gds2 Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

Inhaltsverzeichnis

gds2 Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

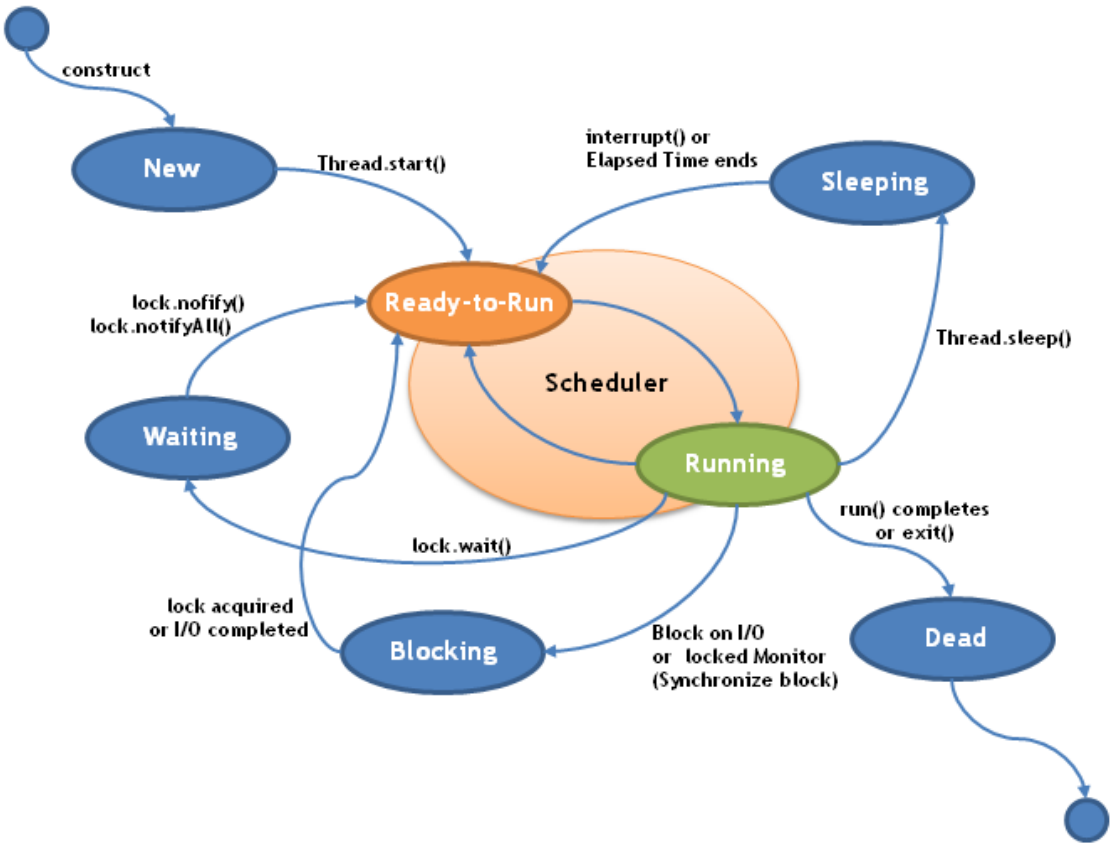
1 Grundlagen

Unter "Nebenläufigkeit" versteht man das (quasi-)gleichzeitige Ablaufen von Teilen *eines* Programms. Die Programmteile, die parallel ablaufen, nennt man Threads (=Faden). Threads benutzen den gemeinsamen Adressraum des Prozesses, können also auf dieselben Variablen zugreifen.[1]

Threads werden eingesetzt

- zur Programmbeschleunigung: wenn z.B. Berechnungen, die zu einem Gesamtergebnis beitragen, parallel ausgeführt werden können. Eine Beschleunigung kann hier aber nur erzielt werden, wenn mehrere Prozessoren oder Prozessorkerne zur Verfügung stehen.
- wenn ein (quasi-)paralleler Ablauf erwünscht ist: wenn z.B. gleichzeitig zur Darstellung einer Animation auch Benutzereingaben bearbeitet werden sollen oder zur gleichzeitigen Bearbeitung von vielen Benutzeranfragen wie z.B. bei einem Webserver.

Zustände eines Threads (Quelle: https://avaldes.com/java-thread-states-life-cycle-of-java-threads)



2 Threads in Java

Bei der Programmiersprache Java ist das Multithreading-Programmkonzept direkt integriert. Daher ist in Java ein Thread im Grunde nichts anderes als ein Objekt einer Klasse, die sich entweder

- von der Klasse `Thread` ableitet oder
- das Interface `Runnable` implementiert.

[1] Und sich (im Gegensatz zur Nebenläufigkeit bei Multitasking) auch gegenseitig zum „Absturz“ bringen

2.1 Erzeugen eines neuen Thread

2.1.1 Ableitung aus der Klasse Thread

Um einen neuen Thread zu erzeugen muss eine eigene Klasse aus `Thread` abgeleitet und die Methode `run()` überlagert werden.
Der Thread wird gestartet, indem die Methode `start()` aufgerufen wird. Diese wiederum ruft die Methode `run()` auf und beendet sich anschließend. Somit kann nach dem Aufruf von `start()` die Anwendung weiterlaufen, während "gleichzeitig" die Methode `run()` abläuft.

`run()` sollte niemals direkt aufgerufen werden!

Beispiel:

```
public class ThreadDemo1_UI
{
  public static void main(String[] args)
  {
    ThreadClass1 thread = new  ThreadClass1();
    thread.start();
    System.out.println("Ende main");
  }
}


public class ThreadClass1 extends Thread
{
  public void run()
  {
    int iZahl=0;
    while (iZahl< 10)
    {
      System.out.println("iZahl = "+iZahl);
      iZahl++;
    }
  }
}
```

Ausgabe:

```
Ende main
iZahl = 0
iZahl = 1
iZahl = 2
iZahl = 3
iZahl = 4
iZahl = 5
iZahl = 6
iZahl = 7
iZahl = 8
iZahl = 9
```

Nebenläufigkeit kann somit erzeugt werden, indem mehrere Threads nacheinander gestartet werden:

```
public class ThreadDemo2_UI
{

  public static void main(String[] args)
  {
    ThreadClass2 thread1 = new ThreadClass2();
    thread1.start();
    ThreadClass2 thread2 = new ThreadClass2();
    thread2.start();
    System.out.println("Ende main");
  }
}


public class ThreadClass2 extends Thread
{
  public void run()
  {
    int iZahl=0;
    /*
     * als Ableitung von Thread können Thread-Infos
     * (z.B. der Name) direkt abgerufen werden:
     */
    String sName = this.getName();

    while (iZahl< 10)
    {
      System.out.println(sName+": iZahl = "+iZahl);
      iZahl++;
    }
  }
}
```

(mögliche) Ausgabe:

```
Ende main
Thread-0: iZahl = 0
Thread-0: iZahl = 1
Thread-0: iZahl = 2
Thread-0: iZahl = 3
Thread-0: iZahl = 4
Thread-0: iZahl = 5
Thread-0: iZahl = 6
Thread-1: iZahl = 0
Thread-1: iZahl = 1
Thread-1: iZahl = 2
Thread-0: iZahl = 7
Thread-1: iZahl = 3
Thread-1: iZahl = 4
Thread-1: iZahl = 5
Thread-1: iZahl = 6
Thread-1: iZahl = 7
Thread-1: iZahl = 8
Thread-1: iZahl = 9
Thread-0: iZahl = 8
Thread-0: iZahl = 9
```

2.1.2 Implementieren von Runnable

Wenn eine Klasse bereits von einer Oberklasse erbt, kann sie wegen der Einfachvererbung nicht zusätzlich von der Klasse `Thread` erben. In diesem Fall kann ein Thread nur erzeugt werden, indem das Interface `Runnable` implementiert wird. Dieses fordert die Implementierung der Methode `public void run()`.
Da damit jedoch noch nicht die Methoden der Klasse `Thread` zur Verfügung stehen, muss diese `Runnable`-Objekt an den Konstruktor der Klasse `Thread` übergeben werden, damit der Thread gestartet werden kann: `public Thread (Runnable r)`.

Beispiel:

```
public class ThreadDemo3_UI
{
  public static void main(String[] args)
  {
    /*
     * Start eines Threads, der aus der Klasse Thread
     * abgeleitet ist:
     */
    ThreadClass2 thread1 = new ThreadClass2();
    thread1.start();
    /*
     * Start eines Thread, der Runnable implementiert:
     */
    ThreadClass3 thread2_runnable = new ThreadClass3();
    Thread thread2 = new Thread(thread2_runnable);
    thread2.start();
    System.out.println("Ende main");
  }
}

public class ThreadClass2 extends Thread
{
  public void run()
  {
    int iZahl = 0;
    String sName = this.getName();
    while (iZahl < 10)
    {
      System.out.println(sName+ ": iZahl = " + iZahl);
      iZahl++;
    }
  }
}

public class ThreadClass3 implements Runnable
{
  public void run()
  {
    int iZahl = 0;
      /*
       * als Runnable können Thread-Infos
       * (z.B. der Name) nicht direkt abgerufen werden,
       * der aktuelle Thread
       * muss erst ermittelt werden:
       */
    String sName = Thread.currentThread().getName();
    while (iZahl < 10)
    {
      System.out.println(sName+ ": iZahl = " + iZahl);
      iZahl++;
    }
  }
}
```

(mögliche) Ausgabe:

```
Thread-0: iZahl = 0
Thread-1: iZahl = 0
Thread-0: iZahl = 1
Thread-1: iZahl = 1
Thread-0: iZahl = 2
Thread-1: iZahl = 2
Thread-1: iZahl = 3
Thread-0: iZahl = 3
Thread-0: iZahl = 4
Thread-1: iZahl = 4
Ende main
Thread-0: iZahl = 5
Thread-1: iZahl = 5
Thread-1: iZahl = 6
Thread-0: iZahl = 6
Thread-0: iZahl = 7
Thread-1: iZahl = 7
Thread-1: iZahl = 8
Thread-0: iZahl = 8
Thread-0: iZahl = 9
Thread-1: iZahl = 9
```

2.2 Beenden eines Threads

Ein Thread wird beendet, sobald das Ende der Methode `run()` erreicht ist.
Ein Thread kann aber auch abgebrochen werden, indem die Methode `stop()` aufgerufen wird. Da aber nicht definiert ist, an welcher Stelle der Thread abgebrochen wird, wurde die Methode `stop()` mit dem JDK 1.2 als `deprecated` markiert und sollte nicht mehr verwendet werden.

```
public class ThreadDemo4_UI
{

  public static void main(String[] args)
  {
    ThreadClass2 thread1 = new ThreadClass2();
    thread1.start();
    System.out.println("Ende main");
    thread1.stop(); // erzeugt eine Warnung: deprecated
  }
}
```

(mögliche) Ausgabe:

```
Ende main
Thread-0: iZahl = 0
Thread-0: iZahl = 1
Thread-0: iZahl = 2
Thread-0: iZahl = 3
Thread-0: iZahl = 4
Thread-0: iZahl = 5
Thread-0: iZahl = 6
```

Besser ist die Alternative, im Thread selbst auf Unterbrechungsanforderungen zu reagieren. Dazu wird im Prinzip nur ein Flag gesetzt, sobald eine Unterbrechungsanforderung gestellt wird. Dieses Flag wird abgefragt und falls es auf `true` steht, kann der Thread kontrolliert beendet werden.
Dieser Mechanismus ist in der Klasse `Thread` bereits implementiert in Form der Methoden `interrupt()` und `isInterrupted()`

```
public class ThreadDemo5_UI
{

  public static void main(String[] args)
  {
    ThreadClass4 thread1 = new ThreadClass4();
    thread1.start();
    System.out.println("Ende main");
     // Aufforderung an den Thread sich zu beenden
     thread1.interrupt();
  }
}
public class ThreadClass4 extends Thread
{
  public void run()
  {
    int iZahl = 0;
     boolean bStop = false;

     while (!bStop && iZahl < 100)
    {
       if (iZahl >= 10 && this.isInterrupted())
       {
         System.out.println("Thread unterbrochen");
         bStop = true;
       }
      else
      {
        System.out.println("iZahl = " + iZahl);
        iZahl++;
      }
    }
  }
}
```

(mögliche) Ausgabe

```
Ende main
iZahl = 0
iZahl = 1
iZahl = 2
iZahl = 3
iZahl = 4
iZahl = 5
iZahl = 6
iZahl = 7
iZahl = 8
iZahl = 9
Thread unterbrochen
```

2.3 Möglichkeiten zur Steuerung von Threads

2.3.1 `join ()`

`public final join()`
Wartet auf das Ende des Threads, für den die Methode aufgerufen wurde.
`public final join(long millis)`
Wartet maximal `millis` Millisekunden auf das Ende des Threads.

Beispiel:
```
t.start();
...
t.join();
// hier erst weiter, nachdem Thread t terminiert hat.
```

2.3.2 `yield()`

Innerhalb von run() aufgerufen gibt der laufende Thread die Steuerung ab, um einem evtl. wartenden Thread mit gleicher Priorität Rechenzeit zu ermöglichen.

```
public void run()
{
  ...
  // Steuerung abgeben:
  this.yield();
  ...
  // hier erst weiter, wenn der Scheduler dem Thread die Steuerung
  // wieder übergeben hat
}
```

2.3.3 `sleep()`

`public static void sleep(long millis)`
Aktueller Thread wartet `millis` Millisekunden.
Beispiel:
```
// 1 Sekunde pausieren:
Thread.sleep(1000);
```

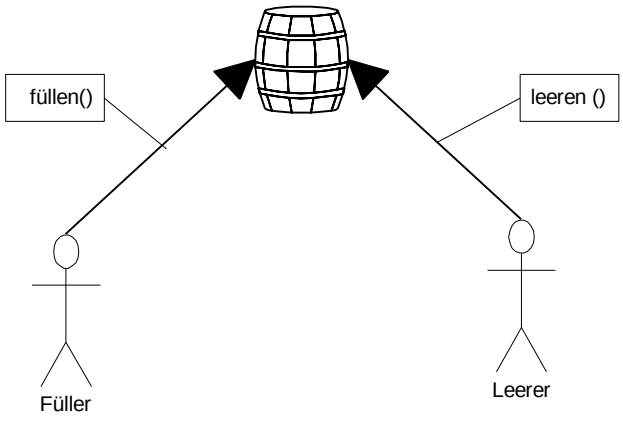
2.4 Synchronisation von Threads

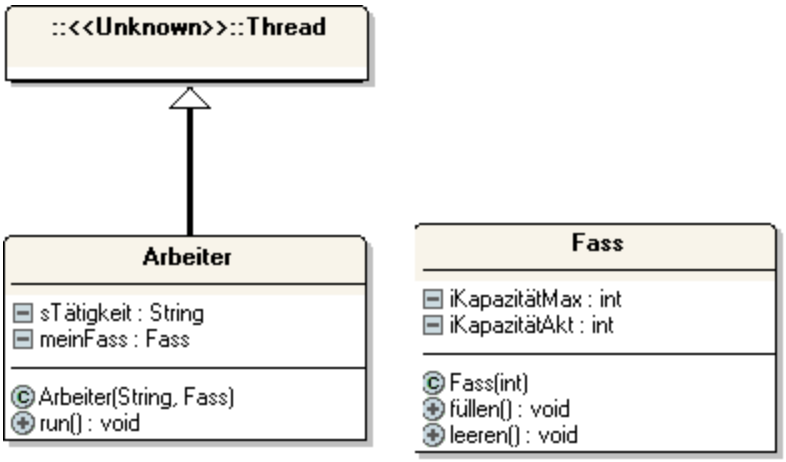
2.4.1 Ausgangssituation

Da Threads im gleichen Adressraum eines Programms arbeiten, können sie auch auf die gleichen Variablen zugreifen. Dies bietet deutliche Vorteile, wenn beispielsweise Ergebnisse zwischen den Threads ausgetauscht werden sollen. Das Ergebnis braucht nur in die gemeinsam bekannte Variable geschrieben zu werden.

Simulations-Beispiel:

Zwei Arbeiter haben die Aufgabe, ein Fass zu füllen bzw. zu leeren. Sie sollen abwechselnd Zugang zum Fass haben.
Jeder Arbeiter ist dabei ein Thread und das Fass die gemeinsam zu benutzende Variable.





gds2 Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

Ein Beispielprogramm startet die beiden Threads

```
public class ThreadDemo6_UI
{
  public static void main(String[] arg)
  {
    Fass einFass = new Fass(30);
    Arbeiter fueller = new Arbeiter("Füller", einFass);
    Arbeiter leerer = new Arbeiter("Leerer", einFass);
    fueller.start();
    leerer.start();
  }
}
public class Arbeiter extends Thread
{
  private String sTätigkeit;
  private Fass meinFass;

  public Arbeiter(String sNeueTätigkeit, Fass meinFass)
  {
    this.sTätigkeit = sNeueTätigkeit;
    this.meinFass = meinFass;
  }
  public void run()
  {
    while (true)
      if (this.sTätigkeit.equals("Füller"))
          this.meinFass.füllen();
      else
          this.meinFass.leeren();
  }
}
public class Fass
{
    private int iKapazitätMax;
    private int iKapazitätAkt;

    public Fass(int iNeueKapazitätMax)
    {
        this.iKapazitätMax = iNeueKapazitätMax;
        this.iKapazitätAkt = 10;
    }
    public void füllen()
    {
        if (iKapazitätAkt < iKapazitätMax)
        {
            iKapazitätAkt++;
            System.out.println("fuellen : " + iKapazitätAkt);
        }
    }
    public void leeren()
    {
        if (iKapazitätAkt > 0)
        {
            iKapazitätAkt--;
            System.out.println("leeren : " + iKapazitätAkt);
        }
    }
}
```

(mögliche) Ausgabe:

```
fuellen : 10
fuellen : 11
fuellen : 12
fuellen : 13
fuellen : 14
fuellen : 15
fuellen : 16
fuellen : 17
fuellen : 18
fuellen : 19
fuellen : 20
fuellen : 21
fuellen : 22
fuellen : 23
fuellen : 24
fuellen : 25
fuellen : 26
fuellen : 27
fuellen : 28
fuellen : 29
leeren : 1
leeren : 29
leeren : 28
leeren : 27
leeren : 26
fuellen : 29
fuellen : 26
fuellen : 27
fuellen : 28
fuellen : 29
leeren : 25
leeren : 29
leeren : 28
leeren : 27
```

Die Ausgabe zeigt, dass beide Threads parallel laufen, dass aber die Threads nicht immer mit dem korrekten Variableninhalt arbeiten.
Grund: beide greifen auf die gemeinsame Variable `iKapazitätAkt` unsynchronisiert zu:

- Der „Füller“ liest den Variablenwert in ein Prozessorregister.
- Der „Leerer“ krätscht dazwischen und ändert den Variablenwert.
- Der „Füller“ kommt wieder dran und hat noch den alten Wert im Register.

In diesem Beispiel muss also dafür gesorgt werden, dass die Methode `leeren()` erst dann ausgeführt wird, wenn die Methode `füllen()` beendet ist.
Eine Möglichkeit, den Zugriff zu steuern sind so genannte Monitore.

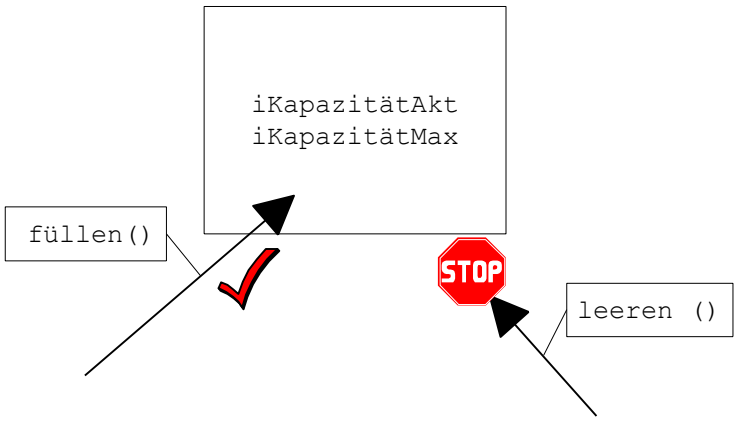
2.4.2 Synchronisation durch Monitore

Das Monitor-Konzept wurde von C.A.R. Hoare eingeführt. Ein Monitor ist ein Objekt, das

- Daten enthält, die nur von bestimmten Zugriffsmethoden bearbeitet werden können (hier: `iKapazitätMax` und `iKapazitätAkt`)
- genau definierte Methoden enthält, die auf die Daten zugreifen (hier: `füllen()` und `leeren()`)
- so genannte Condition-Variablen enthält, die festlegen, ob gerade auf den Monitor zugegriffen werden darf (treten hier nicht in Erscheinung, werden in der Programmiersprache Java intern verwaltet)

Ein Monitor regelt den Zugriff so, dass zu einem Zeitpunkt immer nur eine der zugelassenen Zugriffsmethoden zugreifen kann.
In diesem Beispiel wäre also das Objekt `einFass` der Monitor.

Monitor zum Zeitpunkt des Füllens:



2.4.3 Umsetzung in Java

In Java werden Monitore ganz einfach erzeugt, indem einer Methode, die Zugriffsmethode auf die zu synchronisierenden Daten sein soll, der Modifizierer `synchronized` hinzugefügt wird. Die Steuerung des Zugriffs wird dann automatisch vom System übernommen, der Programmierer muss nichts weiter programmieren.

```
public class ThreadDemo6_UI
{
    public static void main(String[] arg)
    {
        Fass einFass = new Fass(30);
        Arbeiter fueller = new Arbeiter("Füller", einFass);
        Arbeiter leerer = new Arbeiter("Leerer", einFass);
        fueller.start();
        leerer.start();
    }
}

public class Arbeiter extends Thread
{
      ... // s.o.
}

public class Fass
{
    private int iKapazitätMax;
    private int iKapazitätAkt;

    public Fass(int iNeueKapazitätMax)
    {
        this.iKapazitätMax = iNeueKapazitätMax;
        this.iKapazitätAkt = 20;
    }

    public synchronized void füllen()
    {
        if (iKapazitätAkt < iKapazitätMax)
        {
            iKapazitätAkt++;
            System.out.println("fuellen : " + iKapazitätAkt);
        }
    }

    public synchronized void leeren()
    {
        if (iKapazitätAkt > 0)
        {
            iKapazitätAkt--;
            System.out.println("leeren : " + iKapazitätAkt);
        }
    }
}
```

(mögliche) Ausgabe:

```
fuellen : 10
leeren : 9
fuellen : 10
fuellen : 11
fuellen : 12
leeren : 11
fuellen : 12
leeren : 11
fuellen : 12
fuellen : 13
leeren : 12
leeren : 11
fuellen : 12
leeren : 11
leeren : 10
fuellen : 11
leeren : 10
fuellen : 11
fuellen : 12
leeren : 11
fuellen : 12
fuellen : 13
leeren : 12
leeren : 11
fuellen : 12
leeren : 11
leeren : 10
leeren : 9
fuellen : 10
fuellen : 11
```

2.4.4 Weitere Möglichkeiten der Thread Synchronisation in Java

Mit Hilfe von den in 2.4.3 gezeigten „sychronized“ Methoden kann man den Zugriff auf gemeinsam genutzte Ressourcen regeln. Dabei wird der Zugriff auf die Objektattribute für die komplette Ausführungsdauer der entsprechenden Methode reserviert. Dies ist möglicherweise nicht sinnvoll, weil zu umfassend. Deshalb bietet Java noch weitere Möglichkeiten der Synchronisation, die nur Teile einer Methode als „kritischen Abschnitt“ bestimmen. Dies soll hier nur erwähnt und auf einschlägige Literatur verwiesen werden (siehe z.B. Volker Janßen „Angewandte Informatik Java/Softwareentwicklung“ Feb. 2018 Eigenverlag).