Best – Practise (- Regel) zur Benutzung von synchronized[1]

Zusammenfassung:

- Mit **synchronized**, wait, notify und notifyAll lassen sich alle gängigen Synchronisationsaufgaben in Java lösen.

- Ein zentrales Konzept im Zusammenhang mit Threads in Java ist **synchronized**.
  Dieses Konzept ist in die Sprache Java „eingebaut".
  Methoden einer Klasse (static und nicht static) können mit diesem Schlüsselwort versehen werden.

```
class C
{
    public synchronized void sm(...) {...}
    public static synchronized void ssm(...) {...}
}
```

- Außerdem können **Anweisungsblocke mit synchronized geklammert** werden, wobei hierzu die Angabe einer Referenz auf ein Objekt nötig ist:

```
class C
{
    public void sm(...)
    {
        ...
        synchronized(objReference)
        {
            ...
        }
        ...
    }
}
```

Wann setzt man **synchronized** sinnvollerweise und im ersten Ansatz ein?

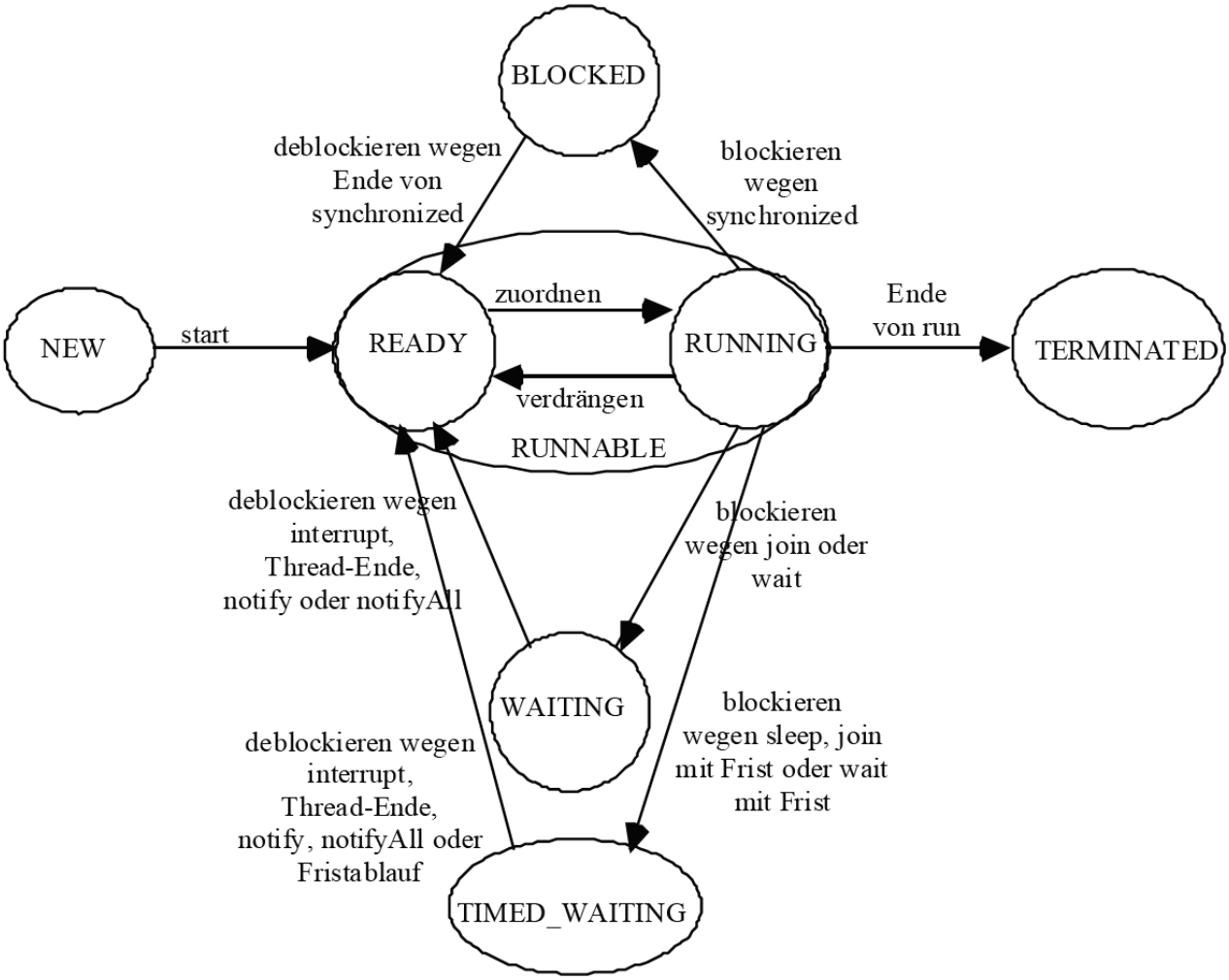
Wenn von mehreren Threads gleichzeitig auf ein Objekt zugegriffen werden kann, wobei mindestens ein Thread den Zustand des Objekts ändert (das heißt dessen Attribute schreibt), dann müssen alle lesenden und schreibenden Zugriffe synchronisiert werden (z.B. mit **synchronized**).

Dabei gilt auch:

Der Zugriff auf gemeinsam benutzte Attribute durch mehrere Threads muss immer **synchronized** erfolgen, auch wenn es nur um das Lesen und Schreiben eines einfachen **int** - Attributs geht.

[1] Alle Texte und Bilder sinngemäß übernommen aus „Parallele und verteilte Anwendungen in Java" - Rainer Oechsele - 4 Auflage 2014

Die Wirkung von einigen Mechanismen auf Thread lassen sich zusammenfassend in einem Zustandsübergangsdiagramm darstellen:



Zunächst muss ein Thread-Objekt erzeugt werden. Damit läuft der Thread aber noch nicht. Der Thread befindet sich im Zustand NEW. Erst wenn die Methode start auf das Thread-Objekt angewendet wird, kann der Thread loslaufen. Das Loslaufen erfolgt aber in der Regel nicht sofort. Ein Thread gelangt deshalb durch start vom Zustand NEW erst in den Zustand READY. In diesem Zustand bleibt der Thread so lange, bis tatsächlich auf ihn umgeschaltet wird. Während der Thread dann tatsächlich ausgeführt wird, befindet er sich im Zustand RUNNING. Die Methode getState der Klasse Thread, mit der die Zustände eines Threads erfragt werden können, unterscheidet die hier eingeführten Zustände READY und RUNNING nicht; diese werden zu einem Zustand RUNNABLE zusammengefasst.

Der Zustand RUNNING kann auf fünf Arten verlassen werden: Erstens könnte die ausgeführte Run-Methode verlassen werden (z. B. durch return oder eine nicht behandelte Ausnahme). In diesem Fall gelangt der Thread in den Zustand TERMINATED. Zweitens könnte der Thread genügend lange ausgeführt worden sein, so dass entschieden wird, auf einen anderen Thread umzuschalten. In diesem Fall wird der Thread verdrängt und wechselt wieder in den Zustand READY (bleibt aber somit RUNNABLE). Die Gründe Nr. 3, 4 und 5 für das Verlassen des RUNNING-Zustands sind, dass der Thread nicht mehr weiterlaufen kann und sich blockiert. Grund Nr. 3 ist, dass ein Thread eine Synchronized-Methode bzw. einen Synchronized-Block betreten will (z. B. auch nach dem Benachrichtigtwerden durch notify

oder notifyAll in der Wait-Methode), das betreffende Objekt aber momentan gesperrt ist. In diesem Fall gelangt er in den Zustand BLOCKED, aus dem er wieder in den RUNNABLE-Zustand (genauer in den READY-Zustand) wechselt, wenn die Sperre des betreffenden Objekts freigegeben wird und von diesem Thread dann gesetzt werden kann. Der Grund Nr. 4, warum ein Thread nicht mehr weiterlaufen kann, besteht darin, dass der Thread durch Aufruf der Methoden wait oder join in einen Wartezustand gelangt. Dieser Zustand wird WAITING genannt. Aus diesem Zustand kehrt der Thread in den RUNNABLE- bzw. READY-Zustand zurück, indem ein anderer Thread die Methode notify oder notifyAll aufruft, oder indem der Thread, auf dessen Ende mit join gewartet wird, terminiert. Eine weitere Möglichkeit für einen Thread, den Zustand WAITING zu verlassen, ist die Anwendung der Methode interrupt auf diesen Thread von einem anderen Thread aus. In diesem Fall wird wait oder join durch eine InterruptedException unterbrochen. Der fünfte Grund für eine Blockierung eines Threads ist der Aufruf von wait oder join mit Angabe einer maximalen Wartezeit oder durch Aufruf von sleep. In diesem Fall gelangt der Thread in den Zustand TIMED_WAITING. Gründe für das Verlassen dieses Zustands sind dieselben wie diejenigen für das Verlassen des Zustands WAITING. Hinzu kommt jetzt als Grund noch der Ablauf der Wartezeit.

In jedem Fall bewirkt das Verlassen des Zustands RUNNING durch einen Thread einen Übergang von READY nach RUNNING für einen anderen Thread.

Befindet sich ein Thread in einem der drei Zustände WAITING, TIMED_WAITING oder BLOCKED, kann nicht auf ihn umgeschaltet werden. Das heißt, wenn ein Thread blockiert ist, wird er nie rechnend und verbraucht keine Rechenzeit. Wir haben schon gesehen, dass dies eine sehr effiziente Art des Wartens auf ein Ereignis ist. Sobald der Grund für die Blockade eines Threads nicht mehr besteht, gelangt er wieder in den Zustand READY. Wie schon beim Starten bewirkt das Deblockieren eines Threads also nicht sein unmittelbares Weiterlaufen. Der Thread gelangt erst in den Zustand READY und muss sich gedulden, bis wieder auf ihn umgeschaltet wird (d. h. bis er wieder in den Zustand RUNNING gelangt).

Mit diesen Konzepten wurden in diesem Kapitel und werden im folgenden Kapitel eine Reihe von Anwendungen entwickelt. Dabei gab es immer *aktive* und *passive Klassen* bzw. *Objekte*. Aktive Klassen sind die Thread-Klassen bzw. die Klassen, welche die Runnable-Schnittstelle implementieren. Passive Klassen sind solche, deren Objekte von mehreren Threads benutzt werden. In **Tabelle 2.4** sind einige der Anwendungen nach diesem Schema noch einmal zusammenfassend dargestellt.

Tabelle 2.4 Zusammenfassung einiger Anwendungen dieses Kapitels

Aktive Klassen	Passive Klassen
Bankangestellte (Klasse Clerk)	Bank und Konten (Klassen Bank und Account)
Autos (Klasse Car)	Parkhaus (Klasse ParkingGarage)
Erzeuger und Verbraucher (Klassen Producer und Consumer)	Puffer (Klasse Buffer)

Bitte beachten Sie, dass die Synchronisation (synchronized, Aufrufe von wait, notify und notifyAll) in der Regel immer in den passiven Klassen realisiert wird.