DHBW
Duale Hochschule
Baden-Württemberg

AIRBUS
DEFENCE & SPACE

# Threadsynchronisierung

Lukas Abelt
lukas.abelt@airbus.com

DHBW Ravensburg
Wirtschaftsinformatik

Ravensburg
20. Mai 2019

# Inhalt

# Inhalt

# Executor Service

## Mehrere Callables abarbeiten

- Über `submit` lässt sich *ein* Callable Objekt bearbeiten
- `ExecutorService` bietet aber auch Möglichkeiten, mehrere `Callable` Objekte zu bearbeiten
- Voraussetzung: Callables haben alle den gleichen generischen Typen (Geben beispielsweise alle einen `int` zurück)
- Hierbei wird eine Liste von Callables übergeben
- Zwei „Modi“:
  - Alle Callables werden bis zum Ende bearbeitet
  - Die Bearbeitung wird unterbrochen, sobald das erste Objekt fertig bearbeitet wurden

# Executor Service

## Mehrere Callables bearbeiten

- Interface bietet folgende Methoden:
  - `List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)` – Bearbeitet alle Callables und gibt eine Liste von Future Objekten zurück
  - `List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` – Bearbeitet alle Callables für eine maximal angegebene Zeit
  - `T invokeAny(Collection<? extends Callable<T>> tasks)` – Bearbeitet die gegebenen Callables und gibt das erste Ergebnis zurück
  - `T invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)` – Bearbeitet alle Callables maximal für die angegebene Zeit und gibt das erste Ergebnis zurück
- **Hinweis**: Alle Aufrufe sind blockierend!

# Zeitsteuerung

## ScheduledExecutorSevice

- Der Start eines bestimmten Kommandos (Als Runnable) lässt sich verzögern
- Oder auch periodisch wiederholen
- Dafür wird spezieller Service genutzt: `ScheduledExecutorSevice`
  - `Executors.newScheduledThreadPool(int nThreads)`
- Dieser kann `Runnable` nach einer bestimmten Zeit und ab dann widerholend ausführen

# Zeitsteuerung

## Methoden

- `schedule (Runnable command, long delay, TimeUnit unit)` – Führt das gegebene Runnable Objekt nach dem gegebenen Delay aus.
- `scheduleAtFixedRate (Runnable command, long initialDelay, long period, TimeUnit unit)` – Führt das gegebene Runnable Objekt nach dem gegebenen Delay aus und startet es dann in dem gegebenen Interval
- `scheduleWithFixedDelay (Runnable command, long initialDelay, long delay, TimeUnit unit)` – Führt das gegebene Runnable Objekt nach dem gegebenen Delay aus und wiederholt es dann regelmäßig. `delay` gibt an, wie viel Zeit zwischen Beendigung und Neustart liegen.

# Inhalt

# Inhalt

# Gemeinsame Zugriffe

## Allgemein

- „Einfache“ Threads kommen sich nur selten in die Quere
- Wenn zur Bearbeitung nur lokale Variablen benötigt werden
- Jeder Thread hat seine eigenen Variablen und Callstack
- Probleme treten bei *gemeinsam genutzten* Daten auf
- ...Oder Ressourcen
- Beispielsweise: Externe Daten(objekte), statische Variablen

# Statischer Zugriff

## Codebeispiel

```
1 class MyThread extends Thread{
2 static int result;
3 public void run(){ ... }
4 }
```

# Gemeinsame Zugriffe

## Weitere Beispiele

- Parallele Zugriffe könnten auch bei globalen Datenstrukturen auftreten
- Wenn diese aus mehreren Threads aufgerufen und bearbeitet werden
- Lesen ist in der Regel das geringere Problem, schreibende Zugriffe sind aber problematisch
- Abstrakteres Beispiel: **Was würde passieren, wenn mehrere Prozesse parallel auf einen Drucker zugreifen, ohne dass dieser die Zugriffe einschränken würde?**
- Problematik liegt in der Umschaltung der Threads
- Es ist nicht absehbar, zu welchem Punkt die Bearbeitung umspringt
- Dadurch muss ein Mechanismus geschaffen werden, der sicherstellt, dass eine Ressource oder Datensatz nur durch einen Aktuer genutzt wird.
- Man spricht hierbei von **kritischen Abschnitten**

# Kritische Abschnitte

## Allgemeines

- Beschreiben Codeblöcke, in denen sichergestellt wird, dass nur ein Thread auf sie zugreift
- Man spricht auch von *gegenseitigem Ausschluss* oder *atomar*, wenn nur ein Thread in einem Programmteil arbeitet
- Lesende Zugriffe sind weniger kritisch
- Aber sobald Daten verändert werden, müssen kritische Abschnitte geschützt werden
- Wenn alle kritschen Abschnitte gesichert sind, spricht man von einer *thread-sicheren* Implementierung (*thread-safe*)
- Immutable Objekte sind immer threadsicher (Da sie nicht verändert werden können)

# Nicht kritische Abschnitte

- Nicht kritische Abschnitte sind „automatisch“ Threadsicher
- Deren parallele Ausführung hat keine unerwünschten Nebeneffekte
- Grundsätzlich alle Abschnitte in denen:
  - Nur lesende Zugriffe erfolgen
  - Keine Objekteigenschaften verändert werden
  - Ausschließlich mit lokalen Variablen (Oder Parametervariablen) gearbeitet wird

# Paralleler Zugriff

## Beispiel

- Folgendes Beispiel: Wir haben ein Objekt `p` vom Typ `Point`
- Thread `T1` möchte `p` mit (1|1) belegen
- Thread `T2` möchte `p` (gleichzeitig) mit (2|2) belegen

| **Thread T1** | **Thread T2** |
|---|---|
| `p.x=1` | `p.x=2` |
| `p.y=1` | `p.y=2` |

# Paralleler Zugriff

## Beispiel

- Durch Umschaltung der Threads können die einzelnen Anweisungen „durcheinander“ geraten:

| Thread T1 | Thread T2 | (x\|y) |
|---|---|---|
| `p.x=1` | | (1\|0) |
| | `p.x=2` | (2\|0) |
| | `p.y=2` | (2\|2) |
| `p.y=1` | | (2\|1) |

# Paralleler Zugriff

- Durch Umschaltung wird somit ungültiges Ergebnis erreicht
- Auch „kleine“ Operationen sind nicht automatisch atomar
- Beispielsweise Benutzung von `i++`
- Da diese Operation (auf unterer Ebene) aus mehreren Teilen besteht:
  - Hole den aktuellen Wert von i
  - Deklariere den konstanten Wert 1
  - Addiere i mit der Konstante
  - Schreibe das neue i zurück in den vorgesehenen Speicherbereich

# Schützen

## Von kritischen Abschnitten

- Wir müssen also einen Mechanismus finden um parallelen Zugriff zu vermeiden
- Java bietet hier zwei Konstrukte, mit denen wir uns beschäftigen:
  - Das Verwenden von *Lock* Objekten
  - Die Nutzung des `synchronized` Keywords

# Inhalt

# Monitore

## Allgemein

- Monitor wird nötig, wenn nur ein Thread in einen Block kommen soll
- In Java realisiert über *Locks*
- Diese können einen Codeabschnitt „abschließen“ und „öffnen“
- Das `Lock` Interface hat zwei wichtige Methoden:
  - `lock()` – Markiert den Beginn des kritischen Abschnitts. Wenn bereits (durch einen anderen Thread) ein kritischer Abschnitt betreten wurde, blockiert die Methode so lange bis dieser das Lock wieder freigibt
  - `unlock()` – Markiert das Verlassen des kritischen Abschnitts

# Lock Interface

## Wichtige Aspekte

- Zu jedem `lock` muss auch ein `unlock` existieren
- Sonst können spätere Threads nie in den kritischen Abschnitt springen
- Das normale `lock` ignoriert den `interrupt` Aufruf des Threads
- Spezielle Methode `lockInterruptedly` realisiert abbrechbaren Aufruf
  - **Achtung:** Insbesondere hier ist für den Fall, das unterbrochen wird (und eine Ausnahme ausgelöst wird), der `unlock` Aufruf wichtig!
  - Wird am besten im `finally`-Block realisiert
- Weitere Methode: `tryLock` – Gibt sofort `true` oder `false` zurück und betritt entsprechend den kritischen Abschnitt

# Inhalt

# Synchronized

## The old way

- Über das `synchronized` Schlüsselwort lässt sich die gleiche Funktionalität wie mit *Locks* erreichen
- Sind seit Java 1.0 Teil der Sprache
- Können jedoch direkt in Methodensignaturen verwendet werden
- Dadurch wird sichergestellt, dass eine Methode eines Objekts nur gleichzeitig durch einen Thread aufgerufen wird
- Auf verschiedenen Objekten kann die Methode weiterhin gleichzeitig aufgerufen werden

# Synchronized

## In der Methodensignatur

```
1 synchronized void foo(){
2   i++;
3 }
```

# Synchronized

- `synchronized` kann beliebig atomare Codeausschnitte umfassen
- Im Block wird angegeben auf welches Objekt ein Monitor gesetzt wird
- Wie beim Aufruf von `lock` wird geprüft ob ein anderer Thread in einem synchronized Block mit dem gleichen Monitor ist
- Und es wird ggf. gewartet bis dieser abgeschlossen ist
- Über `Thread.holdsLock` lässt sich prüfen, ob der aktuelle Thread den Monitor für ein Objekt hält

# Deadlocks

- Die Abhängigkeiten von Monitoren müssen unbedingt berücksichtigt werden
- Insbesondere wenn mehrere Locks in verschiedene Methoden verwendet werden
- Sonst kann es im Programm zu einem sog. „Deadlock“ kommen
  - Zwei Programmabschnitte warten auf die Freigabe eines Monitors
  - Der durch den jeweils anderen blockiert wird

# Deadlocks


Ich will schreiben!
Geben Sie mir Papier!
Geben Sie mir den Stift!

# Inhalt

# Warten und Benachrichtigen

- Synchronisierung bietet einfache Möglichkeiten Nebenläufige Prozesse zu koordinieren
- jedoch in der Praxis sind oft komplexere Abbildungen notwendig
- Wenn Threads Daten austauschen sollen wird oft ein Benachrichtigungsverfahren benötigt
- Ursprünglich wurde dies (Seit Java 1.0) über die Methoden `wait()` und `notify()` realisiert
- Zeitgemäßer ist aber die Verwendung von `Condition` Objekten
- Diese bekommt man über `newCondition()` Methode eines ReentrantLocks

# Warten und Benachrichtigen

## Produzenten-Konsumenten-Beispiele

- Produzenten-Konsumenten-Beispiele sind klassische Anwendungen für Warten und Benachrichtigung:
- Ein Thread liefert Daten, die der andere verarbeiten muss
- Konsument soll keine (kostpsielige) Endlosschleife nutzen um auf Informationen zu warten
- Stattdessen soll Produzent ihn benachrichtigen, wenn Daten verfügbar sind
- Die `Condition` Schnittstelle verfügt dafür über Methoden:
  - `await()` – Betritt den wartenden Zustand
  - `signal()` – Benachrichtigt die wartenden Threads und weckt diese auf

# Condition

## Besonderheiten

- `await` und `signal` können muss man sich in einem kritischen Abschnitt befinden
- Die Methoden lösen sonst eine `IllegalMonitorStateException` austauschen
- Bei Aufruf von `await` wird das entsprechende Lock temporär freigegeben
- Sonst könnte kein anderer Thread den entsprechenden `signal` Call auslösen
- `await` kann auch mit Timeout definiert werden
- Sollten mehrere Threads warten, können über `signalAll` alle wartenden Threads benachrichtigt werden

# Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt