LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

mobile and distributed systems group

Nebenläufigkeit mit Java

Einheit 02: Eigenschaften & kritische Abläufe

Lorenz Schauer
Lehrstuhl für Mobile und Verteilte Systeme

Heutige Agenda

Eigenschaften von Threads

- Name, Priorität, Zustand
- Zustandsübergänge
  - Bewirken und Behandeln
- Auf Ergebnis warten mit join()

Kritische Abläufe und Synchronisation

- Motivation
  - Kritische Abläufe
- Was ist zum Schützen
- Locks und Synchronize

Praxis:

- Join and Sleep

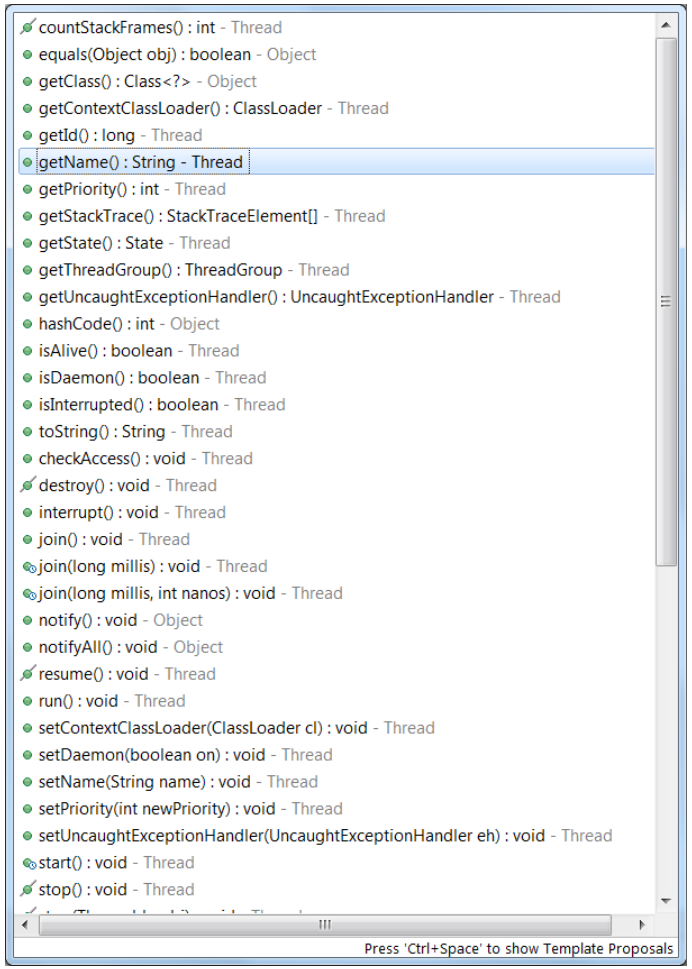
Lernziele

- Weitere typische Eigenschaften von Threads kennenlernen
- Threads beeinflussen können
- Einstieg in Synchronisation und kritische Abläufe

Thread-Eigenschaften

Ein Thread (Exemplar der Klasse `java.lang.Thread`)

- Besitzt neben der `run()` Methode eine Menge an Eigenschaften
  - Zustand
  - Priorität
  - Name …



Thread-Eigenschaft: Name

Ein Thread besitzt einen Namen:

- Implizit „`Thread-<Nr>`“
- Setzen des Namens
  - Üben den Konstruktor
    - `new Thread(Runnable target, String name)`
  - Über den setter
    - `t.setName(String name);`
- Name holen über
  - `t.getName();`

```
@Override
public void run() {

   for(int i=0; i<100; i++)
     System.out.println("Thread: "+this.getName()+" schriebt Zeile: "+i);
}
```

Threads vs. Runnable

Erweiterung der Klasse Thread hat den Vorteil, dass geerbte Methoden direkt genutzt werden können.

Wenn `Runnable` implementiert wird, müssen wir Umweg über statische Methode `currentThread()` gehen.

- Liefert Objektreferent auf den aktuell ausführenden Thread

```
public static void main(String[] args) {

     Runnable r = () ->{

        for(int i=0; i<100; i++)
           System.out.println("Thread: "+Thread.currentThread().getName()+"
             schriebt Zeile: "+i);

     };


     new Thread(r).start();
   }
}
```

Thread-Eigenschaft: Priorität

Threads haben eine Priorität

- Dadurch können höher priorisierte Threads beim Auswahl durch den Scheduler bevorzugt werden
- Ist immer eine Zahl zwischen 1 (`MIN_PRIORITY`) und 10 (`MAX_PRIORITY`)
  - Wird bei der Initialisierung durch den erzeugenden Thread mitgegeben
    - Normalerweise: 5 (`NORM_PRIORITY`)
- Kann mittels `int getPriority()` abgefragt
- Und mittels `void setPriority(int prio)` gesetzt werden

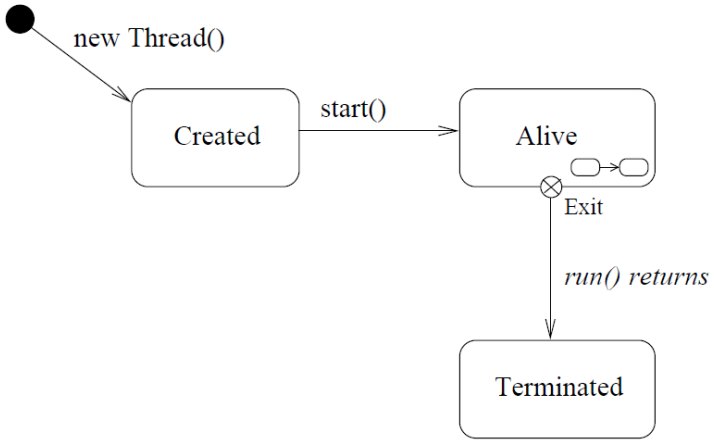
Ob und wie die jeweilige Priorität auch berücksichtigt wird ist nicht immer eindeutig

- Hängt auch vom Betriebssystem ab

Thread-Eigenschaft: Zustand

Wiederholung: Ein Thread besitzt einen Lebenszyklus und mehrerer Zustände

- Mit `isAlive()` kann abgefragt werden, ob ein Thread bereits gestartet wurde, aber noch nicht tot ist.



- Zustandsabfrage mittels `getState();`
- Kann folgende Zustände liefern:

Quelle: Skript Parallele Programmierung. R. Hennicker 2011

Zustand	Erläuterung
NEW	Neuer Thread, noch nicht gestartet.
RUNNABLE	Läuft in der JVM.
BLOCKED	Wartet auf einen MonitorLock, wenn er bspw. einen synchronized-Block betreten möchte.
WAITING	Wartet auf ein notify() bzw. notifyAll()
TIMED_WAITING	Wartet in einem sleep().
TERMINATED	Ausführung ist beendet.

Zustandsübergänge bewirken

Threads warten lassen (Running->Non-Runnable):

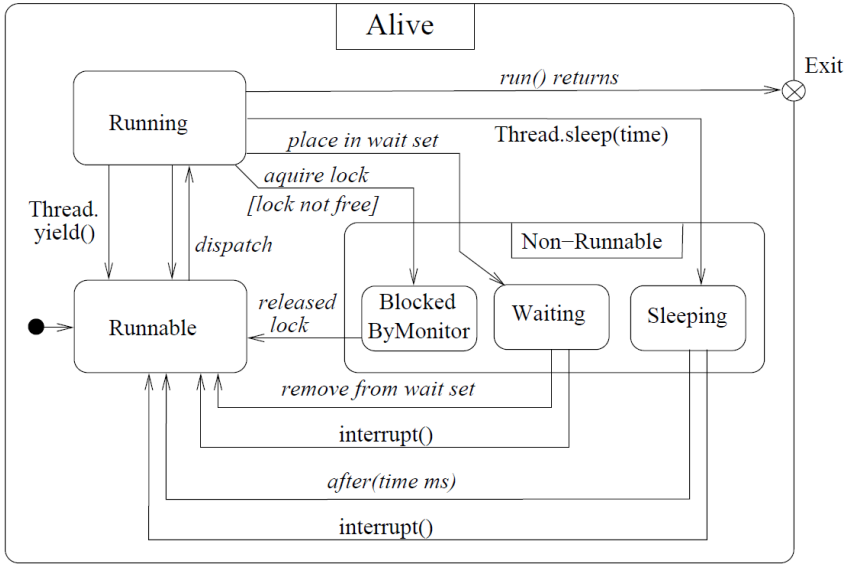
- `Thread.sleep(long millis) throws InterruptedException`
- `TimeUnit.MILLISECONDS.sleep(long millis) throws InterruptedException`

Einen Thread aussetzen (Running -> Runnable)

- Mittels `static void yield()` gibt der laufende Thread freiwillig seine Rechenzeit ab und reiht sich wieder in die Warteschlange der rechenbereiten Threads.
  - Hinweis: Ist für die JVM nicht verbindlich!

Das Ende eines Threads (-> TERMINATED):

- `run()` wird ordentlich beendet
- RutimeException in `run()`
- Der Thread wird von Außen abgebrochen
- JVM wird beendet und beendet damit auch alle Threads



Quelle: Skript Parallele Programmierung. R. Hennicker 2011

Zustandsübergänge bewirken: Threads beenden

Threads zum Beenden auffordern (Running -> Exit)

- per `final void stop();`
  - Wird **nicht empfohlen** uns ist als deprecated (veraltet) deklariert!
  - Bietet die letzte Möglichkeit einen Thread abzuschießen
    - Thread wird radikal beendet, ohne seinen Zustand konsistent abzuschließen
    - Aber: Behandlung durch die `ThreadDeath-Ausnahme` möglich

```
Thread t = new Thread()
{
  @Override public void run()
  {
    try
    {
      while ( true ) System.out.println("Go on and on and on");
    }
    catch ( ThreadDeath td )
    {
      System.out.println( "Und vorbei." );
      throw td;
    }
  }
};
t.start();
try { Thread.sleep( 1 ); } catch ( Exception e ) { }
t.stop();
```

Zustandsübergänge bewirken: Threads beenden

Besser mit Interrupts arbeiten!

- Mittels `void interrupt()` lässt sich von Außen ein internes Flag setzen
  - Kann mittels `boolean isInterrupted()` abgefragt werden
- Häufig bei Threads mit Endlosschleifen benutzt

```
Thread t = new Thread()
{
  @Override
  public void run()
  {
    System.out.println( "Es gibt ein Leben vor dem Tod.  " );

    while ( ! isInterrupted() )
    {
      System.out.println( "Und er läuft und er läuft und er läuft" );

      try
      {
       Thread.sleep( 500 );
      }
      catch ( InterruptedException e )
      {
       interrupt();
       System.out.println( "Unterbrechung in sleep()" );
      }
    }

    System.out.println( "Das Ende" );
  }
};
t.start();
Thread.sleep( 2000 );
t.interrupt();
```

Flag wird geprüft:
Falls true wird while-Schleife verlassen

Achtet ebenfalls auf Interrupt.
Falls Interrupt behandelt wird, wird internes Flag zurückgesetzt. Daher:

In der Interrupt-Behandlung nochmal das Flag setzen, damit While-Schleife endet!

Zustandsübergänge bewirken: Threads beenden

Zusammenfassung:

- `interrupt()`: Setzt das entsprechende Flag für die Beendungs-Routine
  - Beendet aber selbst nicht den Thread!
- Flag lässt sich mit der Methode `isInterrupted()` abfragen wodurch die Beendungs-Routine eingeleitet werden kann
  - Meistens in Schleifen
- Statische Methode `boolean interrupted()` frägt nach dem gesetzten Flag und setzt es gleichzeitig zurück
  - D.h.: Ein erneuter Aufruf von `interrupted()` liefert `false` zurück

- Bei `isInterrupted()`, müssen wir beachten, dass neben `sleep()` auch die Methoden `join()` und `wait()` durch die `InterruptedException` das Flag löschen.
  - D.h. `interrupt()` beendet diese Methoden mit der Ausnahme.

Im letzten Moment…

`UncaughtExceptionHandler`:

- Für eine koordinierte Beendigung bei einem Laufzeitfehler lässt sich einem Thread ein `UncaughtExceptionHandler` mitgeben
  - `void setUncaughtExceptionHandler( Thread.UncaughtExceptionHandler eh )`
- Stell eine innere Schnittstelle in Threads dar
  - `void uncaughtException(Thread t, Throwable e)` muss implementiert werden
  - Wird Im Falle des Abbruchs durch eine unbehandelte Ausnahme durch JVM aufgerufen

```
MyThread t = new MyThread();

UncaughtExceptionHandler eh = new UncaughtExceptionHandler() {

   @Override
   public void uncaughtException(Thread t, Throwable e) {
      System.out.println(t.getName() +"wird beendet" );
      e.printStackTrace();
   }
};
t.setUncaughtExceptionHandler(eh);
```

Auf Ergebnis warten

Manchmal müssen wir auf Ergebnisse eines Threads warten, damit wir mit diesen weiterrechnen können.

- Bsp.: Divide-And-Conquer Ansätze
- Es ist aber nicht immer klar, wann ein Ergebnis zur Verfügung steht.
- Warten auf das Ende der `run()` Methode mittels `join()`
  - Mittels `join(long millis)` lässt sich eine obere Wartezeit definieren

```
public class MyThread extends Thread{
   private int number = 0;

   @Override
   public void run() {

      number = 1;
   }

   public int getNumber(){
      return number;
   }

}
```

```
public class Test {

   public static void main(String[] args)
{

      MyThread t = new MyThread();

      t.start();
      try {
         t.join();
      } catch (InterruptedException e1) {
         e1.printStackTrace();
      }
      System.out.println("Number is:
"+t.getNumber());
   }
}
```

Programmier-Aufgabe

Schreiben Sie in Ihrem Eclipse-Projekt „ThreadsExcercises“ folgendes Programm:

- Ziel der Aufgabe ist es drei Threads zu programmieren die auf das Beenden des anderen Warten und dann eine Zeit schlafen:
  - Die geforderte Aufgabe soll in einer Klasse implementiert werden
    - Drucken Sie jeden neuen Zustand auf der Konsole aus

- Erweitern Sie die Klasse `JoinAndSleep` aus der Klasse `Thread`
- Attribute:
  - Die Klasse hat ein Ganzzahlattribut `sleep` zur Verwaltung der Schlafzeit
  - Die Klasse hat eine Referenz auf ein Objekt der Klasse `JoinAndSleep`
- Konstruktor
  - Der Konstruktor der Klasse erlaubt es die Schlafzeit zu übergeben und eine Referenz auf einen anderen Thread
- `run()` Methode: Diese Methode implementiert die oben genannte Semantik zum Warten und Schlafen
  - Falls ein Thread gegeben ist soll auf sein Ende gewartet werden
  - Anschließend soll eine bestimmte Zeit geschlafen werden
  - Fügen Sie zwischen allen Schritten Konsolenausgaben ein um den Fortschritt zu kontrollieren. Geben Sie hier immer auch den aktuellen Thread aus!
- `main()` Methode
  - Erzeuge Thread 3: Er soll auf keinen Thread warten und dann 4000ms schlafen
  - Erzeuge Thread 2: Er soll auf Thread 3 warten und dann 3000ms schlafen
  - Erzeuge Thread 1: Er soll auf Thread 2 warten und dann 2000ms schlafen
  - Starten Sie Thread 1, 2 und 3

Kritische Abläufe

Motivation

- Threads verwalten ihre eigenen Daten
  - lokale Variablen
  - und einen Stack
- Sie stören sich also selbst nicht
- Auch Lesen von gemeinsamen Daten ist unbedenklich
- Schreiboperationen sind jedoch kritisch!

- Probleme können durch Scheduling entstehen:
  - Ein Thread arbeitet gerade an Daten, die ein anderer Thread bearbeitet.
  - Hier können gravierende und schwer vorhersehbare Inkonsistenzen entstehen!

- Wir brauchen also Mechanismen, die uns davor schützen!

Kritische Abschnitte

Wenn wir mehrere Threads haben und Programmblöcke, auf die immer nur ein Thread zugreifen sollte, dann müssen diese kritischen Abschnitte geschützt werden!

- Ist zur gleichen Zeit immer nur ein Thread beim Abarbeiten eines Programmteils, dann liegt ein wechselseitiger Ausschluss bzw. eine atomare Operation vor
- Arbeitet ein Programm bei nebenläufigen Threads falsch, ist es nicht *thread-sicher* (engl. *thread-safe*).

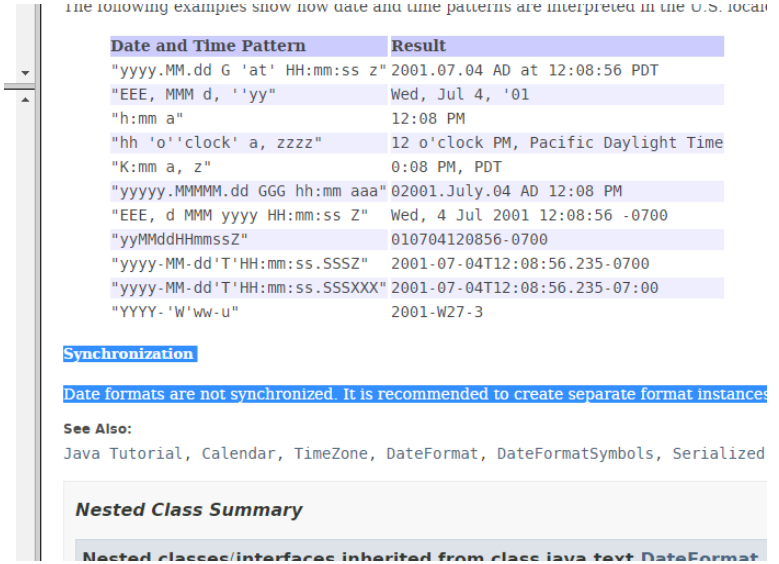
Thread-sichere und nicht thread-sichere Klassen der Java Bibliothek

Thread-sichere Klassen/Objekte

- Immutable Objekte
  - automatisch thread-sicher, da NUR Lesezugriffe und keine Schreibzugriffe
  - Immutable-Klassen wie String oder Wrapper-Klassen kommen daher ohne Synchronisierung aus.

Nicht thread-sichere Klassen/Objekte

- Eher die Regel, statt die Ausnahme
- Bei nebenläufiger Programmierung immer in der Java API-Dokumentation nachschlagen, ob Objekte thread-sicher sind.
- In einigen wenigen Fällen haben Entwickler die Wahl zwischen thread-sicheren und nicht-thread-sicheren Klassen:
  - Bsp.: ArrayList vs. Vector



The following examples show how date and time patterns are interpreted in the U.S. locale

Date and Time Pattern	Result
"yyyy.MM.dd G 'at' HH:mm:ss z"	2001.07.04 AD at 12:08:56 PDT
"EEE, MMM d, ''yy"	Wed, Jul 4, '01
"h:mm a"	12:08 PM
"hh 'o''clock' a, zzzz"	12 o'clock PM, Pacific Daylight Time
"K:mm a, z"	0:08 PM, PDT
"yyyyy.MMMMM.dd GGG hh:mm aaa"	02001.July.04 AD 12:08 PM
"EEE, d MMM yyyy HH:mm:ss Z"	Wed, 4 Jul 2001 12:08:56 -0700
"yyMMddHHmmssZ"	010704120856-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSZ"	2001-07-04T12:08:56.235-0700
"yyyy-MM-dd'T'HH:mm:ss.SSSXXX"	2001-07-04T12:08:56.235-07:00
"YYYY-'W'ww-u"	2001-W27-3

Synchronization

Date formats are not synchronized. It is recommended to create separate format instances

See Also:

Java Tutorial, Calendar, TimeZone, DateFormat, DateFormatSymbols, Serialized

Nested Class Summary

Nested classes/interfaces inherited from class java.text.DateFormat

Was ist zum Schützen?

Prinzipiell sollten kritische Abschnitte und nicht atomare Schreibeoperationen geschützt sein.

- Manche Befehle sehen atomar aus, sind es aber nicht.
  - Bsp.: i++

```
// Was passiert bei i++?

1. i wird gelesen und auf dem Stack abgelegt
2. Danach wird die Konstante 1 auf dem Stack abgelegt
3. Und anschließend werden beide Werte addiert
4. Das Ergebnis wird nun vom Stack geholt und in i geschrieben
```

Aus diesem Grund müsste i++ geschützt ausgeführt werden.

- Java-Konstrukte zum Schutz der kritischen Abschnitte
  - Die Schnittstelle java.util.concurrent.locks.Lock
    - Wird u.a. von ReentrantLock implementiert
  - Synchronized (Nächste Stunde)

Beispiel zu ReentrantLock


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

public class Test {

   public static void main(String[] args) {

      final Lock lock = new ReentrantLock();

      Runnable r = new Runnable()
      {
        int x=0;
        int y=2;
        boolean flag = true;
        @Override public void run()
        {
          while ( flag )
          {
            lock.lock();

            x=x+2;
            y++;

            lock.unlock();

            if ( x==y){
              System.out.println( "Fertig");
             flag=false;
            }
          }
        }
      };
      new Thread( r ).start();
      new Thread( r ).start();
   }
}
```