Threadsichere Datenstrukturen in Java

Jan Müller

12. Januar 2020

1 Einleitung

Threadsicherheit im Kontext von Datenstrukturen bedeutet, dass parallel stattfindende Lese- und Schreibzugriffe ohne Wettlaufsituationen und Inkonsistenzen erfolgen. In Java ist dies nur der Fall, wenn bestimmte Speichereigenschaften und Synchronisationsbedingungen berücksichtigt werden. Javas Standardbibliothek enthält deshalb eine umfangreiche Sammlung an Datenstrukturen sowie Schnittstellen, welche threadsichere Programmierung ermöglichen.

Diese Ausarbeitung stellt verschiedene Ansätze der Standardbibliothek vor und geht dabei auf Eigenschaften, Verwendung sowie Einsatzgebiete threadsicherer `Atomic`-Klassen und *Collections* ein. Diese benötigen weder zusätzliche Synchronisation noch kritische Abschnitte. Darüber hinaus werden Grundbausteine der Implementierung neuer threadsicherer Datenstrukturen erläutert. Dazu gehören das Schlüsselwort `volatile` und dessen Auswirkungen auf die Sichtbarkeit von Schreibzugriffen sowie die sperrfreien Synchronisationsmechanismen der *Variable Handles*. Letztere ermöglichen zudem eine threadsichere Verwendung externer Klassen.

Kapitel 2 erklärt das Schlüsselwort `volatile`. *Variable Handles* werden in Kapitel 3 behandelt. Kapitel 4 thematisiert die `Atomic`-Klassen. In Kapitel 5 werden threadsichere *Collections* der Java Standardbibliothek vorgestellt. Zuletzt fasst Kapitel 6 die Ausarbeitung zusammen.

2 Das Schlüsselwort volatile

In Java sind Schreibzugriffe nicht immer für andere Threads sichtbar, da sie aus Optimierungsgründen *threadlokal* zwischengespeichert werden können. Das Schlüsselwort `volatile` bildet als Modifikator für Attribute die Grundlage für threadsichere Datenstrukturen.

2.1 Eigenschaften

Zugriffe auf Attribute mit dem Modifikator `volatile` (siehe Quelltext 1) finden sequentiell statt. Ferner garantiert `volatile` die Sichtbarkeit von

Schreibzugriffen für alle Threads. Für diese Garantie wird *threadlokaler Cache* nach Schreibzugriffen *geflushed* und *Caching* von `volatile`-Attributen verhindert, weshalb `volatile`-Zugriffe in der Regel langsamer sind. Es gilt zu beachten, dass sich `volatile` bei nicht-primitiven Typen nur auf Referenzen, nicht deren Felder oder Elemente, bezieht.

Seit Java 5 werden Änderungen an regulären Variablen bei nachfolgenden `volatile`-Schreibzugriffen ebenfalls sichtbar gemacht. Andere Threads können solche Änderungen sehen, nachdem sie einen `volatile`-Lesezugriff durchführt haben. Ferner nimmt der Java Compiler keine Umsortierung solcher Zugriffe vor, wodurch eine *Happens-Before*-Beziehung[1] entsteht, aber auch eine weitere Optimierung entfällt.

Der `volatile`-Modifikator garantiert noch keine Threadsicherheit, sondern nur Sichtbarkeit von Zugriffen. In- und Dekrementoperationen sind bei numerischen `volatile`-Attributen beispielsweise nicht threadsicher, da sie mehrere Zugriffe benötigen, zwischen denen andere Threads Schreibzugriffe durchführen können. Um daraus resultierende Wettlaufsituationen zu vermeiden bedarf es zusätzlicher Synchronisationsmechanismen.

```
1 class MyClass {
2   static volatile int classAttribute = 0;
3   volatile int instanceAttribute = 0;
4 }
```

Quelltext 1: Klasse mit volatile Attributen

2.2 Weitere Speichermodi

Neben regulären und `volatile`-Zugriffen existieren weitere Speicherzugriffsmodi, die in dieser Ausarbeitung jedoch nicht betrachtet werden.[3] *Variable Handles* (siehe Kapitel 3) und `Atomic`-Klassen (siehe Kapitel 4) bieten *Getter* und *Setter* für alle Zugriffsmodi.

3 Variable Handles

Java 9 erweitert die Standardbibliothek um systemnahe *Variable Handles*.[4] Diese bieten sperrfreie Synchronisationsmechanismen, die zusammen mit der Sichtbarkeitsgarantie von `volatile`-Zugriffen performante threadsichere Datenstrukturen ermöglichen. Ferner können *Variable Handles* für threadsichere Zugriffe auf reguläre Variablen externer Klassen verwendet werden. Somit ist eine threadsichere Weiterverwendung von bestehendem *Code* möglich.

Eine `VarHandle`-Instanz ist eine getypte Referenz auf ein Attribut oder Arrayelemente. Über sie kann atomar und mit verschiedenen Speicherzugriffsmodi auf Variablen zugegriffen werden. Methoden der Klasse `VarHandle`

sind nativ, das heißt ihre Implementierungen basieren nicht auf Java und sind plattformspezifisch.

3.1 Verwendung

`VarHandle`-Instanzen werden über statische Methoden erzeugt. Für optimale Laufzeitperformanz sollten `VarHandle`-Variablen als `static final` Attribute deklariert und in `static`-Blöcken initialisiert werden.[2]

`VarHandle` stellt Methoden für atomare *Compare-and-Set*-, Additions- und Bit-Operationen zur Verfügung. Eine Anwendung von Methoden auf inkompatible Typen, wie beispielsweise `getAndAdd()` bei nicht numerischen Variablen, führt zu Ausnahmen während der Laufzeit.

Variable Argumentlisten und Rückgabewerte von `VarHandle`-Methoden haben den Typ `Object`. Infolgedessen gibt es keine Typsicherheit zur Kompilierzeit und invalide Verwendung löst entsprechende Ausnahmen aus.

In Quelltext 2 ist ein threadsicherer Zähler auf `VarHandle`-Basis zu sehen. Die erzeugte `VarHandle`-Instanz (siehe Zeilen 11 und 12) wird zum threadsicheren Inkrementieren (siehe Zeile 22) und Lesen per `volatile`-Zugriff (siehe Zeile 20) verwendet.

```
1  import java.lang.invoke.MethodHandles;
2  import java.lang.invoke.VarHandle;
3
4  class Counter {
5    private static final VarHandle COUNTER;
6
7    static {
8      try {
9        // MethodHandles.privateLookupIn() für private Attribute in
             externen Klassen
10       // findStaticVarHandle() für statische Attribute
11       COUNTER = MethodHandles.lookup()
12         .findVarHandle(Counter.class, "counter", long.class);
13     } catch (NoSuchFieldException | IllegalAccessException e) {
14       throw new Error(e);
15     }
16   }
17
18   private long counter = 0;
19
20   long get() { return (long) COUNTER.getVolatile(this); }
21
22   void increment() { COUNTER.getAndAdd(this, 1); }
23 }
```

Quelltext 2: Threadsicherer Zähler mit Variable Handles

3.2 Verwendung für Arrays

`VarHandle`-Instanzen für Arrays können mit Hilfe der Klasse `MethodHandles` und ihrer statischen Methode `arrayElementVarHandle()` erzeugt werden.

Als einzigen Parameter erhält sie einen Array-Typ. Quelltext 3 zeigt Argumentübergaben bei `VarHandle`-Methoden für Arrays.

```
1 int[] array = new int[10];
2 // Erstellt VarHandle-Instanz für int-Arrays
3 VarHandle AVH = MethodHandles.arrayElementVarHandle(int[].class);
4 // Liest Wert an Index 4
5 int value = (int) AVH.getVolatile(array, 4);
6 // Erhöht Wert an Index 2 um 10
7 AVH.getAndAdd(array, 2, 10);
8 // Setzt Wert an Index 1 auf 2, falls er 4 ist
9 AVH.compareAndSet(array, 1, 4, 2);
```

Quelltext 3: Variable Handles für Arrays

3.3 Vorteile gegenüber sperrbasierter Synchronisation

Variable Handles haben dank ihrer systemnahen Methoden und nativen *Compare-and-Set*-Mechanismen einen geringeren *Overhead* als sperrbasierte Synchronisationsmechanismen, wie beispielsweise *Locks* oder `synchronized`-Blöcke. Zudem vermeiden sie *Deadlocks* gänzlich, wodurch das Implementieren von Algorithmen vereinfacht wird.

4 Atomic-Datenstrukturen

Variable Handles (siehe Kapitel 3) sind umständlich und auf Grund fehlender Typsicherheit zur Kompilierzeit fehleranfällig. Javas Standardbibliothek stellt mit dem Paket `java.util.concurrent.atomic` deshalb `Atomic`-Datenstrukturen zur Verfügung, die einen Teil der `VarHandle`-Funktionalität, mit Typsicherheit zur Kompilierzeit und weniger *Boilerplate*, zugänglich machen.[1] Es existieren `Atomic`-Klassen für Variablen und Arrays. Im Gegensatz zu *Variable Handles* sind sie für höhere Abstraktionsebenen geeignet.

4.1 Verwendung

`AtomicBoolean`, `AtomicLong`, `AtomicInteger` und `AtomicReference`[2] bieten einen Teil der `VarHandle`-Funktionalität (siehe Abschnitt 3.1) und verwalten jeweils eine interne Variable entsprechenden Typs. Es ist zu beachten, dass `get()` und `set()`, anders als bei `VarHandle`, `volatile`-Zugriffe sind. Atomare Operationen sind über Methoden wie `compareAndSet()`, `getAndSet()` oder `getAndUpdate()` möglich. `AtomicLong` und `AtomicInteger` besitzen darüber hinaus Methoden für atomares Addieren, Inkrementieren und Dekrementieren. Quelltext 4 zeigt eine Verwendung von `AtomicLong`.

[1] `AtomicLong` und `AtomicInteger` basieren auf der Klasse `Unsafe` mit vergleichbaren Mechanismen (Stand OpenJDK 13.0.1.9).

[2] `AtomicReference` ist eine generische Klasse für beliebige Referenzen.

```
1  // Erzeugt AtomicLong-Instanz mit Startwert 1
2  AtomicLong atomicLong = new AtomicLong(1);
3  // Inkrementiert Wert analog zum Postinkrementoperator
4  atomicLong.getAndIncrement();
5  // Setzt Wert auf 3, falls er 2 ist
6  atomicLong.compareAndSet(2, 3);
7  // Verdoppelt Wert und gibt Ergebnis zurück
8  atomicLong.updateAndGet(oldValue -> 2 * oldValue);
9  // Liest Wert mit volatile-Zugriff
10 long value = atomicLong.get();
```

Quelltext 4: Verwendung von AtomicLong

4.2 Verwendung von Atomic-Arrays

Eine threadsichere Verwendung von Arrays wird durch `AtomicLongArray`, `AtomicIntegerArray` und `AtomicReferenceArray` ermöglicht. Alle Methoden dieser Klassen, mit Ausnahme der Konstruktoren und `length()`, benötigen einen zusätzlichen Parameter, der den Index von Zugriffen angibt. Ihren Konstruktoren muss entweder die Länge interner Arrays oder ein Array entsprechenden Typs übergeben werden, welches anschließend geklont wird. Analog zu numerischen `Atomic`-Klassen aus Abschnitt 4.1 verfügen auch `AtomicLongArray` und `AtomicIntegerArray` über Methoden für numerische atomare Operationen.

4.3 Alternativen

`LongAdder` und `LongAccumulator` sowie äquivalente Klassen für den Typ `double` sind hochperformante Alternativen zu den numerischen `Atomic`-Klassen. Anstelle einzelner Felder verwenden sie Arrays von *Zellen*, die jeweils atomare Zugriffe auf `VarHandle`-Basis unterstützen. Schreibzugriffe werden auf Zellen verteilt, um ihre Operationsdauer zu reduzieren. Dementsprechend sind diese Klassen performanter als ihre `Atomic`-Alternativen[3], jedoch mit eingeschränktem Methodenumfang. Es existieren unter anderem keine Methoden für verschiedene Zugriffsmodi und Aktualisierung von Werten analog zu `updateAndGet()`. Ebenfalls ist zu beachten, dass Methoden zur Ermittlung von Werten, also `sum()` bei `Adder`- sowie `get()` und `getThenReset()` bei `Accumulator`-Klassen, mit jedem Aufruf über alle Zellen iterieren und parallel stattfindende Schreibzugriffe nicht berücksichtigen.

5 Threadsichere Collections

Collections aus dem Paket `java.util` sind im Allgemeinen nicht threadsicher. Ihre Algorithmen garantieren keine konsistenten Datenbestände und vorhersehbares Verhalten, sobald Threads parallel Elemente hinzufügen oder

[3] Für den Typ `double` muss `AtomicReference` mit *Boxing* verwendet werden.

entfernen. Das Paket `java.util.concurrent` enthält deshalb threadsichere Implementierungen und Erweiterungen der Java *Collections*. Methodenumfang sowie Verwendung threadsicherer und nicht-threadsicherer *Collections* ähneln sich entsprechend. Es gilt jedoch einige Besonderheiten und Eigenschaften zu beachten, die im Folgenden erläutert werden.

5.1 Blockierende und nicht-blockierende Methoden

Threadsichere *Collections* weisen blockierendes und nicht-blockierendes Verhalten auf. Rufende Threads verweilen in blockierenden Methoden, bis diese ihre jeweilige Aktion ausführen können. Nicht-blockierende Methoden verfolgen einen anderen Ansatz. Ist ihre Ausführung zum Zeitpunkt des Aufrufs nicht umsetzbar, werden implementierungsabhängig *Exceptions* geworfen oder `null`- beziehungsweise `false`-Werte zurückgegeben.

5.2 Konzepte

Auf Grund des großen Methodenumfangs threadsicherer *Collections* werden an dieser Stelle nur ihre verschiedenen Konzepte vorgestellt und die Klassen nicht im Detail betrachtet. Auch auf die Verwendung von *Collections* im Allgemeinen wird nicht näher eingegangen. Informationen zu spezifischen *Collections* können der ausführlichen Java-Dokumentation entnommen werden. Im Folgenden werden die threadsicheren *Collections* der Standardbibliothek, nach ihren Hauptkonzepten gruppiert, vorgestellt. Besondere Konzepte werden separat betrachtet.

5.2.1 Blocking

`LinkedBlockingQueue`, `PriorityBlockingQueue` und `ArrayBlockingQueue` sowie `LinkedBlockingDeque` sind threadsichere Implementierungen des *Interfaces* `BlockingQueue`. Zugriffe werden mit Hilfe von `ReentrantLock` synchronisiert. `put`-Methoden blockieren, bis Warteschlangen Platz für hinzuzufügende Elemente haben. Aufrufer von `take`-Methoden sind blockiert, bis ein Element entnommen werden kann. Für nicht-blockierendes Verhalten sowie Aufrufe mit *Timeouts* stehen `poll`- und `offer`-Warteschlangenmethoden zur Verfügung. Diese Klassen eignen sich dementsprechend für Algorithmen, bei denen blockierendes Hinzufügen oder Entnehmen von Elementen aus Warteschlangen benötigt wird.

5.2.2 Concurrent

Die Threadsicherheit von `Concurrent`-Klassen basiert auf *Variable Handles* und deren *Compare-and-Set*-Mechanismen[4]. Dadurch wird der von an-

[4] `ConcurrentHashMap` verwendet die Klasse `Unsafe` mit vergleichbaren Mechanismen.

deren Synchronisationsmechanismen verursachte *Overhead* vermieden, jedoch entfällt auch blockierendes Verhalten. Bezüglich ihrer Verwendung unterscheiden sich reguläre *Collections* und ihre entsprechenden `Concurrent`-Varianten deshalb nur in ihrer Threadsicherheit. Neben *Compare-and-Set*-Mechanismen verwenden `Concurrent`-Klassen verschiedene Algorithmen, um auch bei vielen parallelen Zugriffen performant zu bleiben.

`ConcurrentLinkedQueue` und `ConcurrentLinkedDeque` sind für Algorithmen mit Warteschlangen geeignet, welche kein blockierendes Verhalten benötigen und minimalen *Overhead* voraussetzen.

`ConcurrentSkipListMap` sowie `ConcurrentSkipListSet` sind navigierbare sortierte *Collections*. Über ihre Sortierungskriterien können sie geordnete `SubMaps` beziehungsweise Teilmengen erzeugen.

Wird Navigierbarkeit nicht vorausgesetzt, kann `ConcurrentHashMap` verwendet werden. Diese Klasse verfügt zudem über Methoden, welche den `ForkJoinPool` verwenden, um Operationen zu parallelisieren. `forEach()`, `search()`, `reduce()` und weitere Varianten dieser Methoden lassen sich so threadsicher und performant ausführen.

5.2.3 CopyOnWrite

`CopyOnWriteArrayList` und `CopyOnWriteArraySet` führen Schreibzugriffe auf einer Kopie ihrer internen Arrays aus. Bisherige Arrays werden anschließend mit erzeugten Kopien ausgetauscht. Schreibzugriffe sind kritische Abschnitte und werden von `synchronized`-Blöcken geschützt. Das Kopieren von Arrays ist aufwendig, weshalb sich beide Klassen nicht für Programme mit vielen Schreibzugriffen eignen. Ihre Iteratoren verwenden nur das Array, welches bei ihrer Instanziierung vorlag. Dementsprechend sind nebenläufige Veränderungen der Iteratoren ausgeschlossen. `CopyOnWrite`-Klassen eignen sich deshalb für Programme, die wenig Daten einfügen, aber oft über diese iterieren müssen.

5.2.4 Delay

Die Klasse `DelayQueue` implementiert `BlockingQueue` und ermöglicht Zugriffe auf ihre Elemente erst nachdem deren *Delay* abgelaufen ist. Ihre generische Typen müssen das *Interface* `Delayed` implementieren. Dieses erweitert `Comparable<Delayed>` und verfügt über eine Methode, die eine Zeiteinheit erhält und das verbleibende *Delay* in Form eines `long`-Wertes zurückgibt. Elemente werden aufsteigend nach ihrem Ablaufdatum entnommen. Sollte sich kein Element mit abgelaufenem *Delay* in der Warteschlange befinden, werden lesende Zugriffe blockiert, bis ein solches Element verfügbar ist. Analog zu anderen blockierenden *Collections* kann bei diesen Methoden ein *Timeout* definiert werden. Zur Sicherung kritischer Abschnitte wird `ReentrantLock` verwendet.

5.2.5 Synchronous

`SynchronousQueue`, eine Implementierung von `BlockingQueue`, ermöglicht synchrone Verwendung einer Warteschlange ohne Kapazität. „Einfügen“ erfolgt nur dann, wenn ein Abnehmer bereits auf ein Element wartet. Analog dazu ist „Entnehmen“ von Elementen nur möglich, wenn ein anderer Thread am „Einfügen“ ist. Mit `put()` und `take()` findet dies blockierend statt. `poll()` und `offer()` blockieren nur, wenn ein *Timeout* festgelegt wird. Im Kontext anderer Methoden fungiert `SynchronousQueue` als leere *Collection*. Des Weiteren legt sie keine Wartereihenfolge für Zugriffe fest, kann jedoch als fair initialisiert werden und verwendet dann das *FIFO*-Prinzip. Die Threadsicherheit von `SynchronousQueue` basiert auf *Variable Handles*. Ihr blockierendes Verhalten entsteht durch Warte-Algorithmen.

5.2.6 Transfer

Ihrem Name entsprechend eignen sich *Transfer-Collections* zum Datenaustausch zwischen Threads, also beispielsweise für Erzeuger-Verbraucher-Probleme. Seit Java 7 steht dafür das Interface `TransferQueue`, eine Erweiterung von `BlockingQueue`, und dessen Implementierung `LinkedTransferQueue` zur Verfügung. `transfer()` blockiert Erzeuger, bis übergebene Elemente von Verbrauchern empfangen wurden. Alternativ kann mit `tryTransfer()` ein *Timeout* definiert oder ein sofortiges Einfügen versucht werden. Informationen über wartende Verbraucher können mit `hasWaitingConsumer()` und `getWaitingConsumerCount()` abgefragt werden. Die Threadsicherheit aller Operationen wird durch Verwendung von *Variable Handles* garantiert. Warte-Algorithmen ermöglichen blockierendes Verhalten.

5.3 Synchronized-Collections

Über statische Methoden der Klasse `Collections` aus dem Paket `java.util` können verschiedene threadsichere *Wrapper* für `Set`-, `List`- und `Map`-Implementierungen erzeugt werden. Dafür wird eine Instanz der entsprechenden *Collection* den Methoden übergeben. Die Threadsicherheit der *Wrapper* wird dadurch gewährleistet, dass alle relevanten Methoden innerhalb eines `synchronized`-Blocks auf ihre interne *Collection* zugreifen. Dementsprechend groß ist der *Overhead* dieser *Wrapper*. Ferner sind ihre Iteratoren und *Streams* nicht threadsicher.

5.4 Anwendungsbeispiel

Basierend auf den *Interfaces* `ConcurrentMap` und `BlockingQueue` sowie der Klasse `ConcurrentLinkedQueue` wird in Quelltext 5 ein threadsicherer *Request Handler* implementiert, wie er in einem *Webservice* zum Einsatz kommen kann. Dieser wartet auf neue Anfragen an der blockierenden Warte-

schlange `requestQueue` (siehe Zeilen 13 und 25). Anschließend wird die threadsichere *Map* `resultMap` zum Generieren oder Lesen von Antworten verwendet (siehe Zeilen 15 und 27). Zuletzt fügt er abgearbeitete Anfragen, an dieser Stelle um berechnete Antworten erweitert, nicht-blockierend in `finishedQueue` ein (siehe Zeilen 14, 28 und 29).

```
1  import java.util.concurrent.*;
2
3  class Request {
4    final String query;
5    final String origin;
6    private Response response = null;
7    // query und origin Konstruktor, response-Getter/Setter
8  }
9
10 class Response { /* Implementierung */ }
11
12 class RequestHandler implements Runnable {
13   private BlockingQueue<Request> requestQueue;
14   private ConcurrentLinkedQueue<Request> finishedQueue;
15   private ConcurrentMap<String, Response> resultMap;
16   // All-Args-Constructor
17
18   private Response getResponse(final String query) {
19     return new Response(); // Vereinfachtes Beispiel
20   }
21
22   @Override public void run() {
23     try {
24       while (!Thread.currentThread().isInterrupted()) {
25         Request req = requestQueue.take(); // Blockierend
26         Response res =
27           resultMap.computeIfAbsent(req.query, this::getResponse);
28         req.setResponse(res);
29         finishedQueue.add(req); // Nicht-blockierend
30       }
31     } catch (InterruptedException e) { /* Ausnahmebehandlung */ }
32   }
33 }
```

Quelltext 5: Request Handler mit Ergebnisspeicherung

6 Zusammenfassung

Das Java Speichermodell bildet die Grundlage für threadsichere Datenstrukturen, da es unter anderem Sichtbarkeit und Reihenfolge von Zugriffen festlegt. `volatile`-Zugriffe finden sequentiell statt und sind für alle Threads sichtbar. Um Wettlaufsituationen zu vermeiden müssen Zugriffe zudem synchronisiert werden. Gängige Mittel dafür sind `Locks`, unter anderem in Form von `synchronized`-Blöcken, und *Compare-and-Set*-Mechanismen. Die Java Standardbibliothek bietet threadsichere Datenstrukturen für verschiedene Einsatzzwecke. Von `Atomic`-Klassen für einzelne Werte oder Referenzen bis

hin zu *Collections* mit unterschiedlichem Verhalten stehen viele Implementierungen zur Verfügung. Sollten threadsichere Datenstrukturen benötigt werden, welche nicht in der Standardbibliothek enthalten sind, können *Variable Handles* verwendet werden, um diese Strukturen zu implementieren. Darüber hinaus ermöglichen *Variable Handles* eine threadsichere Verwendung regulärer Attribute und Arrays.

Referenzen

[1] Javier Fernández González. *Java 9 Concurrency Cookbook, Second Edition*. Packt Publishing, 24. Apr. 2017. 594 S. ISBN: 178712441X.

[2] Doug Lea. *JEP 193: Variable Handles*. Aug. 2017. URL: `https://openjdk.java.net/jeps/193` (besucht am 25. 11. 2019).

[3] Doug Lea. *Using JDK 9 Memory Order Modes*. Nov. 2019. URL: `http://gee.cs.oswego.edu/dl/html/j9mm.html` (besucht am 25. 11. 2019).

[4] Oracle Corporation. *Class VarHandle*. URL: `https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/invoke/VarHandle.html` (besucht am 25. 11. 2019).

[5] Oracle Corporation. *Package java.util.concurrent*. URL: `https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/package-summary.html` (besucht am 27. 11. 2019).

[6] Oracle Corporation. *Package java.util.concurrent.atomic*. URL: `https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/concurrent/atomic/package-summary.html` (besucht am 25. 11. 2019).