

# Thread Safety

Vortrag von Fabian Stützinger  
im Proseminar Nebenläufige Programmierung am 02.05.2008

# Gliederung

- 1. Einleitung
  - 1.1 Allgemeines zu Threads
  - 1.2 Allgemeines zur Thread Safety
- 2. Definition
- 3. Atomicity
  - 3.1 Race conditions
  - 3.2 Compound actions
- 4. Locking
  - 4.1 Intrinsic Locks
  - 4.2 Reentrancy
  - 4.3. Guarding state with locks
- 5. Liveness und Performance
- 6. Folgerung

# 1. Einleitung

## 1.1 Threads

- Vorteile der Threads:
  - Einsparung von Entwicklungs- und Unterhaltskosten
  - Leistung

# 1. Einleitung

## 1.1 Threads

- Nachteile der Threads:
  - Höhere Anforderungen an Programmierer
  - Sicherheitsrisiken

# 1. Einleitung

## 1.2 Thread Safety

- state /mutable shared state
- Synchronisation beim Zugriff veränderbarer Daten

# 1. Einleitung

## 1.2 Thread Safety

- Möglichkeiten ein defektes Programm zu reparieren
  - Zugang zu den Daten einschränken
  - Variablen unveränderbar machen
  - Zugang zu den Daten synchronisieren

# 1. Einleitung

## 1.2 Thread Safety

- Techniken zum Erstellen von sequentiellem Code helfen auch bei nebenläufiger Programmierung
  - z.B. encapsulation / data hiding
  - höhere Kosten und Sicherheitsrisiken ohne solche Mittel

# 1. Einleitung

## 1.2 Thread Safety

- erst fehlerfreier Code, dann eventuell Leistungsverbesserungen
- Verhältnis von thread-safes Klassen und Programmen

## 2. Definition von Thread Safety

- genaue Definition eher schwierig

- Definition:

Eine Klasse ist dann thread-safe, wenn sie sich weiterhin korrekt verhält, wenn mehrere threads darauf zugreifen, die nicht von sich aus synchronisiert sein müssen.

# 3. Atomicity

Codebeispiel ( thread-safe):

```
public class StatelessFactorizer implements Servlet {  
    public void service(ServletRequest req,  
        ServletResponse resp) {  
        BigInteger i = extractFromRequest (req);  
        BigInteger[] factors = factor(i);  
        encodeIntoResponse(resp, factors);  
    }  
}
```

# 3. Atomicity

Codebeispiel (nicht thread-safe):

```
public class UnsafeCountingFactorizer implements Servlet {
    private long count = 0

    public long getCount() { return count; }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest (req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
```

# 3. Atomicity

- Merkmale einer atomaren Operation:
  - zusammenhängend
  - keine Unterbrechung möglich
  - wird komplett, oder überhaupt nicht durchgeführt

# 3. Atomicity

## 3.1 Race conditions

- häufigste Form von race conditions ist check-then-act
  - andere Form ist read-modify-write, wie bei ++count
- lazy initialization

# 3. Atomicity

## 3.1 Race conditions

Codebeispiel (nicht thread-safe):

```
public class LazyInitRace {  
    private ExpensiveObject instance = null ;  
  
    public ExpensiveObject getInstance() {  
        if (instance == null)  
            instance = new ExpensiveObject() ;  
        return instance;  
    }  
}
```

# 3. Atomicity

## 3.2 Compound actions

- compound actions können LazyInitRace und UnsafeCountingFactorizer thread-safe machen
- compound actions können Operationen atomic machen

# 3. Atomicity

## 3.2 Compound actions

Codebeispiel (thread-safe):

```
public class UnsafeCountingFactorizer implements Servlet {  
    private final AtomicLong count = new AtomicLong(0);  
  
    public long getCount() { return count.get() ; }  
  
    public void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest (req);  
        BigInteger[] factors = factor(i);  
        count.incrementAndGet() ;  
        encodeIntoResponse(resp, factors);  
    }  
}
```

# 4. Locking

- bei einer Variablen in einer Klasse reicht es aus, wenn die Variable thread-safe ist, damit auch die Klasse thread-safe ist
- bei mehreren Variablen helfen locks
- durch locks können miteinander verbundene Statusvariablen in einer atomic operation verändert werden

# 4. Locking

## 4.1 Intrinsic locks

- synchronized block
  - besteht aus zwei Teilen:
    - ein Hinweis auf ein Objekt, das als lock dient
    - Code, der von dem Lock geschützt wird
  - jedes Objekt in Java kann als lock für Synchronisationszwecke benutzt werden
  - ein lock wird automatisch gesetzt, wenn ein synchronisierter Block benutzt werden soll und automatisch wieder aufgelöst, sobald er nicht mehr in Benutzung ist

# 4. Locking

## 4.1 Intrinsic locks

- intrinsic locks in Java können nur einem thread gehören
- synchronisierte Blöcke sind atomar
- locks können bei falscher Nutzung zu massiven Performance-Verlusten führen

# 4. Locking

## 4.1 Intrinsic locks

Codebeispiel (thread-safe):

```
public class SynchronizedFactorizer implements Servlet {  
    @GuardedBy(„this“) private BigInteger lastNumber;  
    @GuardedBy(„this“) private BigInteger[] lastFactors;  
    public synchronized void service(ServletRequest req, ServletResponse resp) {  
        BigInteger i = extractFromRequest(req) ;  
        if (i.equals(lastNumber))  
            encodeIntoResponse(resp, lastFactors) ;  
        else {  
            BigInteger[] factors = factor (i) ;  
            lastNumber = i;  
            lastFactors = factors;  
            encodeIntoResponse(resp, lastFactors) ;  
        }  
    }  
}
```

# 4. Locking

## 4.2 Reentrancy

- jeder intrinsic lock besitzt einen acquisition count, der jedes mal erhöht wird wenn ein thread ein lock nimmt
  - null bedeutet, dass kein thread das lock hält
  - wenn ein thread das lock nimmt, wird der Zähler um eins erhöht, wenn er es abgibt um eins gesenkt
- ohne diesen Zähler würde ein thread bei dem Versuch ein lock zu nehmen, das er bereits hält, blockieren

# 4. Locking

## 4.2 Reentrancy

Codebeispiel:

```
public class Widget {  
    public synchronized void doSomething() {  
        ...  
    }  
}
```

```
public class LoggingWidget extends Widget {  
    public synchronized void doSomething() {  
        System.out.println(toString() + “: calling doSomething“);  
        super.doSomething();  
    }  
}
```

# 4. Locking

## 4.3 Guarding state with locks

- jede gemeinsame veränderbare Variable sollte von einem eigenen lock geschützt werden
- Statusvariablen die gleichzeitig benutzt werden, müssen vom gleichen lock geschützt werden

# 5. Liveness and performance

- lange laufende Prozesse sollten aus synchronized blocks ausgeschlossen werden, wenn sie nicht nicht synchronisiert werden müssen
  - I/O
  - lange Rechenvorgänge

# 6. Fazit

- möglichst wenige veränderbare Statusvariablen benutzen
- Sicherheit und Überschaubarkeit des Codes nicht einfach so dem Streben nach besserer Performance opfern