

Algorithmen & Datenstrukturen

Weiterführende Konzepte

Von der Zukunft, Atomen und Containern

Roger Diehl



Inhalt

- Future- und Callable-Schnittstellen
- Callables und Executors
- Umgang mit Exceptions
- Atomic-Variablen
- Thread-sichere Container
- Blocking Queues
- Zusammenfassung

Lernziele

- Sie kennen die Abstraktion der **Executor**-Services, **Callable<V>** und **Future<V>** und können diese nutzen.
- Sie wissen wie Sie mit Ausnahmen in nebenläufig ausgeführten Tasks umgehen müssen.
- Sie können Thread-sichere Zugriffe auf einzelne Variablen von elementaren Datentypen nachvollziehen, modifizieren und erstellen.
- Sie kennen die Wrapper Klassen um herkömmliche Datenstrukturen Thread-sicher zu gestalten.
- Sie können für die asynchrone Kommunikation die entsprechende Queue-Klasse auswählen und diese nutzen.

Future- und Callable-Schnittstellen

Runnable-Aufgabe mit Rückgabe

- Eine **Runnable**-Interface implementierte, nebenläufig auszuführende Aufgabe besitzt kein Rückgabewert.
- Soll eine Aufgabe einen Wert zurück liefern, geht das nur über ein spezielles «Rückgabe» Attribut.

```
public final class RunnableWithReturn<T> implements Runnable {  
    private T returnValue;  
    private volatile Thread self;  
    // ...  
  
    @Override  
    public void run() {  
        self = Thread.currentThread();  
        // ...  
    }  
    public T get() throws InterruptedException {  
        self.join();  
        return returnValue;  
    }  
}
```

Codeskizze

Attribut für Rückgabewert.

Berechnung und Speicherung des Resultats ins Rückgabe Attribut.

Blockierende Abfrage des Rückgabewertes.

Das «neue» Runnable: Callable

- siehe OOP Input → **O15_IP_Nebenläufigkeit; «Executor API»**
- Das Interface **Callable** kennt nur eine **call()**-Methode.
- Ein **Callable** kann nun einen Rückgabewert haben!
 - Typ des Rückgabewertes wird per Typparameter festgelegt.

```
/**
 * A task that returns a result and may throw an exception.
 */
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable
     * to do so.
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

Quelle: java.util.concurrent Library

Callable-Aufgabe mit Rückgabe

- Um den Rückgabewert einer **Callable**-Interface implementierte, Aufgabe zu bekommen, benötigt man die Hilfe von **FutureTask**.
- **FutureTask** implementiert **Runnable** und kann somit einem Thread zur Ausführung übergeben werden.
- Über die **get**-Methode erhält man Zugriff auf das Ergebnis.

```
final Callable<Integer> callable = () -> {  
    int sum = 0;  
    for (int i = 1; i <= 10000; i++) {  
        sum += i;  
    }  
    return sum;  
};  
final FutureTask<Integer> futureTask = new FutureTask<>(callable);  
new Thread(futureTask, "Future Task Thread").start();  
  
LOG.info("Summe: " + futureTask.get());
```

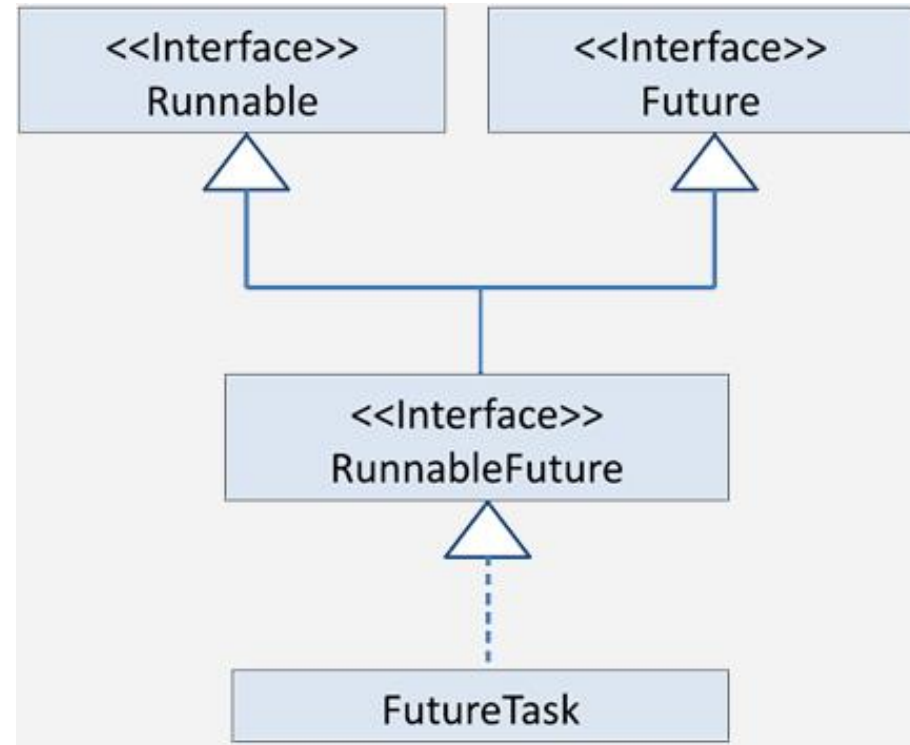
Codeskizze

Callable-Objekt mit Rückgabewert.

Blockierende Abfrage des Rückgabewertes.

Vererbungshierarchie von FutureTask

- **FutureTask** implementiert neben dem **Runnable**- auch das **Future**-Interface.
- Mit dem Interface **Future** kann die Ergebniserückgabe einer asynchronen Berechnung einfach und einheitlich realisiert werden.
- Mit Hilfe von **Future** kann neben dem Ergebnis auch der Status der Berechnung abgefragt werden.



Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

Callables und Executors

Callable, Future und ExecutorService

- Ein **Callable** wird einem **ExecutorService** über die Methode **submit** zur Ausführung übergeben.
- Die Methode **submit** liefert ein **Future**-Objekt zurück.
- Über das **Future**-Objekt kann die Rückgabe erfragt werden.

```
final Callable<Integer> callable = () -> {  
    int sum = 0;  
    for (int i = 1; i <= 10000; i++) {  
        sum += i;  
    }  
    return sum;  
};  
final ExecutorService executor = Executors.newCachedThreadPool();  
  
final Future<Integer> future = executor.submit(callable);  
LOG.info("Summe: " + future.get());  
  
executor.shutdown();
```

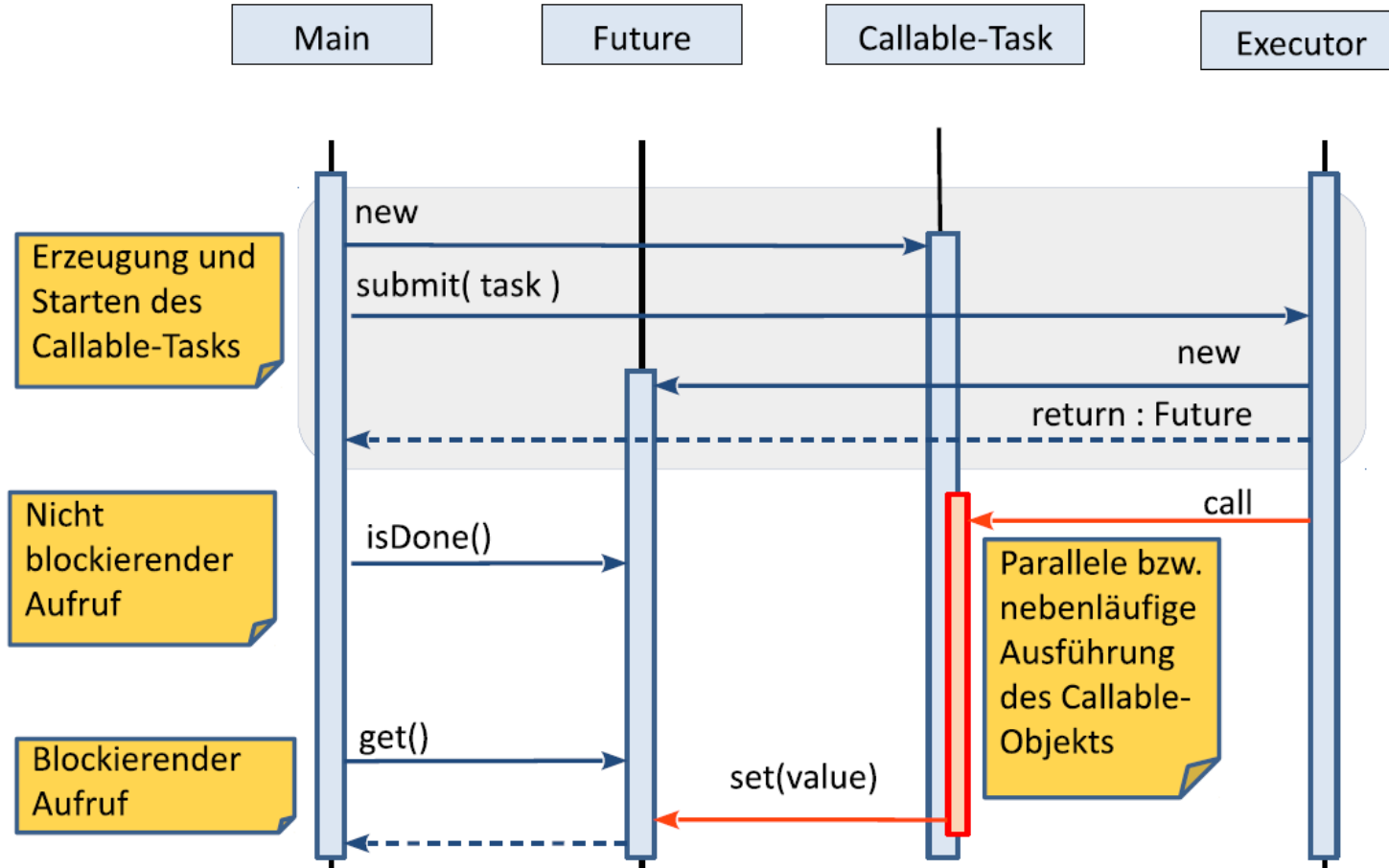
Codeskizze

Callable Rückgabewert.

Executor erzeugen und Callable-Objekt aufgeben.

Blockierende Abfrage des Rückgabewertes via Future-Objekt.

Funktionsweise des Future-Patterns



Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

Future<V>

- **Future<V>** bietet auch **get(long timeout, TimeUnit unit)** an, bei welcher der Aufrufer eine maximale Wartezeit angeben kann.
 - Ist das Ergebnis nach der vorgegebenen Zeit nicht verfügbar, wird eine **TimeoutException** geworfen.
- Mit **isDone** kann der Bearbeitungsstatus abgefragt werden.
- Zum Abbrechen kann **cancel(boolean mayInterruptIfRunning)** benutzt werden.
 - Ist der Task noch nicht gestartet, wird er nicht ausgeführt.
 - Befindet er sich mitten in der Abarbeitung, kann ein Interrupt gesendet werden.
 - Achtung: Falls das Argument **true** ist, muss der Task so implementiert sein, dass er den Interrupt berücksichtigt.
- Mit **isCancelled** kann geprüft werden, ob der Task abgebrochen wurde.

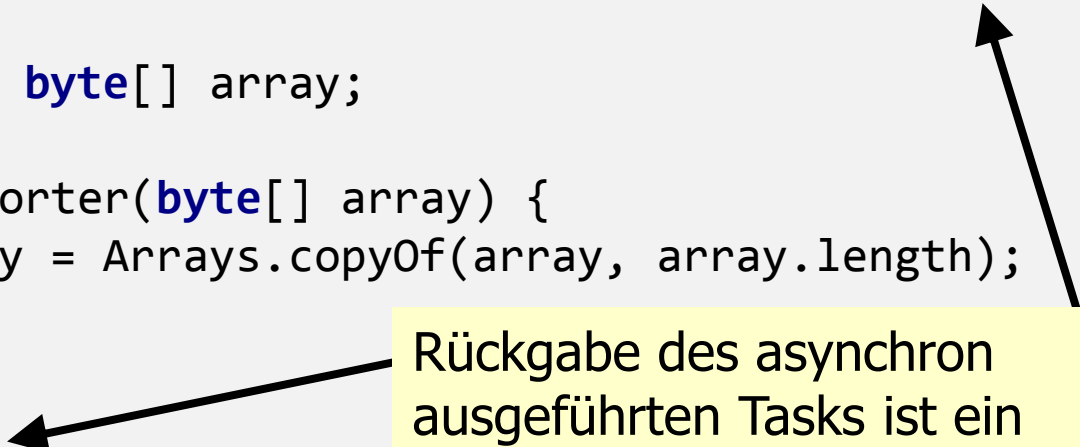
Aufgaben an einen `ExecutorService` übergeben

- `Future<?> submit(Runnable task)`: Das zurückgegebene `Future`-Objekt wird verwendet, um `isDone`, `cancel` und `isCancelled` aufzurufen.
 - Der `get`-Aufruf liefert bei Fertigstellung nur den Wert `null`.
- `Future<T> submit(Runnable task, T result)`: Hier liefert `get`-Methode des `Future`-Objekts das vorgegebene `result`-Objekt als Ergebnis zurück.
- `Future<T> submit(Callable<T> task)`: In dieser Version wird ein `Future`-Objekt zurückgeliefert, mit dem das Ergebnis der Berechnung abgeholt werden kann.

Beispiel – Array sortieren über Callable

- Das Sortieren soll ein **Callable** nebenläufig übernehmen.
- Als Resultat wird das sortierte Array zurückgegeben.

```
public final class ArraySorter implements Callable<byte[]> {  
  
    private final byte[] array;  
  
    public ArraySorter(byte[] array) {  
        this.array = Arrays.copyOf(array, array.length);  
    }  
  
    @Override  
    public byte[] call() {  
        Arrays.sort(array);  
        return Arrays.copyOf(array, array.length);  
    }  
}
```



The diagram consists of two arrows. One arrow originates from the `Callable<byte[]>` interface in the code and points to a yellow text box. The other arrow originates from the `call()` method signature and points to the same yellow text box. Additionally, a cyan text box points to the `Arrays.copyOf` method call within the `call()` method.

Rückgabe des asynchron ausgeführten Tasks ist ein **byte-Array**.

Warum der Einsatz der Methode **Array.copyOf**?

Beispiel – Array-Sortier-Task aufgeben

- Das Programmiermuster ist immer das gleiche:
 1. Arbeit an den **ExecutorService** übergeben.
 2. Etwas anderes machen.
 3. Das Resultat abholen.

```
final byte[] array = new byte[16];  
new Random().nextBytes(array);
```

byte Array erzeugen und
mit Zufallszahlen füllen,
Callable-Objekt erzeugen.

```
final Callable<byte[]> callable = new ArraySorter(array);  
final ExecutorService executor = Executors.newCachedThreadPool();
```

1. **final** Future<byte[]> result = executor.submit(callable);
2. //...hier etwas anderes machen
3. **final** byte[] bs = result.get();

```
LOG.info("lowest value = "+bs[0]);  
LOG.info("highest value = "+bs[bs.length-1]);
```

Codeskizze

Umgang mit Exceptions

Exception-Handling bei execute

- Wie wird mit Fehlern umgegangen, die in nebenläufig ausgeführten Tasks auftreten?
- Beispiel: Ein Task mit einer «Division durch null» wird mit **execute** an einen Pool gesendet.

```
final ExecutorService executor = Executors.newCachedThreadPool();  
executor.execute(() -> System.out.println(1 / 0));
```

- Sobald der Task ausgeführt wird, wird eine Exception geworfen.

```
Exception in thread "pool-1-thread-1"  
java.lang.ArithmeticException: / by zero  
...
```

Exception-Handling bei submit

- Beim Aufruf **submit** wird nicht direkt eine Exception geworfen.

```
final ExecutorService executor = Executors.newCachedThreadPool();  
executor.submit(() -> System.out.println(1 / 0));
```

- Der Grund ist, dass beim Einsatz von **submit** jede nicht behandelte Ausnahme von **Runnable** oder **Callable** abgefangen wird.

```
public void run() {  
    Throwable thrown = null;  
    try {  
        while (!isInterrupted()) {  
            runTask(getTaskFromWorkQueue());  
        }  
    } catch (Throwable e) {  
        thrown = e;  
    } finally {  
        threadExited(this, thrown);  
    }  
}
```

Codeskizze

Exception-Handling bei get

- Erst bei einem Zugriff mit **get** wird die Ausnahme auf das zurückgegebene Future-Objekt ausgelöst.

```
final ExecutorService executor = Executors.newCachedThreadPool();
final Future<?> future =
    executor.submit(() -> System.out.println(1 / 0));
try {
    future.get();
} catch (InterruptedException | ExecutionException ex) {
    LOG.debug(ex);
}
```

Codeskizze

```
2017-03-23 09:45:42,908 INFO - java.util.concurrent.ExecutionException:
java.lang.ArithmeticException: / by zero
```

Exception im Task abfangen

- Alternativ kann man die Exception im Task abfangen und loggen.
- Damit der Aufrufende über die Ausnahme in Kenntnis gesetzt wird, sollte die **Exception** weitergegeben werden

```
final ExecutorService executor = Executors.newCachedThreadPool();
final Future<?> future = executor.submit(() -> {
    try {
        System.out.println(1 / 0);
    } catch (Exception ex) {
        LOG.debug(ex);
        throw ex;
    }
});
try {
    future.get();
} catch (InterruptedException | ExecutionException ex) {
    LOG.debug(ex);
}
```

Exception behandeln oder Loggen.

Exception weiter geben, damit get sie auslösen kann.

Codeskizze

Atomic-Variablen

Atomarer Zugriff

- Der lesende oder schreibende Zugriff auf Variablen eines primitiven Datentyps ist in Java atomar, d. h. nicht unterbrechbar.
 - Ausnahmen: **long** und **double**
- Die Zugriffe auf Referenzvariablen sind dagegen immer atomar, unabhängig davon, ob es sich um eine 32- oder 64-bit-JVM handelt.
- Werden die Variablen mit **volatile** gekennzeichnet, ist der Zugriff garantiert immer atomar, unabhängig vom Datentyp.
- **Aber:** Oft besteht eine Operation auf eine Variable aus mehreren Schritten, obwohl im Code dafür nur eine Anweisung angegeben ist.
 - Beispiel die Anweisung **counter++** besteht aus den Befehlen:
 1. Laden des Inhalts von **counter** in ein Register (Lesen).
 2. Registerinhalt wird inkrementiert.
 3. Das Ergebnis wird in **counter** geschrieben (Schreiben).

Beispiel – Thread-sicherer Zähler

- Wenn einfache Daten mithilfe von Locks geschützt werden, bedeutet dies immer einen Performance-Verlust.

```
public final class SynchronizedCounter {
```

```
    private int counter = 0;
```

```
    public synchronized void increment() {  
        counter++;  
    }
```

```
    public synchronized void decrement() {  
        counter--;  
    }
```

```
    public synchronized int get() {  
        return counter;  
    }
```

```
}
```

Codeskizze

Muss **synchronized** sein damit parallel zugreifende Threads immer den aktuellen Wert von **counter** bekommen.

Atomic-Variablen

- Das Sperren kann im Prinzip nicht vermieden werden.
- Seit Java 5 gibt es das Paket `java.util.concurrent.atomic` mit Kapselungen (Wrapper Klassen) für verschiedene Datentypen.
- Die gängigen Klassen sind `AtomicBoolean`, `AtomicInteger`, `AtomicLong` und `AtomicReference`.
- Mit Atomic-Variablen wird das «Locken» auf die Hardware delegiert.
 - Basierend auf dem `compareAndSet`-Mechanismus.
- Wenn zwei Threads auf verschiedenen Rechenkernen gleichzeitig auf eine Atomic-Variable zugreifen, wird ein Thread den Bus blockieren und der andere muss so lange auf die Freigabe warten.
 - Das bedeutet ein mögliches (sehr kurzes) Blockieren.

AtomicInteger – Compare-and-Set

- Zentral ist die Compare-and-Set-Operation. Mit ihr kann eine Variable atomar gelesen und verändert werden.
 - Die Operation benötigt hierzu drei Angaben:
 1. Eine Speicherstelle,
 2. den erwarteten, alten Wert und
 3. einen neuen Wert.
- **boolean compareAndSet(int expect, int update)**
 - Der Compare-and-Set-Befehl, angewandt auf den Wert des Objekts.
 - Wenn der Inhalt der Speicherzelle mit dem erwarteten, alten Wert übereinstimmt, wird der neue an die Speicherstelle geschrieben.
 - Stimmt der erwartete, alte Wert nicht überein, weil er zwischenzeitlich geändert hat, findet keine Modifikation statt.
 - Eine boolesche Rückgabe signalisiert, ob eine Änderung stattgefunden hat.

AtomicInteger – weitere Methoden

- **int addAndGet(int delta)**
 - Der Wert wird atomar um **delta** erhöht. Der neue Wert wird zurückgegeben.
- **int decrementAndGet()**
 - Der Inhalt wird atomar dekrementiert und der neue Wert wird zurückgegeben.
- **int incrementAndGet()**
 - Der Inhalt wird atomar inkrementiert und der neue Wert wird zurückgegeben.
- **int set(int newValue)**
 - Der Wert wird durch **newValue** ersetzt und der neue Wert wird zurückgegeben.
- **int get()**
 - Liefert den aktuellen Wert.
- Alle «Getter»-Methoden gibt es auch in der **getAnd**-Version.
 - d.h. der Wert welcher VOR der Operation bestand wird zurückgegeben.
- Die Methoden der anderen Atomic Wrapper Klassen sehen ähnlich aus.

Beispiel – Atomarer Zähler

- Thread-sicherer Zähler, ohne das aufwendige **synchronized** und somit ein Performance-Gewinn.

```
public final class AtomicCounter {
```

Codeskizze

```
    private final AtomicInteger counter = new AtomicInteger(0);
```

```
    public void increment() {  
        counter.incrementAndGet();  
    }
```

```
    public void decrement() {  
        counter.decrementAndGet();  
    }
```

```
    public int get() {  
        return counter.get();  
    }
```

```
}
```

Hier können auch **getAnd**-Methoden verwendet werden, weil der Rückgabewert nicht gelesen wird.

Komplexere Änderung

- Wir wollen einen Maximalwert in einer **AtomicLong**-Variablen speichern, der von verschiedenen Threads durch Aufruf einer **update**-Methode geändert werden kann.

```
private static final AtomicLong VALUE = new AtomicLong();
```

```
public static void update(final long newVal) {  
    VALUE.set(Math.max(VALUE.get(), newVal));  
}
```

Falsch!

Codeskizze

```
public static void update(final long newVal) {  
    long alt, neu;  
    do {  
        alt = VALUE.get();  
        neu = Math.max(alt, newVal);  
    } while (VALUE.compareAndSet(alt, neu) == false);  
}
```

Codeskizze

update muss aus mehreren Schritten bestehen, um atomarer Zugriff zu garantieren.

Komplexere Änderung in Java 8

- Um Iterationen zu vermeiden, stehen ab Java 8 in den Atomic-Klassen die Methoden `accumulateAndGet` und `getAndAccumulate` sowie `updateAndGet` und `getAndUpdate` zur Verfügung.

```
private static final AtomicLong VALUE = new AtomicLong();
```

```
public static void update(final long newVal) {  
    VALUE.accumulateAndGet(newVal, Math::max);
```

```
}    newVal wird in VALUE gespeichert, wenn Math::max wahr ist.
```

`mult` gibt den alten Wert von `VALUE` zurück.

```
public static long mult(final long operand) {  
    return VALUE.getAndUpdate((long a) -> operand * a);
```

```
}
```

`operand` wird mit dem Parameter `a` multipliziert und in `VALUE` gespeichert.
Beim Aufruf ist `VALUE` das Argument von `a`.

Codeskizze

Thread-sichere Container

Synchronisierte Collections

- Java besitzt seit seiner ersten Version die standardisierten Container **Vector**, **Stack**, **HashTable** und **Dictionary**.
- Die öffentlichen Methoden dieser Datenstrukturen sind durch **synchronized** geschützt, was jedoch in Singlethreaded-Anwendungen zu unnötigen Performance-Verlusten führt.
- **Bei den Containern des Collection-Frameworks hat man diesen Schutz weggelassen.**
 - Von nebenläufiger Bearbeitung ohne entsprechende Vorkehrungen ist abzuraten.
 - Aber: Es gibt synchronisierte Wrapper-Klassen.
- Für jedes Collection-Interface steht eine öffentliche Klassenmethode von **Collections** zur Verfügung, die eine entsprechend synchronisierte Containerfassade zurückliefert.

Beispiele – Synchronisierte Containerfassaden

```
List<BankAccount> list = new ArrayList<>();  
List<BankAccount> syncList = Collections.synchronizedList(list);  
  
Map<String, String> map = new HashMap<>();  
Map<String, String> syncMap = Collections.synchronizedMap(map);
```

Codeskizze

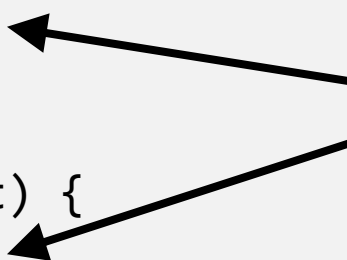
- Das Konzept ist gut geeignet für Datenstrukturen, auf die nur wenig zugegriffen wird.
- Es treten aber in der Praxis gelegentlich folgende Probleme auf:
 - Die Synchronisierung stellt einen Engpass dar, weil sie alle Zugriffe nur mit einem Monitor schützt.
 - Somit ist ein reines, paralleles Lesen nicht möglich.
 - Es kann leicht zu Inkonsistenzen kommen.

Inkonsistenzen bei Containern

- Hintereinander ausgeführte geschützte Methoden können jedoch keine Konsistenz garantieren, da nach jedem Aufruf ein Thread-Wechsel möglich ist.
- Beispiel: Die Datenstruktur wird während einer Iteration durch eine nebenläufige Operation verändert wird.
- Mit normalen for-Schleifen gerät die Datenstruktur sehr leicht in einen unentdeckten inkonsistenten Zustand.

```
for (int i = 0; i < syncList.size(); i++) {  
    // tu was mit syncList.get(i)  
}  
  
for (BankAccount account : syncList) {  
    // tu was mit account  
}
```

Hier sind Thread-Wechsel möglich.



Codeskizze

Iteratoren und Container

- Iteratoren prüfen bei jedem Schritt, ob eine mögliche nebenläufige Veränderung stattgefunden hat, und werfen ggf. eine **ConcurrentModificationException**.
 - Bei beabsichtigten Änderungen durch mehrere Threads ist dies allerdings nicht erwünscht.

```
Iterator<BankAccount> iterator = syncList.iterator();  
while (iterator.hasNext()) {  
    // tu was mit iterator.next()  
}
```

Hier ist, bei nebenläufige Veränderung, eine Exception möglich.

Codeskizze

Wirklich Thread-sichere Iteration

- Um Thread-Sicherheit wirklich zu gewährleisten, muss vor der Iteration der Container selbst geschützt werden.

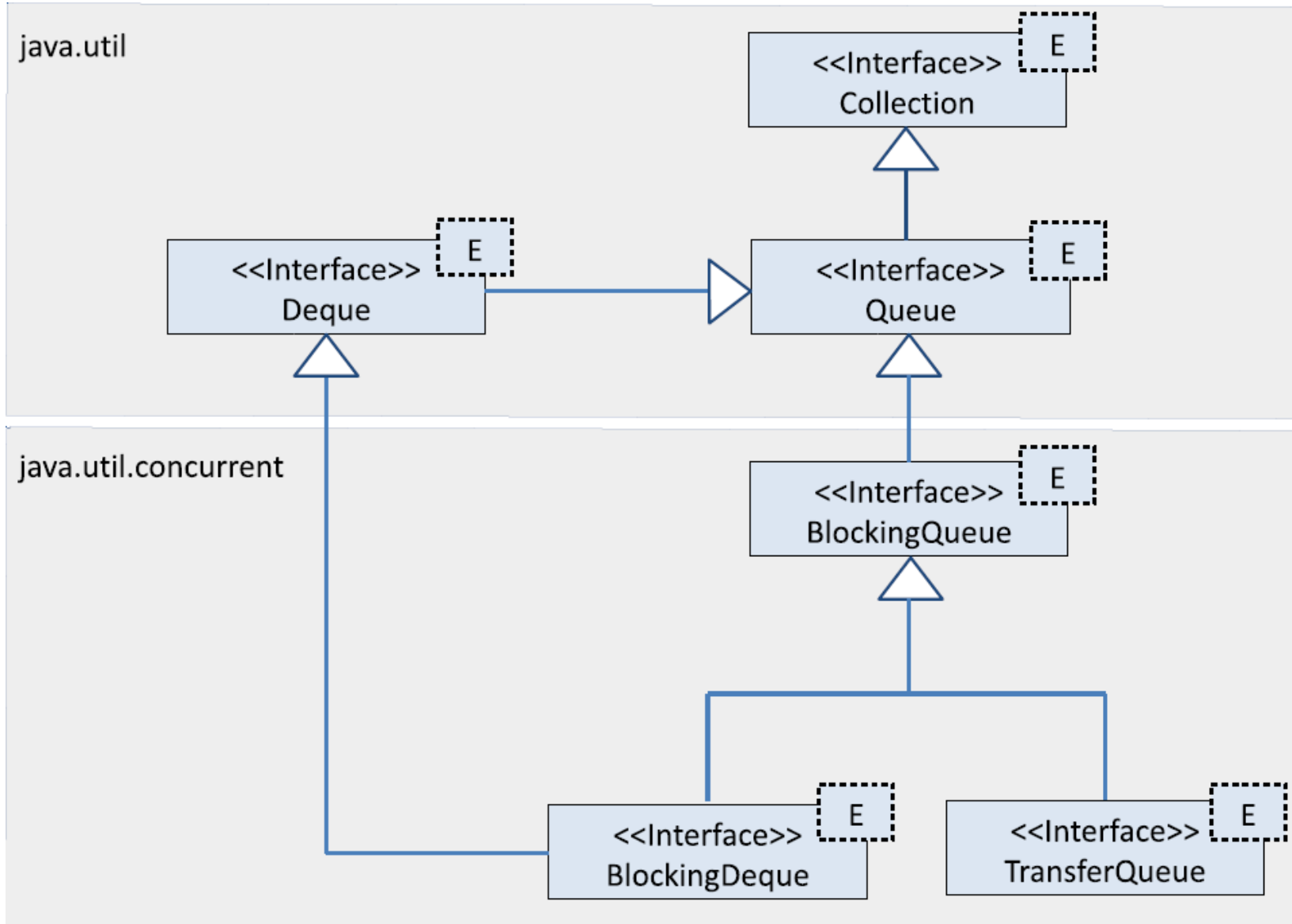
```
synchronized (syncList) {  
    for (int i = 0; i < syncList.size(); i++) {  
        // tu was mit syncList.get(i)  
    }  
}
```

Codeskizze

- Dadurch wird die Nebenläufigkeit stark eingeschränkt, da der Container für alle anderen Zugriffe gesperrt wird.

Blocking Queues (siehe Bounded Buffer)

BlockingQueue – Interface-Hierarchie

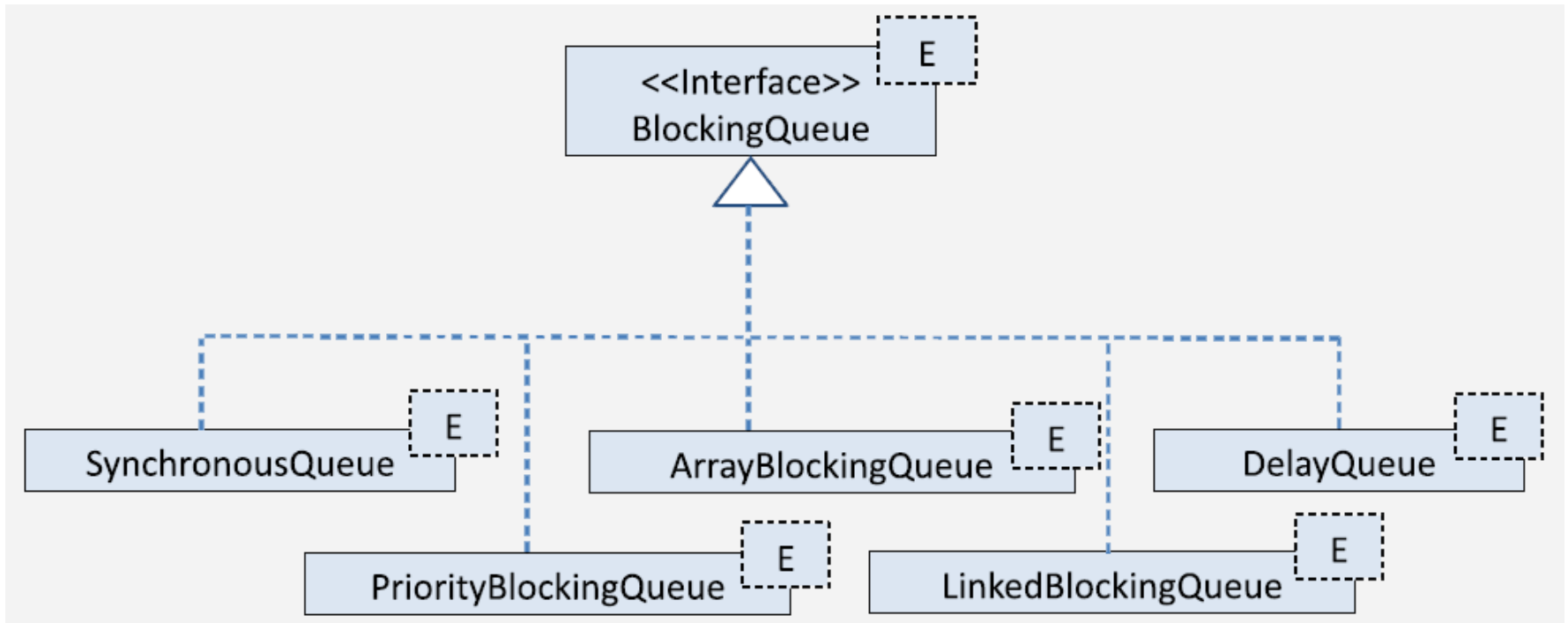


Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

BlockingQueue – Die wichtigsten Methoden

- **boolean offer(E e)**
- **boolean offer(E e, long timeout, TimeUnit unit)**
 - Fügt ein Element am Ende der Queue ein. Die Rückgabe gibt an, ob die Operation erfolgreich (**true**) war oder **false**, wenn nicht. Die Variante mit Timeout gibt ein **false** zurück, wenn die angegebene Zeit abgelaufen ist.
- **E poll()**
- **E poll(long timeout, TimeUnit unit)**
 - Entnimmt ein Element vom Anfang der Queue. Liefert **null**, falls kein Element vorhanden ist oder die angegebene Zeit abgelaufen ist.
- **void put(E e)**
 - Fügt ein Element am Ende in die Queue ein und wartet (blockierend) ggf., bis ein entsprechender Platz in der Queue vorhanden ist.
- **E take()**
 - Entnimmt ein Element vom Anfang der Queue und wartet (blockierend) ggf., bis ein Element vorhanden ist.

Implementierungen von BlockingQueue (Auswahl)



Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

- Mit Ausnahme von **PriorityBlockingQueue** und deren Ableitungen werden Elemente immer am Ende der Queue eingefügt und am Anfang der Queue entnommen (FIFO-Prinzip).

BlockingQueue – Konkrete Klassen

- **ArrayBlockingQueue<E>**

- ist eine Queue mit einer festen Grösse (Kapazität).

- **LinkedBlockingQueue<E>**

- existiert sowohl als kapazitätsbeschränkte als auch als unbeschränkte Queue.

- **DelayQueue<E>**

- kann nur Objekte aufnehmen, deren Klasse das Interface **Delayed** implementiert. Für die interne Organisation werden die Methoden **compareTo** und **getDelay** verwendet.

- **PriorityBlockingQueue<E>**

- sortiert mithilfe der **compareTo**-Methode bzw. mit dem explizit angegebenen **Comparator**-Objekt ihre verwalteten Elemente.

- **SynchronousQueue<E>**

- ist eine blockierende Queue, bei der die beteiligten Threads aufeinander warten müssen. **SynchronousQueue** hat keine Kapazität.

Praxistipp

- Das Interface **Queue** erweitert die **Collection** und bietet daher auch dessen Methode **add** und **remove** zum Hinzufügen oder Entfernen eines Elements in den Container an.
- Im Unterschied zu **offer** wird **add** eine **IllegalStateException** auslösen, wenn das Einfügen nicht möglich ist.
- Im Unterschied zu **poll** wird **remove** bei einer leeren Queue eine **NoSuchElementException** auslösen.
- Da in der Praxis Queues mit fester Grösse der Normalfall sind, sollten die Methoden **offer**, bzw. **poll** bevorzugt werden.

Zusammenfassung

- **Executor-Services, Callable<V> und Future<V>** abstrahieren die Threadverwaltung vollständig. Sie verwalten die Aufgaben in einer Queue und erlauben die Rückgabe von Resultaten.
- Mit dem Paket **java.util.concurrent.atomic** werden lockfreie, Thread-sichere Zugriffe auf einzelne Variablen von elementaren Datentypen unterstützt.
- Um Datenstrukturen Thread-sicher zu gestalten, kann man den Zugriff auf herkömmliche Java-Collections mit entsprechenden Wrapper Klassen synchronisieren.
- Für die asynchrone Kommunikation stehen verschiedene Queue-Klassen zur Verfügung. Das Blockieren und die beschränkte Kapazität garantieren, dass die Queue-Länge nicht unendlich wächst.

Fragen?