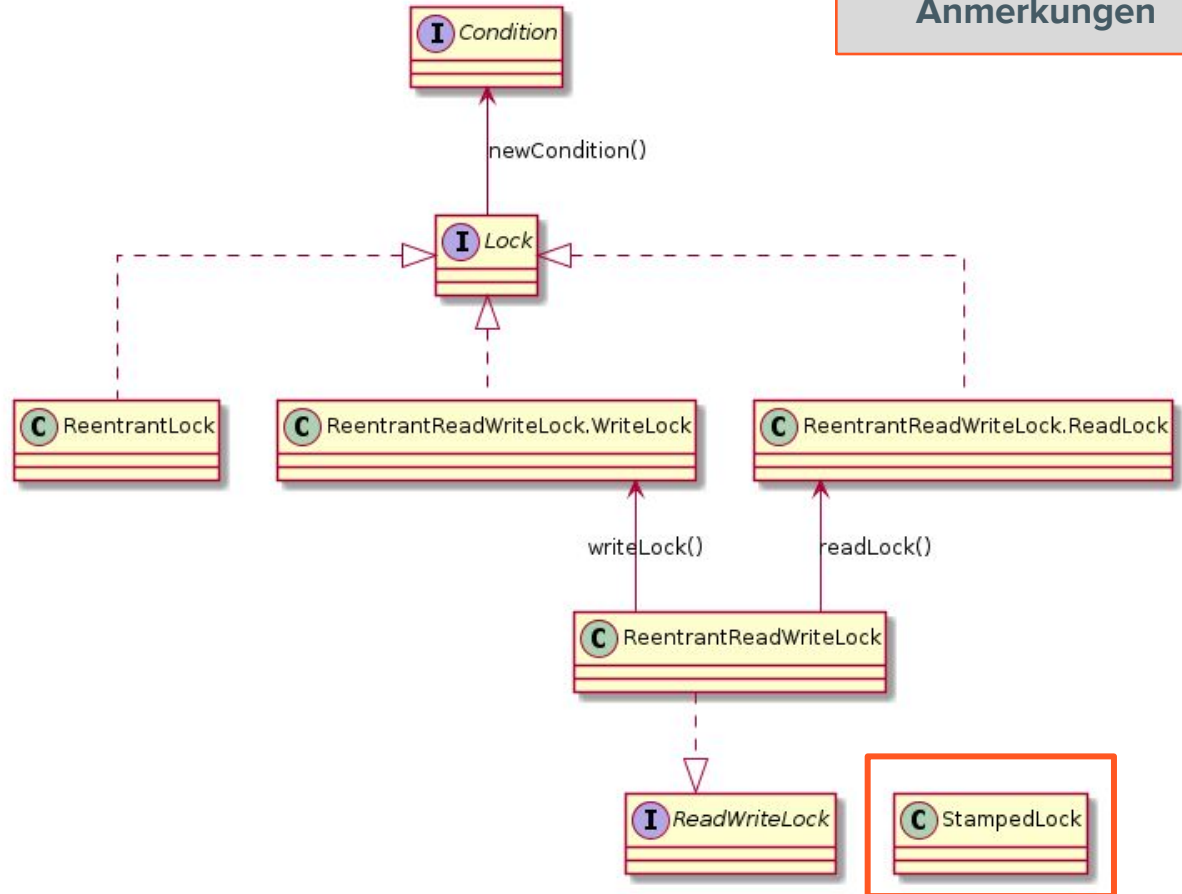
```
public interface ReadWriteLock {  
    public Lock readLock();  
    public Lock writeLock();  
}
```

```
ReadWriteLock lock = new ReentrantReadWriteLock(fairness);  
Lock rLock = lock.readLock();  
Lock wLock = lock.writeLock();
```

# StampedLock

# StampedLock

- Falls die zu schützenden kritischen Abschnitte kurz sind, ist der erforderliche Verwaltungsaufwand für einen Lock relativ hoch. `synchronized` kann von der Laufzeit her schneller sein.
- `StampedLock` nützlich bei großem Leseanteil



# StampedLock Modi

Ein StampedLock besteht aus *Stamp* und *Modus*

- Modus *Writing*: `writeLock()` blockiert für exklusiven Schreibzugriff; liefert eine `long` ID ("*stamp*"), die für `unlockWrite(long)` benutzt werden kann. keine `readLocks` und `tryOptimisticRead` möglich.
- Modus *Reading*: `readLock()` wartet auf nicht exklusiven-Zugriff; liefert eine `long` ID ("*stamp*"), die für `unlockRead(long)` benutzt werden kann.
- Modus *Optimistic Reading*: `tryOptimisticRead()` liefert nur dann eine `long` ID ("*stamp*"), falls der Lock gerade nicht im Modus *Writing* ist. `validate(long)` liefert `true`, falls der Lock nicht zum Schreiben gesperrt wurde, seit die ID vergeben wurde.



# WriteLock beim StampedLock

```
StampedLock lock = new StampedLock();  
long stamp = lock.writeLock();  
try {  
    // ...  
} finally {  
    lock.unlockWrite(stamp);  
}
```

# Optimistisches Lesen mit StampedLock

```
var lock = new StampedLock();
var stamp = lock.tryOptimisticRead();
// ...
if (!lock.validate(stamp)) {
    // nicht erfolgreich => das bisherige ist möglicherweise
    // inkonsistent und muss zurückgerollt werden; eine mögliche
    // Strategie: nochmal pessimistisch gelockt probieren:
    stamp = lock.readLock();
    try {
        // ...
    } finally {
        lock.unlockRead(stamp);
    }
}
```

# **pp.08.02- LockTiming**

# Ergebnisse

	10 Mio 99,99% Read	10 Mio 50% Read	100 Mio 99,99% Read	100 Mio 50% Read
(Unsynchronized)				
Synchronized	228ms	290ms	1900ms	2800ms
ReentrantLock	2200ms	1484ms	24000ms	21000ms
ReadWriteLock	2600ms	2100ms	25000ms	21000ms
StampedLock	21ms	900ms	150ms	23000ms

# Ergebnis abgleichen

## Praxistipp

---

Für den Einsatz von Locks gelten folgende Faustregeln:

- Wenn sehr viel geschrieben wird, ist das `synchronized`-Konzept zu bevorzugen.
  - Der Verwaltungsaufwand für `ReadWriteLock` und `StampedLock` ist relativ hoch. Wenn nur kurz gelesen wird, ist es sogar effizienter, `synchronized` zu benutzen.
  - `StampedLock` führt das optimistische Lesen ein, wodurch in bestimmten Fällen eine Performance-Steigerung erreicht werden kann.
-

# Semaphore

# Semaphore

Semaphore ermöglichen die Begrenzung der Nutzung einer Ressource auf eine bestimmte Anzahl Nutzer.

Kapazität eines Semaphores (`permitCount`): Anzahl an noch erlaubten Nutzern.

Operationen: `release()` und `acquire()` (blockiert, falls `permitCount==0`)

