block[Literatur 1](#_Toc97710684)

[Multithreading in Java 1](#_Toc97710685)

[Synchronisierungsmethoden 1](#_Toc97710686)

[Executors 1](#_Toc97710687)

# Literatur

* Friesen2015
* Rauber2006
* JavaSpec
* JavaDoc
* Hettel

# Multithreading in Java

* Java bietet eine einheitliche Schnittstelle für Threads des Betriebssystems. Die Java Runtime kümmert sich um die Aufrufe der betriebssystemspezifischen Funktionen zur Erzeugung eines Threads. [S. 3]
* Die Klasse Thread bietet die Methode start() mit der der Thread seine Ausführung beginnt. [S. 8]
* Die auszuführenden Anweisungen werden über die Implementierung des Interfaces Runnable an das Threadobject übergeben. [S.3f.]
* Alternativ kann von der Klasse Thread geerbt warden und dort die Methode run() überschrieben warden, da Thread selbst das Interface Runnable implementiert [S. 335]

## Synchronisierungsmethoden

* synchronized
  + Java bietet das Schlüsselwort synchronized das die blockierende Synchronisation bei dem Zugriff auf beliebige Objekte ermöglichtDas Schlüsselwort kann eingesetzt warden, um bei der Ausführung eines Blocks von Anweisungen ein den Zugriff auf ein bestimmtes Objekt zu Synchronisieren. Das Schlüssel Wort kann in der Signatur einer Methode verwendet warden, um automatisch das zugehörige Objekt (oder bei statischen Methoden die Klasse) für andere Threads zu sperren [S. 339 ff.]
  + Das wird dadurch ermöglicht, dass in Java jedem Object ein Monitor zugeordnet ist (Monitor sollte in Synchronisation erklärt warden). [S. 24]
* Volatile
  + Variablen können mit dem Schlüsselwort volatile ausgestattet werden, dadurch nutzen Threads keinen gecacheten Werte, sondern immer den aktuellen Wert im Hauptspeicher. Lesen und schreiben atomar [S. 30 ff.]
  + Einige Quellen warnen davor volatile für long und double zu verwenden da 32-bit Computer zwei Operationen für Lese- und Schreibvorgänge benötigen [S.34]
  + Die Java Spezifikation fordert, dass volatile long und volatile double immer atomar sind. Weiter warden Entwickler angewiesen diese als volatile zu deklarieren, falls sie von mehreren Threads genutzt warden.
  + Lesen und subsequentes Schreiben (x += 1) ist auch bei volatile nicht atomar.
* Atomics und Concurrent Collections
  + Atomares Update für primitive Werte häufig sinnvoll.
  + Java bietet in java.util.concurrenc.atomic Klassen, die atomare Aktionen auf einigen primitiven Datentypen erlauben: AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference [S. 131]
  + Möglichkeit zu threadsicheren Beschränkung auf eine Instanz, ohne globalen Zugriff.
  + Kann auch für Zähler genutzt werden
  + Byte, Short, Float und Double können über Integer und Long dargestellt werden
  + Concurrent Collections bieten verschiedene Collections die threadsafe sind ohne blockende Synchronisierung zu erzwingen [S. 125 ff.] Beispiele sind ConcurrentMap, ConcurrentQueue, BlockingConcurrentQueue – letzteres blockt wenn aus leerer Queue entnommen werden soll oder volle Queue befüllt.
  + Nutzung von CompareAndSwap [S. 133 f.]

## Executors

## CompletableFuture

* Future bietet keine Möglichkeit zur Komposition von einander abhängiger Tasks.
* Muss durch den Entwickler explizit selbst durchgeführt werden.
* Gefahr der Blockierung von Threads, die auf Werte anderer Tasks warten [239]
* Die Klasse CompletableFuture übernimmt die Verwaltung der Komposition und bietet dafür eine Push-API [S.240 ff.]
* Möglichkeit zur Verkettung, Aufspaltung, Zusammenführung und Auswahl von Tasks [S250 ff]