Einstieg in die Programmierung von Computern – Teil X

In Teil 10 beschäftigen wir uns weiterhin mit dem wichtigen Thema Multithreading in Java.
In der Nebenläufigkeit von Prozessen und Threads kommt es zwangsläufig zu parallelen Zugriffen auf gemeinsame Daten.
In diesem Teil beantworten wir die Frage, wie die „Synchronisation von Daten“ in der Programmiersprache Java behandelt wird.

Wir haben folgende Problemstellung: Zwei parallel laufende Threads kommunizieren über gemeinsame Variablen miteinander, d. h. beide Threads können auf die gemeinsamen Variablen zugreifen und diese bearbeiten. Was passiert nun, wenn der eine Thread die gemeinsame Variable in einem Moment verändern möchte, in dem der andere Thread dasselbe auch tun möchte? Anders gefragt: Was passiert, wenn mehrere Threads im selben Moment auf gemeinsame Variablen bzw. Daten zugreifen bzw. diese verändern möchten? Antwort: Ohne gewisse Maßnahmen kann es zu undefinierten Ergebnissen führen. Zum Glück bietet Java für diese Fälle, die die Regel darstellen, sehr gute Schutzmechanismen, auf die nun im Detail eingegangen werden soll. Schauen wir aber zuerst ein kleines Beispiel an (siehe Kasten 1), welches die Problematik zeigt. Im Beispielcode von Kasten 1 werden zwei Threads gestartet, die beide auf die statische Variable „zaehler“ zugreifen und hochzählen (zaehler++). Der Java-Code sieht im ersten Moment in Ordnung aus. Die Ausgabe zeigt am Anfang auch das zu erwartende Ergebnis 1 2 3 4 5 …

Lässt man das Programm etwas länger laufen, so entstehen plötzlich unregelmäßig Lücken im Ergebnis. Auf meinem Rechner fehlt auf einmal die Zahl 42 (38 39 40 41 43 … 58 59 42 …) und taucht aber später in der Ausgabe auf. Läuft das Programm weiter, passiert dasselbe mit weiteren Zahlen. Was ist da geschehen?

Beide parallel laufende Threads greifen unsynchronisiert auf eine gemeinsame Variable „zaehler“ zu, d. h. der Zugriff geschieht ungeschützt. Da die Anweisung „System.out.println (zaehler++)“ nicht **„atomar“** ist (atomar bedeutet: die Anweisung kann in der Ausführung nicht unterbrochen werden), kommt es zu der Situation, dass die Anweisung in der Ausführung unterbrochen wird und der Betriebssystem-Scheduler mit dem anderen Thread weiter macht. Der unterbrochene Thread bekommt wenig später wieder Rechenzeit vom Scheduler und kann dann seinen alten berechneten Wert (in meinem Fall „32“) ausgeben. Der andere Thread hat in der Zwischenzeit natürlich weitergezählt (ist bei der Zahl 59 gelandet). Da scheint ja etwas schiefzulaufen. In der Tat muss der Programmierer für solche Fälle etwas vorsehen und die Variable durch **„Synchronisation“** beider Threads schützen. Werden die Threads nicht synchronisiert, kommt es zu sogenannten **„Inkonsistenzen“**.

Synchronisation durch „Monitore“

Die Programmiersprache Java verwendet für die Synchronisation nebenläufiger Prozesse bzw. Threads die sogenannten **„Monitore“**. Ein Monitor ist eine „automatisch verwaltete Sperre“, die einen „kritischen Bereich“ im Java-Code atomar macht. Ein Monitor kapselt den kritischen Bereich im Java-Code.

Die Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen. Ist sie beim Eintritt in den Monitor bereits von einem anderen Prozess bzw. gesetzt, so muss der aktuelle Prozess bzw. Thread so lange warten, bis der Konkurrent die Sperre freigegeben und den Monitor verlassen hat. Ein nicht atomarer Java-Code-Bereich wird durch einen Monitor atomar gemacht.

Das Konzept der „Monitore“ wird mithilfe des in die Sprache integrierten Schlüsselworts **„synchronized“** realisiert. Durch „synchronized“ kann eine vollständige Methode oder ein Anweisungsblock innerhalb einer Methode geschützt werden. Der Eintritt in den so deklarierten Monitor wird durch das Setzen einer Sperre auf einer Objektvariablen erreicht. Bezieht sich „synchronized“ auf eine vollständige Methode, wird als Sperre der this-Pointer verwendet, andernfalls ist eine Objektvariable explizit anzugeben.

Wir werden nun unser erstes Multithreadingbeispiel so ändern, dass die Ausgabe den Zählerstand korrekt anzeigt, d. h. wir werden die Ausgabeanweisung in den „kritischen Bereich“ nun in einen automatisch verwalteten Monitor verwandeln. Dazu verwenden wir „synchronized“. Der Java-Code im Kasten 2 zeigt die Lösung.

Die zuerst mal nahe liegende Lösung, die Anweisung System.out.println(zaehler++); durch einen synchronized-Block auf der Variablen „this“ zu synchronisieren, funktioniert hier nicht. Die beiden laufenden Threads stellen ja unterschiedliche Instanzen dar, und ein Eintreten in den kritischen Bereich würde somit erlaubt sein. In unserem Fall können wir die Methode „getClass()“ verwenden, die uns für beide Threads dasselbe Klassen-Objekt liefert. Da ja beide Threads mit derselben Klasse erstellt sind, funktioniert diese Variante.

Wie bereits weiter oben beschrieben, kann auch eine ganze Java-Methode „synchronisiert“ werden. Schauen wir uns auch dazu ein kleines Beispiel im Kasten 3 an.

Im unserem Beispiel in Kasten 3 haben wir eine ganze Methode („naechsteZahl()“ in Klasse „Zaehler“) mit „synchronized“ gekennzeichnet und macht die Methode zu einem Monitor, d. h. alle

```
/**
 * Datei: IchBinEinThread.java
 *
 * Dieses Programm startet zwei unsynchronisierte Threads.
 */

package com.oemus.itkolumne;

public class IchBinEinThread Thread {

 static int zaehler = 0; // Gemeinsame statische Variable!

 /**
  * Startet das Java-Programm
  */
  public static void main (String args[]) {

    IchBinEinThread ichBinEinThread_1 = new IchBinEinThread(); // Startet Thread 1
    IchBinEinThread ichBinEinThread_2 = new IchBinEinThread(); // Startet Thread 2

    //**********************************
    // Hier werden die Threads gestartet
    //**********************************
    ichBinEinThread_1.start();
    ichBinEinThread_2.start();
  }

  /**
   * Das tut der Thread.
   */
  public void run(){

      while (true) { // Endlosschleife!
          System.out.println(zaehler++);
      }
  }
```

Kasten 1: Zwei Threads greifen ungeschützt auf eine gemeinsame Variable zu.

```
/**
 * Datei: IchBinEinThread.java
 *
 * Dieses Programm startet zwei unsynchronisierte Threads.
 */

package com.oemus.itkolumne;

public class IchBinEinThread Thread {

 static int zaehler = 0; // Gemeinsame statische Variable!

 /**
  * Startet das Java-Programm
  */
  public static void main (String args[]) {

    IchBinEinThread ichBinEinThread_1 = new IchBinEinThread(); // Startet Thread 1
    IchBinEinThread ichBinEinThread_2 = new IchBinEinThread(); // Startet Thread 2

    //**********************************
    // Hier werden die Threads gestartet
    //**********************************
    ichBinEinThread_1.start();
    ichBinEinThread_2.start();
  }

  /**
   * Das tut der Thread.
   */
  public void run(){

      while (true) { // Endlosschleife!
          synchronized (getClass()) {
              System.out.println(zaehler++);
          }
      }
  }
}
```

Kasten 2: Zwei Threads greifen synchronisiert auf eine gemeinsame Variable zu .

```
/**
 * Datei: Zaehler.java
 *
 * Die Klasse Zaehler beinhaltet eine synchronisierte Methode „naechste
   Zahl()“.
 */

package com.oemus.itkolumne;

public class Zaehler {

    int zaehler;

    public Zaehler(int zaehlerWert) {
       this.zaehler = zaehlerWert;
    }

    /**
     * Synchronisierte Methode.
     */
    public synchronized int naechsteZahl() {
    int ergebnis = zaehler;

    // Hier erfolgen einige rechenintensiven Berechnungen, die leicht
    // durch den  Scheduler unterbrochen werden kann.
      ...

    zaehler++;
    return ergebnis;
  }
}
```

Kasten 3: Zwei Threads greifen synchronisiert auf eine gemeinsame Variable zu .

Anweisungen innerhalb der Methode werden damit zu einem atomaren kritischen Bereich deklariert. Eine Unterbrechung des kritischen Abschnitts durch einen anderen Thread ist dann nicht mehr möglich.

Warten mit „wait()“ & Aufwecken mit „notify()“

Die Programmiersprache Java stellt für Multithreading zwei weitere sehr interessante Methoden **„wait()“** und **„notify()“** zur Verfügung, mit denen eine Steuerung von zeitlichen Abläufen realisiert werden kann. Beide Methoden können nur in einem Monitor, also einem synchronisierten kritischen Bereich, verwendet werden.

Die Methode **„wait()“** nimmt die bereits gewährten Sperren (temporär) zurück und stellt den Prozess bzw. Thread, der den Aufruf von wait() verursacht hat, in die Warteliste des Objekts. Damit wird der Thread unterbrochen und im Scheduler als „wartend“ gesetzt.

Mit **„notify()“** wird ein in der Warteliste befindlicher Thread wieder aus der Warteliste entfernt und aktiviert die aufgehobenen Sperren wieder. Der Scheduler behandelt den Thread wieder ganz normal. „wait()“ und „notify()“ sind damit für elementare Synchronisationsaufgaben geeignet, bei denen es auf die Steuerung von zeitlichen Abläufen ankommt.

```
/**
 * Datei: IchBinEinTread.java
 *
 * Dieses Programm startet zwei Threads.
 */

public class IchBinEinThread Thread {

    private String name;
    private Zaehler zaehler;

    /**
     * Konstruktor der Klasse Zaehler.
     */
    public IchBinEinThread(String name, Zaehler zaehler) {
        this.name = name;
        this.zaehler = zaehler;
    }

    public static void main(String[] args) {
        Thread t[] = new Thread[5];
        Zaehler zaehler = new Zaehler(10);

        for (int idx = 0; idx < 5; ++idx) {
            t[idx] = new IchBinEinThread("Thread-" + idx, zaehler); // Neues Thread-Objekt
            t[idx].start(); // Start Thread
        }
    }

    public void run() {
        while (true) {
            System.out.println(zaehler.naechsteZahl() + " für " + name);
        }
    }
}
```

Kasten 3: Zwei Threads greifen synchronisiert auf eine gemeinsame Variable zu .

```
synchronized(obj) {
    try {
        o.wait();
        // Ich habe gerade gewartet,
        jetzt kann ich wieder weiter-
        machen.
    } catch (InterruptedException e) {
        ...
    }
}
```


Wenn der zweite Thread den Monitor des Objekts obj bekommt, kann er den wartenden Thread aufwecken. Er bekommt den Monitor durch das Synchronisieren der Methode, was bei Objektmethoden synchronized(this) entspricht. Der zweite Thread gibt das Signal mit notify().

```
synchronized(obj) {
    // Habe etwas gearbeitet und
    informiere nun meinen War-
    tenden.
    o.notify();
}
```

Wir haben in diesem Teil gesehen, mit welchen einfachen Mitteln gemeinsame Ressourcen-Zugriffe (auch als „kritische Bereiche“ bezeichnet) synchronisiert werden können. Java bietet im Bereich des Multithreading noch andere sehr interessante Features an, die ich aber in der Serie nicht behandeln möchte, da sonst der Rahmen gesprengt wird.

Ausblick zum Teil 11 der Serie

Ab dem nächsten Teil möchte ich mit der Webprogrammierung in Java beginnen. Auch hier stellt Java sehr mächtige Mittel für die Programmierung von sehr leistungsfähigen Enterprise-Lösungen zur Verfügung. Mit dem Teil 11 startet dann eine umfangreiche Serie zum Thema Java in der Webprogrammierung. Java zeigt gerade im Enterprise-Bereich seine ganze Stärke und wartet mit tollen Technologien auf. Bleiben Sie also dran! ZT

Infos zum Autor

ZT Adresse

Thomas Burgard Dipl.-Ing. (FH)
Softwareentwicklung & Webdesign
Bavariastraße 18b
80336 München
Tel.: 089 540707-10
info@burgardsoft.de
www.burgardsoft.de
burgardsoft.blogspot.com
twitter.com/burgardsoft